Design and Implementation of Visualization Techniques for Subsumption Hierarchies

Anis Zarrad

Thesis
In
The Department
Of
Computer Science
&
Software Engineering

Presented in Partial Fulfillment of the

Requirement for the Degree of

Master of Computer Science at

Concordia University

Montreal, Canada, 2004

September 2004

©Anis Zarrad, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94756-4
Our file Notre référence
ISBN: 0-612-94756-4

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canadä

Dedication

This thesis is lovingly dedicated to my mom, Najet, to my dad, Ali. I extend my deepest love and thanks for their constant moral and financial support and encouragement.

To my sisters, Olfa, Leila and Ines for their encouragement.

To my aunt, Nejla for her continuous support.

Acknowledgement

I would like to thank my supervisor, Dr. Volker Haarslev for his constant support and encouragement and fruitful discussion throughout my graduate studies.

I wish to extend my thanks also to Dr.Nejib Zaguia who provided full and unconditional support during my studies.

Finally, I would like to thank all the people who assisted me in completing this work.

Abstract

Design and Implementation of Visualization Techniques for Subsumption Hierarchies

Anis Zarrad

Data Visualizing is becoming an important research topic in computer science, and has received considerable attention in the last two decades. In several instances, visualization is a crucial step in order to easily access and properly understand the data. With it, the analysis and the decision making is a relatively easier task.

In this thesis, we will focus on the visualization of the concept hierarchies by producing several geometric representations. The main tools used are the graphs where the concepts are represented by vertices and the edges represent the relationships between concepts. Our specific application is the development of a drawing system that interfaces with the description logic reasonner ***RACER***.

Unless there is no error in the ontology, the ***RACER*** system responds to the taxonomy queries correctly. The body of the response must contain information about a relational structure called a concept hierarchy. This information could be saved as a text file.

In the first part of the thesis, we will present our system architecture and discuss its components then we will show how to collect the information about the concept hierarchy using the taxonomy query. We also describe methods for parsing hierarchies and the creation of an appropriate data structure that will be used by the set of algorithms we developed.

The second part of the thesis contains the algorithms used to retrieve the properties of the concept hierarchy, as well as to study the specific structure of these hierarchies. It is well known that graph drawing in general is a very complex issue and, therefore, it is important that our approach in drawing takes into account the specificity of these graphs.

We consider many aesthetic criteria that fit our specific application: the levels should be kept together as much as possible, the drawing area should be as small as possible, the number of crossings should be minimized, etc. Also, we will develop a decomposition technique that will be very useful in many instances.

Contents

Figures List ---------- viii
Tables List ---------- ix
1 Introduction ---------- 1
1.1 Description Logic ---------- 2
1.2 RACER system ---------- 3
1.3 Taxonomy File ---------- 5
1.4 Visualization of the Taxonomy file ---------- 6
1.5 Hierarchical Drawing Algorithms ---------- 9
1.6 System design ---------- 11
1.7 Thesis outline ---------- 11
1.8 Thesis contribution ---------- 12
2 Overview of the system ---------- 13
2.1 Introduction ---------- 13
2.2 Requirements ---------- 13
2.3 TBoxHDI Input/Output ---------- 15
2.4 System Constraints ---------- 17
2.5 TBoxHDI web based system architecture ---------- 18
2.6 Interface description ---------- 19
2.7 Overview of related systems ---------- 22
2.7.1 RICE system ---------- 22
2.7.2 TRIPLE system ---------- 23
2.7.3 Comparison ---------- 24
3 High level design ---------- 25
3.1 Introduction ---------- 25
3.2 Architecture design ---------- 25
3.2.1 Global Architecture ---------- 25
3.2.2 TBoxHDI Architecture ---------- 26
3.3 User interface design ---------- 30
4 Parsing ---------- 32
4.1 Introduction ---------- 32
4.2 Recursive Descent parsing approach ---------- 33
4.3 Grammar transformation ---------- 34
4.4 Parsing algorithm ---------- 35
4.5 Concept hierarchy data structure representation ---------- 35
4.6 Complexity ---------- 37
5 Drawing ---------- 38
5.1 History ---------- 38
5.2 Concept hierarchy properties ---------- 39
5.3 Proposed approach ---------- 40
5.4 High Level Description of the Drawing Algorithm ---------- 42
5.5 Details of the global algorithm ---------- 45
5.5.1 Testing the hierarchy for non-essential edges. ---------- 45
5.5.2 Layout algorithm ---------- 45
5.5.3 Layout orientation decision ---------- 48

5.5.4 Autonomous sets Decomposition algorithm -- 51
5.5.5 Layer permutation algorithm -- 54
5.5.6 Crossing edges reduction algorithm --- 55
5.5.7 Layer Bifurcation algorithm --- 56
5.5.8 The Bypassed edges draw algorithm -- 57
6 Conclusion and Open Problems--- 61
Bibliography: --- 63
Appendix A: Parser algorithm-- 68
Appendix B: Taxonomy file example --- 71
Appendix C: TBoxHDI output --- 72

Figures List

Figure 1: Concept Hierarchy. 2
Figure 2: Language constructs. 3
Figure 3: TBox axioms. 5
Figure 4: Straight line drawing. 8
Figure 5: Hierarchical drawing. 9
Figure 6: System input/output. 17
Figure 7: TBoxHDI Web System. 18
Figure 8: The bar and the file menu of the system interface. 19
Figure 9: The Display menu. 19
Figure 10: The Racer menu. 20
Figure 11: The Tools menu. 20
Figure 12: Right click menu to interact with the hierarchy. 21
Figure 13: Selection of the file that could be displayed. 21
Figure 14: The default view Top to Bottom display. 22
Figure 15: Global Architecture. 26
Figure 16: TBoxHDI architecture. 27
Figure 17: Model subclasses. 30
Figure 18: The internal taxonomy files representation. 36
Figure 19: Bottom-to-Top layout drawing approach. 46
Figure 20: Top-to-Bottom layout drawing approach. 47
Figure 21: The oval marker shows the nodes that must be joined. 52
Figure 22: The first gray rectangle represents the autonomous set Aut (T, BOT) 52
Figure 23: The black node shows the intersection between the bypassed edges. 58
Figure 24: The bypassed edges drawing using the layer bifurcation and "draw bypassed edges "algorithm. 59

Tables List

Table 1: Bottom-to-Top layout strategy. 49
Table 2: Top-to-Bottom layout strategy. 50

1 Introduction

Interfaces have received considerable attention in the context of computer science systems. With the applications getting sophisticated and the data becoming larger and more complex, it is crucial to develop suitable interfaces to allow the users to access and smoothly use the systems. Of course, the complexity of the interface depends on the application and the type of users.

In the context of computer science terms, a ***graphical user interface*** (or **GUI**) is a method of interacting [1] with a computer through direct manipulation of graphical images and "widgets" such as windows, icons, menus, check boxes, radio buttons, push-buttons etc. More specifically, a GUI is a specification for the look and feel of a computer system [4]. The user issues commands via the GUI to computer applications.

We briefly discuss some of the issues and basic concepts in the theory behind user interface design. The UI design can be organized around some basic criteria such as eliminating possible distractions in the UI, providing feedback to the user, avoiding errors or making them easy to handle, or to recover from them, and so on.

User interface design [3] is a complex software component which plays an important role in the usability of an application. From the developer's perspective, usability is important because it can mean the difference between the success and failure of a system. From a management point of view, software with poor usability can reduce the productivity of the workforce to a level of performance worse than without the system.

The design of the interface system must integrate the principle of the simplification of the user's task and only allow minimal interference in order to achieve its goal.

The first graphical user interface was designed by XeroxCoporation's Palo Alto Research Center in the 1970s [1]. But the first real graphical user interface was released in 1981 and was a revolutionary commercial computer workstation called ***Xerox Star*** using a

graphical user interface (GUI) with the nowadays familiar desktop with icon metaphors and a mouse.

The major contribution of this thesis is the development of visualization software for the concepts hierarchy, called TBoxHDI "**TBox** **H**ierarchy **D**isplay **I**nterface".

In our software, we designed a user interface that communicates with the ***RACER*** [8] system through a TCP/IP protocol to respond to a specific query ***Taxonomy***. The response can be visualized as a concept hierarchy using graph drawing algorithms.

1.1 Description Logic

Description logics [33] (DL), also named as terminological logics, are a family of knowledge representation languages. They are meant to express knowledge about concepts, individuals and their relationships *(Figure 1).*

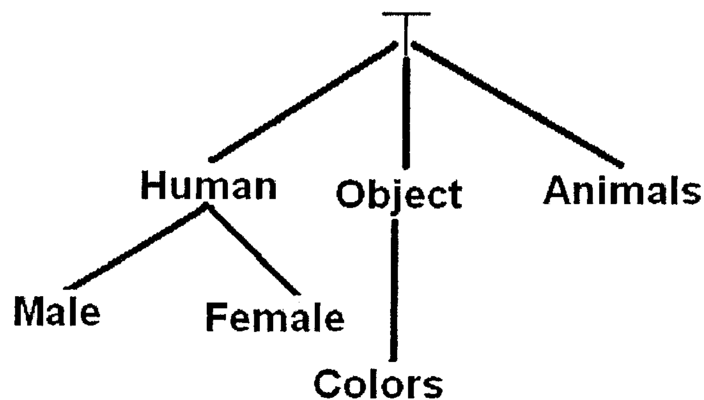

Figure 1: Concept Hierarchy.

The basic building blocks are Concepts (classes), roles, and individuals. Concepts describe the common properties of a collection of individuals and can be considered as unary, first-order logic predicates which are interpreted as sets of objects. Roles are interpreted as binary relations between objects. Description logic also provides a number of language constructs such as intersection, union, negation, role quantification, etc.

(*Figure 2)* that can be used to define new concepts and roles. The main reasoning tasks are classification, satisfiabilty, subsumption, and instance checking. Subsumption represents the *is-a* relation. Classification is the computation of a concept hierarchy based on subsumption.

$C \sqcap D$	(concept conjuntion)
$C \sqcup D$	(concept union)
$\neg C$	(concept negation)
$\exists R.C$	(existential quantification)

Figure 2: Language constructs.

The whole families of knowledge representation systems have been built using these languages and for most of them complexity results for the main reasoning tasks are known. Description logic systems have been used for building a variety of applications including conceptual modeling [38], information integration [39], query mechanisms [40], robotics [35], view maintenance [36], software management systems [41], planning systems [34], configuration systems [41], and natural language understanding [37].

1.2 RACER system

The ***RACER*** system [8] is a knowledge representation system that implements description logic reasoning. It offers reasoning and evaluation services for multiple concepts (TBox) and multiple individuals (ABox) as well. ***RACER*** can answer queries related to description logic. It is implemented in Common Lisp and has been developed at the University of Hamburg. The last version released until now is ***RACER*** 1.7 and it was developed to support different platforms such as *WINDOWS, MAC,* and *UNIX.*

The reasoning services offered by the RACER consist of:

Concept satisfiabilty: checking if a concept has a non-empty interpretation.

Subsumption checking: checking if a concept *C*2 subsumes *C*1.

Instance checking: checking if an individual denoted as *a* belongs to the interpretation of concept *C*.

Also, ***RACER*** can answer queries regarding a specific ontology [9]. An ontology is a formal conceptualization of the world. It expresses a way to bring meaning to data for an area of interest which describes the domain, key concepts, their properties, their relationships, and their associated rules. Ontology provides a set of well-founded constructs that can be leveraged to build meaningful higher level knowledge.

Reasoning [7] is important to ensure the quality of an ontology. It can be employed in different development phases. During ontology design, it can be used to test whether concepts are non-contradictory and to derive implied relations. In particular, one usually wants to compute the concept hierarchy. Information on what concept is a specialization of another one and which concepts are synonyms can be used in the design phase to test whether the concept definitions in the ontology have the intended consequences or not. During the Ontology integration, a Reasoner computes the integrated class hierarchy and checks the consistency. Reasoning may also be used when the ontology is deployed to determine the consistency of facts stated in the ontology or infer instance relationships. However, in the deployment phase, the requirements on the efficiency of reasoning are much more stringent than in the design and integration phases.

The core of a Knowledge Representation System based on Description Logics is its concept language, which can be viewed as a set of constructs for denoting classes and relationships among classes. A general characteristic of DL-systems is that the knowledge base is made up of two components: TBox + ABox.

A ***TBox*** is a collection of definitions of the concepts that describe the ontology domain. Definitions are constructed from a set of axioms (*Figure 3*).
An ***ABox*** describes a set of individuals, their properties, and their relationships.

Axioms (TBox) example:

$$\begin{aligned} \text{Woman} &\equiv \text{Person} \sqcap \text{Female} \\ \text{Man} &\equiv \text{Person} \sqcap \neg\text{Woman} \\ \text{Mother} &\equiv \text{Woman} \sqcap \exists \text{hasChild.Person} \\ \text{Father} &\equiv \text{Man} \sqcap \exists \text{hasChild.Person} \\ \text{Parent} &\equiv \text{Father} \sqcup \text{Mother} \end{aligned}$$

Figure 3: TBox axioms.

ABox Example:

Anis: Man

(Ali,Anis):HasChild

1.3 Taxonomy File

The main idea of the classification is to compute the ancestor (Parents) and descendant (Childrens) concepts for each concept name. The taxonomy is defined as a categorization of concepts based on certain criteria, it lists for each concept it's most specific sub-concepts and super-concepts.

In description logic systems it is often necessary to make legible taxonomy information about the knowledge bases expressing concept and concept hierarchy. There are two major reasons for this: firstly, retrieve the anomalies in the ontology design, secondly, to better understand the ontology's domain.

The user can receive the taxonomy information as a text file. The text file is created when the user instructs the ***RACER*** system to answer the taxonomy query which is a classification ontology process to compute the subsumption relationship between all concepts names mentioned in a TBox. The result is referred as the taxonomy of a TBox and gives for each concept name two sets of concept names listing its "parents" (direct subsumers) and "children" (direct subsumees).

The result may be given in text format:

Result example

```
1:(TOP NIL (AGE ANIMAL CAT))
2:(AGE (TOP) (BOTTOM))
3:(ANIMAL (TOP) (BOTTOM))
4:(CAT (TOP) (BOTTOM))
5:(BOTTOM (CAT AGE ANIMAL) NIL)
```

A quick explication for the output query: starting at line 1, the first token represents a concept name; the second token (NIL) means that the concept (TOP) does not have a super concept, in other words, no concept parents. The last tokens (AGE, ANIMAL, CAT) represent sub concepts for the TOP (Children Concepts).

The file retrieved from ***RACER*** is called a taxonomy file and it has a specific structure that corresponds to a well defined grammar. Our first major task is to use an efficient algorithm to parse the taxonomy file in order to retrieve all needed information about the ontology.

Graphs are frequently used in computer applications as a general data structure to represent objects and the relationships between them. Putting this definition in the context of the ontology classification, the taxonomy file could be represented as a graph. *The ordering relation between the concepts could be either* the relation HasChildrens or HasParents. The concepts may be a class. The visualization of the taxonomy file is related then to the graph visualization or graph drawing.

1.4 Visualization of the Taxonomy file

Graph drawing [12] addresses the problem of visualizing structural information by constructing geometric representations of abstract graphs. Visualization allows one to better understand the domain ontology. It essentially takes nodes and relations from an application and creates a picture or layout for them on the screen. Graph theory and order

theory may play a major role in getting a good layout that can help the user to understand the application.

There are many special kinds of graphs which are used in computing systems for different purposes: Petri Nets, Entity-Relationship Diagrams, Flowchart Graphs [13], objected oriented diagrams [14], PERT Diagrams, etc. These graphs are used in many design systems even without the aid of a computer, and each kind of graph has a special set of conventions and notations.

The ontology classification fields require a specific kind of drawing to understand the ontology design (concepts and their relation) which is called the hierarchy graph or layered graph [31].

A *layered graph* is an *oriented acyclic graph.* The *vertices* of a layered graph can be partitioned into sets ($L_0=\{s\}$, L_1, L_2, etc.), called *levels* or *layers*, such that each *edge* of the graph, that is an ordered pair of vertices, has the first component in a level L_i and the second component in a level L_j where $i<j$. Hence, there are no edges between two vertices in the same layer.

The interpretation process has to first apply rules for layer 0 as long as possible, then rules for layer 1, etc. Using rule layers makes it possible to specify a hierarchical view of ontology concepts.

The layer rule is to assign concepts without parents in layer 0 then place the children of the concepts in layer 0 into layer 1, and so on.

It is a fact that there is no generic approach for drawing graphs. The same observation also applies to the size of the graph. It is essential to study the special properties of the graph representing the concept hierarchy and to understand to which class of graphs it belongs to in order to decide which specific drawing algorithms to apply.

Using a hierarchical approach to represent the taxonomy file has a number of advantages.

-When we examine the graphical representation of a model we use our visual cognitive apparatus which has some millions of years of evolutionary advantage over our text-reading abilities. The two-dimensional representation of a diagram is a lot more expressive than text, which is typically scanned from left to right and from top to bottom. Diagrams can be viewed following different directions to gain distinct insights:

- Keeping in mind the ontology concept design which is related to the classification process;
- Ergonomically friendly presentation of the ontology;
- Easy browsing and navigation through the ontology;
- Efficient retrieval and searching of a specific concept name,

The layout must satisfy several aesthetics, to create a "good" layout. Aesthetics differ from one application to another, but typically include items such as the avoidance of edges and node crossings, reducing the space, favoring short straight edges, and the preservation of a minimum distance between nodes, etc. To achieve the aesthetic goal we may adopt three major drawing parameters for the concepts hierarchy display:

Straight line drawing: each connection is drawn as a straight line segment (*Figure 4*).

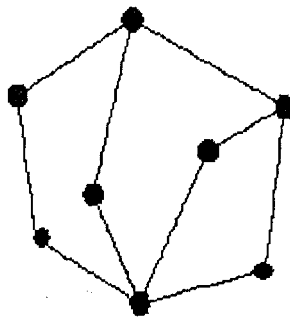

Figure 4: Straight line drawing.

Aesthetics: properties applied to the drawing achieve the understandability and readability of concepts. The major important element in this parameter is to reduce edge crossing. This problem was first studied by Warfield [43] and similar methods were discovered by

Carpano [44], Sugiya, Tagawa, and Toda [45]. There has not been many results dealing with this problem, probably due to the difficulty of the problem. In this work we follow Di Battisa and Tamassia's approach [12].

Space minimization: the concepts hierarchy display should fit in one screen, if the output is larger than the screen width, many techniques may be applied to reduce the display width.

1.5 Hierarchical Drawing Algorithms

In general, a concept hierarchy drawing algorithm (*Figure 5*) reads as input a concept hierarchy description and produces a drawing of this hierarchy as output according to a given graphic standard. The drawing is described in terms of graphic methods such as *AssignLayer*, *Draw_Line*, *Draw_Concept*, etc.

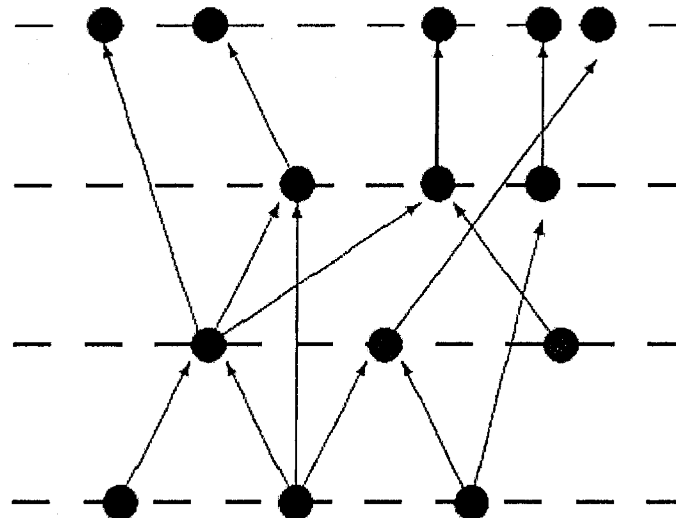

Figure 5: Hierarchical drawing.

The hierarchical concept drawing algorithm has four major steps or sub-algorithms.
The first step places the nodes in discrete ranks using a layout algorithm. This step will be applied in both orientations: start assigning layer from Top to bottom or from bottom to top, and then choose the best orientation using specific criteria to do the next step.

For the second step we will use a decomposition technique. We noticed that most of the examples we are dealing with, especially large sized ones, have many subsets with the following characteristic: all elements in the subset have the same nodes neighborhood. In graph theory literature these are called *autonomous sets*. An autonomous set is a subset of concepts S such that all concepts in S have exactly the same relationships with the concepts outside S. Since all elements in an autonomous set behave the same way, then we could collapse all elements in the drawing into one object and allow the user to interactively see the content of the object. This will greatly simplify the drawing without any loss of information, since all relationships are still represented. Therefore, concepts having the same set of parents and children will be in one homogenous node. Of course, the user has the ability to turn this feature on and off.

The crossing between edges is a major readability problem for the human user. However, there are no efficiency algorithms that minimize the number of crossings in a general graph. However, we could design heuristics that fit well in specific applications. In our case, and since we are interested in a layered drawing, the permutation of the elements used to draw each layer is crucial in order to minimize the crossings. We will present an algorithm that finds the right permutation to be used for each layer and mainly depends on the inclusion relations between the sets of children and parents of each element in the layer. This algorithm will decide the actual layout coordinates of the concepts.
Layer_Bifurcation and Draw_edges algorithms deal with the edges that cross at least one layer. The drawing of those specific edges needs a special attention which will be explored in the last chapter. We provide the main procedure that contains the most important functions that must be considered in this thesis.

Procedure Draw_Graph

Begin

Layout ()

Autonomous_sets()

Cross_Reduction ()

Position ()

Layer_Bifurcation

Draw_Edges

End

1.6 System design

The system design is based on a two-tiered system. An applet will be downloaded at the user terminal and will communicate with the ***RACER*** system that resides on the server. The user may use the layered design because of the following reasons:

- The reusability and maintenance of the system; and
- To simplify and separate the major components in the design system.

1.7 Thesis outline

In chapter 2, we discuss the necessity of a web based system and present a detailed overview of the system we developed. We go through the constraints, requirements, input, output, etc. in detail.

In chapter 3, we introduce two existing systems and their functionality. Then, we do a comparison study with our own system and show its advantages.

In chapter 4, we focus on the high level design of our system and give a detailed view of the system architecture as well as the interface design.

In chapter 5, first we study the structure of the graph representing the concepts hierarchy then we describe in detail the approach that will be used to create an understandable and readable drawing.

In chapter 6, we will be interested in the parsing process. We design a suitable algorithm that will be used to parse the taxonomy file, and then we create the appropriate data structure to handle the information about the concepts hierarchy. The complexity of these algorithms will be presented.

In chapter 7, we present what we consider to be a very important part of this thesis. We will show the design and the implementation of a set of algorithms essential for the drawing. These include the layout algorithm, autonomous sets, and cross reduction and position algorithms. The complexity of these algorithms will also be discussed. Finally, a conclusion and a set of proposals for future work will be presented in Chapter 8.

1.8 Thesis contribution

The main contributions of this thesis are as follows:

-We designed and implemented a system interface that will query RACER for the data related to a hierarchy of concepts in a TBox, and then use several drawing algorithms in order to visualize the hierarchy in a desirable view that is easier to understand by the user and reduces the chance of a confusing and misleading concept hierarchy drawing.

-An efficient way of ranking the nodes; the target of this step is to assign a y-coordinate to each node;

-An efficient algorithm to find the autonomous sets that consist of merging nodes that have the same neighbors;

-Because the crossing can reduce the readability and visibility for the user we produce an improved heuristics to reduce edge crossing;

-A method for computing the coordinates x and y for each node; this can be seen as a rank assignment problem;

-We must provide an intelligent algorithm to draw the edges that pass through more than two consecutive layers, in many cases those edges intersect with the node shape and this could affect the view display.

2 Overview of the system

2.1 Introduction

The main task for the TBoxHDI system is to generate a hierarchical graphical view using the *Taxonomy file* generated by the *RACER* system.

