

# **LARiSSA:**

## **A prototype implementation of evolving ontologies**

R. Siebes

August, 2001

**Master's thesis of R. Siebes**

Email: [ronny@cs.vu.nl](mailto:ronny@cs.vu.nl)

Department of Artificial Intelligence  
Division of Mathematics and Computer Science  
Vrije Universiteit Amsterdam  
The Netherlands

**Supervisor:**

Dr. F. van Harmelen

**Second reader:**

Dr. D. Fensel

## **Acknowledgements**

In this interesting new millennium, the technocratic society of the western world has major drawbacks, but also leads to astonishing new developments. The Internet is one of the greatest achievements of human kind. The Web seems like a autonomous organism, which is impossible to control. Many discussions with friends about this 'creature' made clear that we are only at the beginning of exploring the possibilities. A fruitful conversation with Frank van Harmelen where my ideas to enhance the WWW are torpedoed by his rational critics, laid the basis for this document. Therefore, I want to thank him and all the people who inspired me for working on this document. Especially, I want to thank Sander van Splunter en Bas Vermeulen, who helped me to make the four years at the university a great time. I thank my parents for making my life possible and keeping me on the track. At last I thank my beloved girlfriend Larissa for supporting me with love and sojakaas.

## Summary

The Web poses unique problems for the use of ontologies because of the rapid evolution and autonomy of web sites. We discuss the dynamic and evolving aspects of ontology languages in distributed environments such as the Web. Some of those languages have reasoning tools to check the structure of the ontology for inconsistent and false information when new information is provided. We call these ontology languages dynamic, because they provide sufficient means to handle new information. The WWW is besides dynamic also a heterogeneous area where information is provided by many different sources. It is not always easy to know the validity of information found at the different (sometimes anonymous) sites. However, when a lot of web-documents refer to a specific site, one has more reason to believe the validity of the information on that site. The same goes for ontologies. Ontologies can contain information from many different users. When a user provides information where many other users agree with (by giving afterward the same information), the information becomes more valid and the user gains more reliability in that specific area. A way to implement this into an ontology language is to work with *ratings*. Every statement gets a sort of *meta-information* which is an indication of the validity of the statement. The rating of a statement increases when other users provide the same statement to the ontology. When a user makes a new statement, the system where the ontology resides, looks at the user's ratings in the surrounding area and calculates the rating for the new statement. This rating mechanism also can be used for implementing the devaluation of validity from statements in time. Especially *relative* statements like 'a pentiumIII processor is *fast*' are vulnerable for devaluation of validity. If we did nothing with this problem, the validity of the ontology decreases. The rating mechanism can be used to automatically decrease the validity of all the statements in time. We assume that statements which are valid, even after a long period of time, are in the meanwhile strengthened by identical statements which increases the rating. This assumption assumes that enough users add information to the ontology periodically. The rating mechanism can be viewed as an extension of dynamic ontologies. When implemented properly and working with enough users, the ontology should increase or at least, not decrease its overall validity whereby the amount of information increases in time. When an ontology uses this mechanism we indicate this as an *evolving* ontology. Not only new statements provided by the user have to increase the amount of information. Reasoning methods like analogous reasoning, heuristic-based generalisation/specialisation and standard deduction techniques can be used to automatically extract information. These methods all contribute to make the ontology evolving. The literature study in this paper shows that these dynamic and especially evolving aspects of ontologies largely have been ignored so far. We also conclude that all/most ontology languages are based on classical logic with hard notions of true/false, not allowing for different grading of statements. We introduce a system called LARiSSA, which will be used as an experimental platform to implement several evolving methods. This system we use for experimental purpose, where we implement the rating method and some of the other methods and conclude that these methods contribute to a useful way to handle ontologies. Therefore we do not only have a paper about this new subject, but also an implemented system which can be used for future experiments.

# Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>5</b>
<b>2</b>	<b>LITERATURE STUDY .....</b>	<b>5</b>
2.1	DEFINITION OF 'DYNAMIC' AND 'EVOLVING' .....	6
2.2	ONTOLOGY LANGUAGES AND APPLICATIONS FOR THE SEMANTIC WEB .....	7
2.2.1	<i>Oil</i> .....	7
2.2.2	<i>OntoBroker and On2Broker</i> .....	8
2.2.3	<i>Shoe</i> .....	10
2.3	SUMMARY .....	12
<b>3</b>	<b>EVOLVING METHODS.....</b>	<b>13</b>
3.1	MAKING IMPLICIT INFORMATION EXPLICIT .....	13
3.2	GENERALISATION/SPECIALISATION.....	14
3.3	COMBINE RELATIVE AND ABSOLUTE STATEMENTS .....	16
3.4	DEDUCTION BY ANALOGY.....	19
3.5	RATINGS .....	23
3.5.1	<i>Ratings and semantic distance</i> .....	23
3.5.2	<i>Calculation of a user-specific reliability rating for a statement</i> .....	24
3.5.3	<i>Upgrading individual ratings</i> .....	28
3.5.4	<i>Aggregating user-specific ratings into a single statement rating</i> .....	29
3.5.5	<i>Devaluation function</i> .....	34
3.6	SUMMARY .....	35
<b>4</b>	<b>LARISSA .....</b>	<b>37</b>
4.1	THE ONTOLOGY ENVIRONMENT .....	37
4.2	STRUCTURE OF THE ONTOLOGY .....	39
4.3	REQUIREMENTS OF THE SYSTEM .....	44
4.4	SOFTWARE & CO. ....	45
4.5	INTERNAL REPRESENTATION OF THE DATA .....	46
4.6	QUERY AND MANIPULATION LANGUAGE.....	47
4.6.1	<i>Adding and retrieving object information</i> .....	48
4.6.2	<i>Adding and retrieving property information</i> .....	50
4.6.3	<i>Adding and retrieving functions</i> .....	51
4.7	SUMMARY .....	52
<b>5</b>	<b>RESULTS.....</b>	<b>52</b>
<b>6</b>	<b>CONCLUSIONS .....</b>	<b>62</b>
<b>7</b>	<b>REFERENCES.....</b>	<b>64</b>

## 1 Introduction

The need is growing to make the sources on the web not only readable by human readers. Computers should also be able to extract various information sources. A way to organise this information is in an ontology. The role of ontologies is to capture domain knowledge and provide a commonly agreed understanding of a domain. Most ontologies are often seen as basic building blocks for the Semantic Web. However, those pieces of knowledge are often not static, but change over time. Domain changes, adaptations to different tasks, or changes in the conceptualisation require modifications of the ontology. Therefore an ontology language should provide proper means to handle this dynamic environment. The web is also a heterogeneous environment, where different providers of information exist. As such, the reliability of such information is questionable, and it is inevitable that inconsistencies will arise. When information is treated as possible truth instead of absolute truth, by assigning a relative value to it, inconsistencies don't have to give problems. [Heflin, 1999]. This paper tries to examine the dynamic and heterogeneous part of an ontology language. A new method of dealing with the reliability of information is proposed. Another issue where we look at in this paper is to extend an ontology not only by adding new information but also by looking at the existing information to derive implicit information.

This paper is organized into five main chapters. This chapter is a brief introduction into the subject and shows the structure of the document. In the second chapter, we give the definition of *dynamic* and *evolving*. We examine some ontology languages by this definition, and conclude that the evolving part is lacking. The third chapter introduces new methods, to enhance the ontology environment to make it evolving. Chapter four introduces a new system LARiSSA that is an experimental environment to implement the methods from chapter three. We make up the requirements of the ontology structure and about the performance of the system. Chapter five shows the results of the implementation of LARiSSA. In the last chapter, we discuss the results and make conclusions.

## 2 Literature study

There is a lot of information about ontologies. Several languages are developed, to pave the way for a common method of handle semantic information to the WWW. Some also reached the level of W3C recommendation. It is not the purpose of this paper to explore these languages into detail. We only give a brief overview and how they cope with the special attributes of web information.

## 2.1 Definition of 'dynamic' and 'evolving'

In the literature, the terms *dynamic* and *evolving* are not sufficient explained. Often these terms are used mixed-up and used together. Therefore we try to give a clear definition of both terms.

### 2.1.1.1 Dynamic

We define that a dynamic ontology provides efficient means to alter or extend its structure. When information (ranging from a single fact to a complete ontology) is added or altered, the ontology should remain a correct and consistent structure. [Klein 2001] emphasises the need of a versioning methodology to handle revisions of ontologies. Techniques like the use of namespaces, are used to discriminate objects with an identical name to indicate the source of the information. Another technique to maintain a correct ontology structure is by using a consistency checking tool like FaCT<sup>1</sup>. Each time when a new statement or ontology is provided, the tool checks the ontology for correct class hierarchies, prevention of circular definitions etc. We define an ontology language 'dynamic' when it provides sufficient means to maintain a consistent structure after the addition/altering of information.

### 2.1.1.2 Evolving

The definition of *dynamic* does not prevent 'contradictions' that are not logically inconsistent. If claimant *A* says 'father(Mark,Katherine) and claimant *B* says father(Katherine,Mark), the apparent contradiction is because one claimant is misusing the father relation. However, this does not change the fact that *A* and *B* made those claims. Similar problems may occur in ontologies where inference rules derive a conclusion whose interpretation would be inconsistent with another ontology. Web systems simply cannot assume that all of the information has been entered solely under a knowledge engineer's watchful eye, and is therefore correct and consistent. As authority on the Internet is distributed, it cannot and does not make any such a promise. Since there is often no editorial review or quality control of Web information, each page's *reliability* must be questioned [Heflin, 1999b]. A method, which will be discussed in more detail, is the increase of reliability of the provider when other people give exactly the same information. The reliability of a user is an indication of the *validity* of its provided information. An evolving ontology, which deals with a heterogeneous environment, should provide means to express this validity information about statements. Time also could influence the validity of information. Especially relative information like 'a pentiumIII has a *fast* processor' is vulnerable for decay. Again, if the ontology has the predicate 'evolving', it should provide means to express this.

We think that the words 'evolving ontology' mean more than a dynamic ontology where the validity of the information doesn't decrease. Evolving also indicates a growing amount of information in time. A dynamic ontology should be sufficient to guarantee this, because new information is added periodically. However we could also think about extraction of

---

<sup>1</sup> <http://www.cs.man.ac.uk/~horrocks/FaCT>

new information out of existing information. Methods, which we discuss in the next section, like analogous reasoning, heuristic-based generalisation and deduction, can expand existing information with derived information. In that case, the ontology evolves automatically to fill the gap between existing information and new information provided by the user. We indicate these methods as *evolving methods*.

## 2.2 Ontology languages and applications for the semantic web

In this section, we give a brief overview of some different languages and their application tools, which are developed to build ontologies to organise web-based information. Some of these languages rely on inference engines (classifiers) to compute a class hierarchy and to determine class membership of instances based on the properties of classes and instances. We show in this paper that inference engines are the key for evolving ontologies, we skip languages which do not provide possibilities for an inference engine or are too complex for it, like Ontolingua ([Gruber, 1992], [Farquhar et al., 1997]). For each language we look at the dynamic- and evolving possibilities.

### 2.2.1 *Oil*<sup>2</sup>

The Ontology Inference Layer OIL is a proposal for a web-based representation and inference layer for ontologies, which combines the widely used modelling primitives from frame-based languages with the formal semantics and reasoning services provided by description logics. It is compatible with RDF Schema<sup>3</sup> (RDFS), and includes a precise semantics for describing term meanings (and thus also for describing implied information).

#### 2.2.1.1 Dynamic possibilities

OIL contains a construction to modularise ontologies. This mechanism is identical to the namespace mechanism in XML<sup>4</sup> and XML schema<sup>5</sup>. It amounts to a textual inclusion of the imported module, where name-clashes are avoided by prefixing every imported symbol with a unique prefix indicating its original location. However, much more elaborate mechanisms would be required for the structured representation of large ontologies. Means of renaming, restructuring, and redefining imported ontologies must be available. Future extensions will cover parameterized modules, signature mappings between modules, and restricted export interfaces for modules [Fensel et. al, 2000].

OIL has the possibility to use the FaCT<sup>6</sup> reasoner to check the consistency of all the class definitions in an ontology. FaCT (Fast Classification of Terminologies) is a Description

---

<sup>2</sup> <http://www.ontoknowledge.org/oil/>

<sup>3</sup> <http://www.w3.org/TR/rdf-schema/>

<sup>4</sup> <http://www.w3.org/XML/>

<sup>5</sup> <http://www.w3.org/XML/Schema>

<sup>6</sup> <http://www.cs.man.ac.uk/~horrocks/software.html>

Logic (DL) classifier that can also be used for consistency checking in modal and other similar logics. Although OIL provides this mechanism for inheriting values from superclasses, such values cannot be overwritten. As a result, such values cannot be used for the purpose of modelling default values. If an attempt is made at "overwriting" an inherited attribute value, this will simply result in inconsistent class definitions that have an empty extension. For example, if we define the class "CS professor" with attribute "gender" and value "male", and we subsequently define a subclass for which we define the gender attribute as "female", this subclass will be inconsistent and have an empty extension (assuming that "male" and "female" are disjoint).

#### 2.2.1.2 Evolving possibilities

The FaCT reasoner can, in addition to consistency checking, be used to discover subclass/super-class (subsumption) relations that are implied by the definitions in the ontology but not explicitly stated.

### 2.2.2 *OntoBroker and On2Broker*

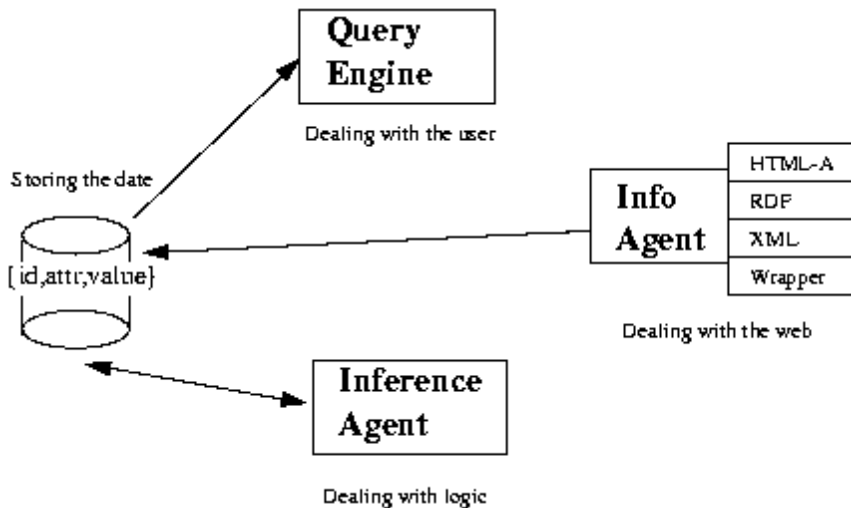
[Fensel 2001 Ontologies: a silver bullet for knowledge management and electronic commerce]

Ontobroker ([Fensel et al., 1998], [Decker et al., 1999]) applies Artificial Intelligence techniques to improve access to heterogeneous, scattered and semi-structured information sources as they are presented in the World Wide Web or organization-wide intranets. It relies on the use of ontologies to annotate web pages, formulate queries, and derive answers. The ontologies can be written in an *annotation* language called HTML<sup>A</sup> to enable the annotation of HTML documents with machine-processable semantics.

Ontobroker relies on two tools: a webcrawler and an inference engine. The webcrawler collects web pages from the Web, extracts their annotations, and parses them into the internal format. The inference engine takes these facts together with the terminology and axioms of the ontology, and derives the answers to user queries. To achieve this it has to do a rather complex job. First, it translates frame logic into predicate logic and, second, it translates predicate logic into Horn logic via Lloyd-Topor transformations. As a result it obtains a normal logic program. Standard techniques from deductive databases are applicable to implement the last stage: the bottom-up fixpoint evaluation procedure.

On2broker [Fensel et al., 1999] is the successor system to Ontobroker. The major new design decisions in On2broker are the clear separation of the query and inference engines and the integration of new web standards like XML and RDF. The inference engine works as a demon in the background. It takes facts from a database, infers new facts, and returns these results back into the database. The query engine does not directly interact with the inference engine. Instead it takes facts from the database. An overview of On2brokers architecture is given in Fig.1

Ontologies are the overall structuring principle in On2broker. The info agent uses them to extract facts, the inference agent to infer facts, the database manager to structure the database, and the query engine to provide help in formulating queries



**figure 1: The gist of On2broker**

### 2.2.2.1 Dynamic possibilities

Here we only look at On2broker because the only difference with Ontobroker is the separation of the query and inference engine and the integration of some new web standards, which have no direct influence on the dynamic possibilities. On2broker can read various input formats like F-Logic, a Datalog/Prolog like syntax and RDF. Thus it provides a homogeneous access to an inhomogeneous set of information sources and input formats. RDF and XML make use of name spaces to identify a resource. On2broker generates out of a RDF fact or a XML fact an F-Logic fact where the name space is included as following:

```
"http://www.w3.org/Home/Smith"["http://description.org/schema$Creator"->>literal("John Smith")]
```

There are two places where consistency checking could be done, namely directly by the 'info agent' or afterwards by the 'inference agent'. When done directly, the server can get to overloaded if too many users provide information at the same time. The advantage of direct checking, is that the database always contains consistent information, therefore answering queries can be done without errors. Afterward checking can spread the server load, because the reasoning process can be done at quiet times. However, one should think how to handle possible inconsistent data in the database. Unfortunately, On2broker does not consistency check at all, it would have just reasoned based on inconsistent information.

### 2.2.2.2 Evolving possibilities

The inference engine uses facts and ontologies to derive additional factual knowledge that is only provided implicitly. It frees knowledge providers from the burden of specifying each fact explicitly. Because On2broker doesn't check the consistency of the data, consistent facts should be provided by the user, otherwise the inference engine could infer nonsense.

### 2.2.3 *Shoe*

SHOE<sup>7</sup> [Heflin et al., 2000] is an ontology-based knowledge representation language designed for the Web. The SHOE project uses a language that, like Ontobroker, is embedded in HTML. SHOE's basic structure consists of *ontologies*, entities which define rules guiding what kinds of assertion may be made and what kinds of inferences may be drawn on ground assertions, and instances, entities which make assertions based on those rules. Because SHOE exists in a distributed environment with little central control, SHOE treats assertions as claims being made by specific instances instead of facts to gather and intern as generally-recognised truth. As said SHOE's syntax is an extension of HTML, but an almost identical XML syntax is also available.

#### Dynamic possibilities

SHOE's design philosophy avoids the possibility of contradictions between agent assertions. SHOE does this in four ways [Heflin, 2000]

1. SHOE only permits assertions, not retractions.
2. SHOE does not permit negation.
3. SHOE does not have single-valued relations, that is, relational sets which may have only one value (or some fixed number of values).
4. SHOE includes the claimant as part of a claimed assertion.

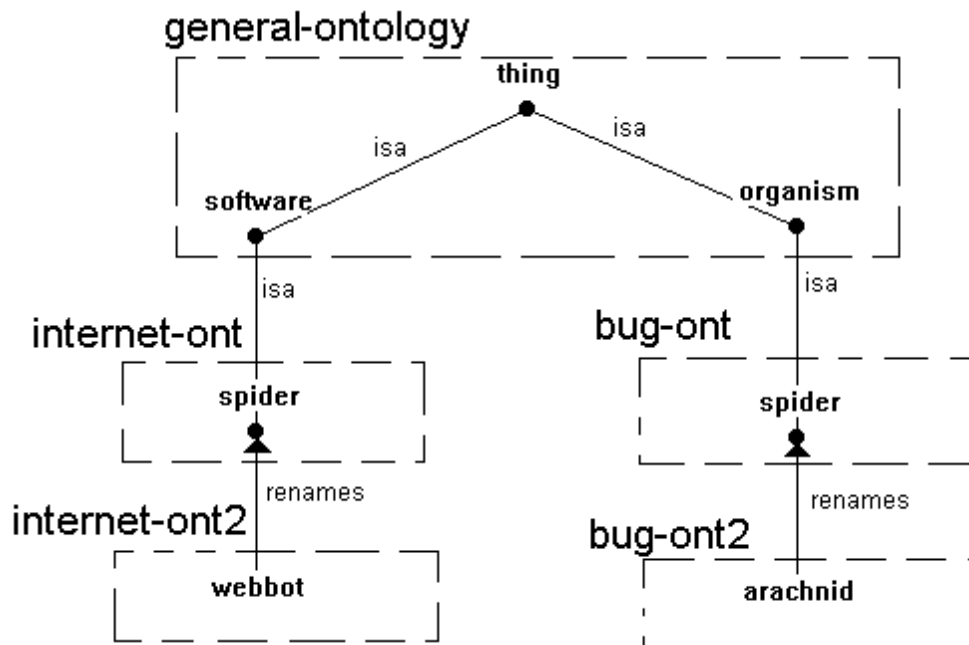
SHOE ontologies are made publicly available by locating them on web pages. SHOE ontologies build on, or extend, other ontologies, forming a lattice with the most general ontologies at the top and the more specific ones at the bottom. An ontology extension is expressed in SHOE with the <use-ontology> tag, which indicates the id and version number of an ontology that is extended. An optional url attribute allows systems to locate the ontology if needed and a prefix attribute is used to establish a short local identifier for the

---

<sup>7</sup> <http://www.cs.umd.edu/projects/plus/SHOE/>

ontology. When an ontology refers to an element from an extended ontology, this prefix and a period is appended before the element's name. In this way, references are guaranteed to be unambiguous, even when two ontologies use the same term to mean different things. By chaining the prefixes, one can specify a path through the extended ontologies to an element whose definition is given in a more general ontology.

The primary means of achieving interoperability in SHOE is through use of the ontology extension and renaming features. Two categories are similar to the extent that they share the same supercategories. As a result, ontologies are interoperable to the extent that they share the same ancestor ontologies. For example, in Figure 2, the term spider means different things in internet-ont and bug-ont because the categories have different ancestors. However the term webbot in internet-ont2 means the same thing as spider in internet-ont because a def-rename indicates that it is an alias of the term. Achieving interoperability in this manner depends on the ontologies being constructed correctly; therefore the most general ontologies (the ones that affect the most people) should be designed and maintained by experts.



**figure 2: Ontology Interoperability in SHOE.**

In this figure, *is a* refers to the presence of a def-category tag and *renames* refers to the presence of a def-rename tag

About consistency checking tools [Heflin, 1999b] wrote:

"We have mapped SHOE into a first order logic definite program and used this to discuss how different types of revisions to an ontology affect existing data sources. We

have shown that revisions that add categories or relations will have no effect. Revisions that modify rules may change the answers to queries against that modify rules may change the answers to queries against the data sources, and revisions that remove categories or relations may make the ontology incompatible with the data sources. This knowledge should be used in weighing the benefits and costs of any revision. Although ideally integration is a by-product of ontology extension, in a large, distributed environment manual ontology integration will need to be performed periodically."

Although this quote shows that revision is possible, no ready-to-use tools are written to check the consistency of a SHOE ontology.

### Evolving possibilities

Like On2Broker, SHOE ontologies permit the discovery of implicit knowledge through the use of taxonomies and inference rules, allowing content providers to encode only the necessary information on their web pages, and to use the level of detail that is appropriate to the context. Shoe-enabled web tools can then process this information in novel ways to provide more intelligent access to the information. Interoperability is promoted through the sharing and reuse of ontologies. However, in contrary to On2broker, SHOE does not have such a tool yet to extract implicit information. Heflin [1998] only describes that a SHOE ontology is suitable for this type of reasoning.

## **2.3 Summary**

Especially in dynamic and heterogeneous environments like the WWW, ontologies should be easy maintainable, whereby manipulation of the information doesn't lead to inconsistent structures. Those ontologies we call 'dynamic ontologies'. We stated that the definition of a dynamic ontology is not enough to deal with the problems of a heterogeneous environment, where also the validity of the provided information is important. This validity is influenced by the reliability of the different users, and time aspects. Therefore we introduce 'evolving ontologies' which deal with these aspects. The word 'evolving' implies that the structure should get 'better' in time. Therefore providing methods to maintain the validity of the ontology is not enough. When new information is provided to the ontology, the structure grows and when the validity of this structure at least stays the same, we can say that it evolves, i.e. gets 'better'. Not only new information has to lead to a growing structure. Reasoning mechanisms like analogous reasoning, heuristic-based generalisation/specialisation and classical deduction can be used to extract information out of existing data. These methods are useful to 'glue' scattered information and contribute to let ontologies evolve.

We examined three web-related ontology languages, where we looked at the dynamic and evolving aspects. To begin with the dynamic aspect, all languages provide means to handle objects with identical names and different meanings by the use of name spaces. However, tools to detect inconsistent data in the ontology are too slow [FaCT in combination with

OIL] or even are not provided [On2broker and SHOE]. Regarding the evolving aspect, only OIL in combination with FaCT and On2broker provide a method to extract implicit information and is implemented in a ready-to-use tool. They do this to transform the data into a kind of first-order logic and derive additional facts. Other methods, like allowing different gradings of statements instead hard notions of true/false based on classical logics, are not provided by those languages. Therefore, we conclude from the literature study that dynamic and especially evolving aspects of ontologies have been largely ignored so far. In the next chapter we look at some different methods which make an ontology evolving.

### 3 Evolving methods

In this chapter we look at different methods which contribute to the evolving structure of the ontology. First we look at the already used method, introduced at the literature study, namely deduction of implicit information. Then we look at new methods and look at their contribution to evolving ontologies. Those methods are already used in other areas, and we perhaps just have to fit them into the ontology. We look at the possibilities and the difficulties of these methods.

#### 3.1 Making implicit information explicit

As we have seen in the literature study, ontologies can be combined with an inference engine to derive implicit information. Inference methods, like unification, forward or backward resolution, and inheritance, can be viewed as evolving method because they extend the structure of an ontology. There are two main possibilities where to extract this, namely (1) at the developers or (2) at the user's side.