The hierarchical graphical display of the TBox structure remains the universal and most familiar and understandable representation for the user.

The TBoxHDI system is meant to be suitable for visualizing large hierarchical data sets as well. For a useful display, there is a need for a set of flexibilities and features to be developed: zoom, rotations, improved animation, etc. This leads to a better interactivity and an easier browsing of the hierarchy. Using a fixed sized shape for the nodes, the TBoxHDI display should help the user to better manage and comprehend the ontology.

This could be achieved by focusing on the important characteristics of the concept hierarchy and by an interface design that simplifies the user's task as much as possible.

2.2 Requirements

For the TBoxHDI we have chosen to use a direct-manipulation, with an interaction between the taxonomy file and screen representations.
Using the system interface the user requests a taxonomy file through a specified communication protocol and then, using the *taxonomy* command, *RACER* sends the requested file back.

In the design of the TBoxHDI system, we took care of several visualization concerns and issues of the users. As we already mentioned, the system should respond to our most important concern, which is the readability of the concepts hierarchy described in the taxonomy file as well as the simplicity to use the different features of the system by the user.

Several goals have to be achieved at the same time.

-Make a decision about the ontology design approach that has been used.

-The ability to evaluate the ontology and to detect the anomalies in the design. For instance, we should be able to easily check whether the ontology does fulfill the requirement and specification of the domain. It is also important to be able to locate duplicate concepts, the meaning of the ontology, and which design methodologies have been used.

-Easy search for a specific concept name.

-Easy access to the detailed information about a specific concept.

-The system must be extensible because the number of available display algorithms and tools is constantly growing.

To achieve these goals we incorporated several ideas and features in our system:

-A web-based system: nowadays it is obvious that our system should be web-based. The TBoxHDI is accessible over the normal HTTP Web protocol. It should be modular and flexible for the user.

-The use of mouse clicks is an important feature. It should allow easy interaction between the different parts of the display. With mouse clicks, it will be possible to have access to the granular information about a specific or group of concepts. Also, we may be able to select a specific area of the screen in order to investigate the details of the local information or to have a better view of a subset of concepts.

-We should be very careful about minimizing the area of the screen used for the display. The best situation would be to use a single screen. Scrolling has shown to be a big distraction to the user.

-We do not pretend that there is a single approach that could resolve all drawing issues. Graph drawing is simply a very difficult task which has been shown in the

literature. Therefore, it is very helpful to propose to the users several ways or algorithms to do the drawing. The choice of the specific way to display the concepts depends on the specific properties (small, large, dense, etc.) of the concepts hierarchy.

-The use of different colors is also important. Coded colors will greatly simplify the presentation of the data and minimizes the use of text on the screen. Edges with different colors will have different meanings; highlighted parts on the screen will help the user to focus in the desired part of the data, etc.

-The system uses two different shapes for representing the concepts in the hierarchy: circles and ovals. Their meanings will be specified later. We did avoid the use of more shapes in order to not confuse the user. Of course, for aesthetics reasons and in order to control the area used, the size of the different used shapes cannot be arbitrary. However the user could access all text information related to a specific node by executing a mouse click.

-The ability to rotate the display could help to better understand the data and is an important feature implemented in our system.

-In many cases, the concepts hierarchy is very large (the concepts could number in the thousands) and therefore zooming features are very important. It will allow access to the structural information of the different parts of the display.

2.3 TBoxHDI Input/Output

The system accepts as input four kinds of files as described below (*Figure 6*) those files are transmitted to the racer to be classified:

RDF file : The Resource Description Framework (RDF) is a framework for representing information on the Web. This document defines an abstract syntax on which RDF is based, and which serves to link its concrete syntax to its formal semantics. It also

includes a discussion of design goals, meaning of RDF documents, key concepts, data typing, character normalization and handling of URI references.

DAML file: The DARPA Agent Markup Language (DAML) is a language designed to express information so that it can be easily used by computer programs (analogous to the way HTML is designed to express information so that it can be easily used by the user). This is expected to foster the use of intelligent agents and to aid both humans and programs in finding and using information.

XML file: Extensible Markup Language (XML) provides a portable means of representing data structure with one type of relation: containment. Containment and different tag names (e.g. <mother>, <father>, <son>, <daughter>) are used to model different relationships among otherwise identical objects. XML vocabularies are generally fixed by a DTD or XML Schema definition. One can copy or include someone else's definition in your own, but can not add additional information to their definition.

RACER file: defines the vocabulary for discussing and representing information about a particular domain. A Racer File consists of Classes, Properties, the relationships between them, and constraints on their use.

The output of the system is a concept hierarchy for a specific file in a layered graph structure.

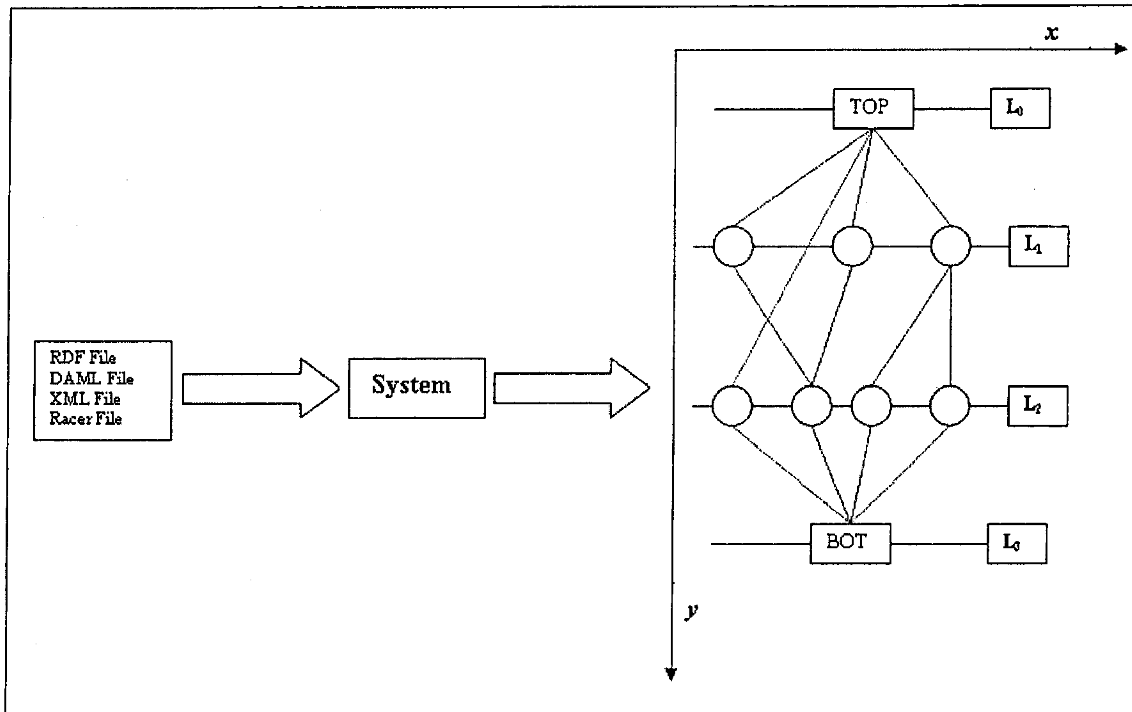

Figure 6: System input/output.

2.4 System Constraints

The term 'Constraint' refers to the specification of the graph. The TBox hierarchy may contain, at many concepts and we would use a predefined shape with a fixed size to draw those concepts (classes).

When we have a large concepts hierarchy, it is clear that we cannot represent that many nodes on the same screen. In the best case scenario, the screen resolution could be set at 1024*800 pixels. The TBoxHDI needs a high screen resolution since the presence of a graph of a size over 1024*800 pixels will almost surely force the user who has a low resolution display to use horizontal and vertical scrollbars.

It becomes really annoying when the viewer tries to understand the whole graph and has to scroll every few seconds to look at the next part and hide another one. It then becomes hard for the user to get a complete idea of the graph information, which means one of the requirements is discarded.

By using a high screen resolution we augment the number of nodes that can fit in one layer. Of course, we assume we use efficient and sophisticated drawing algorithms.

The system can run on the following operating systems: MAC, Windows 98, Windows 2000, Windows NT, and Windows XP.

2.5 TBoxHDI web based system architecture

As internet technology has become essential to software development, the software might be useful and efficient using a web-based system in order to be accessible over the internet. The TBoxHDI web system provides a mechanism of communication between two remote systems that are connected through the Web Service's network.

Using a web based system for TBoxHDI should give the system more modularity and flexibility for the user, since it will be accessible over a standard HTTP Web protocol (*Figure 7*).

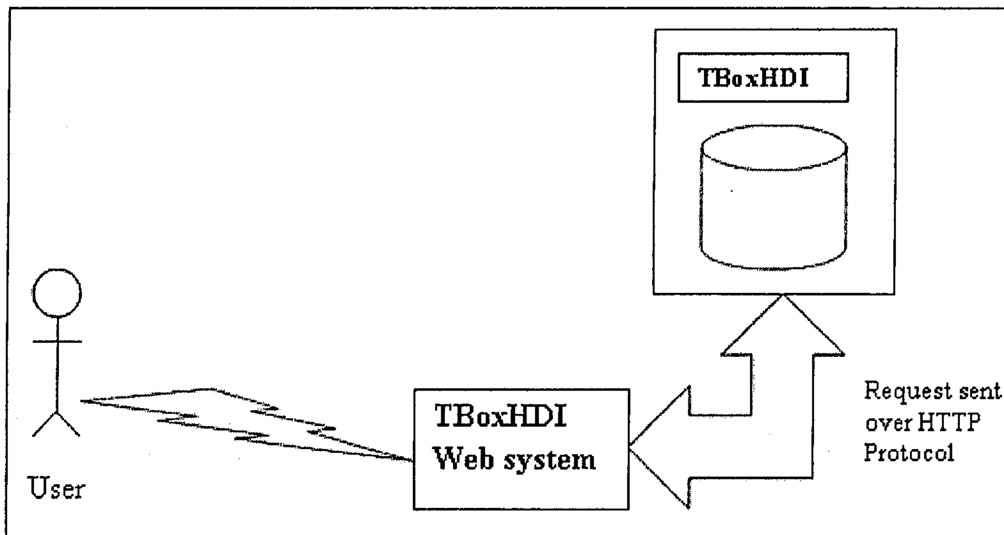

Figure 7: TBoxHDI Web System.

TBoxHDI Web-based System Manager can be configured to run in Client server mode. The client requests server services from a managed machine through intend port 8080

(default port). Client-server mode needs to be enabled on the servers destined to be managed as remote machines

2.6 Interface description

Figures 8 to 14 show a series of screen captures of the main display area with some principles scenarios. They Show progressive steps to display and interact with the hierarchy concept. The user can use the mouse and the menu icons to retrieve the necessary information about the selected concept name.

Several layout displays may be applied to allow adjustments of the spacing between nodes, alignment, clear vision, and the best display in the available screen space.

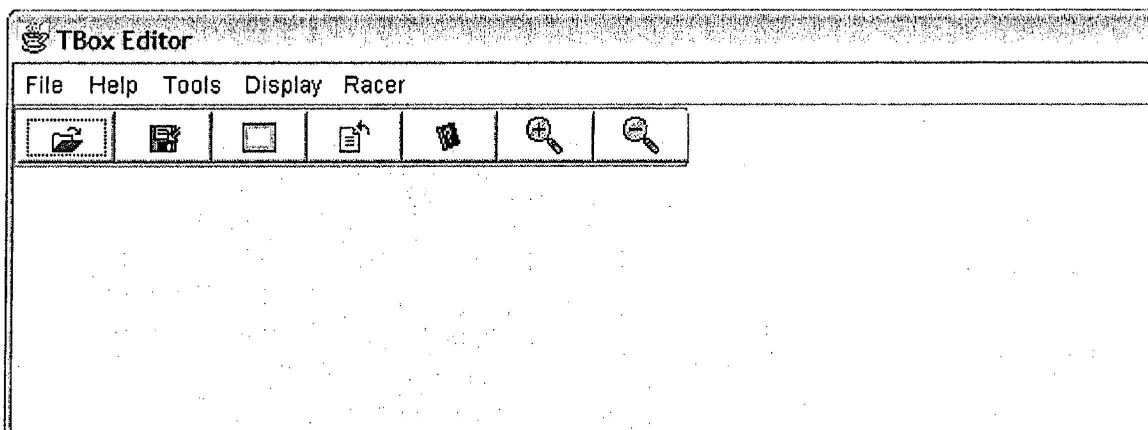

Figure 8: The bar and the file menu of the system interface.

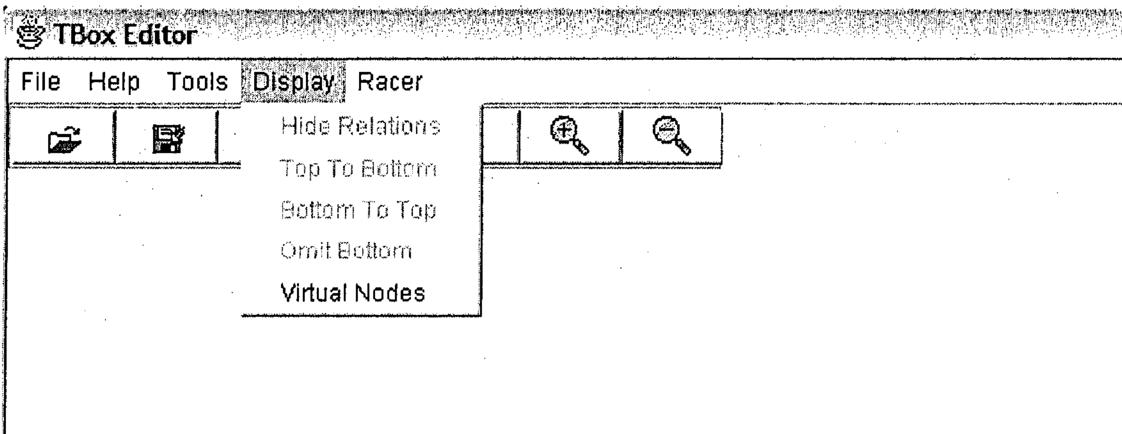

Figure 9: The Display menu.

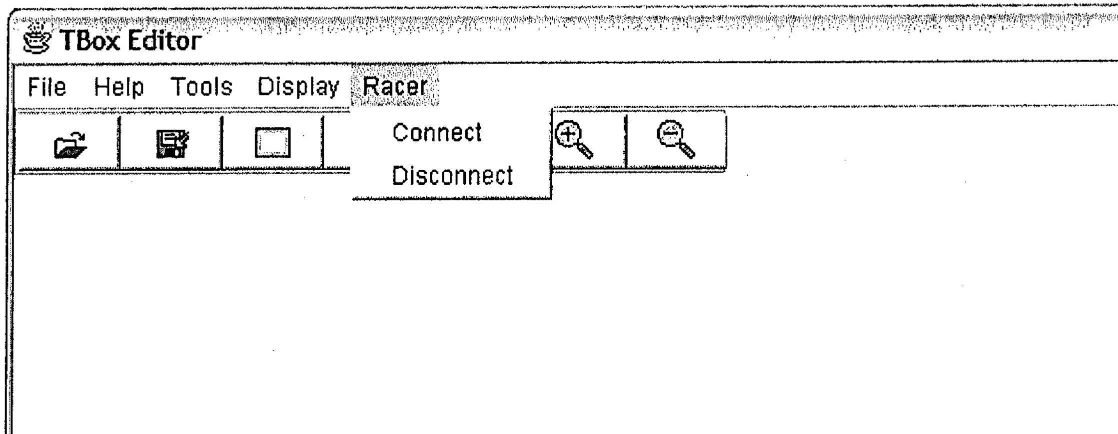

Figure 10: The Racer menu.

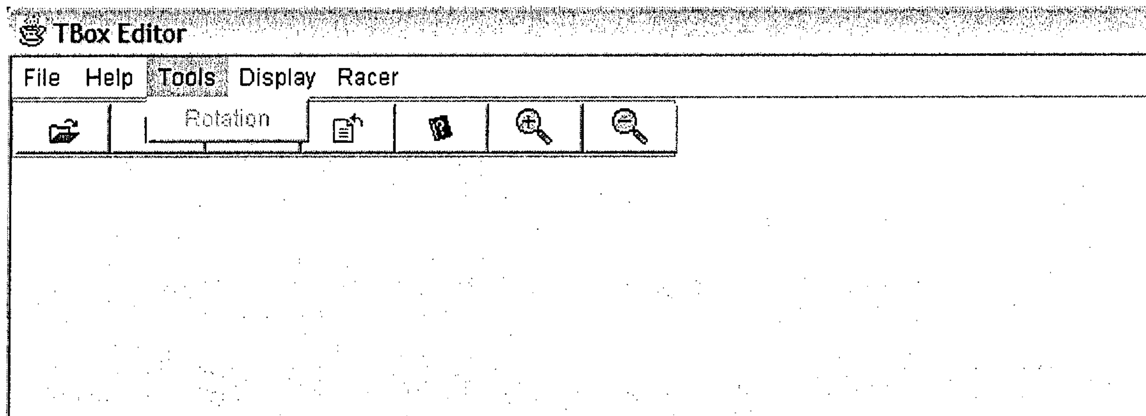

Figure 11: The Tools menu.

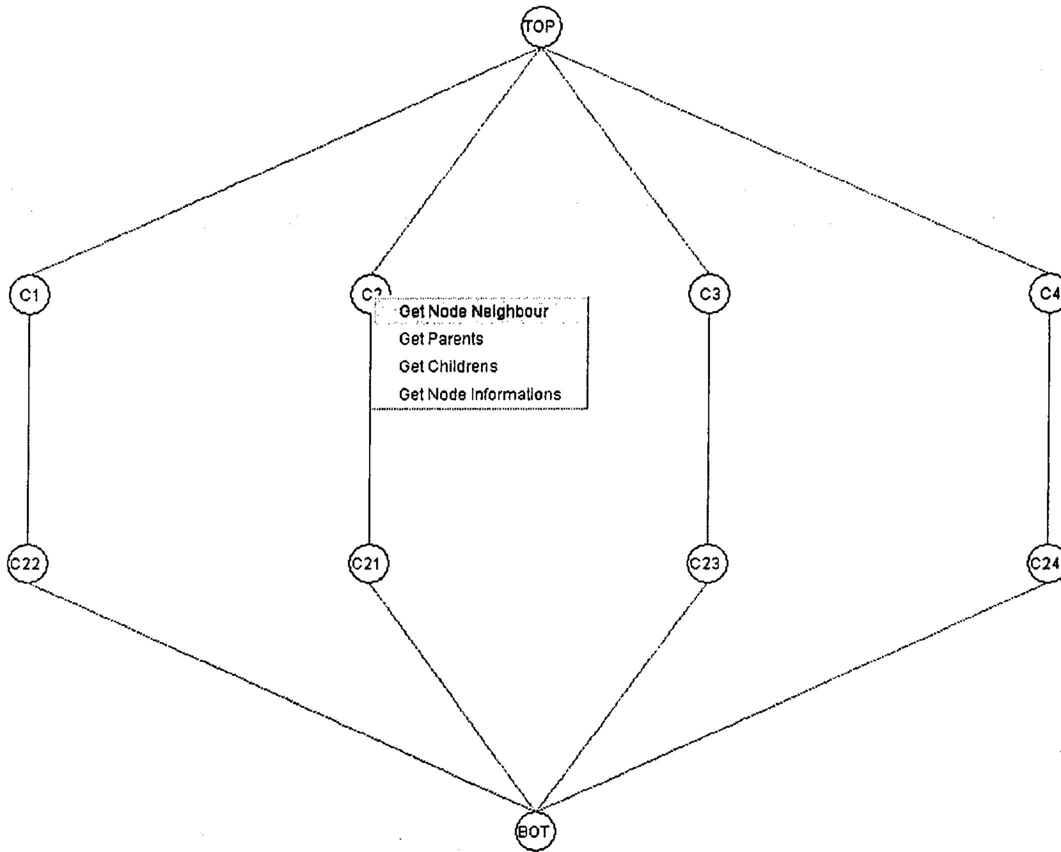

Figure 12: Right click menu to interact with the hierarchy.

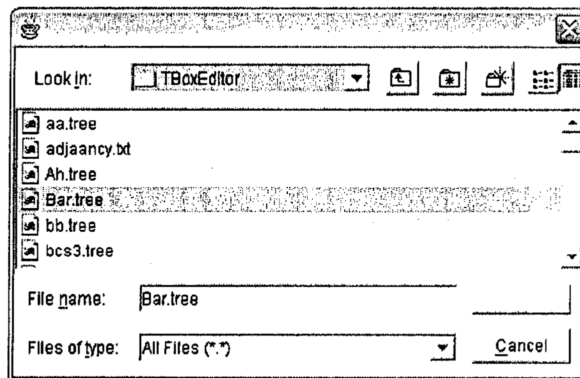

Figure 13: Selection of the file that could be displayed.

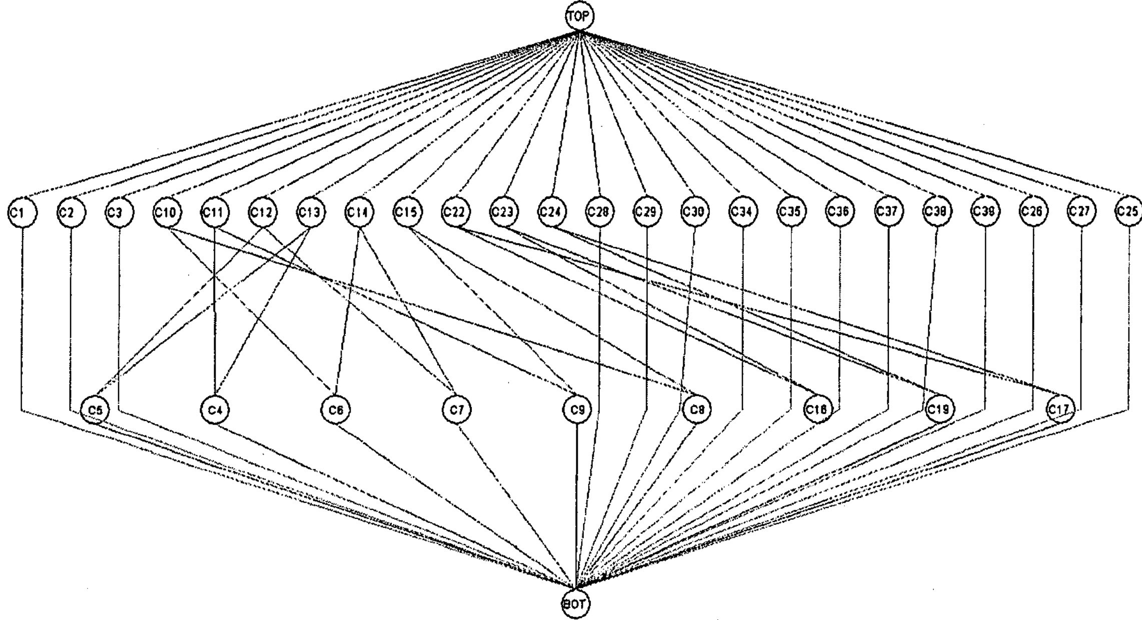

Figure 14: The default view Top to Bottom display.

2.7 Overview of related systems

There are two well known visualization techniques for the concept hierarchy: classical representation and exploring tree representation. Many of those visualization techniques work effectively for hierarchies of 1000 nodes. But as the number of nodes increases toward 5000, these techniques tend to break down. When the hierarchies contain a large number of nodes, we usually require an efficient technique to display the concept hierarchies.

There are at least two systems that offer the display of the concepts hierarchy, namely RICE and TRIPLE which use a classical visualization technique.

2.7.1 RICE system

The RICE system [46] (*RACER Interactive Client Environment*) has been developed by Ronald Cornet at the Department of Medical Informatics-University of Amsterdam. The

RICE system provides an interface that manipulates the description logic as processed by the DL reasoner.

It connects with the DL reasoner and displays the classified knowledge in a MS Windows file manager style. Individuals for each concept are displayed on demand.
The major problem with RICE is: it use a naïve drawing technique to display the edges and the individuals. The notion of ancestor and descendant concept is lost. Moreover, it does not give a global view of the hierarchy, a crucial aspect, in order to understand and properly analyze the structure.

2.7.2 TRIPLE system

The TRIPLE system is an RDF query, inference, and transformation language for the Semantic Web. Instead of having a built-in semantics for RDF Schema (as many other RDF query languages have), TRIPLE allows the semantics of languages over RDFs (like RDF Schema, Topic Maps, UML, etc.) to be defined with rules. For languages where this is not easily possible (e.g., DAML+OIL), access to external programs (like description logics classifiers) is provided.

As a result, TRIPLE allows RDF reasoning and transformation under several different semantics; it also offers a web-based system interface to display the class hierarchy in a layered graph.

TRIPLE is a joint project with Stefan Decker (Stanford University Database Group) and Michael Sintek (DFKI GmbH Kaiserslautern, Knowledge Management Department and Stanford University Database Group) as its creators.

Although it displays the hierarchy in a layered fashion and keeps a global view, it lacks the sophistication in the drawing algorithm so it does not seem to be able to have acceptable drawings, especially when the hierarchy is large. Also, there is no possibility for the user to interact with the given drawing in order to adjust it if necessary. It only

gives a static graph. We find the curve lines used in the drawing confusing, especially when there are a lot of intersecting edges in the drawing.

2.7.3 Comparison

Many techniques were developed to visualize and explore large hierarchies [1,2,3]. Generally, when the user knows the ontology domain, how it is structured, and what he is searching for, then the display technique could be easier to implement. Rice uses the exploring tree (File manager windows) to categorize each concept according to it is subsumers and subsumees. With this approach a user cannot have a global view if the graph does not fit into the available screen space.

It seems that a system such as TRIPLE uses a very standard and basic technique for drawing the hierarchy. This technique has very big limitations. In fact, it is enough to have a hierarchy where the number of edges is large to see the problem. The drawing becomes completely unusable. Roughly this algorithm is:

Decompose the graph by layers and then distribute the vertices of every layer around the center of the screen in a regular way.

We attempt to provide a simple yet powerful method to explore very large hierarchies. The strategy maintains context within a complex hierarchy while providing easy access to details without using techniques such as clustering, which could affect the visual stability of a hierarchy. Several algorithms could be used for drawing and the choice should depend on the characteristics of the hierarchy. Our system should be able to first check the properties of the hierarchy and then automatically choose the right drawing methodology. In this case the visualizations of the TBox relational structures should be useful and understandable. The properties of the input graph are an essential parameter of a graph drawing. This approach is discussed more deeply in the following sections.

3 High level design

3.1 Introduction

We are designing a complex application to display a layered graph view. It is composed of a large number of components across multiple levels of abstraction between the data, the user interface, and the interaction between them. The application must support such operational requirements as maintainability, reusability, and probably security.

Many design patterns were introduced in the domain of computer science to reduce the complexity of such task. One commonly used implementation to look at complex software is a high level design pattern called *Model-View-Controller* (MVC). It splits the application into less dependent pieces.

The *Model-View-Controller* pattern is one of the earliest and one of the most successful design patterns. It was developed by Trygve Reenskaug in 1979. The MVC architecture was introduced as part of the Smalltalk-80 version [20] of the Smalltalk programming language invented by Alan Kay. The *Model-view-Controller* design pattern is one of the fundamental ways to architect a program that involves both user interfacing, a "view", and an internal data processing, a "model". The goal of the MVC pattern is to separate the application object (model) from the way it is represented to the user (view) from the way in which the user controls it (controller).

The MVC paradigm splits the design of the software into three key areas: ***View***, ***Controller*** and ***Model***.

3.2 Architecture design

3.2.1 Global Architecture

The system architecture is based on two-tiered system architecture (*Figure 15*). On the user's side, an applet will be downloaded and then a network connection using a simple

TCP/IP protocol could be set up to communicate with the ***RACER*** system that resides on the server side.

The TBoxHDI system is a server-based software that allows users to operate in a Windows environment and on most workstation platforms. The supported platforms include Macintosh and UNIX. The server runs the ***RACER*** application using an independent architecture. The RACER System must classify the ontology and then respond to the taxonomy query of the user, and send the response back to the client application for viewing and control. The client users see and work only with the user application's interface. Users are unaware that the ***RACER*** application is running on a server (the provider) and not on their PC. The only two parameters that the users will need to set up the connection are: the residual ***RACER*** IP address and the port number.

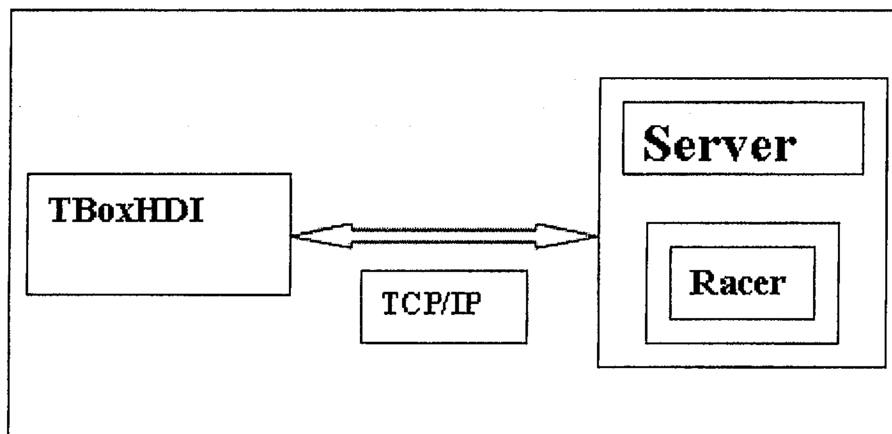

Figure 15: Global Architecture.

3.2.2 TBoxHDI Architecture

The challenge of the system architecture is to manage the software complexity. Layered solution based on the MVC pattern may be applied (*Figure 16*) in order to provide simplicity, flexibility, and adaptability to the system.

The benefits of the MVC architecture are: set out the strategy for the future developments and maintenance, code reuse, clear definition of the system, and dependency management.

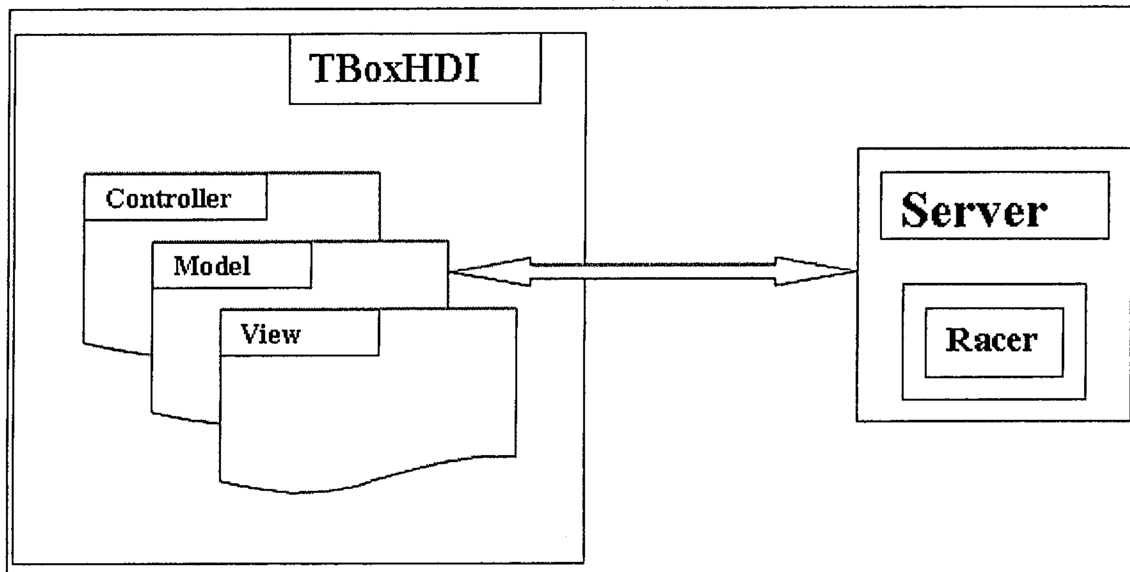

Figure 16: TBoxHDI architecture.

We define the services that can be offered by each layer:

First layer: A ***View*** provides graphical user interface (GUI) components for a model. This class coordinates the appearance of the GUI. It decides where all the controls and allocated displays (labels, text fields, buttons, concepts Box, relation line, etc.) are positioned. When notified by the model that it has changed its state, the view updates itself by calling methods in the model that return all the information that the view must display. The View class contains lots of specifics about how the GUI looks (fonts, sizes, foreground colors, background colors, as well as several details required for the placement of its components). When a user manipulates a view of a model, the view informs a controller of the desired change. In Java the views are built with AWT or SWING components. In general, all needed graphical components such as buttons, menu icons, panels, selected menus, etc. are listed in the java library. However, views must implement the java.util.Observer interface.

Second layer: The ***Model*** is the central class that represents the most important part in the design because it carries out huge tasks and operations. It has a set of public functions that can be used to achieve all of its functionality. Some of those functions are query methods that permit a "user" to get information about the current state of the model. Others methods permit the state to be modified and the rest deal with the parsing process, data collection, drawing algorithms, and network connections.

The model layer handles the most important methods for the system and has a complex structure design. We may use the layered approach [19] in the model class itself *(Figure 17)* to reduce the complexity. When we apply this approach, there are typically two major different parts that can be explored, namely *modeling* the problem and then *solving* it.

Problem *Modeling* can be viewed as breaking the problem up into layers and defining the functionality of each layer. We separate the components system into layers [19]. The components in each layer should be cohesive roughly and at the same level of abstraction. Each layer should be loosely coupled to the layers underneath. We define the set of subclasses in the ***Model*** class (*Figure 17*).

The **first subclass** represents the network connection. It delivers the connection between the TBoxHDI and the ***RACER*** system to classify the ontology and to respond to queries. This layer is related to the network service and to devices that require connectivity. In this layer, we use the TCP/IP protocol as the underlying transfer protocol. This protocol allows the developers to use existing libraries for implementation. A richer security protocol may be adopted in future work if we need a secure connection to exchange important ontology data between the system and ***RACER*** application. However, for the purposes of this work we felt that the standard protocol is sufficient.

The system calls the socket function to create a socket and connect to the racer socket once the connection is in place. The client sends the ontology file to ***RACER*** through this connection and asks the receive function to get the taxonomy query response from the

RACER application. When all the data has been received, the system calls the close function to close the socket.
The default port number for the ***RACER*** system is 8088.

The **second subclass** creates an appropriate data structure that must be used for the upper layer through a set of procedures such as the data collection process and the parsing algorithm. The use of appropriate data structures is required in order to have more efficient algorithms.

The **third subclass** contains all the drawing algorithms for constructing the geometric concept hierarchy graph. Several result and graph proprieties must be applied to implement such an algorithm in a sophisticated and efficient way to reflect the major thesis contributions.

Finally, the **last subclass** manages the behavior of the data. It should be able to register the views, notify all of its registered views when any of its function causes its state to change. (The view should then call other methods in the model to know what to display).

In Java, the View Model Class consists of one or more classes that extend the class java.util.Observable. This super class will provide the register/notify infrastructure needed to support a set of views. Most code of this class is implemented using methods and attributes already provided by the java library.

Then we define the *Solving* problem which can be viewed as showing how the layers interact with each other in order to achieve the desired outcome. To make the system code more usable and comprehensible by the developer, each layer must interact with the other layer specifically through well defined interfaces. We use the down to top approach (*Figure 17*) to interact between the **Model** subclasses. The down most subclass use one or more services of the upper level layers.

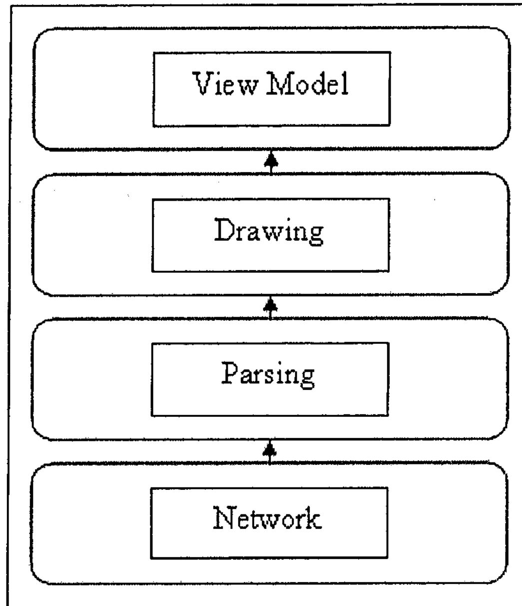

Figure 17: Model subclasses.

Third layer: this represents the ***controller*** class and updates the model as necessary when a user interacts with an associated view. The controller can call update methods of the model to get it to update its state, in which case the model will notify all registered views that a change has been made, so they will update what they display to the user as appropriate.

In many programming languages such as java, the controller is presented as listener; it can be used and called directly from the event structure java library.

3.3 User interface design

When we design a user interfaces, we must work hard to ensure the interface is themed appropriately and integrates seamlessly with the content, as well as being logical and functional for the user.

Interface graphics design is concerned with how effectively users complete the tasks and how the interface looks like. It is also related to the aesthetics and the user behavior as

well. The primary goal of this user interface application is to display a hierarchical view in the panel drawing area. This can be accomplished by placing concept names in a circle shape and the subsumption relationships between them using a colored straight line in the drawing area.

The challenge for the designer is to create an acceptable user interface that incorporates multiple inputs to produce a layered graph viewed as output. The TBoxHDI interface is a tool for prototyping the concept hierarchy display in user interfaces. This tool creates a hierarchical view using a bar menu icon or the file menu depending on the user ergonomics. The user can interact with the system interface using the mouse to sketch some query and operations for a specific concept name selected by the user to better understand the hierarchy.

To facilitate the transition from a global view to a local view, a pop up menu was built using the right mouse click with an option for accumulating ideas and understanding the appropriate local view.

4 Parsing

4.1 Introduction

After sending a taxonomy request to the ***RACER*** system, the specified ontology will be classified. The response is received by the user application through the network connection and then used as the basis for generating concepts hierarchy display.

The body of the response query is a list of triples and each triple consists of

-A name set of the atomic concept and its synonyms.

-A list of concept-parents name sets where each entry is a list of concept parents and their synonyms.

-A list of concept-children name sets where each entry is a list of concept children and their synonyms.

The response to the taxonomy query follows a special grammar described below:

Triple → <Triple> | (<Node> < Parents> {Children})

Node → (<NameList>) | <Name>

Parents → (<NodeList>)

Children → (<NodeList>)

NodeList → (<NameList>) | <Name>

NameList → <NameList> | <Name>

Name → <String>

The goal of the parsing process is to create a data structure that can be used as input for the drawing process.

There are two primary methods used when building parsers: Top-Down and Bottom-Up. The Top-Down process attempts to construct the parse tree for some sentence of the language by starting at the root of the parse tree. In top-down parsing/recognition we start at the most abstract level (the level of sentences) and work down to the most concrete

level (the level of words) using *left-to-right* rules. Bottom-Up attempts to build the parse tree by starting from leafs (the level of words) of the tree until we reach the root.

Top-Down parsers are generally easy to write manually. Bottom-up parsers are usually created by parser-generators and tend to be faster.

4.2 Recursive descent parsing approach

One of the most straightforward forms of parsing is recursive descent parsing [23]. It is a Top-Down process in which the parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. A basic operation necessary for this involves reading token from the input stream and matching them with terminals from the grammar that describes the syntax of the taxonomy file. When proper matches occur the input stream reading pointer will look ahead one token and advance.

What a recursive descent parser actually does is perform a depth-first search of the derivation tree for the string being parsed. This will allow exploring the possibilities of rules that could apply.

In general, for each non-terminal we write one procedure, and we call it the non-terminal's procedure. For each terminal, we compare the current token with the expected terminal and decide which rule to use. If there is only one production rule then it is chosen. However, if there are multiple productions for a non-terminal, we use an if-then-else statement to choose which rule can be applied. The structure of the procedure is dictated by the production for the corresponding non-terminal. If there was a left recursion in a production, we would have had an infinite recursion that causes a recognition problem.

The recursive descent parsing is quite efficient. Each non-terminal in the parse tree requires a simple table lookup. Each terminal node requires a simple match. The running time is polynomial to the size of the taxonomy file.

4.3 Grammar transformation

Unfortunately, there are some problems with this simple scheme. A grammar rule is said to be *directly left recursive* if the same non-terminal is both on the left hand side of a rule and the first element on the right hand side of a rule. Hence, after a directly left recursive rule has been selected, the first action of the corresponding parsing procedure will be to call itself immediately without consuming any of the input. It should be clear that such a recursive call will never terminate and a recursive descent grammar will go into an infinite loop. Hence, a recursive descent parser cannot be written for a grammar which contains such directly (or indirectly) left recursive rules.

The problem with the taxonomy grammar is the recursive rules as found in the *Triple*; the right hand side calls *Triple* that causes a self loop. When we start parsing the file, we first call the *Triple* rule; we start by trying its first alternatives. Which is *Triple* , trying *Triple* means trying *Triple* again, etc. This causes an indefinitely loops. The Top-Down approach cannot work on this grammar because the same problem may happen again when we call NameList terminal symbol. Fortunately, it is possible to transform the taxonomy grammar to resolve the infinite loop problem caused by the left recursive rules. The usual way to eliminate left recursion is to introduce a new non-terminal to handle all but the first part of the production.

TripleList → Triple| TripleNode

TripleNode → TripleList

TripleNode → ε

Triple → (<Node> <Parents> <Child>)

Child → <Children>

Child → ε

Parents → (<NodeList>)

Node → (<NameList>) | Name

Children → (<NodeList>)

NameList → <Name> | <NameChain>

NameChain → NameList

NameChain→ ε

NodeList→ <Node> | <NodeChain >

NodeChain → NodeList

NodeChain→ ε

Name → <String>

4.4 Parsing algorithm

We treat the grammar as a pattern of tokens. It is practical to tokenize the taxonomy file, and specify a grammar for the file in term of token terminals. The token terminal are "(", ",", ")", spaces, and names. The name is a set of characters that does not contain any token terminal. We create a procedure for each terminal in the transformed grammar which returns a Boolean variable when the parsing is done.

The Top-Down algorithm is based on the idea of using a generative grammar to produce all possible sentences in a language until one is found which fits the input sentence.

Construct the root of the parser.
Repeat until the fringe of the parsing tree matches the input string.
1- At a node labeled *A*, select production that have *A* on its left hand side and, for each symbol on its right hand side, construct the appropriate child.
2- When a terminal symbol is added to the fringe and it does not match the fringe, backtrack.
3- Find the next node to be expanded.

The parse procedure code is shown in the appendix A.

4.5 Concept hierarchy data structure representation

The TBox hierarchy can be seen as a *graph* $G=(V,E)$ that consists of a finite set V of vertices and a finite set E of edges, that is a set of ordered pairs (u,v) of vertices in V. In

the ontology classification, the vertices may be a concept name or class name and the edges are the ordering relation between the concepts.

As experience shows, and for large data, building the right data structure has a crucial effect on how easy the implementation could be as well as on the quality of the final result. An adjacency list data structures is chosen because certain algorithms will be used that work best with this particular data structures, and provides a compact way to represent graphs. We create two standard ways to represent the TBox hierarchy as a collection of parent's adjacency lists and a children's adjacency list (*Figure 18*).

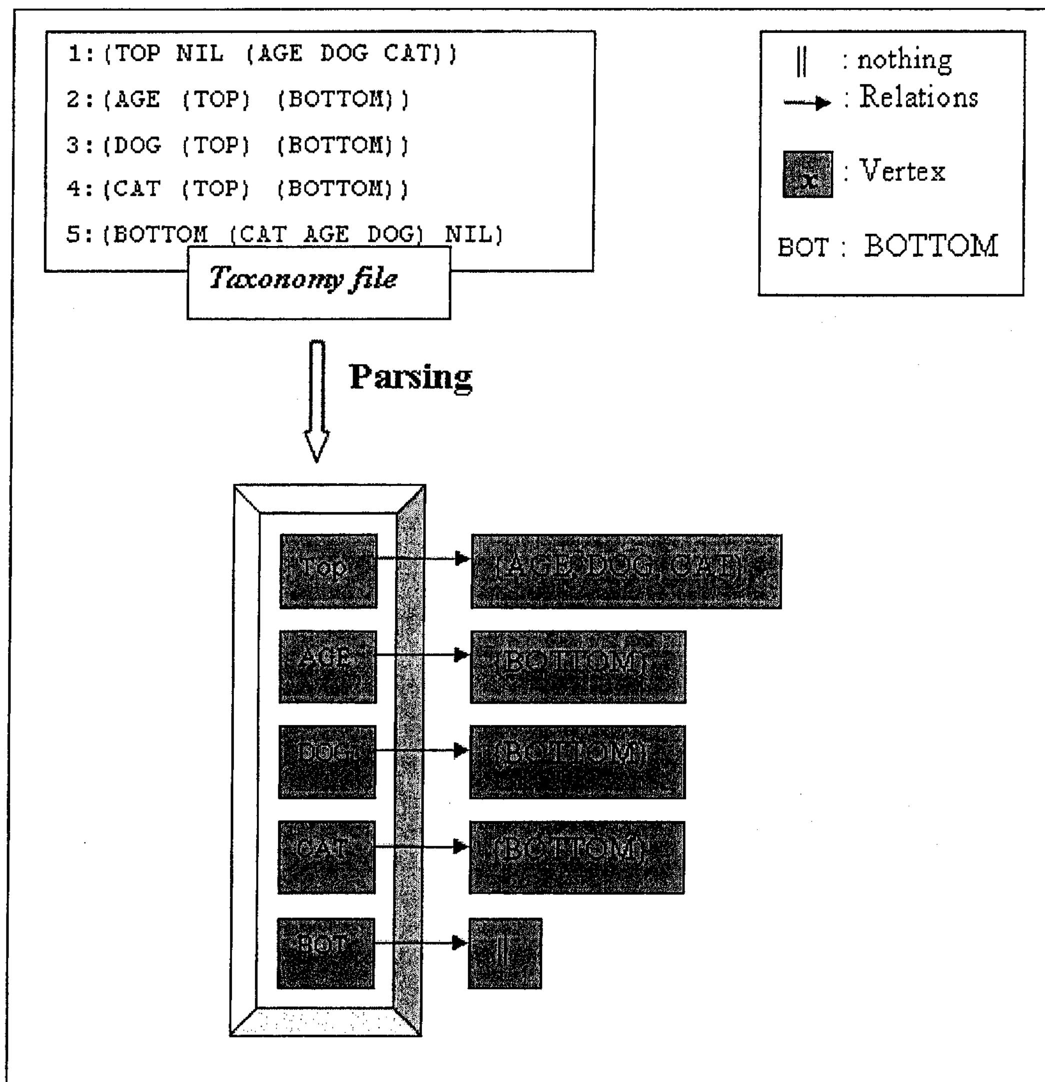

Figure 18: The internal taxonomy files representation.

The parents adjacency list representation of a TBox hierarchy $G = (V, E)$ consists of an array *ParAdj* of $|V|$ lists, one for each vertex in V. For each $u \in$ V, the parents adjacency list *ParAdj* [u] contains all the vertices v such that there is an edge $(u, v) \in$ E where v is a parent concept of u .

The Children adjacency list representation of a TBox hierarchy $G = (V,E)$ consist of an array *ChlAdj* of $|V|$ lists, one for each vertex in V. For each $u \in V$, the children adjacency list *ChlAdj* [u] contains all the vertices v such that there is an edge $(u,v) \in$ E and v is a children concept of u.

We introduce two basic methods for the concept hierarchies data structure: the **GetParents** and **GetChildrens** procedure.

GetParents procedure: This procedure provides a list of concept parent names for a selected concept. The input is the parents' adjacency list and the concept name. The output is the list of parents' concepts (direct subsumers) for the specific concept name.