- (1) A tool, which makes a connection with a reasoning engine to derive implicit information at the time of the development of an ontology, is *OilEd*<sup>8</sup>. *OilEd* is a simple ontology editor, which allows the user to build ontologies using OIL. The intention behind *OilEd* is to provide a simple, freeware editor that demonstrates the use of, and stimulates interest in, OIL. *OilEd* is not intended as a full ontology development environment - it will not actively support the development of large-scale ontologies, the migration and integration of ontologies, versioning, argumentation and many other activities that are involved in ontology construction. Rather, it is the "NotePad" of ontology editors, offering just enough functionality to allow users to build ontologies and to demonstrate how we can use the FaCT reasoner to check those ontologies for consistency. This FaCT reasoner also extracts implicit information, like adding the same properties to object 'car' when the object 'automobile' with its properties is given, and a synonym statement between both is provided.
- (2) Derive implicit information at the time of using the ontology. In the design of Ontobroker, the web crawler and the inference engine are separated. The web crawler periodically collects information from the web and caches it. The inference engine uses this cache when answering queries. The decoupling of inferencing and fact collection is

---

<sup>8</sup> <http://img.cs.man.ac.uk/oil>

done for efficiency reasons. On2broker refines this architecture by introducing a second separation: separating the query and inference engines. The inference engine works as a demon in the background. It takes facts from a database, infers new facts and returns these results back into the database (fig 2). The query engine does not directly interact with the inference engine. Instead, it takes facts from the database.

When information is made explicit and added into the ontology, the structure is evolved, because it contains more information. Logicians would argue that the information already is implicit in the ontology, thus no new information is derived. However, when information is made explicit, the quality of the ontology is increased by the fact that reasoning time isn't needed anymore to extract the implicit information.

### 3.2 Generalisation/specialisation

When an ontology is organised in an 'isa' structure, i.e. class hierarchy, we can generalise some frequently occurring features from classes to their superclasses. This heuristical method often is used in machine learning [Carbonell, 1986]. Next we describe how such a method could be useful in our context. We explain the use of the generalisation method with an example:

Assume the next small ontology about printers:

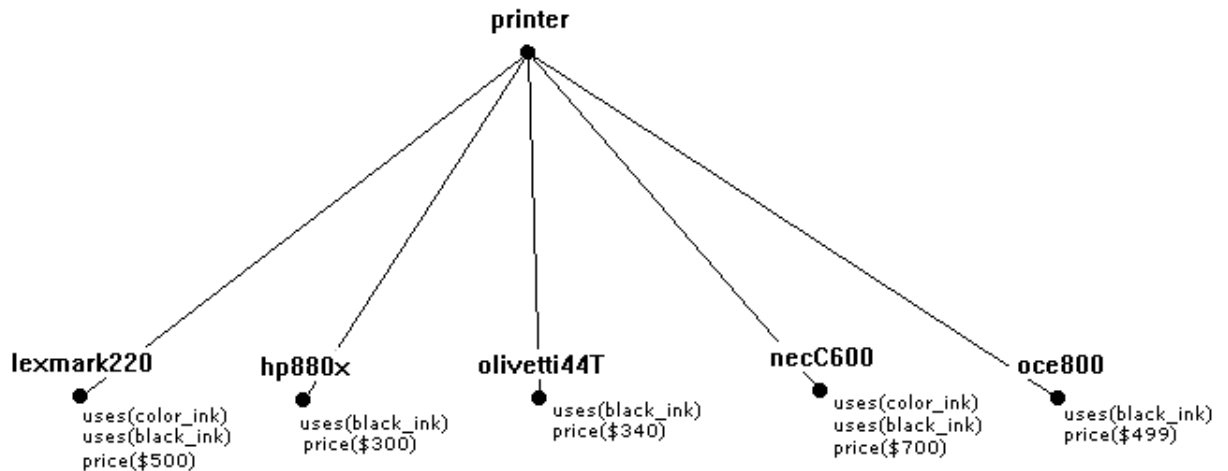


figure 3

Here *every* printer uses black ink, thus we could make an assumption that the use of black ink is a general property of a printer. We can now retract the statement 'uses(black\_ink)' from all the children of the class printer. After this procedure we get the following ontology:

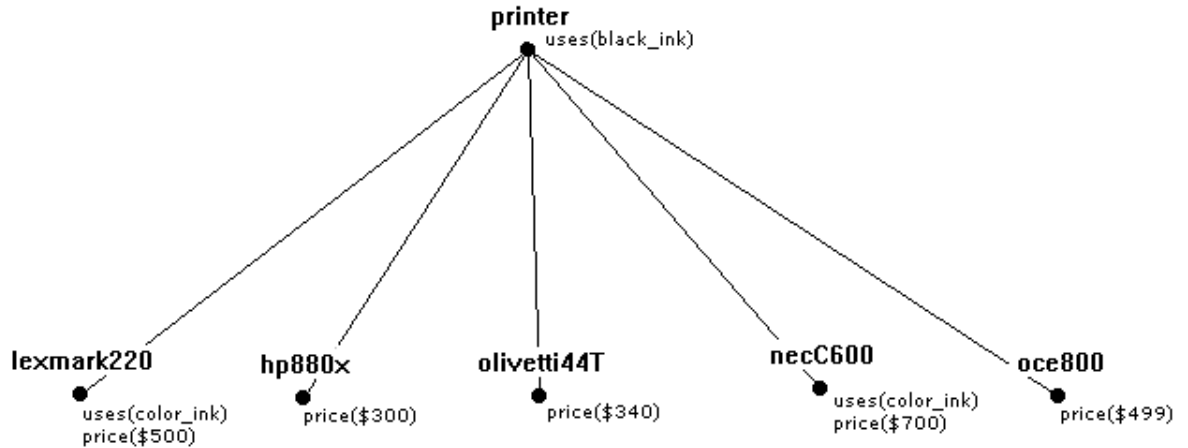


figure 4

We can see in figure 4 that the ontology "learned" new information that was not already presented in the old ontology, namely the fact that all printers use black ink. This means that the ontology is changing/evolving through such an operation.

The validity of such a heuristic depends of course on the size of the ontology, therefore statistic methods are necessary to examine when an heuristic can be applied. After the generalisation, some information of the individuals has become redundant, namely the generalised information itself. The retraction of redundant statements reduces the specific redundancy problems (e.g. updating errors) and we have a smaller ontology, which is more efficient. Another advantage is the possible assistance from the system when adding new information to the ontology: when a user adds a new printer into the ontology, the system adds, with possible intervention from the user, that this printer uses black ink.

In our example ontology about printers it also provides information about the price of the different printers. If the system has a notice of the range between prices, it could deduce *price(\$300-\$700)* for the object *printer*, indicating that the price of printers lies between \$300 and \$700. Although it is not a generalising method but merely a summarising method, it could be useful in the e-commerce domains to compare different hardware companies.

Another possibility of generalisation could be automatic detection of standard relations. For example: imagine an ontology where the most individual computers have a relation which indicates the type of CPU, e.g. 'CPU(AMD-Athlon 600MHz)' or 'CPU(PentiumIII 800MHz)'. The system could automatically detect that this relation is standard for computers, so that when a new computer is added without a CPU, the system could ask the user if he/she forgot to add the CPU(...) relation.

Because we are working with dynamic ontologies, there is a possibility that heuristically inferred information becomes incorrect. Therefore there should be a way to *retract* information. *Specialisation* is such a method. When too many inconsistencies arise from a statement in a parent with statements from its children, this statement is retracted from the

parent and added to the children which don't have this inconsistency. But how can we detect those inconsistencies? It would be simple, when *negative* information is provided like 'a necP2200 *not* uses black ink'. In such a case we could compare the number of children with the negative statement with the positive general statement at the parent. If this ratio is too high, the statement at the parent should be retracted. Retracting generalised information becomes more difficult, when statements *semantically* exclude each other. An example can also be found in the printer domain, namely that of toners and ink-cartridges: when a printer uses toner, it doesn't use ink. When, for example, the ontology generalised that printers use ink-cartridges, and after that several printers are added to the ontology which use toner, the generalisation should be retracted. However, the system could not do it on the base of negative information (not having ink-cartridges). The only solution for such a problem is to explicitly state this to the inference engine, or include it into the ontology like 'exclude(ink-cartridges,toner)'.

### 3.3 Combine relative and absolute statements

At this moment I am working on a Pentium 166MHz, good enough for the text processing tool, but really too slow for compiling the Java code from LARiSSA. When someone asks me what the relative speed is of this computer, I would indicate it as **slow**. However, four years ago, when my parents bought this thing, I went to bed with a smile, thinking 'wow, we really have a supercomputer in our house!' What I want to show here is that an absolute statement (166MHz) can have relative indicators (slow or fast), which can differ from time to time. The introduction of relative statements in an ontology has the advantage that users which query the system, don't have to know the technical specification of objects like 'give me a computer with a 1.4 GHz processor', but can suffice with 'give me a computer with a fast processor'. Our definition of an evolving ontology is that the statements, at least, remain their validity in time.

When we look at the computer example, we could introduce a function that describes the relative devaluation of absolute computer speed in time. We can use *fuzzy logic* techniques to express these functions. We illustrate this with the following example:

We assume an ontology that contains information about computers with their absolute processor speed. For simplicity we say that a 'fast' computer only means that it has a 'fast' processor (we assume here that the rest of the computer is configured correctly to use the full processor's capacity).

We express the relative translation of processor speed in the following fuzzy function:

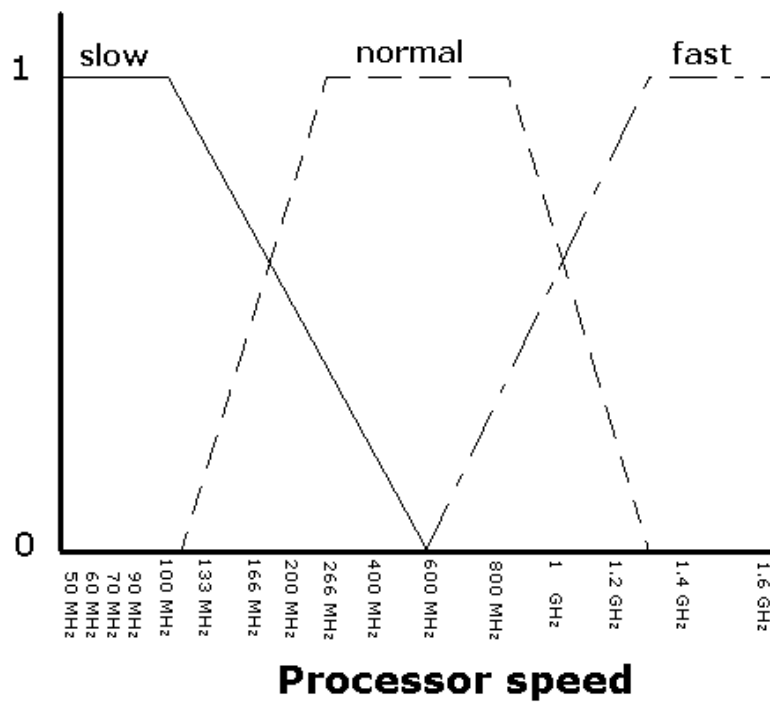


figure 5

In figure 5, a fast computer has a processor with a speed between 1.3 and 1.6 GHz. It is important to note that the scale of processor speed does not have to be a linear scale.

In the next figure we show the state of relative translation after a period where several faster processors have been added to the ontology:

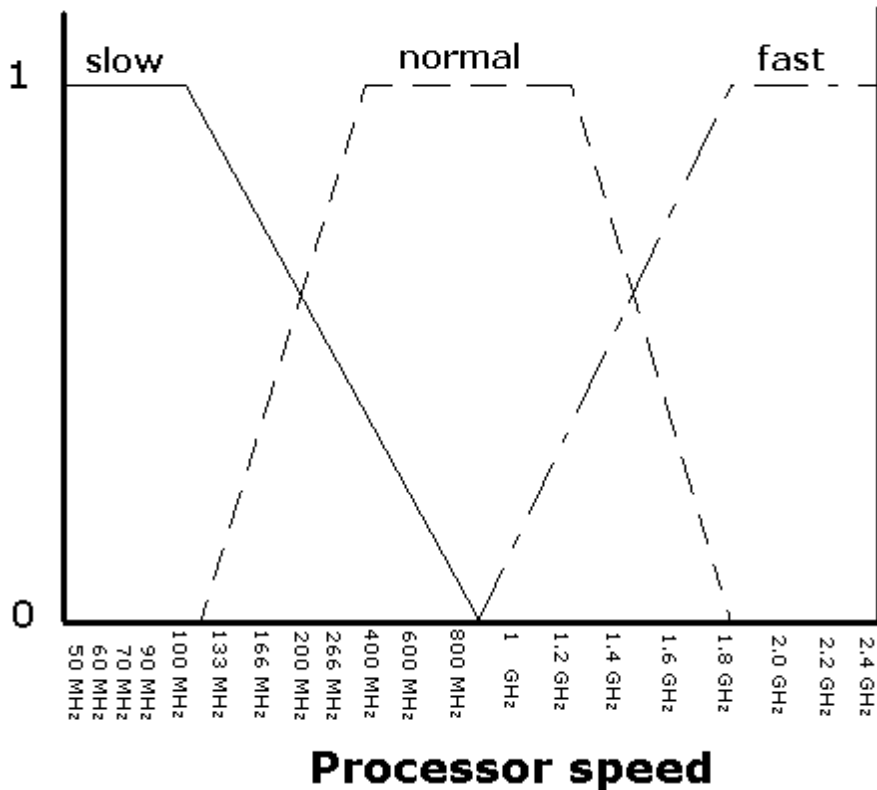


figure 6

There are now several new processors released with speeds that goes up to 2.4 GHz. figure 6 shows the way how the new speeds are inserted.

The example shows the adaptation of the function according to the new information. Note that these functions could also be dynamically generated. For example, a user provides a new processor to the ontology (with its properties like absolute processor speed). Next, the user also indicates that it is a fast processor. With this knowledge, the function would indicate processors with speeds close to its absolute speed as 'fast'.

Although the proposed way works well in our examples, there are relative statements to objects which are neither scalable nor predictable. When we look, for example, at fashion, the colour 'pink' is 'nice' in February 2001, but 'awful' in March of the same year. If then a user would ask for a 'nice' pullover in March, and the system returns a pink one, he/she never consults the ontology again. The only way to solve this problem is to explicit state to the ontology that pink cloths are nice cloths to the ontology in February and that they are awful in March (to overcome contradictive situation, we could introduce ratings to statements, which devaluate in time. This option will be discussed in section 3.5).

### 3.4 Deduction by analogy

Innovation and creativity are traditionally considered to be among the highest traits in human intelligence. They abound in all sorts of human reasoning, ranging from everyday activities like cooking to artistic activities like painting to complex forms of activities like engineering design. Analogy plays an important role in reasoning underlying innovation and creativity [S. Bhatta 1992]. Analogical reasoning is the process of retrieving knowledge from a familiar situation (source analogue) that is similar to the current situation (target) and transferring that knowledge to solve the current problem.

We identify the following crucial elements of an analogy:

- a source situation S
- a target situation T
- a good (possibly partial, but still good) match between S and T.
- some facts/properties which are known about S, but are unknown about T, and which can be inferred about T on the basis of the match.

The ability to make analogies between distant situations or domains appears to be crucial for innovation and creativity. This subchapter proposes some methods, which make use of analogical reasoning. We show that this reasoning process can be used to let an ontology evolve.

[Owen S. (1990), *Analogy for Automated Reasoning*, Academic Press] gives some characteristics of analogical reasoning:

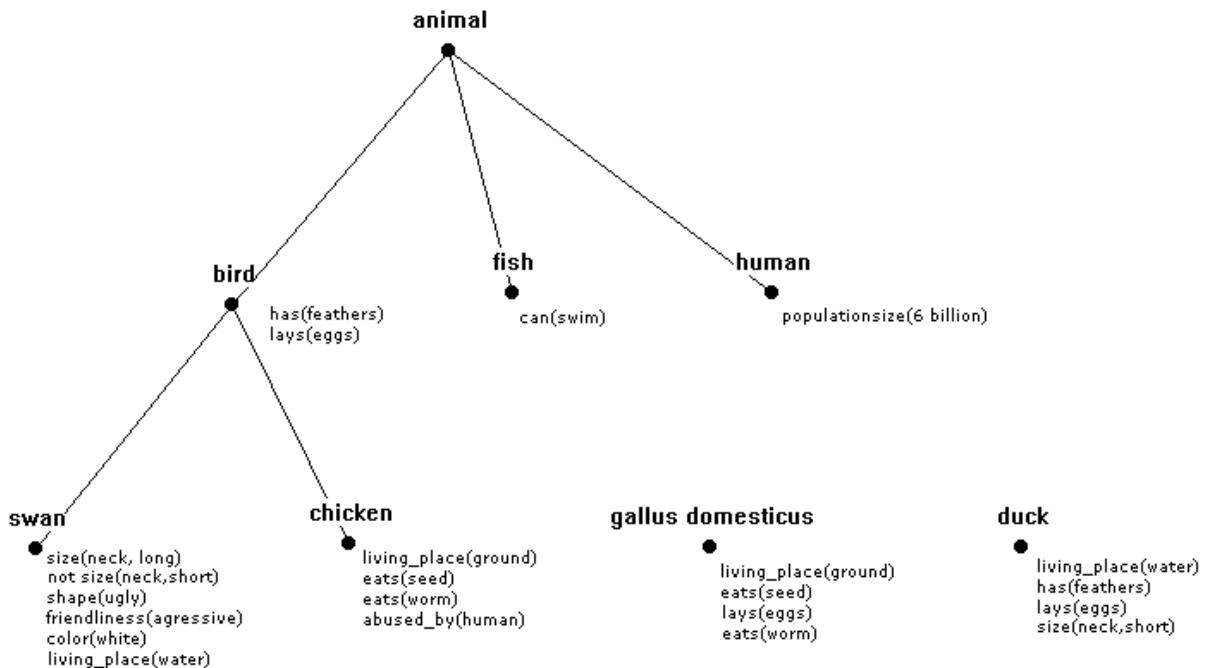
- There seems to be no general, formal rule that tells us what constitutes a reliable analogy match. It is in the nature of analogy that it is partly empirical - we only know for sure if a particular match is good or not after the application procedure has used it in trying to solve the target problem.
- Almost all analogy researchers have adopted the heuristic view, though some more explicit than others. Therefore analogy-matching algorithms use **heuristic criteria** to guide them in searching for good analogies. The heuristics that a particular matcher uses determine its (or rather its designer's) idea of what constitutes a promising analogy.
- Since analogy is heuristic, it is likely that different criteria will be successful in different situations. It is therefore desirable for analogy matchers to be flexible with respect to the criteria that they use: That is, the specific criteria used should not be engrained in the architecture of the matcher.

The big question is how we can use analogical reasoning in combination with ontologies. To answer this question, we give two scenarios where analogical reasoning is used in an ontology environment.

#### Scenario 1

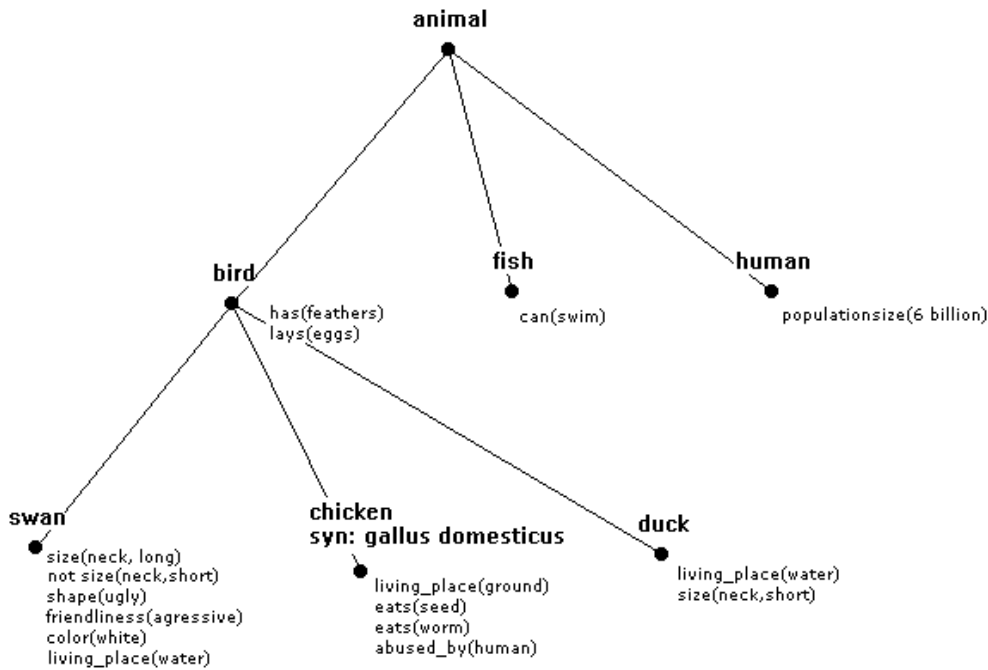
Assume an ontology that is organised in a 'isa' network, where the properties of the object are a description of the object itself. When a user adds an object with its properties to the ontology, an analogous reasoning engine looks for objects with similar properties. If the engine finds an object where the most properties are the same like the user's object, it could ask if the found object is a synonym of the user's object. It could also be that the user didn't provide the superclass of the object, and that the engine proposes the superclass of the found object to the user's object. We clarify this all with an example:

Given the next small ontology where the 'gallus domesticus' and the 'duck' are not processed by the analogy engine:



**figure 7**

The system can infer (with possible confirmation of the user), based on heuristics, that the 'gallus domesticus' has almost the same properties as chicken (inclusive the inherited properties from bird). Therefore the system merges the two objects, and sees them as synonyms. The 'duck' case seems to be a bird, because it has the same properties, but it is not a synonym for bird, because the property 'living\_place(water)' is not a common property of birds. It is also not a synonym for swan because the size of the neck is short (that a short neck is the opposite of a long neck should be made explicit to the system, which could be done with description logic and will be discussed in the 3.5). After reasoning we get:



**figure 8**

It is obvious that the heuristics have to be 'calibrated' by experiments, to assure that the system infers more sense than nonsense. We classify this scenario with the gallus domesticus as analogous reasoning because it contains the crucial elements described at the beginning of this chapter. The target is the gallus domesticus, the rest of the objects are the source. There is a good match between gallus domesticus and chicken and the properties of the chicken can be used to identify the gallus domesticus as a chicken. This example shows that analogous reasoning techniques can be used to classify objects, therefore it has a close relation with classification reasoning.

### Scenario 2

A more advanced scenario with analogous reasoning is when the interpretation of queries needs more creativity from the system.

Image the next ontology:

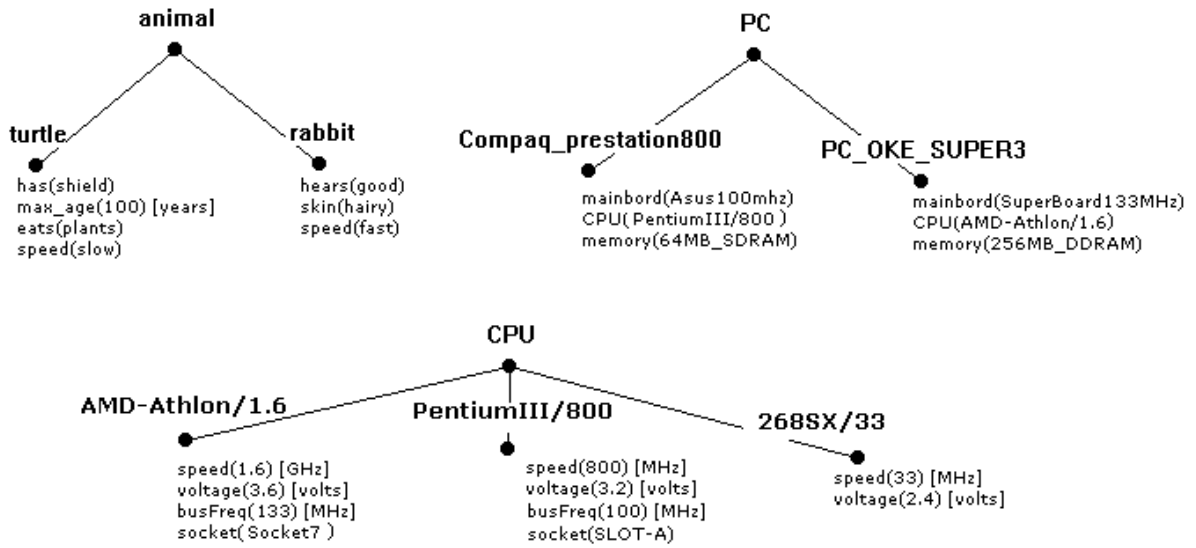


figure 9

The system gets the following query from a user with the following meaning 'give me a computer with rabbit speed'. Although it perhaps is a silly question, because computers and rabbits do not have much in common, a human computer salesman probably knows what a customer means with that question. What the user really means is that he/she wants a computer with fast processor. The user uses the analogy between the number of calculations of the processor and the transported distance of the rabbit in a period of time. A relative indicator like 'fast' has the same meaning in both cases: a relative high speed against the sort members. When the system wants to answer the question of the user it first has to look for information about the speed of rabbits. In this case it is simple because relative information is provided, namely 'fast', otherwise it first had to determine its relative speed (on the same way as described in section 3.3). Next it has to look for 'speed' information for computers. There is no direct information about 'speed' in the computer ontology, therefore it has to look at speed information at the objects, like 'PentiumIII/800', which occur in the properties from the computer. The specific CPU's all have a speed relation, and their relative speed could be calculated on the same way as in section 3.3. Now that the system found the relative speed of processors, it takes those processors that have the same relative predicates as the relative predicate 'fast' of the rabbit. The only thing what now has to be done is to return a computer that contains the found processor.

The two scenarios gave an overview how analogical reasoning could be used to infer information. However, it is not easy to determine which features are important for an object and which are not. A solution could be that the system learns by trial and error. A wrong reasoning step means that it took properties of the source object, which are not important for the target. As a result, these properties are not taken so easily again for the target object.

## 3.5 Ratings

The former four evolving methods all infer implicit knowledge by using a reasoning engine to extract new knowledge. Therefore these methods are used to *extend* information. A new method of making ontologies evolve is by the use of a natural selection mechanism to weed out erroneous information. Assigning a rating to a statement can be viewed as a sort of *meta information*, because they say something *about* the data in the ontology. Rating methods are frequently used in the Internet domain to rank various sort of information like music, web-sites, software, etc. Different techniques can be used to rate information [Smid-Isler 1998]. In our definition ratings are numeric values given to a user's statement as a measure of the quality of this specific statement. In other words: a way to express the influence of this statement in the ontology. There are different possibilities of giving ratings. First it is important to decide *who* gives these ratings. If the user itself gives this rating, the responsibility of the quality of the network also lies in it's hands. The other possibility is that the system itself indicates this rating to the specific statement.