GetChildrens procedure: This procedure provides a list of concept children names for a selected concept. The input procedure is the children list and the concept name. The output is the list of children concepts (direct subsumees) for the specific concept name.

4.6 Complexity

In term of accessing the information the adjacency list structure is more efficient than other data structure that can be used for both procedures.

The running time for both procedures is linear O(n), where n is the number of concept names in the concept hierarchies. This complexity may be reduced when we use a binary search instead of sequential search.

5 Drawing

5.1 History

Graphs began to appear around 1770 and became commonly used only around 1820. The origins of graph drawing are not well known. Although Euler [24] (1707-1783) was credited with originating graph theory in 1736. He proposed the Königsberg Bridge, a planar graph problem.

Graph drawing problems were in limited use centuries before Euler's time. Moreover, Euler himself does not appear to have made significant use of graph visualizations. The use of graph drawing did not begin to be useful until decades later. In 1871 E.Steinitz gave special attention to the hand graph drawing problem. Century's later (1960) computer scientists began to use graph drawing as diagrams to assist with the understanding of software. In 1963 D.E.Knuth [25] introduced the drawing flow chart which was perhaps the first paper to present an algorithm for drawing a graph for a visualization purpose.

Today, the research in graph drawing has many applications such as software engineering, database, biology, semantic web, etc. Although a lot of research has been done in this area; most of the questions are still unsolved for instance.

-To accomplish a planar drawing the number of crossing edges must be zero, but not for every graph we can achieve a planar drawing. This problem is NP-complete [30].

-For any practical drawing technique do we have an efficient algorithm?

-Do we have an efficient algorithm to minimize the drawing area for any drawing?

-Computing the maximum planar subgraph in a general graph is NP-complete [47].

5.2 Concept hierarchy properties

It is important to take into consideration some properties of the concept hierarchies, to decide the best drawing approach that must be used.

A concept hierarchy is a relational structure, consisting of a set of concepts (classes) and relationships between those concepts. This structure is modeled as a *graph* $G=(V, E)$ already defined.

In the ontology classification we have two kinds of relationships: HasParents and HasChildrens. The main idea of the classification is to compute the ancestor (HasParents) and descendant (HasChildrens) concepts for each concept. In the system display those notion must be clear to the user.

There is a natural decomposition of the hierarchy into levels or layers. A concept *C* belongs to the layer i, if the longest path between the top concept and C has i edges. It is a way to regroup vertices which are at the same distance from the top.

Definition: Let $G = (V, E)$ be an acyclic directed graph. A vertex is called maximal if it has no parents in G. A vertex is called minimal if it has no children in *G*. It is obvious that for acyclic directed graphs there are always minimal and maximal elements.

Definition: Let $G = (V, E)$ be an acyclic directed graph. We define the Top-to-Bottom decomposition into levels as follows: the level L_0 consists of all vertices without parents (or set of maximal elements in G). For every i>0, the level L_i consists of all maximal elements in $G-\{L_0, L_1, \ldots, L_{i-1}\}$.

Note that for every node v in a non-empty layer L_i, there exists at least one simple path P_v starting from v and passing through every level L_k for $0 \leq k \leq i$. That is a sequence of vertices $v_i = v, v_{i-1} \ldots v_0$ such that $v_k \in L_k$, for every $k \leq i$ and (v_k, v_{k+1}) is an edge in *G*.

A similar decomposition could be done from Bottom-to-Top: The level L_0 consists of all vertices without children (or set of minimal elements in G). For every i>0, the level L_i consists of all minimal elements in G-{L_0, L_1, …, L_{i-1}}.

Clearly, any vertex in layer (or level) L_i should have at least an ancestor in layers L_{i+1}.

In a layered (or leveled) drawing, all vertices of the same layer will be drawn on a horizontal line and the edges are represented by straight lines between different layers. Of course, and since a TBox hierarchy is an acyclic digraph because the ***RACER*** would collapse a cycle in the concept hierarchy during the reasoning process, it is impossible to have edges between vertices in the same layer.

Note that we could have edges connecting vertices from non-consecutive layers. These edges are called ***Bypassing*** edges. ***Direct Edges*** are defined as edges that connect two nodes in a consecutive layer.

We did notice however that in the examples we dealt with, most of the outgoing crossing edges are from the Top vertex or the Bottom vertex depending on whether we use the Top-to-Bottom or the Bottom-to-Top layout strategy.

The Bottom node in the hierarchy can be viewed as an imaginary virtual node; it is used for encapsulating and covering the ontology domain. The deletion of this concept does not affect any ontology parameters.

5.3 Proposed approach

Our goal is to produce a readable drawing of a given taxonomy file by analyzing its properties and then making a decision about which drawing algorithm should be applied. For instance, one major problem is caused by the crossing edges which could seriously affect the readability of the drawing. Aesthetic criteria should reflect the assumptions about how people read graphs and how to achieve the best readability. Reduction of

crossing between edges seems to be the most crucial parameter to minimize. It creates the biggest problem in readability, especially when the graph is large.
The computation of the number of crossing edges is needed, since it is used as a major parameter whenever the readability of a drawing algorithm is analyzed.

Horizontal orientation versus Vertical orientation of the drawing.

It is very common not to draw the direction of the edges for an acyclic directed graph. Instead, the position of the vertices on the plane will indicate the directions. With a vertical (also called upward drawing) orientation and for an edge (u, v) in the graph, the vertex y will be drawn higher up than x (the y-coordinate of u is strictly smaller than the y-coordinate of v). However, for the horizontal drawing the vertex y will appear at the right of the vertex (the x-coordinate of u is strictly smaller than the x-coordinate of v).

Relationships between concept hierarchies are drawn with a vertical orientation, so the directions of the edges are omitted but the y-coordinates of the nodes on the screen define the direction (upward). Vertices are drawn as circles or rectangles (for groups of vertices) and edges are drawn as straight-line segments.

With a rectangular screen, the horizontal lines could accommodate a larger number of vertices. In most cases the number of layers of the hierarchy is not so big; however, the sizes of the layers could be very large. So it is more appropriate to use a horizontal orientation in our drawings.

The choice of the horizontal layer orientation will not, of course, resolve the main issue of readability of the drawing.

The number of concepts related only to the top or only to the bottom is usually very large and these concepts could increase the complexity of the drawing, although there are simple ways to deal with theses relationships.

The size of the graph could be very large and it becomes difficult if not impossible to fit the display in a single screen.

Layer distribution.

The distribution of the vertices on the different layers could be more or less proportional depending on the drawing algorithm we use. It is another parameter to take into account in order to have a readable drawing.

Designing an approach that solves all of the above mentioned problems and which could be used for all taxonomy files is known to be a very hard task. However we believe that our approach is useful to obtain an acceptable drawing in many cases.

5.4 High Level Description of the Drawing Algorithm

Our proposed drawing approach consists of the following steps:

1. Test the hierarchy by removing the non essential edges (Cycle remove step).
2. Layer decomposition for both strategies orientation Bottom-to-Top and Top-to-Bottom: this step consists of assigning a layer for each node.

The layout approach consists of computing a layer for each node. The first step of our layout algorithm is to assign nodes with no parents on the first level and then place the direct parents of the previous nodes located in the first level on the second level. Third level nodes are the direct parents of nodes of the second levels and so on. This process continues until all nodes have been assigned a level.

3. Distribution decision between the two strategies: is defined as choosing the best strategy which produces a homogenous distribution between layers.

The decision about the strategy is based on two drawing parameters: The number of nodes in each layer and the number of layers.

4. The autonomous set decomposition: Consists of grouping several nodes that have the same properties into one virtual node.

This approach computes the set of nodes that have the same relationships with other nodes in the graph. It is a technique used to reduce the number of nodes in the same layer by grouping them into one virtual node. The autonomous sets are represented as a rectangle in the display area.

5. Layer permutation: Deals with the order of nodes that must be chosen for each layer.

This procedure consists of computing the direct degree for each node in each layer. : Let us denote the direct degree of a node u as Deg (u) = n where n is the number of nodes v, where $u \in L_i$, $v \in L_{i+1}$, and $(u,v) \in E$. Once the computing of the children is done for a specific layer, a sort procedure could be applied to sort them using their direct degree number in an appropriate data structure. First we place the nodes with the highest degree in the middle layer as a pivot, then place the rest of the nodes on each pivot side starting from the left until all nodes are placed.

6. Cross reduction for each two consecutive layers using the layer by layer sweep algorithm. In this step we discard the crossing edges that pass through more than one layer.

This procedure is called to place the nodes on the horizontal line with respect to each other.

We use the ***layer by layer swap*** algorithm [28]. It is described as follows. First choose a concept ordering of a layer L_1. Then for each layer i= 2, 3, 4....h the concept ordering of layer L_{i-1} is fixed when ordering the layer L_i.

The ordering technique is based on the number of crossings. The number of crossings between two nodes u and v in the same layer L_i is denoted as C_{uv} and defined as: the number of crossing that edges incident with u make with edges incident with v when $x_u < x_v$. More formally, for $u,v \in L_i$, $u \neq v$, $x_u < x_v$, C_{uv} is the number of pairs (u, w), (v, z) of edges with $x_z < x_w$, where z and $w \in L_{i-1}$, and x_u, x_v, x_z and x_w represent respectively the coordinate for the nodes u, v, z and w.

This procedure is done for each two consecutive layer. All nodes in the layer L_i are fixed, then all nodes v in the next layer L_{i+1} must permuted if $C_{uv}>C_{vu}$ where $u,v \in L_{i+1}$. If any nodes changed position during this process the whole procedure is repeated. This continues until either no more nodes move in a pass or until a specified number of iteration. This continues until we reach the last layer.

7. Layer Bifurcation: Consists of finding the layer number where bypassed edges need to be bifurcated.

This technique must be used for each crossing edge $(u,v) \in E$ where $u \in L_i$, $v \in L_j$ and $j>i+1$. We try to find the layer number where we have to bifurcate the crossing edge. The idea is based on two parameters, x_u and x_v, that represent respectively the x coordinate for u and v. This procedure is done for each two consecutive layers starting from the layer L_i until the layer L_j.

Calculate the crossing number for the edge ($x_{u,i}$ and $x_{v,i+1}$), for a given drawing. $x_{u,i}$ and $x_{v,\ i+1}$ respectively represent the x_u coordinate in the layer L_i and the x_v coordinate in the layer L_{i+1}. We compute the crossing number for each iteration and then we save the smallest one with the layer number.

8. Draw the bypassed edges: show the technique used to draw the crossing edge with respect to the node order already made in step number 5.

Having the layer number where we should draw the bifurcations done, we start drawing a vertical straight line between two consecutive layers from $x_{u,i}$. until we reach the layer bifurcation, then we cross the line and keep drawing the vertical straight line between two consecutive layers until we reach the node $x_{v,j}$.
In most cases the drawing of a vertical straight line between two consecutive layers may pass through a node (circle with a radius of 15 pixels). Three cases must be considered.

1-When the edges pass through the middle node shape, this should be the worst case scenario.

2-When the edge passes on the left side of the node.

3-When the edge passes on the right side of the node.

As a first step we calculate the intersection distance D between the line and the node.

If D<= 15, then diverge the line with D pixels to the right.

If D> 15, then diverge the line with D pixels to the left.

5.5 Details of the global algorithm

Since the size of the TBox Hierarchy could be very large and the system we developed is web-based it is very crucial to pay close attention to the efficiency of our algorithms. It is important to have an acceptable response time.

Usually graphs with up to 200 vertices are considered to be small graphs. Graphs ranging between 200 and 400 vertices are considered medium size graphs. Graphs with size 400 to 700 vertices are considered large size and then anything larger than 700 are considered to be graphs with a very large size.

5.5.1 Testing the hierarchy for non-essential edges.

An edge (u, v) is called a redundant transitive edge if there exists a directed path of edges from u to v, that does not contain the edge (u, v).

Although concept hierarchies are not supposed to contain transitive edges, it is important to check if there are any and delete them. In upward drawings the transitive edges, if they exist, they could be deleted without losing any data. The existence of such an edge will create a problem in defining the layers. For instance, if (u, v) is a transitive edge, then the vertex v will belong to different layers at the same time.

5.5.2 Layout algorithm

5.5.2.1 Introduction

The algorithm consists of assigning a layer to each node. To determine the layer of each node we may use two different strategies and then decide the orientation (*Figures 19, 20).*

The layout procedure first identifies all the roots nodes or the leaf nodes in the graph depending on the orientation of the drawing.

The first step of our layout algorithm is to assign nodes with no parents on the first level and then remove those nodes from the adjacency list, next place the children of the previous nodes on the second level. Third level nodes are the direct parents of nodes of the second level and so on. This process continues until all nodes have been assigned a level.

The layout algorithm computes the number of layers in the graph as well as the number of nodes on each layer. These parameters are useful to decide about the final drawing strategy to be used.

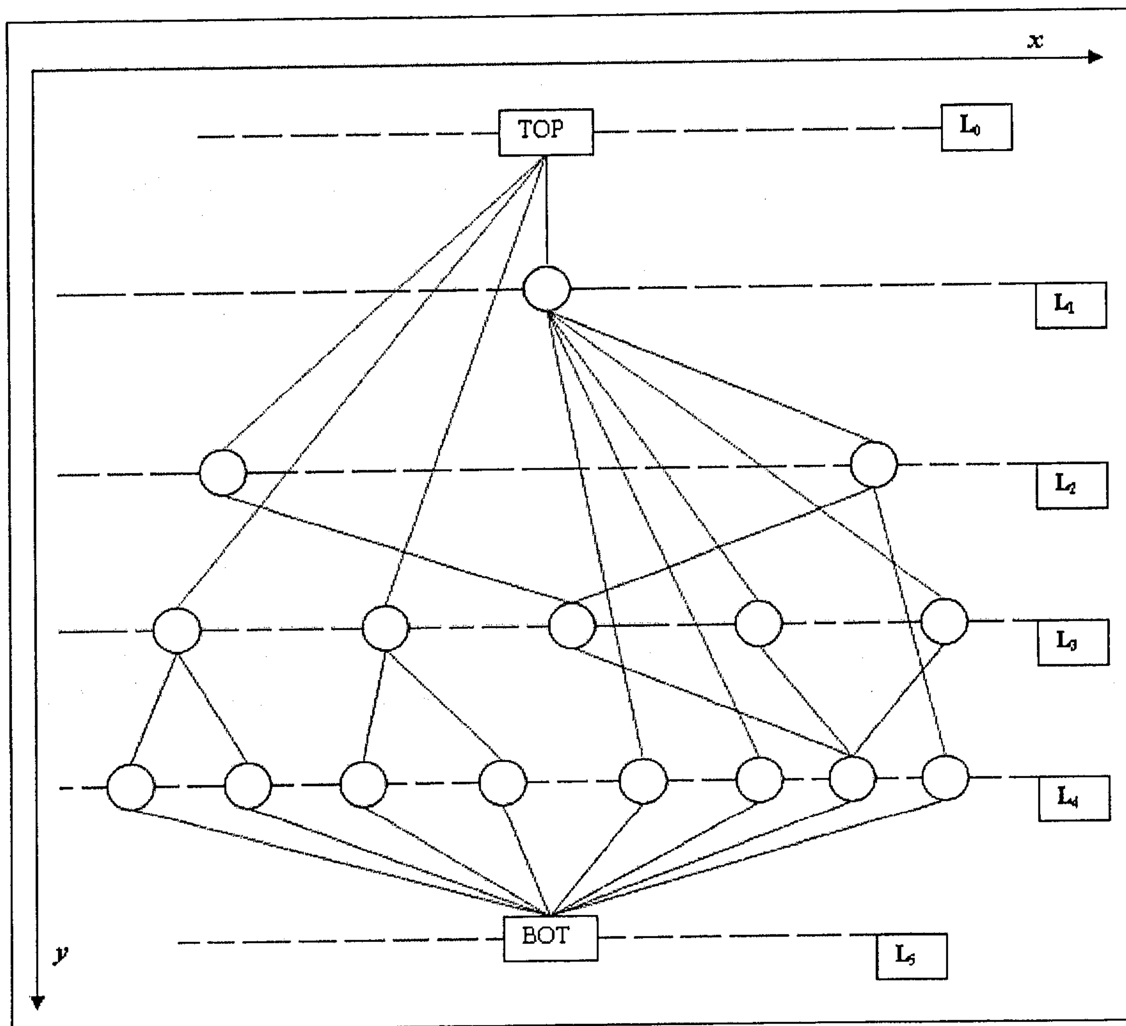

Figure 19: Bottom-to-Top layout drawing approach.

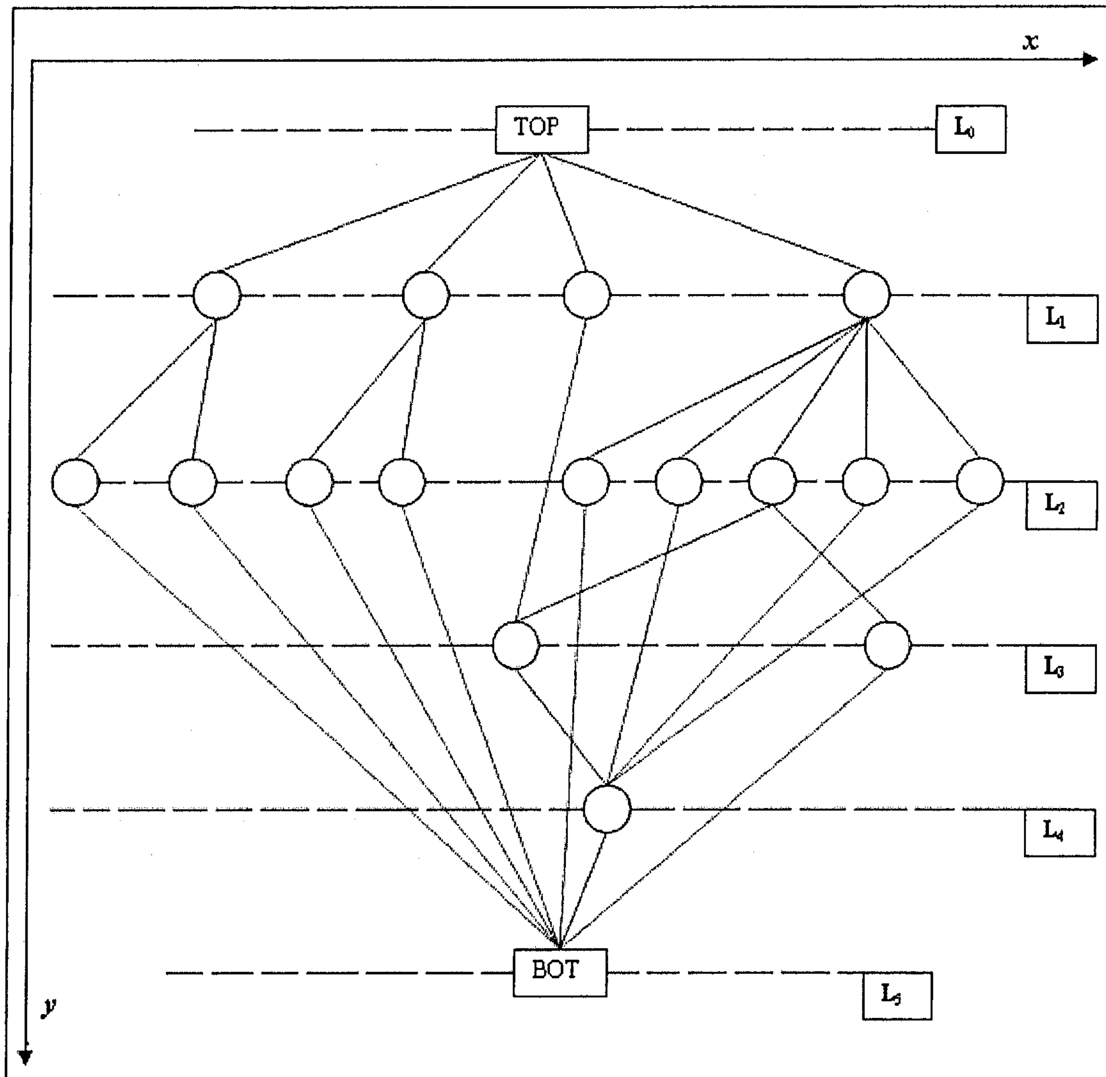

Figure 200: Top-to-Bottom layout drawing approach.

5.5.2.2 Algorithm

The following algorithm traverse the data structure and assign a layer number for each vertex.

Algorithm *Layering (G);*
Input: an adjacency list ***L*** that represents the graph *G*= (*V,E*) and a ***ParAdj*** list that contains the parents node for a given node name.
Output: layering of *G* as a linked list L_l, where each element contains a node name and layer number.
Begin
NumLayer=0;

```
Initially all nodes are unlabeled
Repeat
        For each node u in L
                If (ParAdj[u] is empty)
                        Add the u node and the NumLayer to the list Ll
                        Label the vertex u.
                End if
        End for
        NumLayer++;
        Remove the labeled vertex u from the list L.
Until no vertices are left in the List L
End
Return Ll
```

5.5.2.3 Complexity analyses

The total running time of the layout algorithm is $O(n^2)$, where n is the number of nodes in the graph.

5.5.3 Layout orientation decision

5.5.3.1 Introduction

We call the layout algorithm for both orientation strategies: Top to Bottom and Bottom to top. The layout orientation decision is based on two drawing parameters: The number of nodes on each layer and the number of layers. Taking into consideration those parameters we can conclude the best display orientation. Using a statistical measure of dispersion, we calculate the distance of a typical score of the number of nodes in the layer denoted as x it is given by subtracting it from the mean: $x - \mu$. Therefore, the square distance is $(x - \mu)^2$. Then we can get the average of these (the population variance) by adding and dividing by the number of points.

The ***Layer variance*** (noted as σ^2) is the average square distance from the mean node:

$\sigma^2 = ((x_1 - \mu)^2 + (x_2 - \mu)^2 + \ldots + (x_n - \mu)^2) / n$, where $\mu = (x_1 + x_2 + \ldots\ldots + x_n) / n$ and x_n is the number of node in the n th layer.

The ***layer standard deviation*** (noted as σ) $\sigma = \sqrt{\sigma^2}$. This parameter must be close to the mean layer μ to get a proportional layer distribution.

A simple procedure ***OrientationGraph*** has been implemented to calculate the σ parameter for both layout strategies. Then we calculate a new parameter $\Delta = |\mu - \sigma|$. The best approach with the smallest Δ is chosen. In some case both strategies give the same number, then the Top Bottom approach is chosen.

We noticed that in most examples we worked with, the Top-Down design approach seems to be more appropriate to create a drawing for the ontology. The number of crossing edges in each file has proven very small for the Top Bottom layout with the omit Bottom concept option. Moreover, with this approach the distribution of the nodes on the different layers seems to be more proportional than with the Bottom Top layout.

Below (Table 1 and Table 2) we show a sample of examples of ontologies retrieved from the TBoxHDI output; these examples belong to different size classes (small, medium, large, and very large).

For every case we computed the number of bypassed edges and the number of nodes in each layer depending on the type of drawing: Top-to-Bottom or Bottom-to-Top.

Number of Nodes	Number of bypassed Edges	Number of Nodes on each Layer
35	15	1-9-24-1
52	20	1-12-38-1
72	25	1-15-55-1
112	66	1-2-4-23-81-1
112	56	1-6-26-78-1
109	55	1-6-25-76-1
124	62	1-2-7-28-85-1
131	104	1-1-1-1-1-2-3-6-11-22-82
17	5	1-1-2-4-8-1
18	8	1-1-2-4-8-1
300	92	1-1-3-7-12-15-22-32-73-128-1
317	147	1-3-15-46-251-1
441	217	1-2-1-2-3-3-5-11-17-25-62-308-1
495	153	1-1-1-3-6-7-14-20-36-99-306-1
657	217	1-1-1-2-3-7-7-15-24-43-130-422-1
1121	386	1-1-2-3-5-5-11-17-30-51-81-205-708-1
2750	1488	1-1-1-1-1-2-5-10-12-18-33-43-76-170-452-1924-1

Table 1: Bottom-to-Top layout strategy.