It is also important *what* we want to express in this rating. It could, for example, be an indication of the *certainty* from the user (or the system) of his/her statement. Another possibility is an indication of the *importance* of the statement to the ontology. In other words: the influence it has to other statements in ways of priorities. Another way to look at ratings is that they could be an indication of the *reliability* about the statement given by the user. It is (for obvious reasons) preferable that the system, or another person than the person itself, provides this 'trust value' to the statement.

The last option, about reliability, is a way we want to make an ontology dynamic. We first explain this method with an example:

User X provides a system with a statement about an object O. The system has no further reliability information about statements from user X about object O, therefore the system assigns to this statement a low initial reliability rating. After this some other users give exactly the same statements about object O. Now the system has more reason to believe that user X knows more about object O than initially expected. When user X, again makes another statement about object O, it gets a higher reliability rating than the initial value.

In the example we see that the rating of a specific statement can be different when new information is added into the ontology. The rating of the statement increases when other users agree with it, by providing the same information. The rating of the statement relatively decreases when other users say the opposite.

Now we look in more detail at some aspects of ratings.

### 3.5.1 Ratings and semantic distance

We already mentioned that ratings are an indication of the reliability of statements. When a user provides new information, the system looks at earlier statements of that user. If those earlier statements exist, it looks at the *semantic distance* of the provided statement to the other statements. If, for example, a previous statement from the user has a close semantic distance, it will influence the new statement's rating more than if it had a far semantic distance. The reason for introducing the semantic distance is that it models the real-world situation from knowledge about objects. For example, an expert on gorillas knows much about gorillas. Probably the knowledge about orang-utans is also more than the average user, because they are both monkeys. However, his/her knowledge about gorillas is probably no indication of the knowledge about e.g. crocodiles. Now we will give a closer look at *what* semantic distance means, and *how* to calculate it. Semantic distance is an indication of the *degree of semantic similarity*, or, more generally, *relatedness*, between two lexically expressed concepts [Budanitsky, 1999a]. Measures of similarity or relatedness are used in such applications as word sense disambiguation, determining discourse structure, text summarisation and annotation, information extraction etc. Research on the topic in computation linguistics has emphasised the perspective of *semantic relatedness* of two lexemes in a lexical resource, or its inverse, *semantic distance*. It's important to note that semantic relatedness is a more general concept than similarity. Similar entities are usually assumed to be related by virtue of their likeness (*bank-trustcompany*), but dissimilar entities may also be semantically related by lexicate relationships as meronymy (*car-wheel*) and antonymy (*hot-cold*), or just by any kind of functional relationship or frequent association (*pencil-paper*, *penguin-Antartica*). Budanitsky & Hirst ([Budanitsky et al. 1999]) present an extensive survey and classification of measures of semantic relatedness. One category of such measures has been spurred by the advent of networks such as MeSH<sup>9</sup> and WordNet<sup>10</sup>. These vary from simple edge counting to attempts to factor in peculiarities of the network structure by considering link direction, relative depth, and density. In our first prototype we only use the first option, namely edge-counting. We do this to keep it simple to implement, later we could make use of more sophisticated techniques.

### 3.5.2 Calculation of a user-specific reliability rating for a statement

When a user provides the ontology with a statement, the system calculates a reliability rating for this statement. The system looks at (all) the previous statements of the user with their attached ratings. Previous statements, which have a close semantic distance, influence the new statement's rating more than those with a higher distance. To calculate the new rating we use the following formula:

$$\frac{\sum (\text{Distance}^{-1} * \text{Rating})}{1 + \sum \text{Distance}^{-1}}$$

<sup>9</sup> <http://www.nlm.nih.gov/mesh/>

<sup>10</sup> <http://www.cogsci.princeton.edu/~wn/>

The intuition behind this formula is the normalised weighted average of all the existing ratings of the user, with the inverse distance used as weights. With distance we mean the number of edges between two statement plus one<sup>11</sup>. A depth analysis about the implications of this formula is going beyond the boundaries of this work, and hopefully will be discussed in more detail in the. However, we should ask ourselves some questions to tackle unforeseen problems.

*Is the formula commutative?*

With commutative we mean, if the user adds the same statements but in a different order, the final outcome is different. We already mentioned that identical statements from **other** users which are added after the original statement, increases the rating. The system prevents the situation that a user can state the same statement more than once, by giving a warning that he/she already gave that information. By this way, we prevent that a user alters his/her own ratings. When a user provides different statements, whereby no other users provide statements in the meanwhile, the order of the additions shouldn't influence the final outcome. Therefore the formula should be commutative. For now, our formula doesn't meet this requirement because a lack of time. Further research should solve this problem.

*Why a linear decrease of weights by increasing distance?*

For now, the weight is the inverse of the distance (number of edges plus one). This is the simplest option and therefore easy to implement into the system. Again, further theoretical research is needed.

*Why counting 1 upon the sum of weights?*

First assume the following formula, where this addition of 1 is skipped.

$$\frac{\sum (\text{Distance}^{-1} * \text{Rating})}{\sum \text{Distance}^{-1}}$$

When every node from a ontology contains a statement from a user, accompanied by the rating of the statement, the formula indeed calculates a normalised rating where the weights indicate the influence of the individual ratings. A problem arises when only one statement is provided by a user that adds a new statement where the rating has to be calculated. The distance has then no influence on the calculation. For example, user X provided a statement S1. Because it is the first statement from the user to the system it get an initial rating R1. After this, he adds a new statement S2. Now assume that the distance between S1 and S2 is four. To calculate the rating R2 of statement S2 we fill in the formula and get:

---

<sup>11</sup> To prevent division by zero when calculating the weight factor

$$R2 : \frac{1/4 * R1}{1/4} = R1$$

Of course this is not what we want, because the semantic distance is not expressed in the result. We can solve this problem by giving the system the instruction to calculate the rating in case of only one known statement, not by the formula but by dividing the rating by the distance. The system calculates R2 simply by dividing R1 by 4 (the distance). This is not only a dirty hack, but also isn't a solution to another problem. We show the problem in the next example:

We introduce two small ontologies :

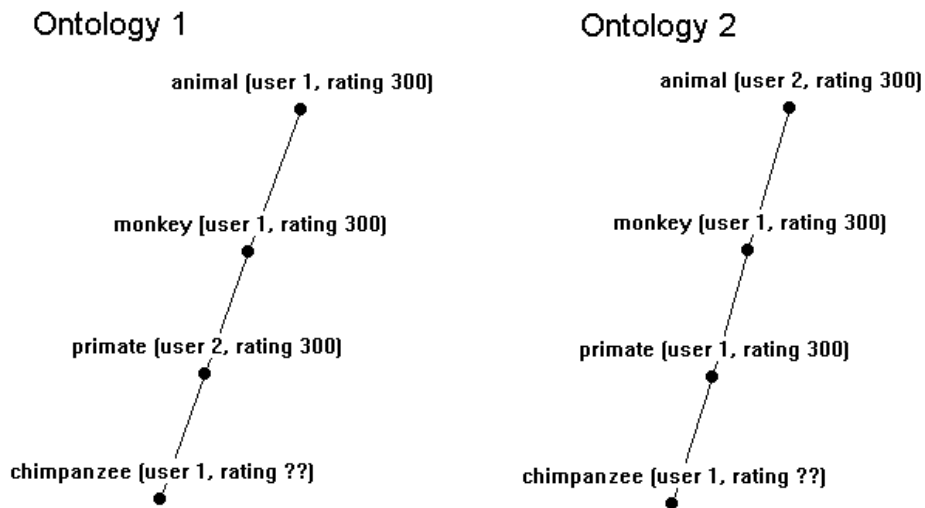


figure 10

The calculation of the rating from user 1 about the chimpanzee in both ontologies leads to the following answers:

$$\text{Ontology 1: } \frac{1/4*300 + 1/4*300}{1/3 + 1/4} = 300$$

$$\text{Ontology 2: } \frac{1/2*300 + 1/3*300}{1/2 + 1/3} = 300$$

Both results are the same, while the semantic distance of the statements in ontology 1 is further than from ontology 2. Also without the second ontology we could see the problem,

because the calculated rating from ontology 1 should be lower than 300 due to a semantic distance higher than one.

The problem is solved when we use the initial proposal. If we use it on the both ontologies from figure 10 we get:

$$\text{Ontology 1: } \frac{1/3 * 300 + 1/4 * 300}{1 + 1/3 + 1/4} \approx 95$$

$$\text{Ontology 1: } \frac{1/2 * 300 + 1/3 * 300}{1 + 1/2 + 1/3} \approx 136$$

The results now reflect the influence of the semantic distance.

Also the first problem with the single statement is solved. For example, if the semantic distance between the new statement and the existing statement is 4 and the rating of the existing statement is 300, the calculation of the new rating is:

$$\frac{1/4 * 300}{1 + 1/4} = 60$$

The result shows the influence of the distance between the existing and the new statement.

Now that we discussed some important aspects of the formula, we close this subject with an example. Figure 11 shows an ontology, structured by 'isa' relations. When a user makes an 'isa' statement, the name of the user and the calculated rating are displayed on the connecting edge. Objects also can have properties, which are displayed beneath the object, accompanied with the name of the user and the calculated rating.

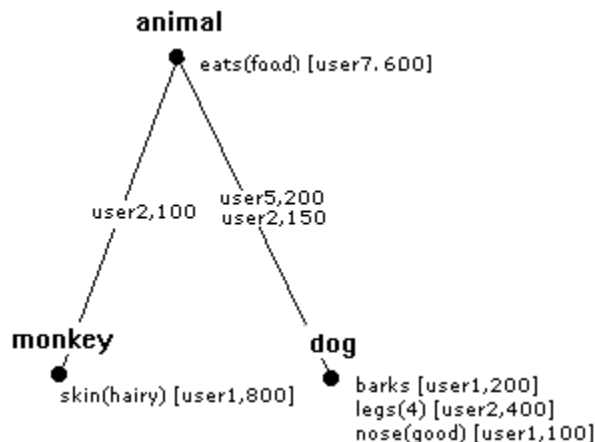


figure 11

Next, user 1 adds a new statement with content 'a dog has an high intelligence'. The system looks at all previous statements of user 1, with their semantical distance:

Statement	Rating	Semantical distance
Dog: barks	200	1
Dog: nose(good)	100	1
Dog: skin(hairy)	800	3

When we fill in the formula, we get:

$$\frac{1/1*200 + 1/1*100 + 1/3*800}{1 + 1/1 + 1/1 + 1/3} = 170$$

The ontology includes the new statement, with its ranking:

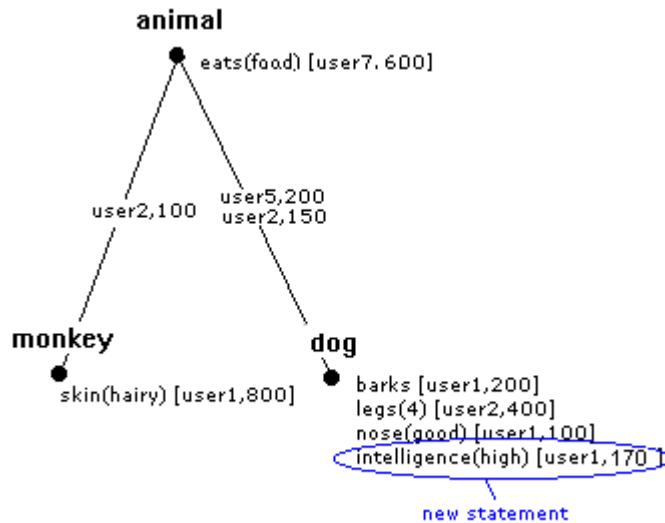


figure 12

### 3.5.3 Upgrading individual ratings

In 3.5.2 we have seen how the ratings for new statements are calculated out of the ratings from previous statements. In this subchapter we explain how the existing ratings are influenced by new information. When a person tells information you already know, it is reasonable to assume that this increases your belief in the known information. The same principle we can use to update ratings in the ontology. The simplest solution is to increase the rating from an existing statement, when the same statement is provided. We again clarify this with an example:

Assume again the following small ontology from figure 11. Now user 1 adds a new statement with content 'a dog has 4 legs'. We see in the ontology that user 2 also provided this information. The only thing the system has to do is to increase the credits for the statement of user 2 and add the new statement from user 1 as described in 3.5.2. We have to decide *how* to increase this rating. One possibility is to multiply it with a factor, but we can also add up a specific rating. In fact we could use any function to manipulate the rating, but investigating the possibilities goes beyond the scope of this paper. We see no reason to choose the simplest option, just adding up a certain number to the existing rating. In this example we add a number of 100 at the rating. Note that this is random chosen, experiments should calibrate this rating more precisely.