Of course, in a Bottom-to-Top drawing there will be no crossing edges coming from the Bottom, since all nodes connected to the Bottom will be in the first layer. Similarly removing the Top in a Top-to-Bottom drawing will not decrease the number of crossing edges.

Number of Nodes	**Number of bypassed Edges** With Bottom/Without Bottom	**Number of Nodes on each Layer**
35	15 / 0	1-9-24-1
52	20 / 0	1-18-32-1
72	25 / 0	1-30-40-1
112	81 / 2	1-2-18-41-49-1
112	62 / 0	1-16-46-48-1
109	61 / 0	1-15-44-48-1
124	84 / 0	1-2-25-58-37-1
131	53	10-14-7-11-29-29-20-6-3-1-1
17	7 / 0	1-1-3-8-3-1
18	10 / 3	1-1-2-9-4-1
300	144 / 17	1-1-2-14-38-45-65-53-32-21-9-18-1
317	234 / 8	1-25-170-93-27-1
441	319 / 16	1-5-7-30-41-29-70-73-88-40-10-46-1
495	311 / 12	1-7-50-113-110-77-61-41-19-8-7-1
657	417 / 3	1-8-38-45-106-82-139-130-69-21-11-6-1
1121	719 / 19	1-8-10-43-70-196-184-284-182-87-27-9-19-1
2750	1887 / 440	1-9-55-252-271-310-319-468-433-373-132-65-26-22-2-12-1

Table 2: Top- to-Bottom layout strategy.

Screen shots of the TBoxHDI output for each kind of file using different options are shown in the appendix C.

5.5.3.2 Algorithm

The following algorithm decides the best display approach by analyzing the number of nodes in each layer.

Algorithm *OrientationGraph* (R1, R2) ;
Input: R1, R2 arrays that contains the number of nodes in each layer for each respective strategy (Top-to-Bottom and Bottom_to-Top) .
Output: BottomTop or TopBottom
Begin
BT=0; TB=0;
    For (all element x in R1)
        BT=BT + x
    End for

```
        μ1= BT / R1.size ();
        For (all element x in R1)
                BT=BT + (μ1-x) ^2
        End for
        σ1 ^2= BT / R1.size ();
        For (all object x in R2)
                TB= TB + x
        End for
        μ2= TB/R2.size ();
        For (all element x in R2)
                TB= TB + (μ2-x) ^2
        End for
        σ2 ^2= TB/R2.size ();

        Δ1=| μ1- √σ1 ^2 |
        Δ2=| μ2- √σ2 ^2 |

        If (Δ1<=Δ2)
                Return BottomTop
        End if
        Else
                Return TopBottom
        End else
End
```

5.5.3.3 Complexity analysis

The algorithm uses a simple naïve process with a run time $O(N+N')$ where N and N' is the number of layers for each strategy.

5.5.4 Autonomous sets Decomposition algorithm

5.5.4.1 Introduction

The second major technique used to simplify the drawing without losing data is a decomposition technique. It combines any group R of nodes that have the same relationships with all nodes outside of R.

Definition: Let $G= (V, E)$ be a directed graph. A non-empty set of nodes A is called autonomous set and denoted $Aut\ (v,w)$ if for every node u in A and v in V-A, there exists

an edge (v, u) and if and only if there is an edge (u, w) where w in V-A for every u in A. (*Figure 21 and 22*)

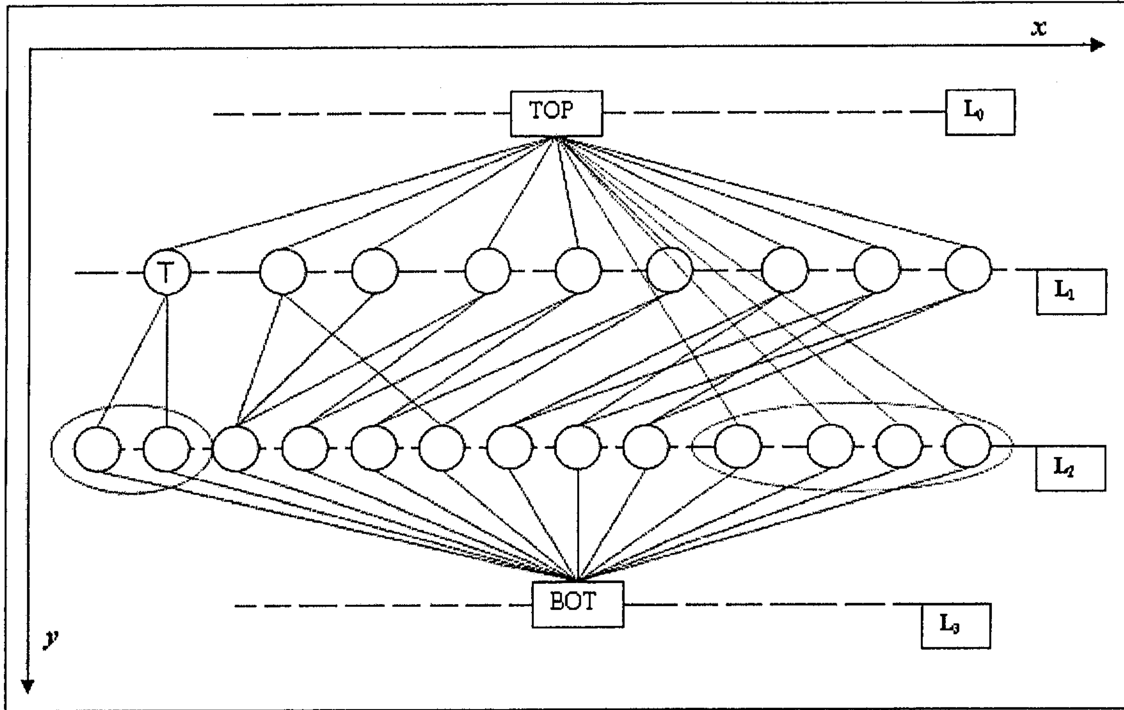

Figure 21: The oval marker shows the nodes that must be joined.

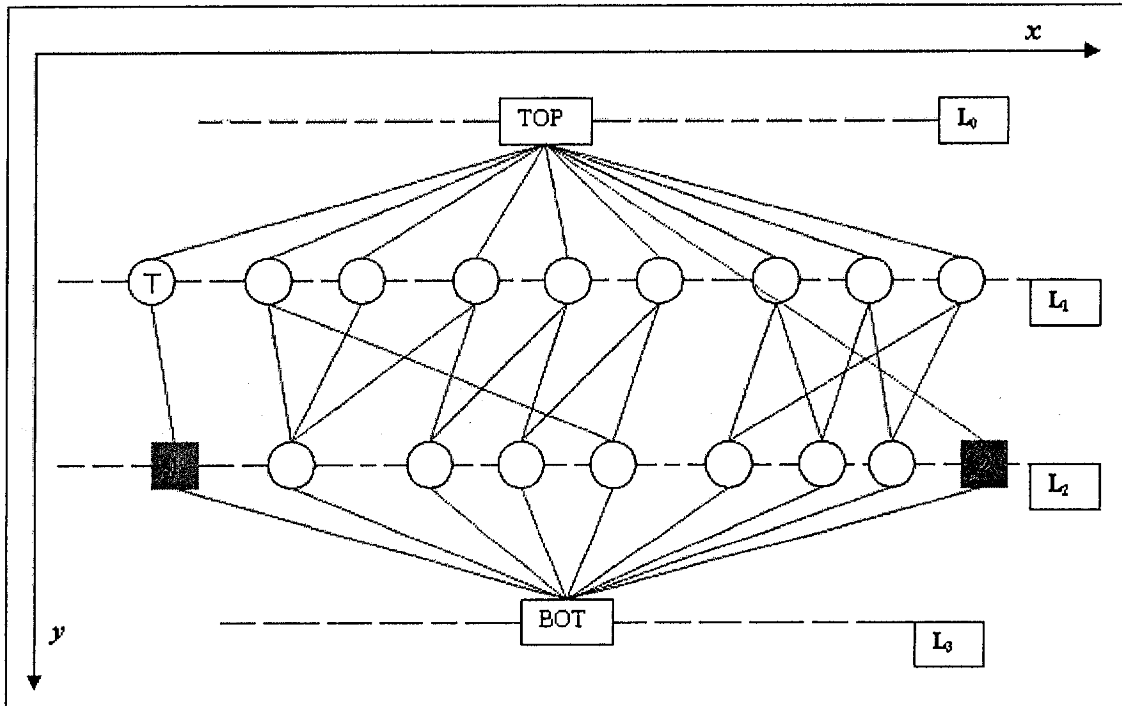

Figure 22: The first gray rectangle represents the autonomous set Aut (T, BOT)
The second gray rectangle represents the autonomous set Aut (TOP, BOT)

By grouping all nodes of an autonomous set into one virtual node, we greatly simplify the drawing without losing any data.

The goal of this part is to create a data structure with a reduced number of vertices without affecting the displayed information. This technique is related to the large graph aesthetic parameters because many nodes are joined and the size of the graph may be reduced.

Once the nodes are placed in their layer, the procedure ***AutonomousNodes*** is called.
Here is the algorithm which finds all the autonomous sets in the graph G.

CandidateNode: procedure checks if the node may be a candidate to be joined into the autonomous set. It returns a Boolean. There are two conditions that are checked for a node: the node must only have one parent and if the node is already in an autonomous set, then it is discarded.

UpdateAdjacencyList: procedure that updates the adjacency list. If a join decision is made, we remove all nodes that must be joined and replace them by a new single node.
Input: L a Link List (Adjacency List), N_1, N_2 nodes must be joined
Output: L a new Link List (Adjacency List) with homogenous nodes

The algorithm will use an adjacency list as input. The process is done for all nodes and starts from the top to bottom or bottom to top depending on the orientation of the graph.
Adj[u] is defined as: the set of children of node u.

5.5.4.2 Algorithm

The following algorithm creates an updated linked list of the graph G that contains the autonomous sets.

Algorithm: *AutonomousNodes* (G);
Input*: *L a Link List (Adjacency List represents the concept hierarchy graph) and ***Adj***[u] that represent the children set for a given node u.
Output*: *L a new Link List (Adjacency List with autonomous sets)

Begin
For each node *u* in the ***L***
    For each element x_i in ***Adj[u]***
        If (***CompareLists*** (Adj [x_i], Adj [x_{i+1}]) =**true**)
            If (***CandidateNode*** (x_i) =**true** and ***CandidateNode*** (x_{i+1}) =**true**)
                1-Create autonomous set with x_i and x_{i+1}
                2-***UpdateAdjacencylist*** (L, x_i, x_{i+1})
            End if
        End if
    End for
End for

Return ***L***
End

5.5.4.3 Complexity analysis

The maximum number of iteration that must be made is related the number of pairs node. Suppose the number of nodes is n then, the total number of iteration is $n*(n-1) / 2$.
The running time of the autonomous sets algorithm is $O(n^2)$ where n is the number of nodes in the graph.

5.5.5 Layer permutation algorithm

5.5.5.1 Introduction

Unfortunately, the algorithm used in the crossing step uses a random order of nodes for each layer, and especially for the first layer. The challenge of this part is in choosing an optimal ordering within the layers that could be useful and helpful to draw pretty concept visualizations. The choice of the order is on the computation of the direct degree. The layer permutation must be applied to each layer.

5.5.5.2 Algorithm

The following algorithm permutes the nodes order for a selected layer.
Algorithm *LayerPermutationNodes*();
Input*:** random node order as a linked list ***L for a selected layer.
Output*:** a new permuted node order as a linked list ***L.
Begin
For each element *u* in ***L***

Compute the direct degree n
Add n to the list *Lst*
End for
Sort *Lst* using the descendent order.
Place the first node v in the *Lst* in the middle of the layer fixed as a pivot P.
Remove v from the list *Lst*.
For each element u_i in the *Lst*
If (i is even)
Place u_i in the left of P
Else
Place u_i in the right side of P
End for
End

5.5.5.3 Complexity analysis

This simple procedure runs in linear time $O(n)$ where n is the number of nodes in the graph.

5.5.6 Crossing edges reduction algorithm

5.5.6.1 Introduction

The minimization of the number of crossing edges is a very difficult problem in general. In fact, it is a NP-complete problem even if the graphs contain only two layers [30]. However, in the research literature there is a number of heuristics that have been developed for this problem.

Of course the relative positioning of the nodes within every layer will decide about the number of crossings in the drawing.

In our case, we will present a heuristic in order to minimize the crossings between edges.

Now we use the definition of the crossing number C_{uv} described in the section **5.4**.

The basic idea of this step is using the ***layer by layer swap*** algorithm which has received a great attention [31, 32]. It is described as follows. First we choose a concept ordering of a layer L_1. Then for each layer i= 2, 3, 4….h, the concept ordering of layer L_{i-1} is held fixed when we reorder the vertices in layer L_i. Unfortunately the "two layer crossing problem" is NP-complete [29].

This procedure is called to place the nodes in the vertical line with respect to each other.

This procedure is done for all nodes in the top (bottom) layer depending on the orientation, then for all nodes in the next layer and so on until the bottom (top) layer is done. If any node changed position in any layer during the process, the whole procedure is repeated again. This continues until either no node moves in a pass or until a specified number of iterations is reached.

5.5.6.2 Algorithm

The following algorithm exchanges adjacent pair of vertices.
Algorithm *LayerByLayerSwap(G, x_1)*
Input: two layered graph G = (L_1, L_2, E) and a fixed vertex order x_1 for L_1
Output: vertex order x_2 for L_2
Begin
We choose a random order for L_2
Repeat
    Scan the vertices of L_2 from left to right, exchanging an adjacent pair *u, v* of vertices, whenever $C_{uv}>C_{vu}$

Until the number of crossings remain unchanged.
End

5.5.6.3 Complexity analysis

The computation of the crossing number C_{uv} depends on the relative position of *u* and *v*. Each scan in the repeat loop can be done in O ($|L_2|$) and there are O ($|L_2|$) scans. The time complexity of the layer by layer swap algorithm is O ($|L_2|^2$). The total running time is O ($n\ |L_2|^2$), where n is the number of iterations to get a stable crossing number.

5.5.7 Layer Bifurcation algorithm

5.5.7.1 Introduction

Once we have done all the previous steps we may get an acceptable drawing with an optimal number of crossings. Unfortunately, in many cases the bypassed edges (*u, v*) may reduce the visibility and the readability by causing some node shape intersections. We

compute the upper layer number that represents the beginning of the bifurcation for each bypassed edge.

5.5.7.2 Algorithm

The following algorithm returns a layer number that represent the beginning of the bifurcation.

Algorithm LayerBiffurcation()
Input: a Bypassed edge (u,v) with $u=(x_1, y_1)$ and $v=(x_2, y_2)$
Output: an integer y representing the upper layer number
Begin
n=GetLayer(u) // return the layer number that the vertices u belongs.
m = GetLayer(v)
$C=+\infty$ // initialization for a big number
For ($i=n$ to m)
Create an imaginary node u' with coordinates (x_1,i)
Create an imaginary node u'' with coordinates $(x_2,i+1)$
Create an imaginary edge (u', u'')
Compute the crossing number $C_{u''w}$ for each node w in the layer $i+1$,

If ($C> C_{u''w}$) **Then**
$C= C_{u''w}$
$y=i$
End if
End for

Return y.
End

5.5.7.3 Complexity analysis

The running time of this algorithm is dominated by the compute crossing procedure (step 4). The crossing complexity is O ($|L|^2$) where L is the number of edges between two consecutive layers. The total running time for one bypassed edge is $|L|^2$*(m-n-1). Where m and n are respectively the upper and lower layer number.

5.5.8 The Bypassed edges draw algorithm

5.5.8.1 Introduction

The basic idea is drawing the bypassed edges as a straight line through all layers, but with respect to the nodes that intersect with the edges using the layer bifurcation technique

already described in the previous step (*Figure 23 and 24*). This algorithm adopts the polyline drawing edges with a minimum number of bends to provide flexibility instead of curved edges which may be difficult to track by the users.

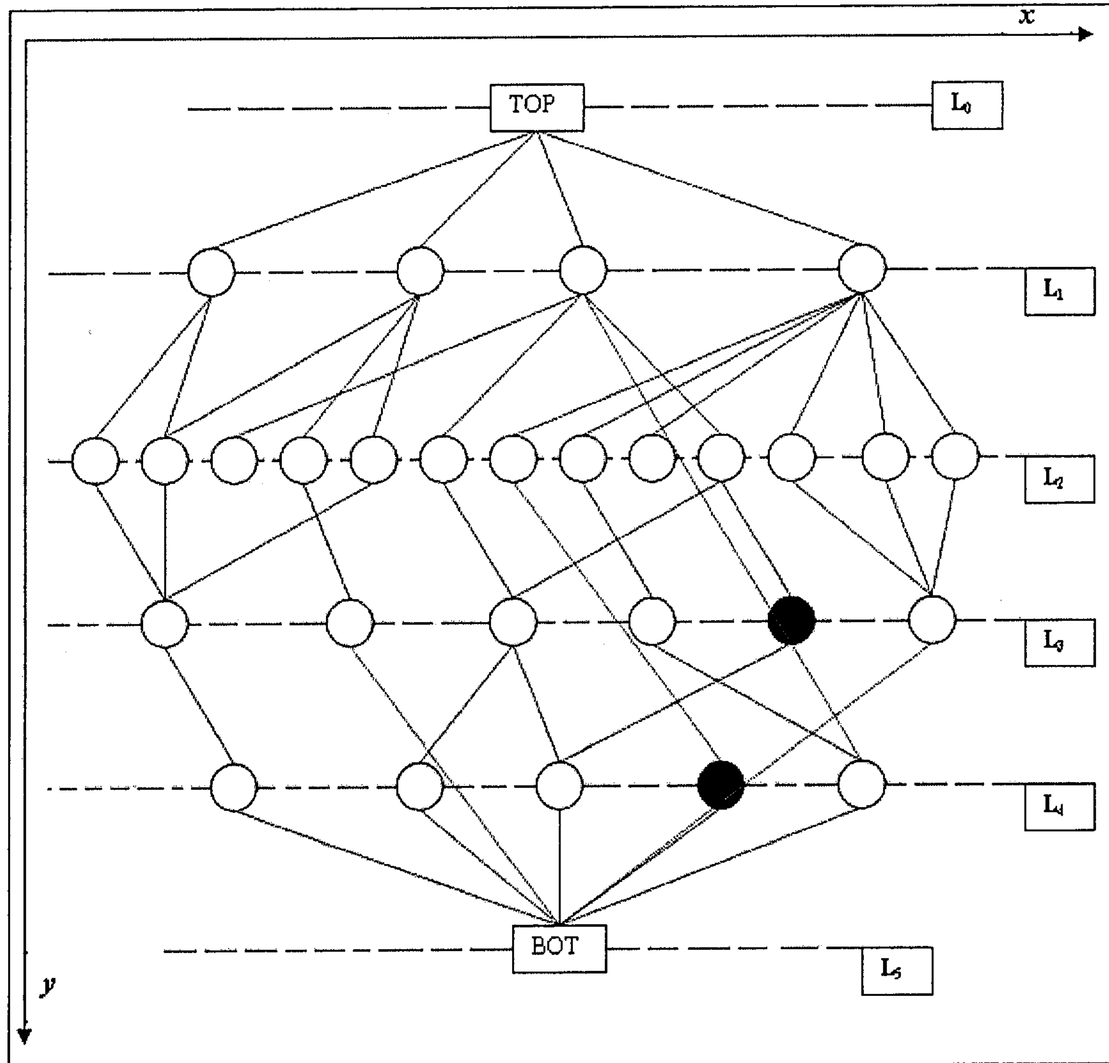

Figure 23: The black node shows the intersection between the bypassed edges.

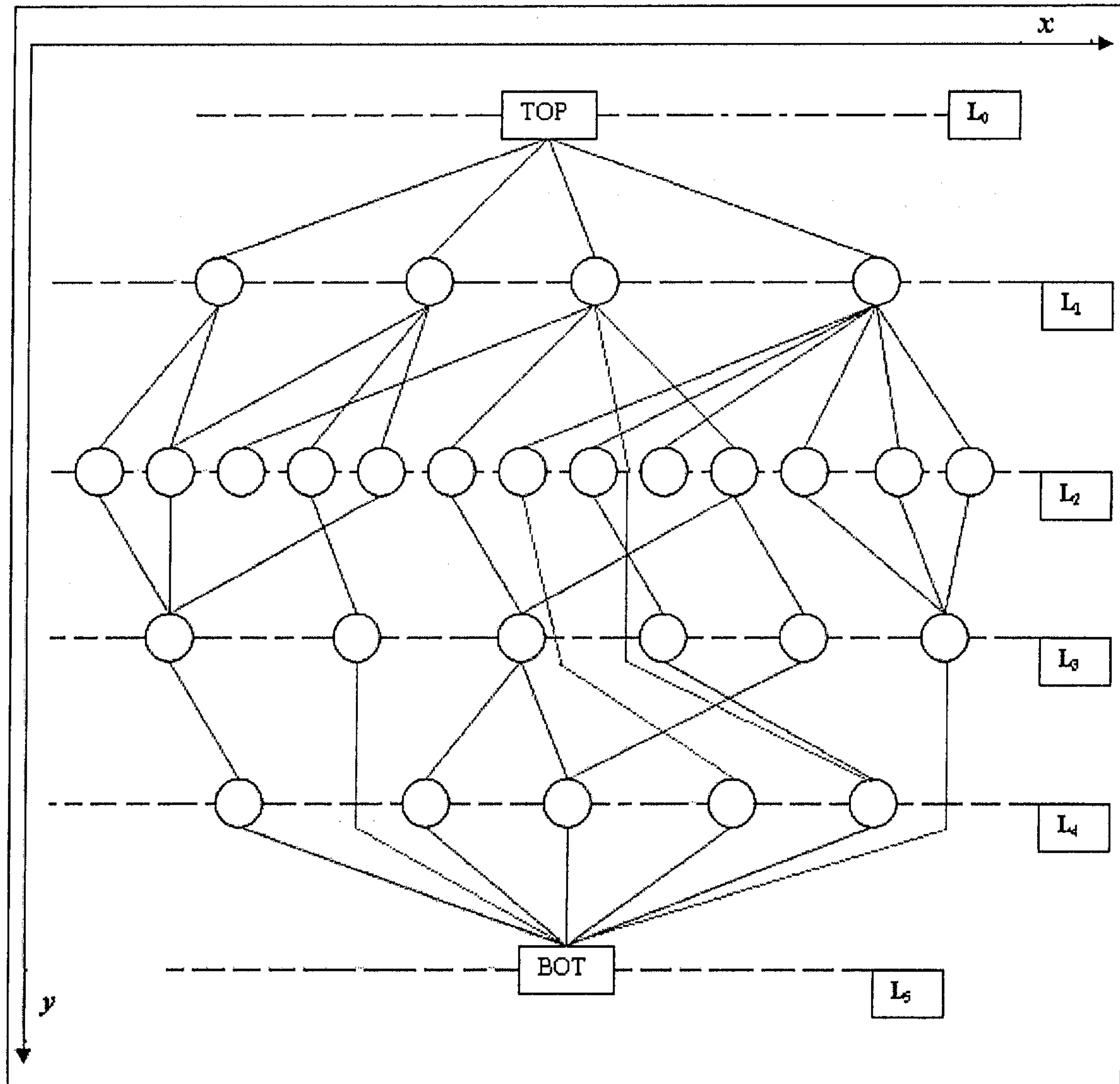

Figure 24: The bypassed edges drawing using the layer bifurcation and "draw bypassed edges "algorithm.

5.5.8.2 Algorithm

Algorithm BypassedEdgeDraw(T)
Input: a drawing T with optimal number of crossings
Output: a new drawing T' with a bypassed edge drawing
Begin
For each bypassed edge $e=((x_1, y_1),(x_2, y_2))$
Integer i = ***Layer Bifurcation (e)***
If a layer number#i
Calculate the intersection distance D between the line and the node.
If D<= 15 **then** diverge the line with D pixels to the right.
If D> 15 **then** diverge the line with D pixels to the left.
If D=0 **then** draw a straight line

End if
Else
Bifurcate the line from the (x_1, y) coordinate to the $(x_2, y+1)$ Coordinate. //y and $y+1$ represent respectively the coordinate for the layer L_i and L_{i+1}
End else
End for
End

5.5.8.3 Complexity analysis

The time complexity is strongly related to the layer bifurcation procedure already computed in the previous step. The total running time of this algorithm is $O(K*Q)$ where K in the number of Bypassed edges in the drawing and $Q = |L|^2*(m-n-1)$. Where m and n are respectively the upper and lower layer number for each bypassed edges and L are the number of crossing between two consecutive layers.

6 Conclusion and Open Problems

In this thesis we presented the techniques of visualizing subsumption hierarchies. We have presented two main parts: In the first part we presented methods for parsing the incoming ***RACER*** information and store them in data structures. In the second part we announced an approach for best viewing the data.

The last part was mainly dedicated to the description of the algorithms used to visualize the taxonomy. Unfortunately, we cannot easily create an efficient drawing algorithm for all taxonomy file sizes. It is a very labor intensive process in terms of time complexity, crossing edges and the available display size. A technical description of the system was also provided in this part by presenting its architecture and discussing its main components.

While dealing with a very large taxonomy file, we encountered the problem that occurs when the number of nodes in the same layer is very big. That is why we proposed a new approach called the virtual node. In order to guarantee the readability of the drawing we have to respect the following criteria:

1-A layered graph must be used.
2-Compute the autonomous sets.
3-Number of crossing edges must be minimal.
4-A special drawing for the bypassed edges.

Much more work remains to better visualize a large taxonomy file size with a minimum number of edges crossing. This can be an area for future research.

There are still some issues which have not been addressed yet and that we considered as an open problem, here we describe three of them:

1-The data structure is very large.

2-Minimize the crossing number even for the bypassed edges that goes from the top or the bottom.

3-An efficiency search tool.

4-Visualization in 3-D can be explored.

Bibliography

[1]: http://www.webopedia.com/TERM/X/Xerox.html. June 28, 2004

[2]: B. A. Myers. "*A Brief History of Human Computer Interaction Technology.*" *ACM interactions.* Vol. 5, no. 2, March, 1998. pp. 44-54.

[3]: B. Shneiderman. "*Designing the User Interface, Third Ed*".Reading, Massachusetts: Addison Wesley. 1998.

[4]: G. Bonsiepe. "*Interpretations of Human User Interface*". Visible Language. Vol. 24. No. 3:262-285.

[5]: N.F. Natalya F, and D.L. McGuinness. " *Ontology Development 101: A Guide to Creating Your First Ontology*" . Stanford University, Stanford, CA, 94305.

[6]: C. Christine. "*Graphical User Interfaces: Keep Them Sleek and Simple*". Computer world . Vol. 25. No. 16:37-40.

[7]: I. Horrocks, U. Sattler, and S. Tobies. "*Reasoning with individuals for the description logic SHIQ*". In David MacAllester, editor, Proceedings of the 17th International Conference on automated Deduction (CADE-17), number 1831 in lecture Notes in computer science, Germany, 2000.

[8]: V. Haarslev, and R. Moller. "*Racer System Description* ." Proceedings of International Joint Conference on Automated Reasoning, Springer-Verlag, June 18-23, 2001, Siena, Italy.

[9]: F. Baader, I. Horrocks, and U. Sattler."*Description Logics as Ontology Languages for the Semantic Web*". Lecture Notes in Artificial Intelligence, 2003.

[10]: W. Kocay. "*The Hopcroft-Tarjan planarity algorithm*". Computer science department. University of Manitouba. October 1993

[11]: G.D. Battista, and R. Tamassia. *"Algorithms for plane representations of acyclic digraphs* ". Theoretical Computer Science, 61:175 n 198, 1988

[12]: G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. *"Graph Drawing: Algorithms for the Visualisation of Graphs"*. Prentice Hall, 1999.

[13]: D.E. Knuth. *"Computer draw flowcharts"*. Commun. ACM, 6, 1963.

[14]: M.S. Marshall, I. Herman, and G. Melançon . *"An Object-Oriented Design for Graph Visualization"*. Software: Practice and Experience, 739-756. 2001.

[15]: K. Heidegger. *"Visualizing and exploring large hierarchies using cascading, semicircular disks". Proc of IEEE infovis'98 late braking hot topics* IEEE 9-11.

[16]: C. Hao, H. Meichun , D. Umesh , and A. Krug.*"Web-Based Visualization of Large Hierarchical Graphs Using Invisible Links in a Hyperbolic Space Ming"*. Software Technology Laboratory HP Laboratories Palo Alto,HPL-2000-2, January, 2000.

[17]: J. Lamping and R. Rao, *"Laying out and Visualizing Large Trees Using a Hyperbolic Space"*. ACM /UIST'94, 1994.

[18]: L. Beaudoin, M.A. Parent, and L. C. Vroomen. *"A Compact Explorer For Complex Hierarchies* ". Conference , San Francisco, California , 1996.

[19]: F. Buschmann. *"Pattern-Oriented Software Architecture"*. Vol 1. Wiley & Sons, 1996.

[20]: S. Burbeck. *"Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)"* .Paper, 1992

[21]: M. Fowler. *"Patterns of Enterprise Application Architecture"*. Addison-Wesley, 2003.

[22]: G. Helm, and Vlissides. "*Design Patterns: Elements of Reusable Object-Oriented Software*". Addison-Wesley, 1995.

[23]: D.R. Hanson." *Compact recursive-descent parsing of expression*".Department of electrical engineering and computer science, Princeton University, Princeton NY 08544 USA .

[24]: E. Kruja, J. Marks, A. Blair and R. Waters. "*A Short Note on the History of Graph Drawing*". Proceedings of Graph Drawing 20001 (Springer LNCS, Vol.2265), TR2001-49.

[25]: D.E. Knuth, "*Computer Drawn Flowchart*", Commun. ACM, 6, 1963.

[26]: T.H. Cormen, C. E. Leiserson, and R.L. Rivest. "*Introduction to Algorithms*." McGraw Hill, New York, 1990.

[27]: M. Forster. "*Applying Crossing Reduction Strategies to Layered Compound Graphs*". University of Passau, 94030 Passau, Germany

[28]: G. Di Battista, P. Eades, R. Tamassia, and I.G Tollis. "*Graph drawing, Algorithms for the visualization of graphs*": Chapter 9, 280–290, 1999.

[29]: P. Eades, and N. Wormald. "*Edge crossings in drawings of bipartite graphs*" *Algorithmica*. 11(4):379–403, 1994.

[30]: M.R. Garey, and D.S. Johnson. "*Crossing number is NP-complete*". SIAM Journal of AlgebraicDiscrete Methods, 4, no3, 312–316, 1983.

[31]: E. Makinen. "*Experiments of Drawing 2-leveled Hierarchical Graphs*". Technical Report A-1988-1, Department of computer science, University of Tampere, Jan 1988.

[32]: T. Catarci, "*The Assignment Heuristic for Crossing Reduction in Bipartite Graphs*". In Proc. 26th Allerton Conf. Commun. Control comput, 1988 .

[33]: F. Bader, D. Calvanese, D. McGuinness, D. Nardi and P. Patel-Schneider, "*The Description Logic Handbook*". Cambridge University Press. January 2003.

[34]: L. Badea. "*Planning in Description Logics: Deduction versus Satisfiability Testing*". Proceedings of the International Workshop on Description Logics - DL-98, pp 106-115, Trento, Italy, 1998.

[35]: A. Artale and E. Franconi. "*Representing a robotic domain using temporal description logic*"', AI EDAM Artificial Intelligence for Engineering, Design, Analysis and Manufacturing, *Vol 13(2), pp 105-117, 1999.*

[36]: A. Simonet and M. Simonet. "*Object Persistency, View Maintenance and Query Optimization in OSIRIS, a DL-like OODBMS*". Proceedings of the International Workshop on Description Logics - DL-97, pp 138-143, Gif sur Yvette, France, 1997.

[37]: F. de Beuvron, F. Rousselot, and D. Rudloff. "*Interpretation of Description Logics for Natural Language and for Databases*". Proceedings of the International Workshop on Description Logics - DL-97, pp 159-162, Gif sur Yvette, France, 1997.

[38]: P. Lambrix, and L. Padgham. "*Conceptual Modeling in a Document Management Environment using Part-of Reasoning in Description Logics*". Data & Knowledge Engineering, Vol 32, pp 51-86, 2000.

[39]: D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. "*Description Logic Framework for Information Integration*". Proceedings of the 6th Int. Conference. on the Principles of Knowledge Representation and Reasoning, 1998.

[40]: P. Bresciani. "*Querying Databases from Description Logics*" Reasoning about Structured Objects: Knowledge Representation Meets Databases - KRDB-95, 1995.

[41]: L. Padgham. "*Combining Description Logic Systems with Information Management Systems*". Proceedings of the International Workshop on Description Logics - DL-95, pp 56-58, Roma, Italy, 1995.

[42]: A. Zeller. "*Versioning System Models through Description Logic*". Proceedings of the 8th International Symposium on System Configuration Management (SCM-8), Brussels, 1998.

[43]: J. Warfield. "*Crossing theory and hierarchy mapping*". IEEE Transactions on Systems, Man and Cybernetics, SMC-7(7):502--523, 1977.

[44]: M.J. Carpano. "*Automatic display of hierarchized graphs for computer aided decision analysis*". IEEE Trans. on Syst. Man Cybern., SMC-10,no.11,705-715,1980.

[45]: K. Sugiyama, S. Tagawa, and M. Toda. "*Methods for Visual Understanding of Hierarchical Systems*". IEEE Trans. Sys Man Cybern., SMC-11,no.2,109-125,1981.

[46]: R. Cornet. " *RICE manual Technical report 2003-01*". department of medical informatics university of Amsterdam April 11 ,2204.

[47] : M.R. Garey, and D.S. Johnson. "*Computers and Intractability A Guide to the Theory of Np-Completeness*". WH Freeman & Co, 1979 ISBN: 0716710455.

Appendix A: Parser algorithm

Parse procedure:

Givens: V vector of tokens
Results: return true if the file follow the grammar other wise false.
Intermediates: none
Header: Boolean <-Parse(V)
Body:

```
If (TripleList()==true)
    return true
Else return false
```

Grammar terminal procedures

Boolean TripleList()

```
      Token ->Next token
  if (Triple()==false)
      { return false;}
     else if (TripleNode()==false)
      { return false;}
     else { return true;}
```

End TripleList()

Boolean TripleNode()

```
    if (V.isEmpty()==false)
      {TripleList(); return true;}
     else return true;
```

End TripleNode()

Boolean Triple()

```
If (Token ==”(”)
     Token ->Next token
     Node()
     Parents()
     Child()
     if ((Token==")")&&(V.isEmpty()==false))
              { Token ->Next token;
                if(Token=="("){ Triple(); return true;}
                else { return false;}
              }
       else if ((Token==")")&&(V.isEmpty()==true))
             { return true;}
            else {return false;}
        }
else {return false;}
```

```
End Triple()

Boolean Child ()
if (Children()==false)
   { return false;}
else {return true;}
End Child()

Boolean Parents()
  If (Token ==”(”)
       Token ->Next token
       NodeList()
        if(Token==")"){
              Token ->Next token
               return true;}
      Else {return false;}
   }
}
else return false
End Parents()

Boolean Node()
If (Token ==”(”)
       Token ->Next token
       NameList()
       Return true
 Else if( Name ()==true) { return true}
 Else return false
End Node()

Boolean Children()
  If (Token ==”(”)
       Token ->Next token
       NodeList()
        if(Token==")"){
              Token ->Next token
               return true;}
      Else {return false;}
   }
}
else return false
End Children()

 Boolean NameList()
  If (Name()==true)
        Return true
```

```
  Else {NameChain(); return true ;}
End NameList()

Boolean NameChain()
   if (NameList()==false){
        return false;
        }
  else return true ;
End NameChaine()

Boolean NodeList()
   if (Node()==true){
           return true;
          }
   else {NodeChaine();return true;}
End NodeList()

Boolean NodeChain()
  if (NodeList()==false){
        return false;
       }
  else return true ;
End NodeChain()

Boolean Name()
If (IsString(Token) == true  )
    {Token ->Next token
     Return true
    }
Else return false
End Name()

Boolean IsString(Token)
  If (Token dos not contain any token terminal)
        return true
   Else retun false
End IsString()
```

Appendix B: Taxonomy file example

```
;FaCT version 1.6  13:24:36 GMT  4/1/1999
;   Allegro CL 5.0.beta [Linux/X86] (6/11/98 23:02)  Linux/X86 Linux
;   Loaded TBox files:
;      /localhome/ian/FaCT-distribution/FaCT/Tboxes/galen.tbox
;Features and optimisations:
;   Transitivity:                            ON
;   Concept Inclusions:                      ON
;   Blocking                                 ON
;   Subset S-equivalence:                    ON
;   Encoding & Normalisation:                ON
;   GCI absorption:                          ON
;   Backjumping:                             ON
;   Obvious Subsumption Detection:           ON
;   Use caching in subsumption tests:        ON
;   Use caching in satisfiability tests:     ON
;   SAT branch to minimise clashes:          ON
;   Branching heuristic:                     OLDEST+JW
;   Cyclical primitive definitions:          ON
;   Profiling:                               ON


(TOP NIL (C1694 C1692 C1690 C1688 C1686 C1684 C1682 C1680 C1678 C3 C2
C1))
(C1 (TOP) (C4))
(C2 (TOP) (C309))
(C3 (TOP) (BOTTOM))
(C4 (C1) (C104 C103 C66 C32 C28 C7 C5))
(C5 (C4) (C220 C58 C54 C47 C43 C6))
(C6 (C5) (C273 C223))
(C7 (C4) (C2275 C17 C8))
(C8 (C7) (C16 C9))
(C9 (C8) (C15 C14 C11 C10))
(C11 (C9) (C13 C12))
(C12 (C11) (BOTTOM))
(C13 (C11) (BOTTOM))
(C14 (C9) (C116 C115))
(C15 (C9) (C110 C109 C108 C107 C106 C105))
(C16 (C8) (C130 C129 C126 C124 C123 C122 C121 C120 C119 C118 C117))
(C17 (C7) (C20 C19 C18))
(C18 (C17) (C134))
(C19 (C17) (C140 C136))
(C20 (C17) (C1728 C1704 C201 C200 C167 C164 C147 C67 C1717 C21))
```

Appendix C: TBoxHDI output

System screen shot to show the evolution of the system

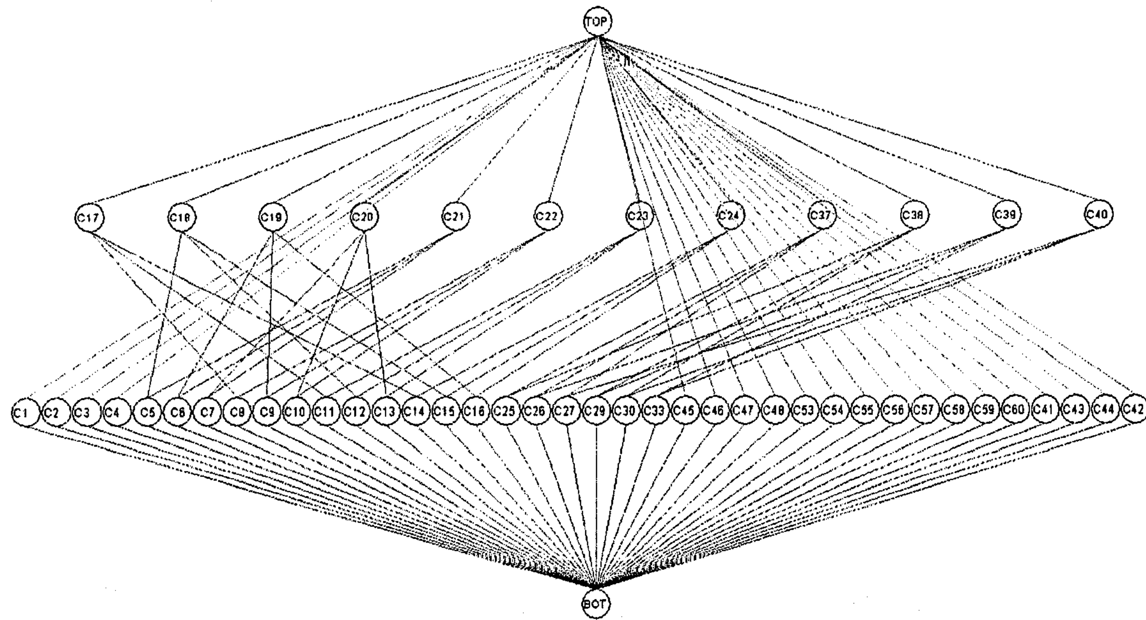

Figure 1: Display using only the basic algorithms.

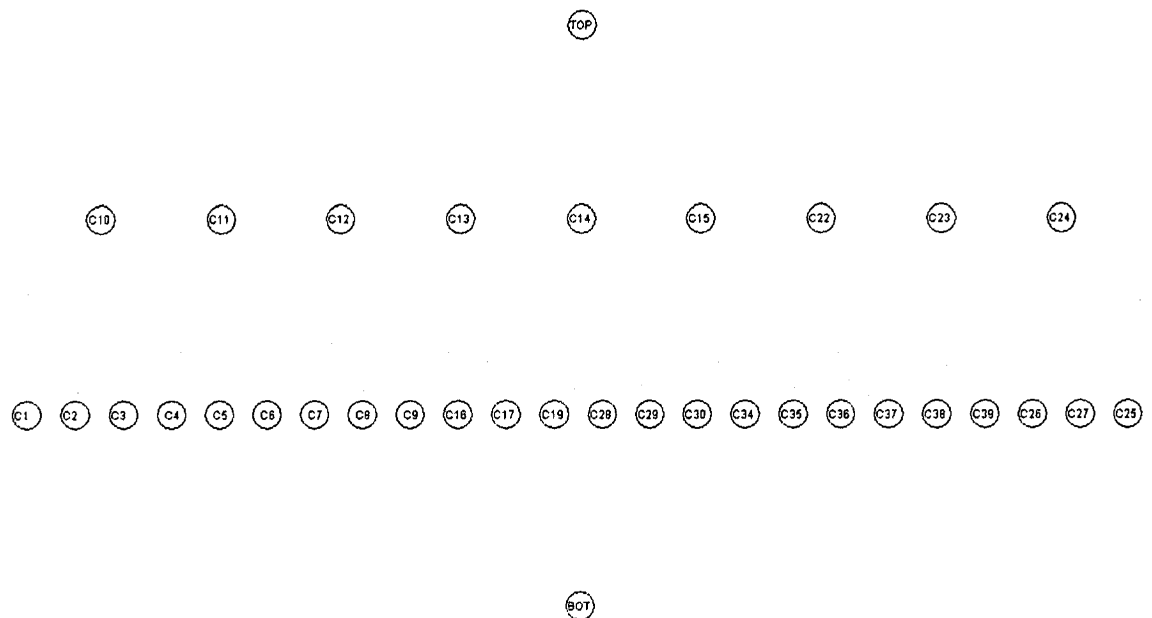

Figure 2: Display with the hide relation option.

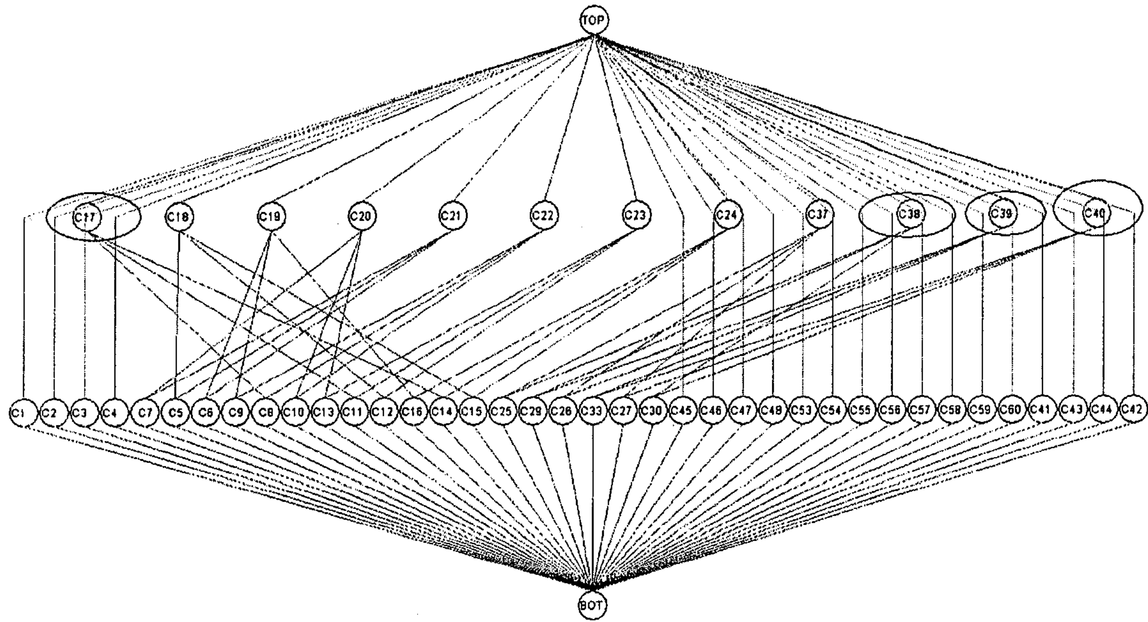

Figure 3: Display using the crossing algorithm. The ellipses show that the bypassed edges still pass through nodes.

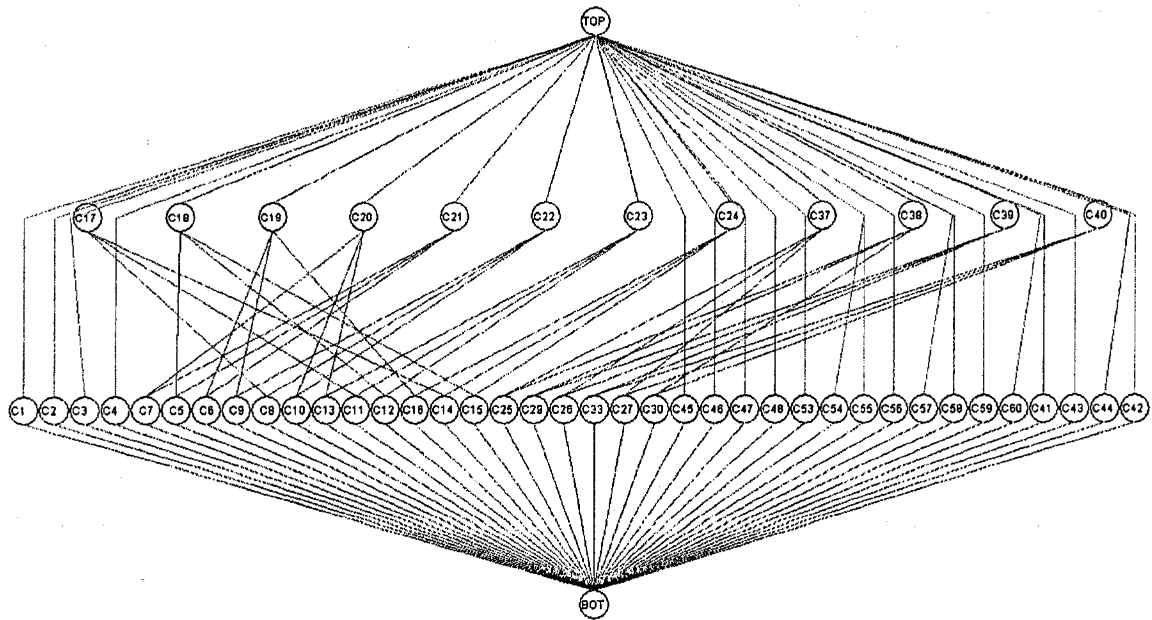

Figure 4: The previous problem is resolved using the modified bypassed edges Layer bifurcation algorithm.