After the update the ontology looks like this:

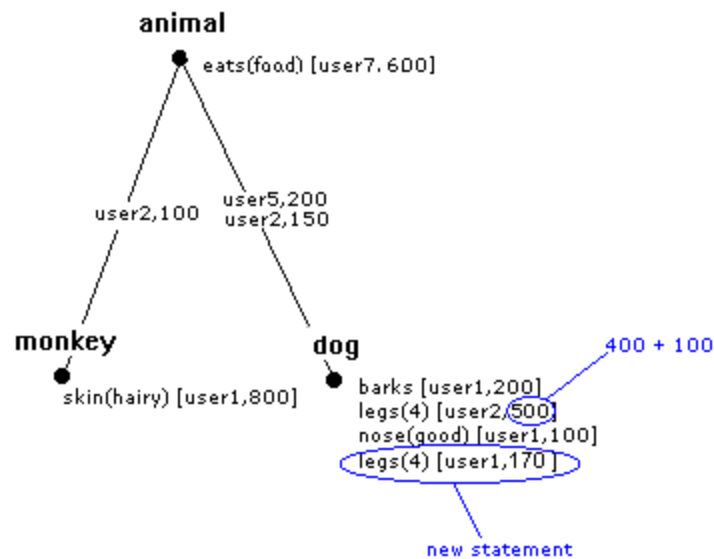


figure 13

What can we do more with these user-specific ratings assigned to each statement? In the next section, we introduce a method to aggregate user-specific ratings into a single rating to a statement. Next, the single rating is compared with the other statements to express the *relative validity*.

### 3.5.4 Aggregating user-specific ratings into a single statement rating

In 3.5.2 and 3.5.3 we introduced a way to assign a user-specific rating to a statement. We stated that a rating is an indication of the validity of the statement and is also an indication of the reliability of the user. We now exploit some methods to aggregate user-specific ratings from a statement into a single rating. In our situation, the aggregated rating is the

highest user-specific rating for the statement, because the updating function already upgrades the rating when another user provides the same statement. When this aggregated rating is compared with the other aggregated statements in the ontology, we call it the *relative aggregated rating* of a statement.

We now discuss some possibilities to calculate this relative aggregated rating:

#### 3.5.4.1 Calculating the global percentile rank of the aggregated rating

The percentile rank is the percent of statement ratings in the whole group of all aggregated statement ratings that scored *at or below* the statement of interest. The standard formula to calculate the percentile rank is: of an aggregated statement is:

$$Percentile = 100 \cdot \left( \frac{\frac{Worse}{Total} + \frac{(Total - Better)}{Total}}{2} \right) = 50 \cdot \left( \frac{Worse + Total - Better}{Total} \right)$$

In our context, *Worse* means the number of statements with a lower aggregated rating than the statement where we want to know the percentile rank. *Total* means the total of statements and *Better* the number of statements with a higher aggregated rating.

The next example explains the use of this method.

Please look at the next figure:

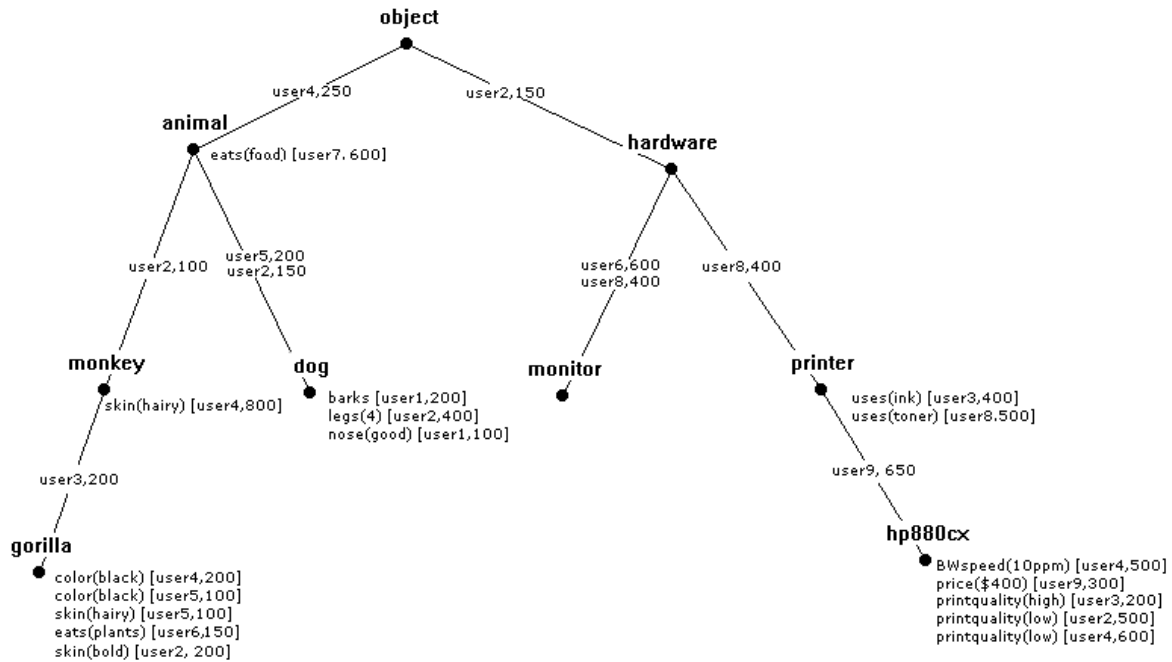


figure 14

As an example, we want to calculate the percentile rank of the statement ‘a printer uses ink’.

In the figure, some identical statements exist (provided by different users). In that case, we take the highest rating and skip the rest, which results in the following table:

An animal is an object	(user 4, rating 250)
An animal eats food	(user 7, rating 600)
A monkey is an animal	(user 2, rating 100)
A monkey has a hairy skin	(user 4, rating 800)
A gorilla is a monkey	(user 3, rating 200)
A gorilla has a black color	(user 4, rating 200)
A gorilla has a hairy skin	(user 5, rating 100)
A gorilla eats plants	(user 6, rating 150)
A gorilla has a bald skin	(user 2, rating 200)
A dog is an animal	(user 5, rating 200)
A dog barks	(user 1, rating 200)
A dog has 4 legs	(user 2, rating 400)
A dog has a good nose	(user 1, rating 100)
Hardware is an object	(user 2, rating 150)
A monitor is hardware	(user 6, rating 600)
A printer is hardware	(user 8, rating 400)
A printer uses ink	(user 3, rating 400)

A printer uses toner	(user 8, rating 500)
An hp880cx is a printer	(user 9, rating 650)
An hp880cx has a BWspeed of 10ppm	(user 4, rating 500)
An hp880cx has a price of \$400	(user 9, rating 300)
An hp880cx has an high printquality	(user 3, rating 200)
An hp880cx has a low printquality	(user 2, rating 500)

Table 1

Now that we have the aggregated ratings, we look which ones perform better, even or worse than our focused statement about the printer. We get the following results:

*Worse* = 13 (count instances with a rating below 400)  
*Better* = 7 (count instances with a rating above 400)  
*Total* = 23 (count all the instances)

When we fill in the formula we get:

$$percentile = 50 \times (13 + 23 - 7) / 23 \approx 63\%ile$$

So this means that the rating of this statement performed *as well as or better than* 63% of the overall rating of all the statements.

Although this global percentile is a good indication of the knowledge of the statement, it only works well in ontologies where the objects are close related, i.e. small semantic distance. In our example the ontology has information about 'animals' and 'printers', which are not close related. When a lot of information is provided to the ontology about 'printers', the possibility exists that ratings are relative higher than the ratings from information about 'animals'. The information about 'animals' could be perfectly correct, however the system would indicate that statements with a relative high local ranking have a low percentile rank. To overcome this shortcoming we introduce the next option.

#### 3.5.4.2 Calculating the local percentile rank of the aggregated rating

Another possibility to calculate the percentile rank of a statement is to look only at the direct local information of a specific statement. This overcomes the problem from the first option with looking at all statements in the ontology. In the next example we show how we calculate the local percentile rank:

We now want to know the percentile rank of the statement 'a gorilla has a black color'  
 We only have to look at the local statements about gorilla and get the aggregated rankings:

A gorilla is a monkey	(user 3, rating 200)
A gorilla has a black color	(user 4, rating 200)
A gorilla has a hairy skin	(user 5, rating 100)
A gorilla eats plants	(user 6, rating 150)
A gorilla has a bald skin	(user 2, rating 200)

Filling in the formula:

$$50 \times (2+5-0) / 7 = 50\%ile$$

(Note that the global percentile rank would be  $50 \times (6+23-12) / 23 = 37\%ile$ )

We indeed solved the problem with the global percentile rank calculation, but we now encounter a new problem. If we look at figure 4, we see that the statement 'a monkey has a hairy skin' has a very high reliability. We can see that a gorilla is a subclass from a monkey, therefore the properties of a monkey also are properties from the gorilla. To overcome this problem, we introduce the next option, which takes the semantic distance into account.

#### 3.5.4.3 Calculating a aggregated rating indication corrected by semantic distance

The problems of both global and local rank calculation are that they are too rigorous in the 'view-distance'. In global rank calculation, statements that have nothing to do with the asked statement have the same influence than statements that are semantically 'close' to it. In contrary, local rank calculation only views at statements that directly say something about the object in the asked statement, without looking at sub- or superclasses or even further. Due to this problems we introduce the semantic distance, as described in 3.2.1. When a statement has a 'far' semantic distance from the asked statement, it has less influence than the 'closer' one. There are several options to calculate the semantic distance in a network, and again we choose here the simplest version, namely counting the edges. We take the same formula as used for calculating the individual rating, corrected by semantic distance (see 3.2.2). The result from the formula is the mean overall rating from the network, corrected by the semantic distance (note that this overall rating can be different, when looking at another statement). After this we divide the focused rating by this overall rating to get the rating indicator. When this indicator is 1, the specific statement is as important as the mean value of the network, corrected by the semantic distance. If it is lower then the statements in that area have higher ratings, if, in contrary, the specific rating is higher, it is more important. To clarify this, let us look at an example:

Again we use figure 4 and table 1 as the example ontology. We want the rating indication from the statement: 'a gorilla has a bald skin'

To calculate the new rating we use the following formula:

$$\frac{\sum \text{Distance}^{-1} * \text{Rating}}{1 + \sum \text{Distance}^{-1}}$$

When we fill in the formula, with the information of table 1, we get:

$$\begin{array}{l}
 \text{(animal)} \quad 1/3*250 + 1/3*600 + \\
 \text{(monkey)} \quad 1/2*100 + 1/2*800 + \\
 \text{(gorilla)} \quad 1/1*200 + 1/1*200 + 1/1*100 + 1/1*150 + 1/1*200 + \\
 \text{(dog)} \quad 1/3*200 + 1/3*200 + 1/3*400 + 1/3*100 + \\
 \text{(hardware)} \quad 1/4*150 + \\
 \text{(monitor)} \quad 1/5*600 + \\
 \text{(printer)} \quad 1/5*400 + 1/5*400 + 1/5*500 + \\
 \text{(hp880cx)} \quad 1/6*650 + 1/6*500 + 1/6*300 + 1/6*200 + 1/6*500
 \end{array}$$


---


$$\begin{array}{l}
 1 + 2*(1/3) + 2*(1/2) + 5*(1/1) + 4*(1/3) + 1/4 + 1/5 + 3*(1/5) + 5*1/6 \\
 \approx 226
 \end{array}$$

Next we divide the specific rating by the overall rating:  $200/226 \approx 0.88$

A way to interpret the rating indication, is by providing a table with ranges like:

Rating indication of statement	Validity of statement
> 2	very high
between 1.2 and 2	high
between 0.8 and 1.2	medium
between 0.5 and 0.8	low
< 0.5	very low

Table 2

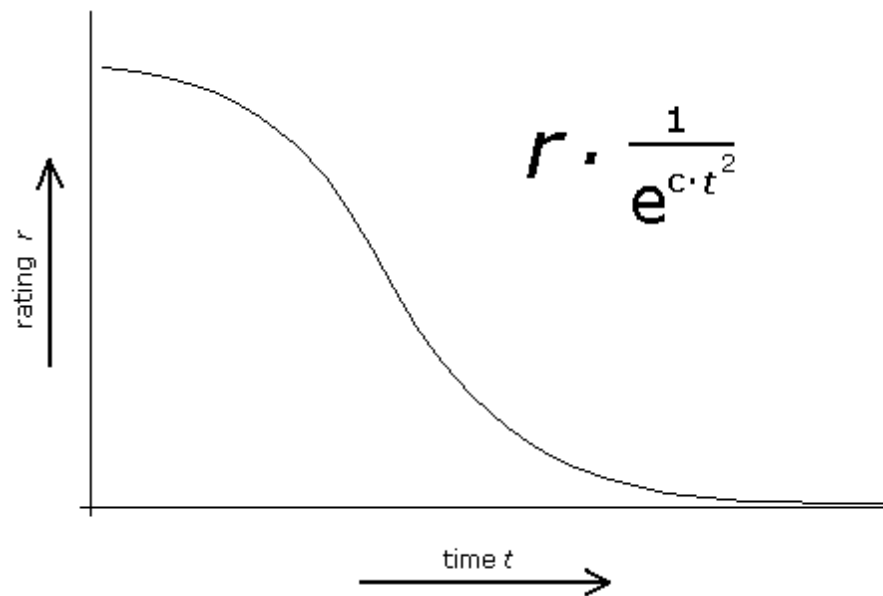
The table is based on intuition, experiments should calibrate the interpretations of the indications more precisely.

The conclusion of the three methods: The problems of both global and local rank calculation are that they are too rigorous in the 'view-distance'. Therefore we prefer the third method, which uses the semantic distance to determine the influence of the aggregated ratings to the focused statement. The closer the semantic distance, the more influence of the rating.