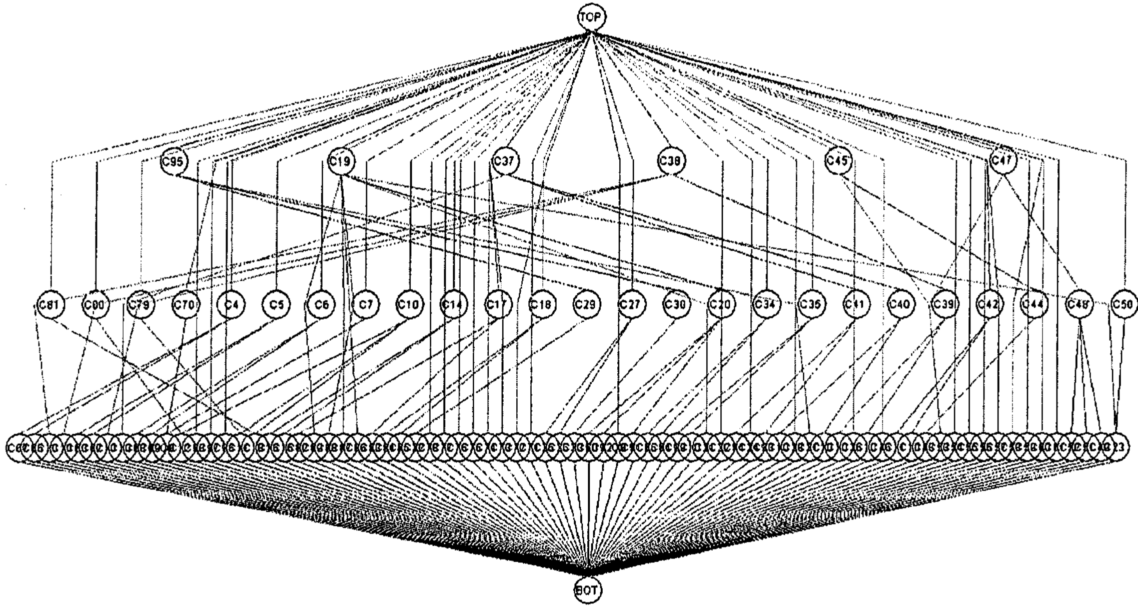

Figure 5: The Bottom to Top display.

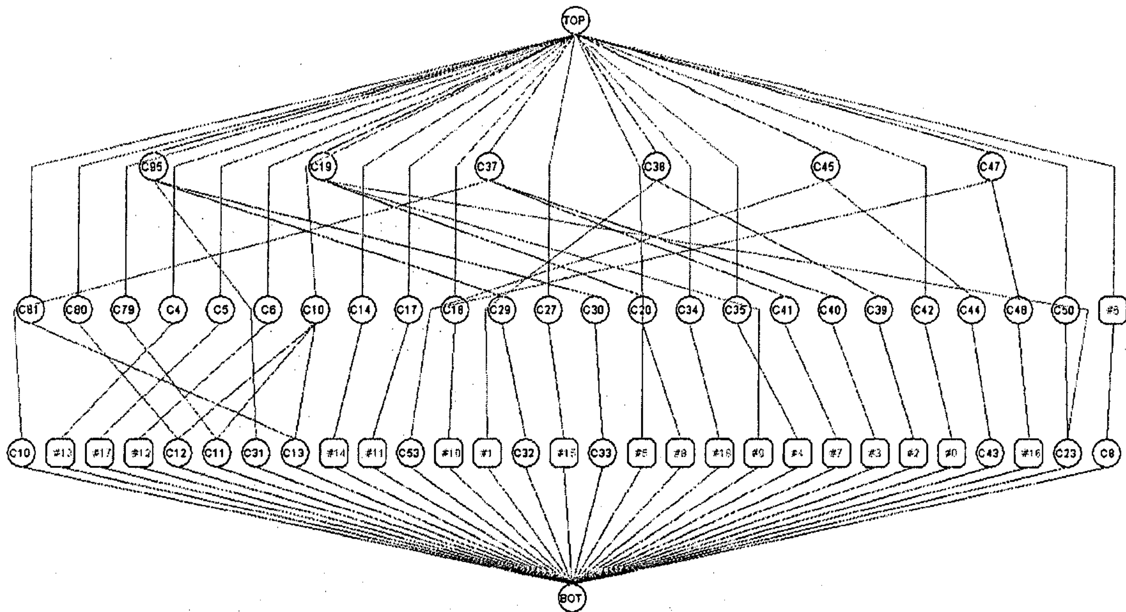

Figure 6: The Bottom to Top display with the virtual node option.

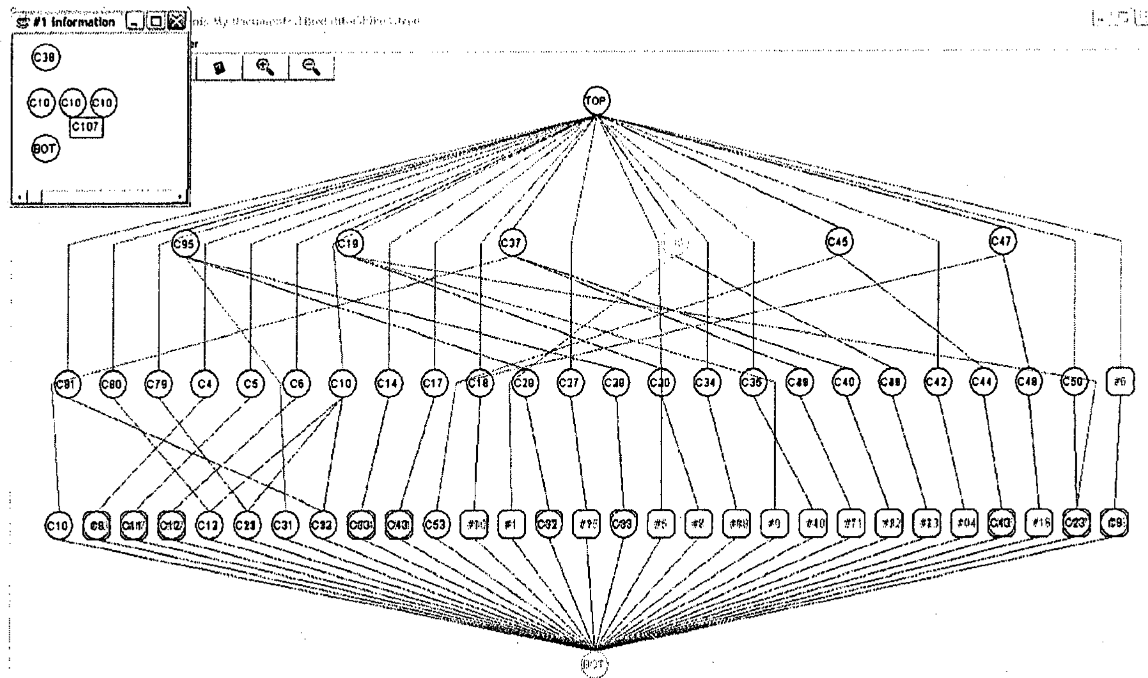

Figure 7: The Bottom to Top display with the virtual node option using the zoom on the virtual node #1.

Screen shots for different size files with different options

Small file

Top to Bottom

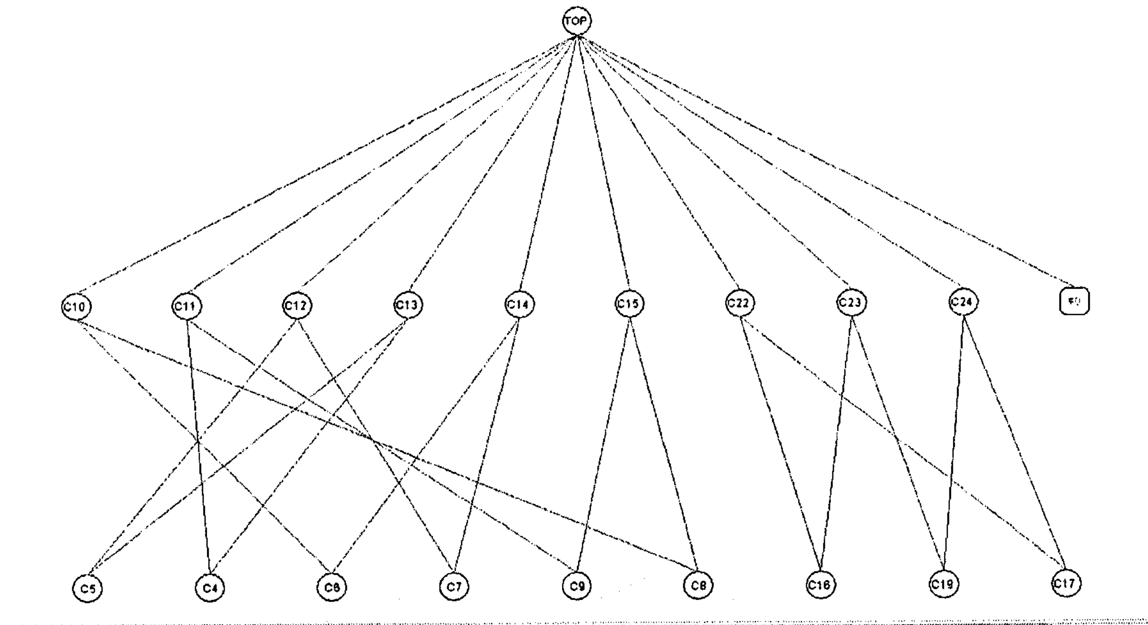

Figure 8: The best display is given using the Top to Bottom, the virtual node and the omit Bottom options.

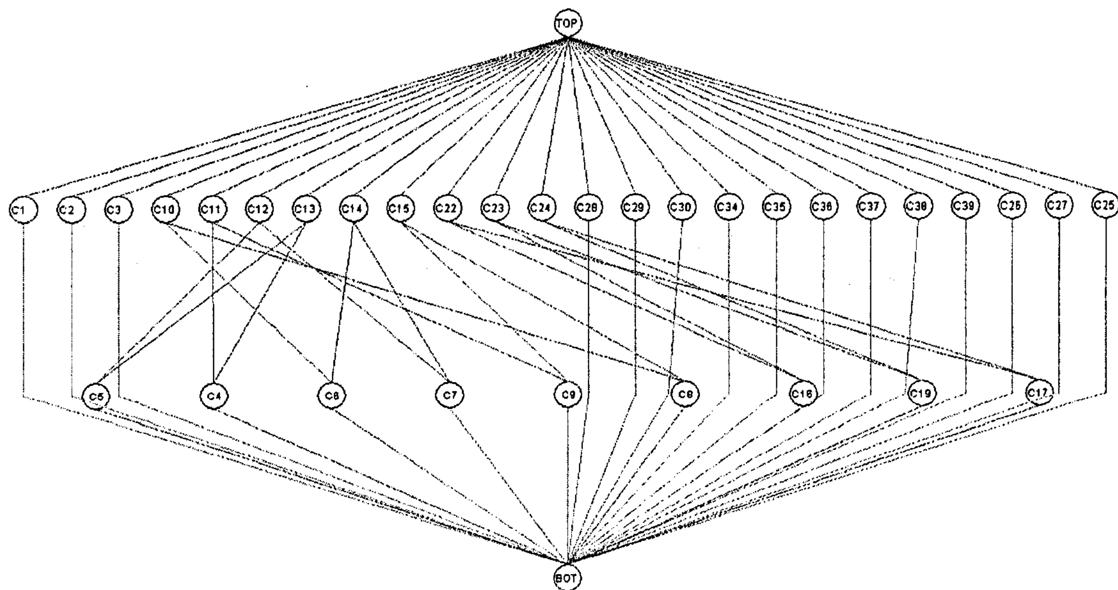

Figure 9: The Top to Bottom display without any options.

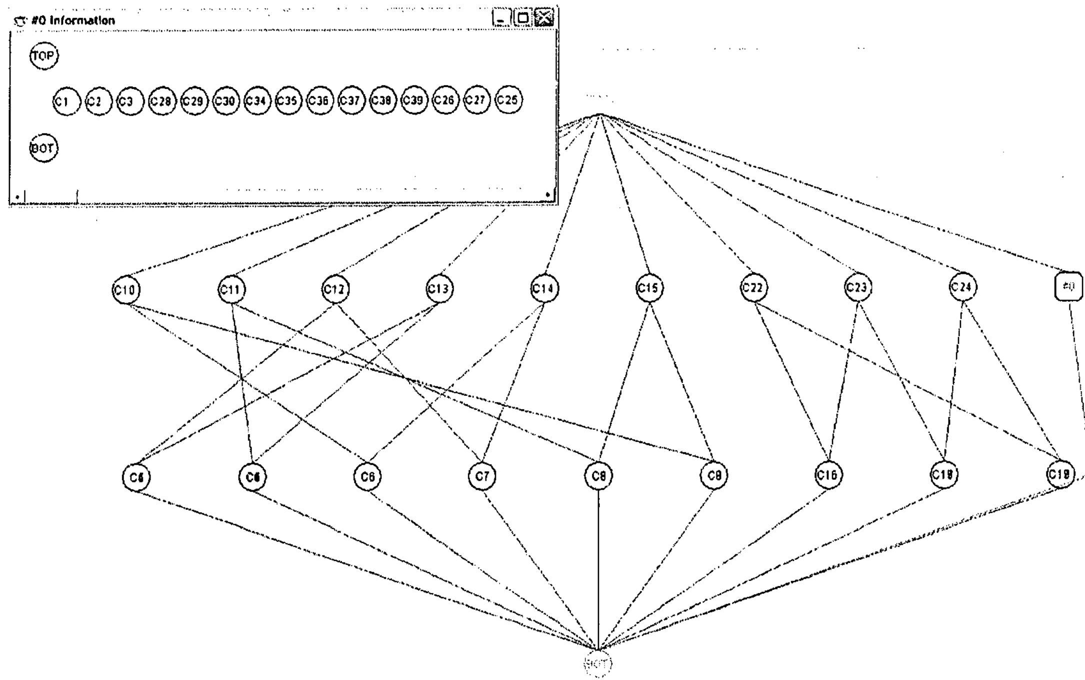

Figure 10: The Top to Bottom display with virtual node option and zoom.

Bottom to Top.

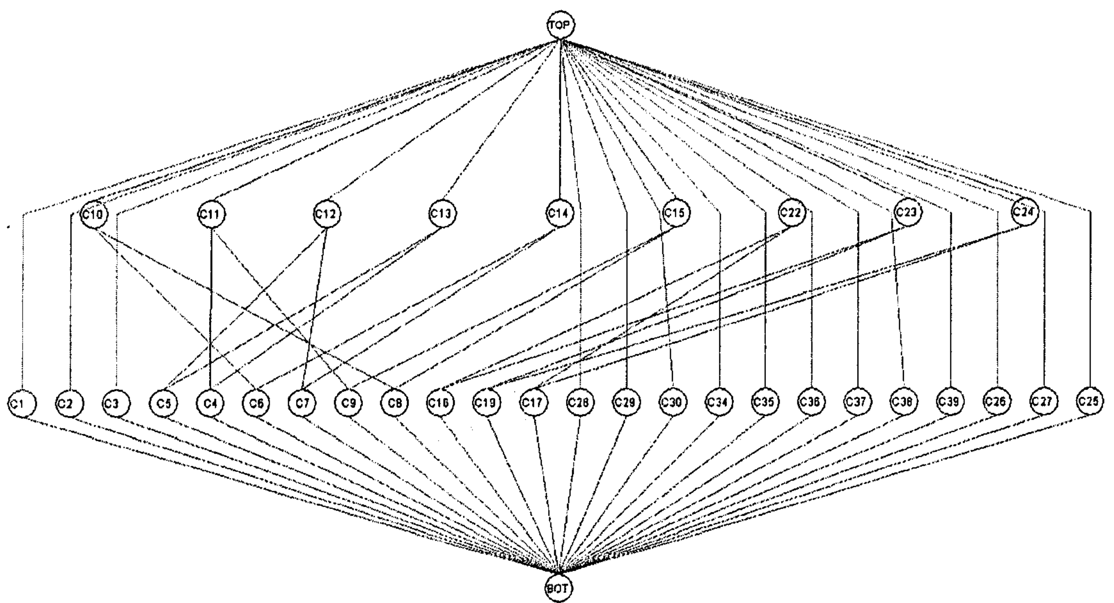

Figure 11: The Bottom to Top display without any option.

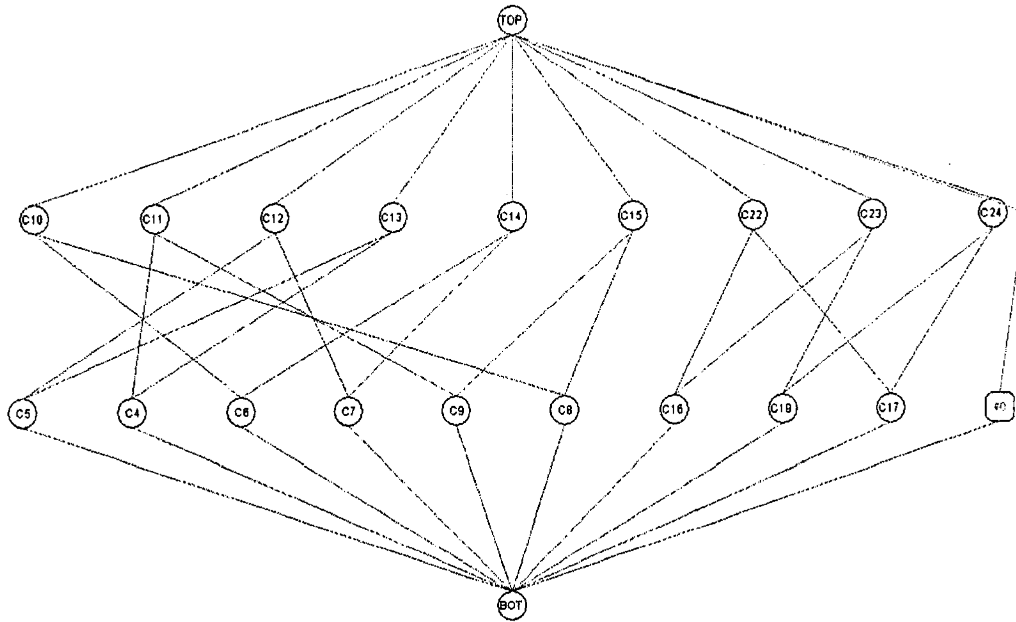

Figure 12: The Bottom to Top display with virtual node option.

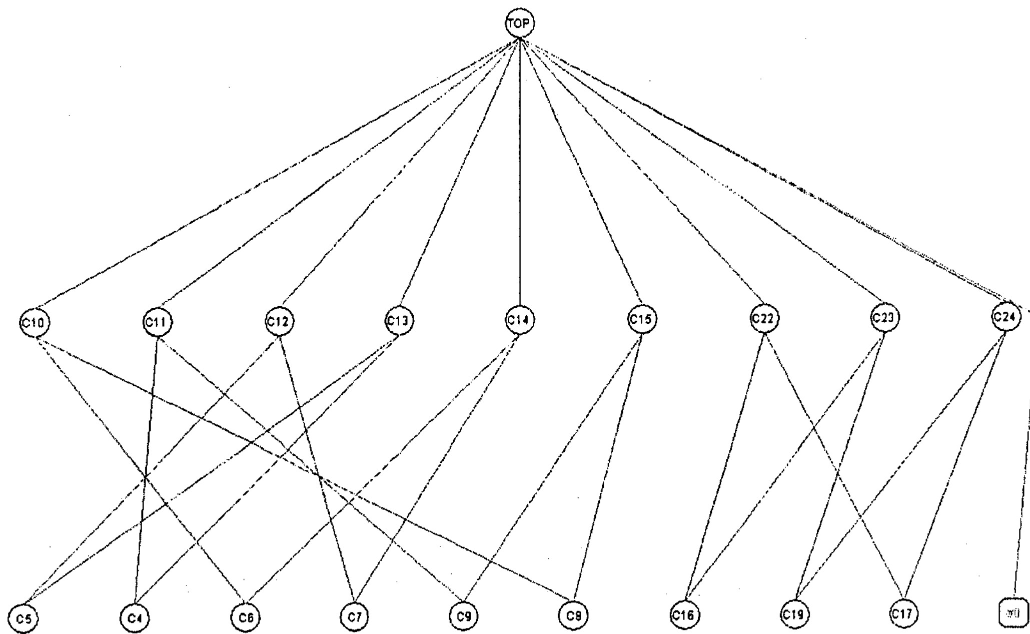

Figure 13: The Bottom to Top display with virtual node and omit Bottom options.

Large file

Top to Bottom

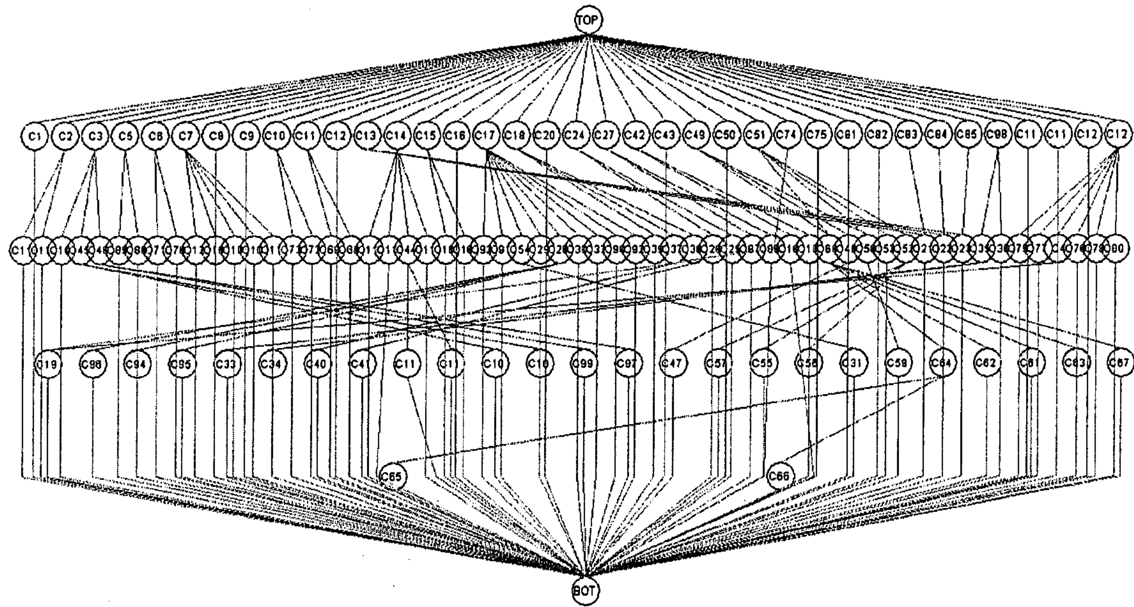

Figure 14: The Top to Bottom display without any option.