### 3.5.5 Devaluation function

Every time when a new statement is provided to the ontology, the system upgrades the ratings of identical statements, which are already known. This method causes an inflating

effect on the ratings. To prevent the inflate working of the overall increase of the individual ratings, we want to introduce a devaluation function. Such a function results in a decrease of the ratings in time. This function gives the network also an extra evolving aspect, because only the strong statements will survive, because they are strengthened now and then. We use here a standard devaluation function. The theoretic aspects of devaluation functions goes beyond the scope of this thesis, we only show how the function looks like. In our setting the function could look like the s-curve in figure 15.



**figure 15**

In this figure, the rating stays almost stable for a short period, next it decreases very fast and then after a long period it slowly goes to the under limit. We could set this limit on various values. When the limit is set on the initial rating (the rating which every unknown statement gets from an unknown user), statements always remain in the ontology. More efficient and more closely to the notion of evolving is to set this limit at a zero. If the rating nears this limit, the statement is removed out of the ontology.

### **3.6 Summary**

Several methods can be used to make an ontology evolving. The five methods proposed in this chapter should be seen as a novel way to handle ontologies. The way they are described in this paper should be seen as an introduction with some ideas to implement them. A complete theoretical basis should be provided in the future. The first method converts the information from an ontology into an internal format, where inference engine derives new information. Logicians would argue that the information is already implicit available in the ontology, thus the ontology contains nothing new. However, the ontology does not need the time and resources of a reasoning engine to express the implicit information. Therefore, the ontology is 'grown' after the reasoning process. The second

method makes use of heuristics to generalise or specialise information. When, for example, many subclasses of an object contain all the same property, this property is generalised to the object itself. How many subclasses are needed to ‘trigger’ the generalisation process, is determined by the heuristic. The third method combines relative information like ‘a rabbit has a *fast* speed’ with absolute information like ‘a rabbit has a maximum speed of 40 km/h’. The validity of relative information can be influenced by time. For example, ‘a Pentium 133MHz processor is fast’, was correct some years ago but now the most people wouldn’t agree anymore. The proposed method makes use of *fuzzy logic* to combine the two types of information. The fourth method uses analogous reasoning methods, to detect similarities between objects. Like the second method, this one uses heuristics to decide if an analogy exists between two objects. The last method deals with the validity of the different statements. Especially in heterogeneous environments, where different (anonymous) users provide information to the ontology, it is difficult to determine the reliability of the user which make the statements and the validity of the statement itself. The proposed method assigns ratings to statements, where these ratings can be viewed as a sort of *meta-information*. When a user makes a statement about an object, the system looks into the ontology for the ratings from other statements from that user. From the found ratings, it calculates the new rating. In the calculation procedure, the method calculates the *semantic distance* to determine the influence of the individual ratings to the new rating. How ‘further’ an object in an existing statement stands from the object in the new statement how higher the semantic distance. The semantic distance is therefore calculated out of the number of edges between the two statements. The reason to introduce the semantic distance is that an expert about an object has a lot of knowledge about the object itself, but also more knowledge about the objects close in the neighbourhood of the object than the average user. The further object X is removed from object Y, the higher the probability that the expert of object X has an average knowledge of object Y.

The way to upgrade ratings, is kept very simple. If a new statement is provided, the system looks into the ontology for equal statements and upgrades the accompanied ratings with a specific amount. To prevent the situation that a user can upgrade its own ratings, only other users than the user itself can upgrade a statement from a user.

We explored a way to aggregate user-specific ratings from a statement into a single rating. This single rating can be used to determine the validity of the statement itself, independent from the different users. The upgrading method already aggregates the single-user ratings, because it upgrades every rating from the identical statement. Therefore we only need the highest rating to express this single aggregated rating. We compare this aggregated rating with the ratings of other statements to express the relative validity of the focussed statement. We introduced three methods to calculate this relative validity. The first method divides the single aggregated rating from the focussed statement by the aggregated ratings from all the other statements. The drawback of this method is that the ontology can contain sorts of information which have nothing to do with each other and in this method the rating of a unrelated statement can influence the focused rating. To overcome this problem, we introduced a method that only looks at the statements in the same node. However, now the ‘sight’ is too limited: the ratings in the area of the node can be an indication of knowledge in the node itself, but are overseen with this method. Therefore we introduce a method which takes the semantic distance into account. The closer the semantic distance of a

statement to the focussed statement, the more influence of the rating to the focussed statement. A formula is introduced to calculate the overall rating of the network towards a focussed statement that takes the semantic distance into account. Next, the rating of the focussed statement is divided by the result of the formula. The result of this division is translated by a translation table into an understandable predicate like 'high validity' (which means that the focussed statement has a high validity). At the end of the section, we introduced a devaluation function. This function lowers the individual ratings in time. We do this to prevent the inflate working of the addition of ratings. This function also contributes to the evolving structure: only the strong statements will survive (because the identical statements regularly strengthen them). When the rating comes below a certain threshold, the statement will be deleted out of the ontology.

## 4 LARiSSA

In chapter 2 we discussed some different ontology languages which make use of reasoning tools. We showed that some of these tools only are capable for consistency checking and have a form of making implicit information explicit. There are no reasoning tools developed for ontologies that work with the other four evolving methods from chapter 3. In the introduction we showed the need for evolving ontologies therefore we start the development of a new experimental system called LARiSSA which stands for *Learning Analogous Reasoning in Simple Structured Areas*. The purpose of this system is implement the evolving methods and see what the problems are. With *structured areas* we mean ontologies that are organised in some way, e.g. an 'isa' network. With *simple* we indicate the complexity of the ontology, where simplicity now lies in the size of the experimental ontologies. The ontologies shouldn't be too large in an experimental environment, because we want to be able to trace the reasoning steps of the engine. With *analogous reasoning* we point at one of the evolving methods, from chapter 3. Because this is the most exotic, and probable the most difficult one to implement, we used it in the name, as a sort of end goal. With *learning* we indicate that the reasoner should also be able to evolve. The system should contain some *parameters*, which can be adjusted to tune the reasoner. In our first implementation, a human individual should experiment with the system to tune the parameters, but we see no reason that this can't be done by the system itself in the future. It is important to know that the methods in LARiSSA are easy to adapt to the three existing languages. The structure of the three languages makes it possible to add information, e.g. about ratings. Therefore, we can adapt/extend the reasoning engine with the reasoning part of the evolving methods.

### 4.1 The ontology environment

We can choose to implement our methods in an existing environment, or to build our own environment. Some advantages of choosing an existing environment instead building from scratch:

- We only have to work on the things we want to work at, namely the implementation of the reasoner. We don't have to bother about the fundamentals from the system itself, like development of the ontology language or the communication with the reasoner and the database etc.
- We can make use of the available tools for editing and debugging the ontology, which makes the work easier.
- Available documentation where different choices are explained, therefore we only have to refer to these and not to write them ourselves.
- More attention from the developers and other involved persons of the existing (and concurring) environments, thus more readers of this work.

But there are also some disadvantages

- In commercial environments, there is often no access to the source code. In that case it is impossible to see *why* some things work due to information hiding. However, in an experimental environment like LARiSSA it is sometimes very important to know these things.
- When building the system from scratch, it is completely adjusted to the specific methods, which makes it probably more efficient (faster reasoning). There is no ballast of things we don't need, thus the software is smaller, therefore faster to compile and easier to spread (smaller download size).
- When writing the complete environment from scratch, the developer knows every detail of the software, which could be very useful when problems arise. When others make the core software, one is also dependent on them, which could lead to communication problems (long response times, language difficulties etc.).
- When some parts of the software (e.g. the information database), runs somewhere else and the developer doesn't have direct access, he/she is dependent of others when, for example, the program should be rebooted.

Due to time constraints for completing this thesis, it is probably better to use an existing tool where we can completely focus on the development of the evolving methods. When we look at the environments from chapter 2, Oil with the FaCT reasoner and On2broker are the only candidates for building LARiSSA as an alternative for writing from scratch. SHOE is not an option because it doesn't provide a satisfying reasoner. Unfortunately there were some problems with FaCT and On2broker. First, the experience with the FaCT reasoner is that it has much problems with *instances*, therefore the reasoning process takes a lot of time, before it completes. We have tried it with a medium size ontology (40 classes and 50 instances) and the reasoner took about 35 minutes (!) of time before it completed. After reporting it to the developers of FaCT, they made some adaptations and the reasoning time could be reduced to 10 minutes. This is still much too long, therefore FaCT is not a good option for now. On2broker was the second option. Because it is a commercial product the licence stuff had to be organised. Due to some problems at the software side and at my system requirements, this took about five weeks. Meanwhile I started my own implementation, just in case. In those weeks my own system evolved rapidly, and On2broker still had some problems, so I made the decision to continue my own work. This decision doesn't have a big impact, as it perhaps looks like. Of course we miss the advantages of an existing environment, but the experience, made by the scratch work is

very useful when we want to implement the evolving method in existing environments. More information about the implementation is described in chapter 5.

## 4.2 Structure of the ontology

In this section we look how the ontology should be organised to make the evolving methods possible. It is important to note that the ontology language itself is of secondary importance, the main focus of this paper is about evolving methods. However, we should make some decisions about the structure of the ontology to make the methods possible.

### Objects: identify by relation

Our ontology is organised in objects. Every object is unique, thus every object should be identifiable. It is not necessary that all objects have different names, they could also be identifiable by their place in the structure. For example, there could be two objects with the name 'mouse', where the superclasses 'hardware' and 'animal' discriminate it. This solution not only is a model for homonyms in the real world, it is also useful in analogous reasoning methods (namely, it can be a hint for identical properties of both objects, like the animal mouse object has some identical properties as the computer mouse object).

### Object hierarchy

The objects in the ontology should be organised in a hierarchical manner, because objects should be able to inherit properties from other objects that are from a more general class. For example the *generalisation/specialisation* methods need this ontology structure, because they retract information from the object and place it at the parent of the object. For this reason we can't suffice with a semantic network without any hierarchy.

### Multiple inheritance

Ontologies like Oil have the possibility of multiple inheritance. This means that a child can inherit properties from more than one parent. This is in contrary to the OO model that only allows one superclass. We choose to use the possibility of more than one parent because it is a better description of the real world. For example we want to have the option to express 'Nixon is a Quaker' but also 'Nixon is a republican'. A problem with multiple inheritance is that contradictions may arise in the object structure, like 'a Quaker is a pacifist' and 'a Republican is not a pacifist', but also in the object's properties. We avoid this problem with the rating method where every statement has a relative validity.

We already mentioned that the names of the objects are not the discriminating factor between them, but its relation with the superclass. However, this choice can lead to the following situation:

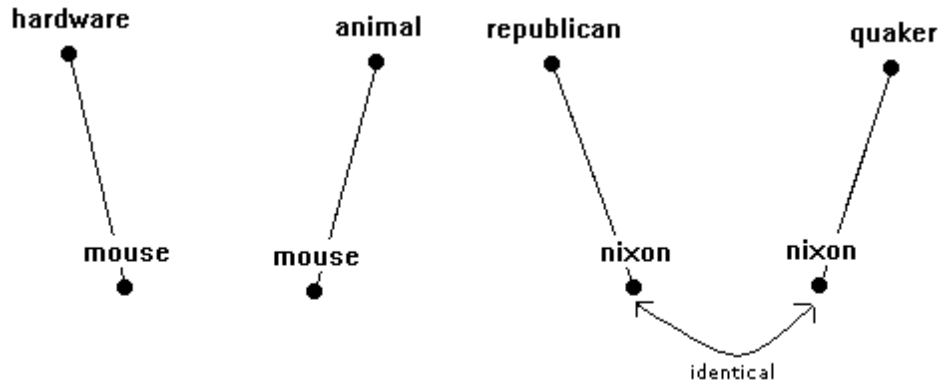


figure 16

The figure shows that the two mice are really two different objects, but the two 'nixons' are the same object. We can solve this problem when we make this information explicit to the ontology system, e.g. `identical((nixon, republican), (nixon, quaker))`.

Unique pairs

We restrict the ontology by allowing only *unique* parent-child pairs. If a relation is not unique, the child couldn't be identified. We illustrate the problem that arises when the relation is not unique:

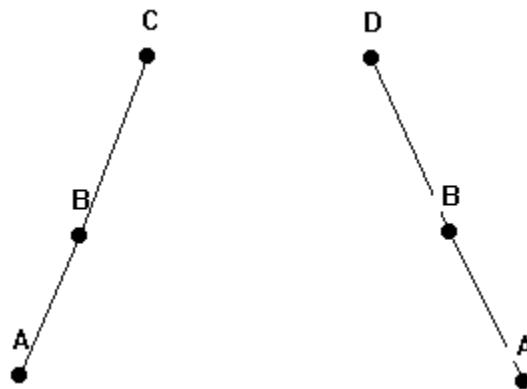


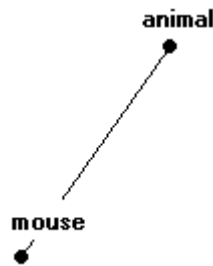
figure 17

We see in figure 17 that there are two equal 'isa' relations, namely that an 'A' is a 'B'. When we talk about 'A', which one do we mean? We only could identify it when we knew the 'super-super-class' of A, which is 'C' or 'D'. But this option is also a temporary one, because

theoretical it is also possible that there could be two identical triples (e.g. a--b--c--d1 and a--b--c--d2) etc. In other words, we only moved the problem but it still is there. In real world situations, equal relations occur very rare, and if they occur, we always can use another name for one of the children in both relations. Therefore we restrict our ontology for allowing no identical relations in the ontology.