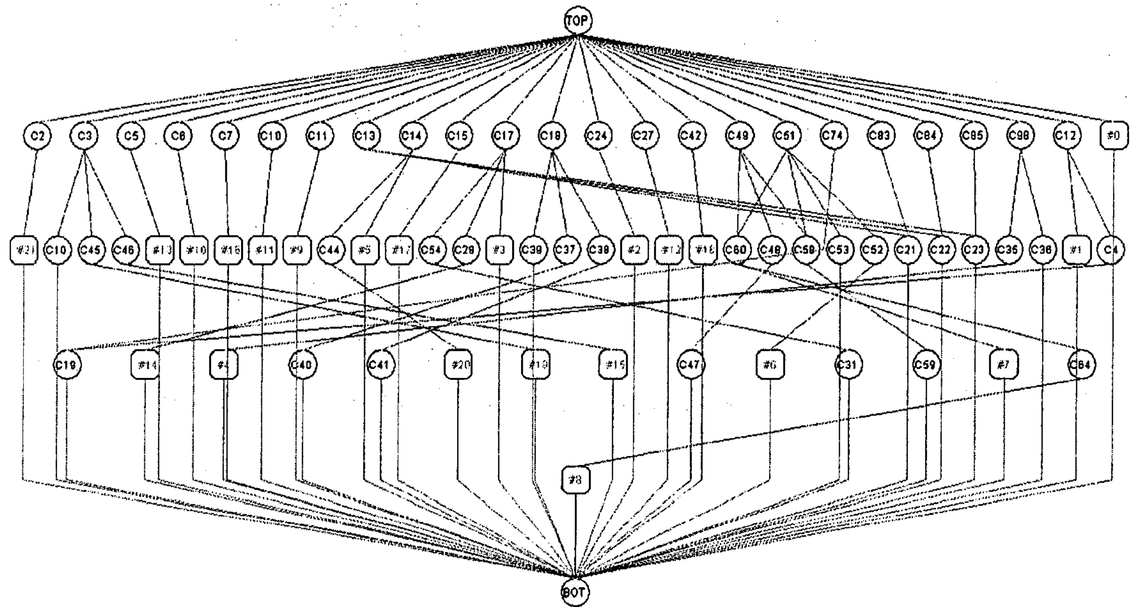

Figure 15: The Top to Bottom display with the virtual node option.

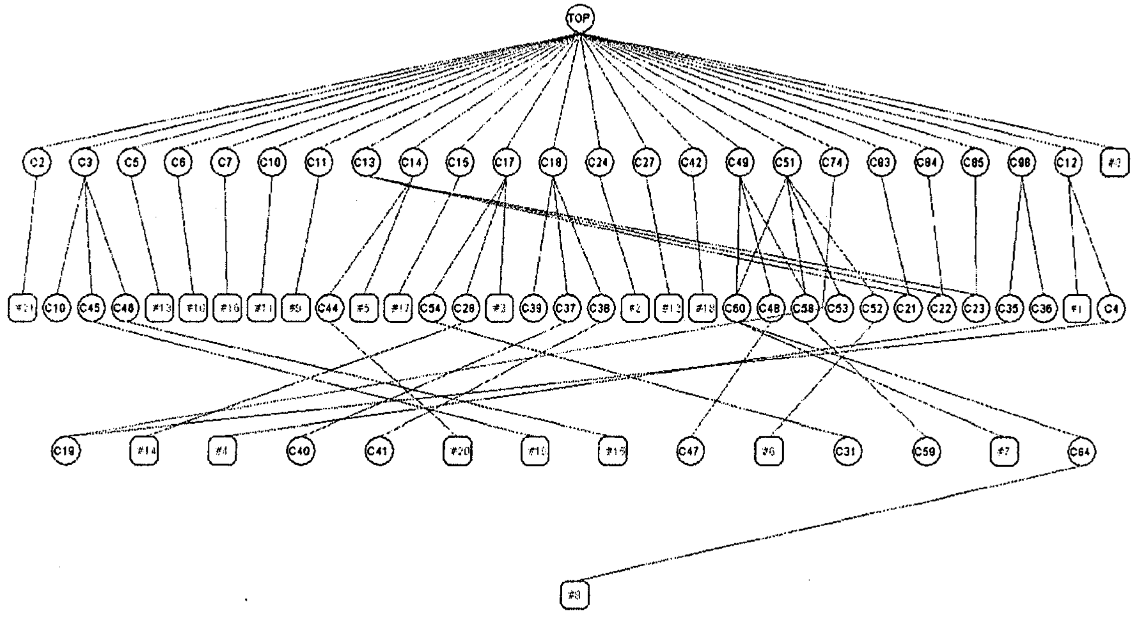

Figure 16: The Top to Bottom display with the virtual node and the omit Bottom option.

Bottom to Top

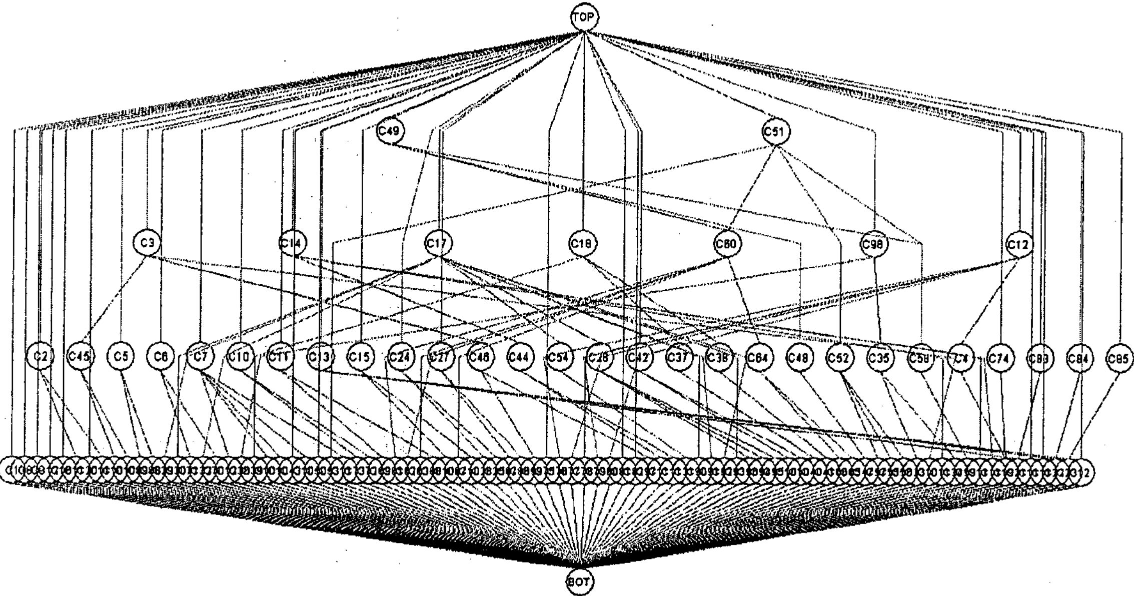

Figure 17: The Bottom to Top display without any option.

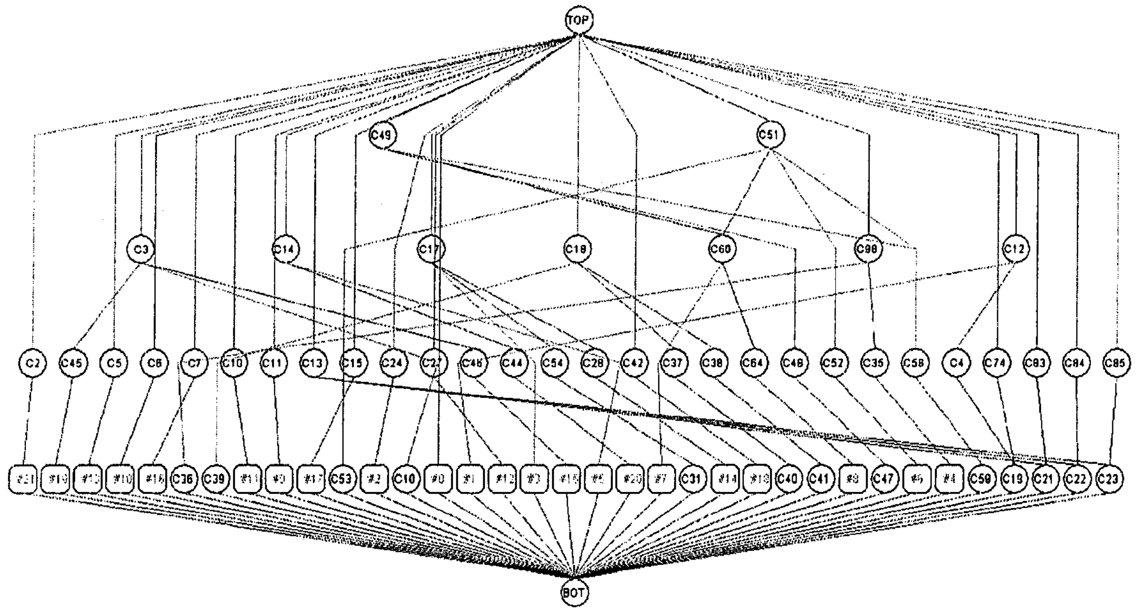

Figure 18: The Bottom to Top display with the virtual node option.

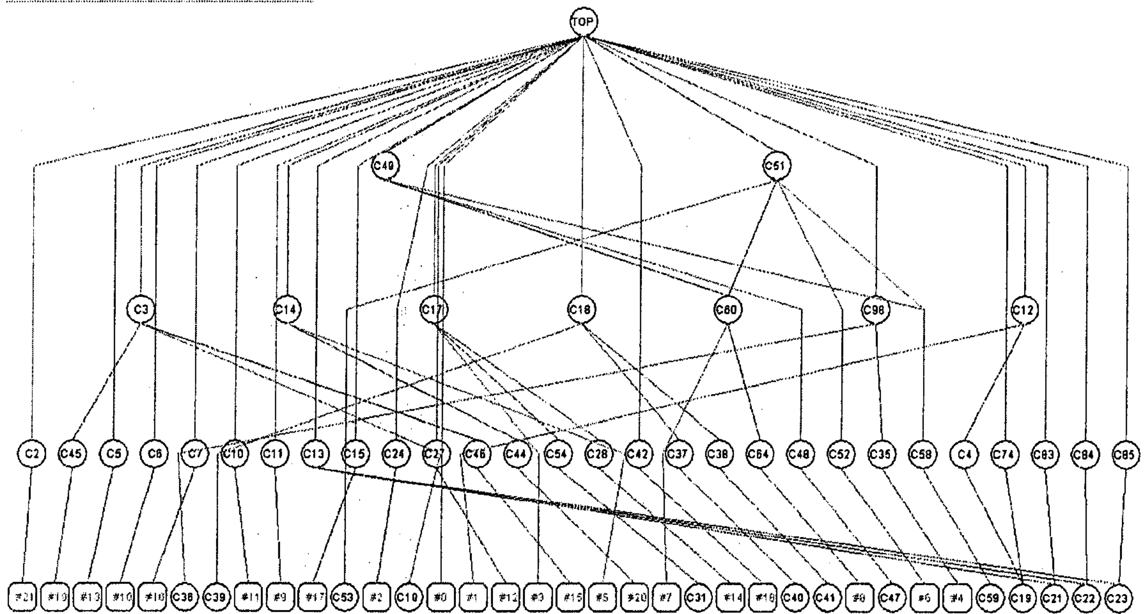

Figure 19: The Bottom to Top display with the virtual node and the omit Bottom option.

Very large file

Top to Bottom

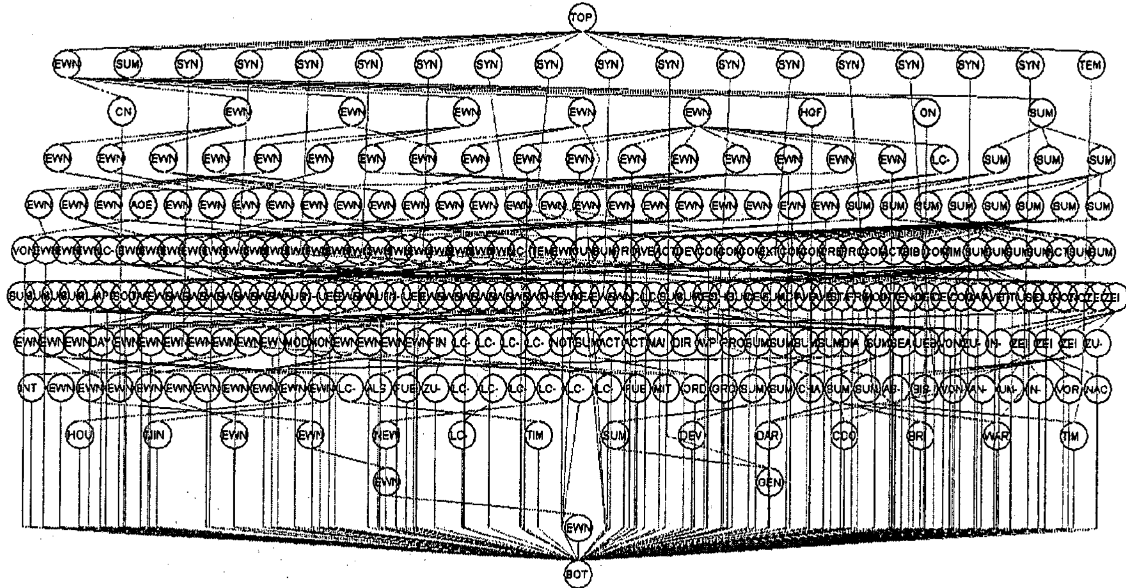

Figure 20: The Top to Bottom display without any option.

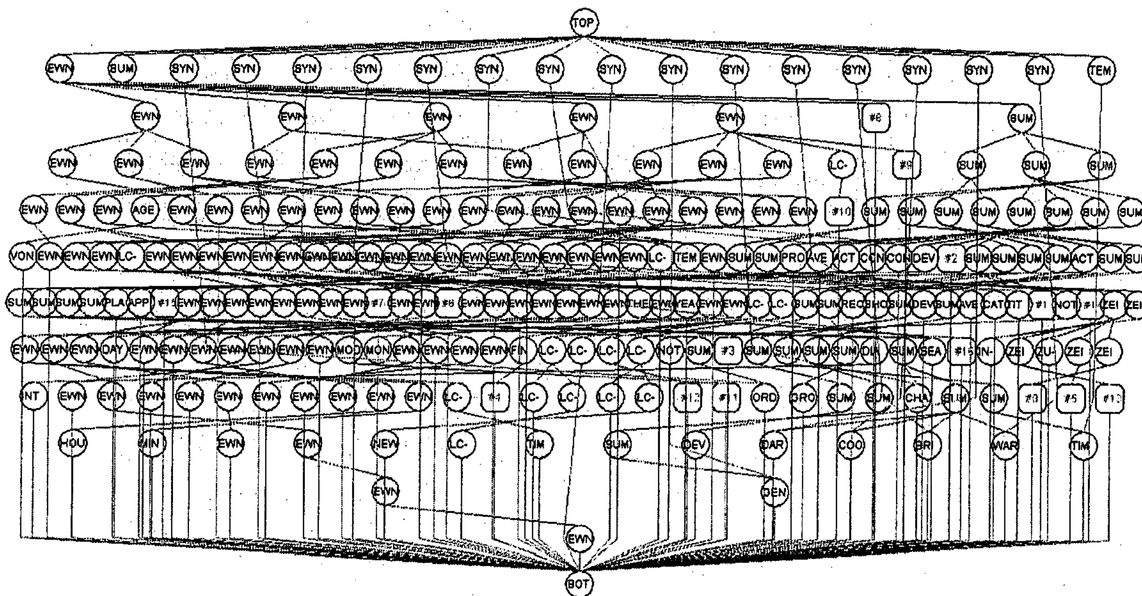

Figure 21: The Top to Bottom display with the virtual node option.

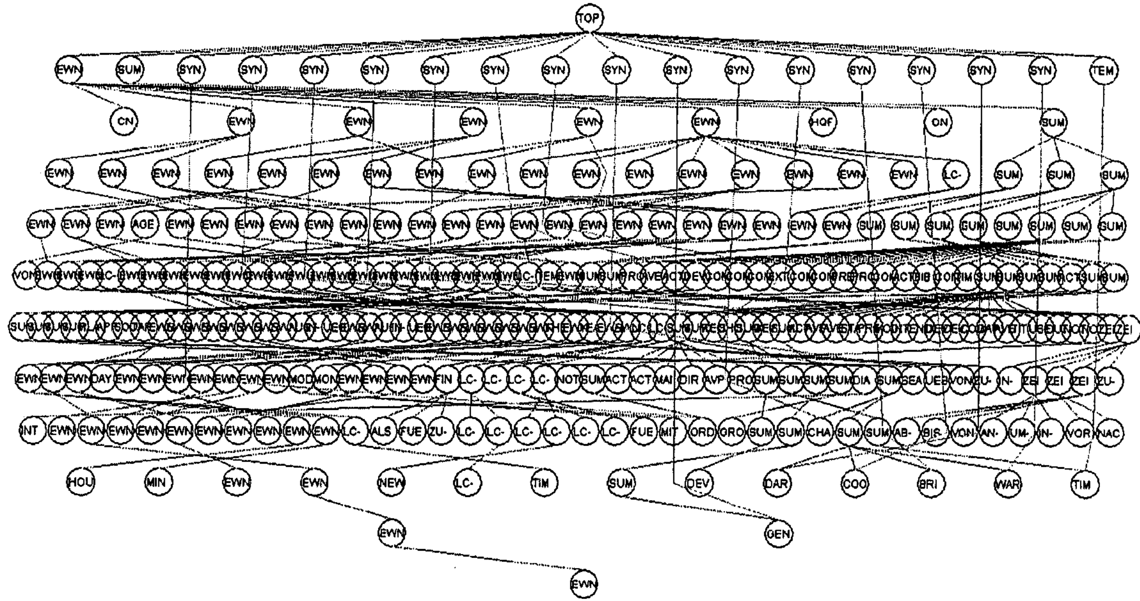

Figure 22: The Top to Bottom display with the Omit Bottom option.

Bottom to Top

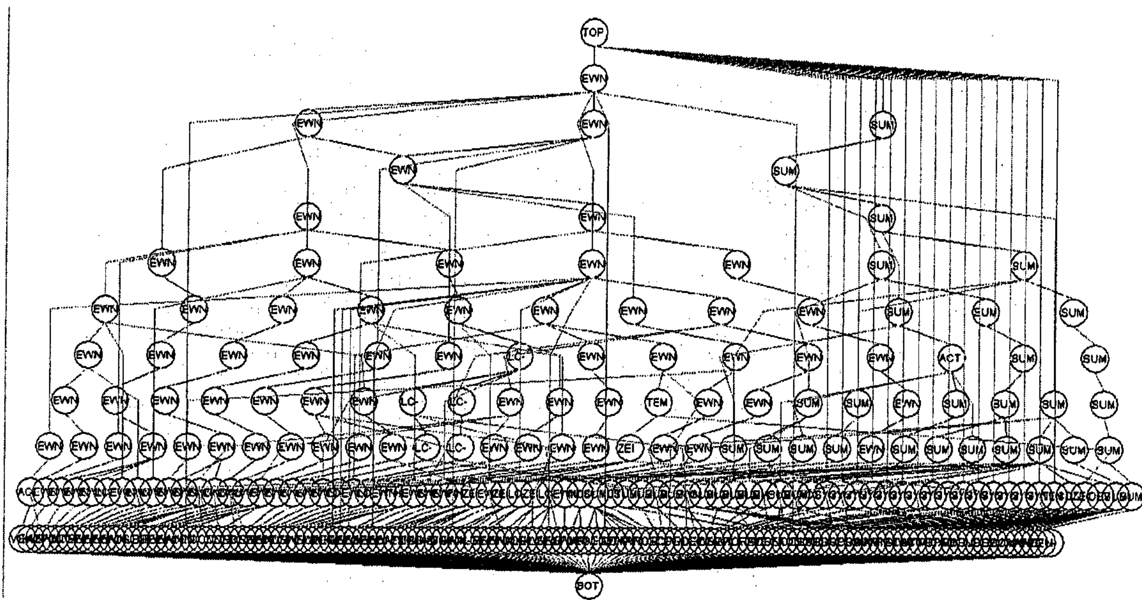

Figure 23: The Bottom to Top display without any option.

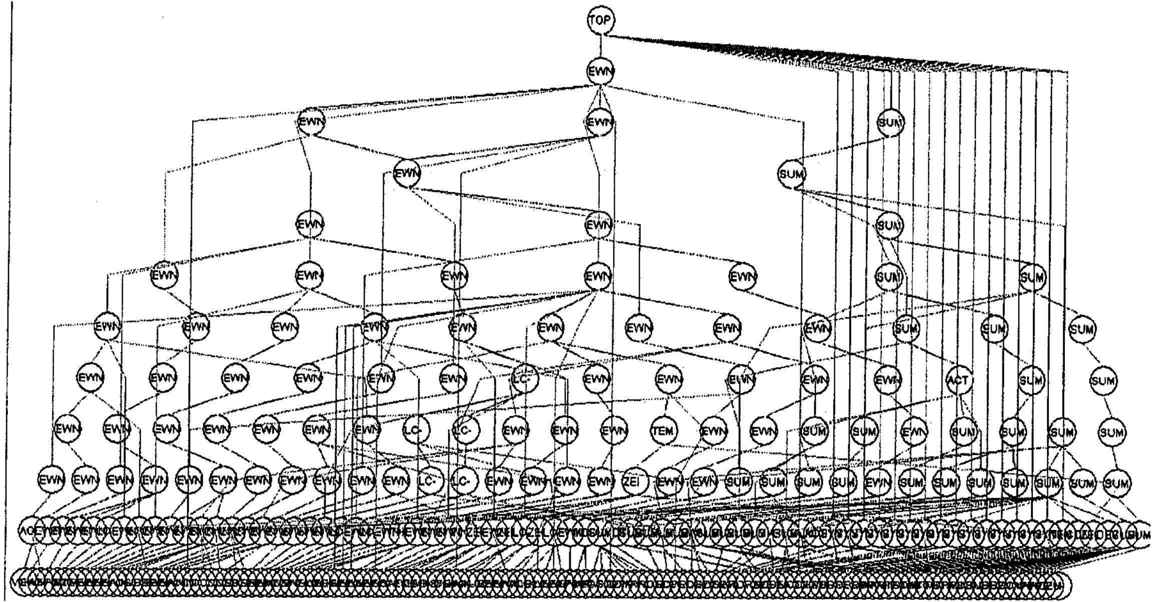

Figure 24: The Bottom to Top display with the Omit Bottom option.

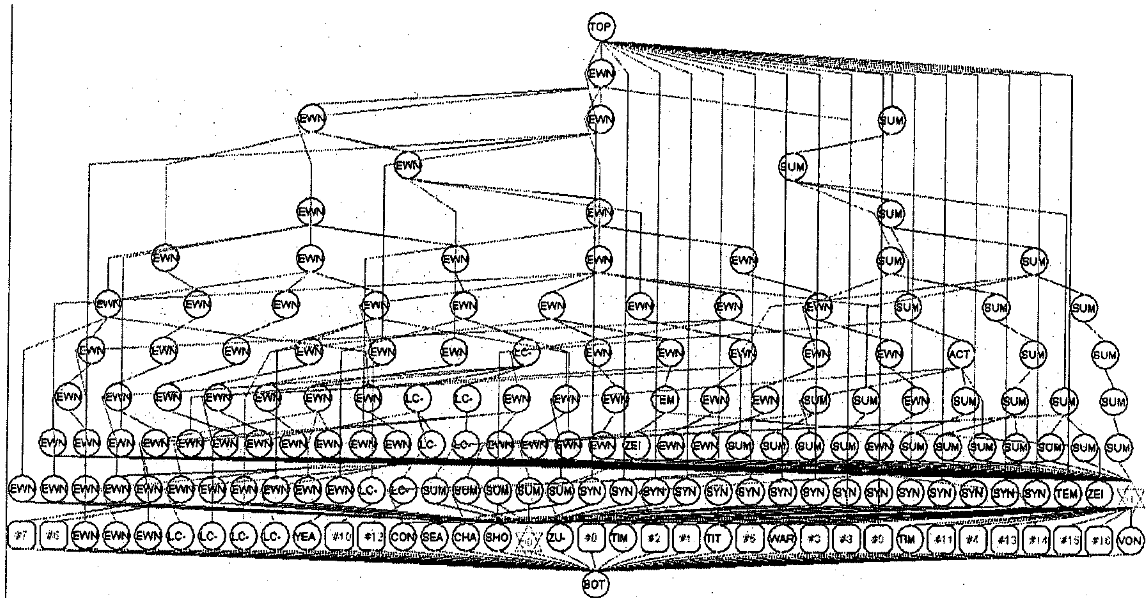

Figure 25: The Bottom to Top display with the virtual node option.

Figure 26: The Bottom to Top display with the virtual node and the omit Bottom option.