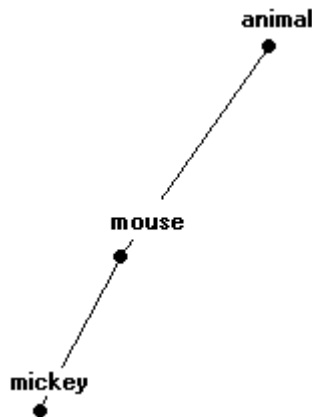
The next example shows the way we can handle non-unique name situations.

First, the following information is given to the ontology: 'A mouse is an animal', which leads to the following structure (fig 18)



**figure 18**

Next, the system gets the following statement: 'Mickey is a mouse' (fig. 19)



**figure 19**

After this, the system gets the statement 'a mouse is hardware' (fig. 20)

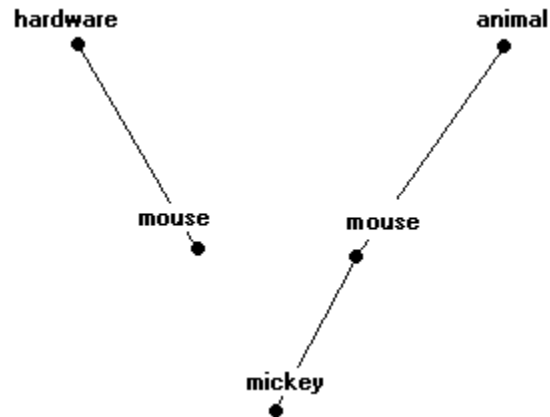


figure 20

The new information leads to the situation that the name 'mouse' has two meanings. The system now assumes that all the old subclasses of 'mouse' are also a subclass of the object animal, because the mouse is an animal. We legitimise this assumption with referring to the real world. Twenty years ago the word 'mouse' only had one meaning, so every human directly knows what you mean with mouse. When the first computer mouse was introduced, the old information in the minds of humans, all was about the animal mouse. From that moment, it first should be clear about which type of mouse we are talking. When a user now provides the statement 'clicky is a mouse', it is not clear which mouse he/she means. The user should indicate it by explicitly state it, e.g. 'clicky is a (animal)mouse'

### Relative and absolute properties

Objects have properties like size, color, number of legs etc. These properties all have a 'has' relation with the object like 'a mouse has a long tail'. We need to incorporate these properties into our ontology structure because they are needed in the most evolving methods. Method 3.3, about relative and absolute statements needs the distinction between relative and absolute properties. To relate these two types of properties, functions like in figure 5 should be incorporated into the system.

### Quantifiable and non-quantifiable relative properties

Relative properties can be divided into quantifiable and non-quantifiable ones. Quantifiable properties are like 'that processor has a *fast speed*' or 'that shirt has an *intense color*'. Non-quantifiable ones like 'that processor has a *good speed*' or 'that shirt has a *stupid color*'. We make this distinction because it helps the analogous reasoner to know that this property adapts the function like figure 5. Only quantifiable relative properties can be added to such a function, because quantities can be scaled on the absolute scale of the function. The non-quantifiable properties still can be used for explicit information retrieval like 'Kings\_X is a *good band*' where a user asks 'give me a *good band*'. More advanced

reasoners can detect absolute and quantifiable properties which are responsible for the non-quantifiable statement.

## Functions

Objects can have functions. Almost every function has also a direct object, for example 'a human breaths air', where human is the object, breaths the function and air the direct object. Sometimes it is difficult to find the direct object, especially in real world situations where implicit information is so obvious that we never think about it and therefore leave it in sentences. For example 'A human thinks' where the direct object is *information*, or 'a sun shines' where the direct object is 'radiation'. When we have a sentence like 'the queenspeech is annoying', it can be translated into a sentence where the direct object is involved, like 'the queenspeech annoys ronny'. The direct object could be very useful in generalisation/specialisation and analogous reasoning methods. We explain this with a generalisation example:

Imagine the following statements:

*annoys(queenspeech,ronny)*  
*annoys(queenspeech,wim)*  
*annoys(queenspeech,alexander)*

We also have the following small ontology (fig 21)

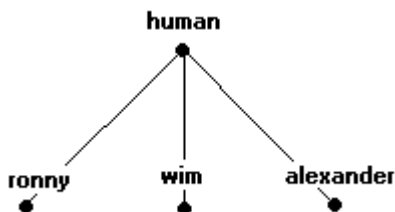


Figure 21

The generalisation process now generalises the statements to:

*annoys(queenspeech,human)*

The heuristic-based parameter to 'trigger' the generalisation step in this example is set on three identical properties. Experiments should calibrate this parameter more precisely.

## Synonyms, inverses

It would be useful, but not necessary for the evolving methods, to express some more *meta*-information about objects like their synonyms and inverses in the ontology. Especially in the case of analogous reasoning the detection of synonyms and inverses can be useful. When the analogous reasoner gets a description of an object by its properties, it could detect an object, which has exactly the same properties and proposes the synonym.

When the object has opposite properties from an object, it proposes the inverse. In a more advanced system, the reasoner could adapt his reasoning process, on the base of correcting user information (feedback). For example:

The analogous reasoner proposes a synonym for an object on the base of identical colors. Assume that colors are not important in that area, therefore the user corrects the system, indicating that it was a wrong proposal. After this correction the analogous reasoner decreases the importance of colors in that area.

### Instances vs. classes

We make no difference between classes and instances, which is an important decision, where some ontology languages differ in their approach. Some people argue that there is a difference between the name 'hp880cxi' and the instance 'hp880cxi' which stands on my table, but we don't see the importance of this in an ontology structure. In an ontology we can see the instances as leaves from the object tree. The advantage of ignoring the difference between them is that also users do not have to think about this issue.

### Quantifiers and negation

In this paper, we only use statements without quantifiers and negations, because it would take too much time to examine the theoretical and practical implications. The absence of quantifiers, e.g.  $\forall$  and  $\exists$ , is not dramatic. We assume for every statement about an object, that it holds for all subclasses of the object, itself included ( $\forall$ ). Every subclass of an object could be viewed as an instance ( $\exists$ ). We also skip the use of negation. We do this for the same reasons as with quantifiers. Negation from a statement implies that the statement itself is wrong. The decreasing rating function guarantees that 'wrong' statements do not survive. Therefore negation is not indispensable.

## **4.3 Requirements of the system**

Because we develop LARiSSA from scratch, it is possible to set some system requirements, which contribute to a comfortable environment to do experiments.

### Storing the data

To see how an ontology evolves in time, we should look how it develops a longer period. When the system is switched off for any reason, the information should not be lost. Therefore, information should be permanently stored. A database system is a way to do this efficiently.

### Simultaneous multiple user access

The rating method is based on the input of different users. It assigns a reliability indicator to each statement from a user. When we implement this rating method, the system should offer multiple user access at the same time. This has implications on the way queries and additions are handled. To store the data we again should use a database system, which can efficiently handle simultaneous input, in such a way that the internal structure of the database stays correct.

### Access everywhere

The system should be easy to access, so that many users can work with it, which is a prerequisite for good testing. The Internet is a good medium for this: the system is running somewhere on a server, who could be accessed with a browser anywhere in the world. Querying the system should be as easy as querying a search-engine.

### Fast reasoning

At last, it is important that the server responds within a reasonable amount of time. If not, users will not use the system, and experiments will fail.

## **4.4 Software & co.**

In this section, we take a brief look at the software decisions. In the requirements, we saw that Internet accesses and database storage are recommended. Java Servlets are software programs, which runs on a server and can be accessed by a browser. These servlets have the advantage that the software stays on the server, thus the clients don't have to download the software. Another advantage is that only the software on the server has direct access to the database, which is more secure. Java Servlets are written in Java code. Java is an easy and widely used language. Its object oriented way of programming makes it easy to plug in classes that could be useful. Java Servlets can be extended with standard drivers who are able to communicate with databases. MySQL and PostgreSQL are two database languages that are free available, well documented and widely used. The difference between the two is that MySQL is a relational database environment and that PostgreSQL is an object oriented one. In our environment, the latter fits better in our project, because then it is easier to get, for example, the subclasses from an object. A disadvantage is that MySQL is much simpler to install on a windows 98 machine. PostgreSQL needs Unix. I am working on a Windows 98 machine therefore it is easier to install MySQL for now. It is much to difficult and time consuming to build a theoretical sound object oriented language on top of a relational database, therefore we only implement the functions we need.

## 4.5 Internal representation of the data

As mentioned earlier, the data from the ontology is stored in a MySQL database. The internal structure of a database exists out of different tables. There are several ways to organise the information, which could all be even correct. However, we think the following is the most logical one.

### UserTable

First we need a table where individual user information could be presented. When a new user logs at the system, he/she provides a new password, which will be stored in the table. A unique id is provided to every user, which will function as a *primary key*.

Id	Username	Password
1	Ronny	idonttell
2	Clicky	meneither

### Object table

In this table, the different object names are represented with their superclasses. In case of non-unique names, the superclass of the object's superclass is provided. It is also possible to provide a synonym to an object. When a user wants to add more than one synonym, a new statement has to be made, which results in a new row in the table. The user identification and the rating for the statement represented. We also have a timestamp on each statement. This timestamp is necessary for the implementation of the automatic rate decrease. We also need a unique id for every statement.

ID	Name	Superclass	Super-Superclass	Synonym	UserId	rating	Time-stamp
1	Mouse	Animal			2	150	01-08-2001
2	Chicken	Animal			2	200	03-08-2001
3	Chicken			Gallus_domesticus	2	150	03-08-2001
4	Mouse	Hardware			3	150	05-08-2001
5	Mickey	Mouse	Animal		4	200	05-08-2001

### 'has' table

In this table, the three types of 'has' properties from an object are stored, namely the absolute, the quantifiable relative and the non-quantifiable relative one. When the object name is not unique, we also need the superclass of the object. The syntax of absolute statements should contain the value, the type and eventual the range (like 'max' and 'avg'). More information about the syntax is provided in the next section.

ID	Object	Superclass	Property	Quantifiable	Non-quantifiable	Absolute	Rating	Time-stamp
1	Mouse	Animal	Speed	Fast			150	01-08-2001
2	Chicken		Speed			30 [km/h [max]]	200	03-08-2001
3	Chicken		Character		Nice		150	03-08-2001
4	NecP220		ink-cardridge			2 nr	150	05-08-2001

### Function table

A function table expresses the different functions of objects. The object name, function and the name of the direct object are expressed in the table. Both the object name and the direct object name can be possible not unique, therefore the table offers the possibility to express their superclasses. It is useful to offer the possibility of expressing the synonym of a function name, like talk-speak or split-divide. When the system knows that two functions are synonyms they are threaten as the same in for example the generalisation method.

ID	Function	Object	Object superclass	Direct-object	Direct object superclass	Syn	User id	Rating	Time-stamp
1	Eat	herbivore		Plant			1	150	01-08-2001
2	Eat	chicken		Seed			2	200	03-08-2001
3	Scan	scanner	Hardware	Document			4	150	03-08-2001
4	Use	hp880cx		Ink		need	26	150	05-08-2001

Now that we know the internal representation of the data, we explain how we can manipulate and retrieve the information with an own query language

## 4.6 Query and manipulation language

In this section we describe the language which can be used to ask questions and add information to LARiSSA. We mentioned already the relative value of this language. We only use it for this system, it is not our purpose to invent a complete and sound new language to query and manipulate ontologies.

The first version of the language should be able to add and retrieve objects, functions and properties. We only show the syntax that is also implemented in the system. The reason for this is that the implementation of the system often alters syntax for various reasons. Adding and retrieving synonyms are not implemented yet, therefore we don't mention it in the language description.

#### 4.6.1 Adding and retrieving object information

In case of identical names for different objects, every object can be identified by its relation with the superclass. Therefore every object should be added with its superclass. 'Object' is the most general object, i.e. every other object than 'object' is a subclass of it. When the superclass of an object is unknown, the user should give 'object' as the superclass. The next example states that a gorilla is a monkey:

```
isa(gorilla,monkey);
```

When the superclass is unknown the superclass should be 'object'

```
isa(human,object);
```

When a superclass has more than one meaning, the superclass of the superclass should be provided:

```
isa(gorilla,(animal)monkey);
```

Every time when a user adds information, the system should indicate if the action was successful, in the case of error, the system should give the appropriate information like:

```
object 'monkey' has more than one superclass, please specify...
```

Retrieval of object information is the same syntax as adding an object, only a question mark in front of the sentence discriminates the two:

```
?isa(gorilla,monkey);
```

The system should respond, in the positive case, the ratings of the individual confirmations like:

```
isa(gorilla,monkey) by user 1, rating 350  
isa(gorilla,monkey) by user 3, rating 200
```

When nobody confirmed the question the system should respond like:

```
sorry, nobody confirmed your question...
```

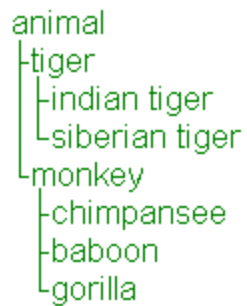
When the superclass has not a unique name, it should response on the same way as the adding example:

```
object 'monkey' has more than one superclass, please specify...
```

The language should offer a possibility to get a class hierarchy overview with a specific object as root. The statement:

```
?isa(_,animal)
```

gives the hierarchy of the subclasses of object 'animal':

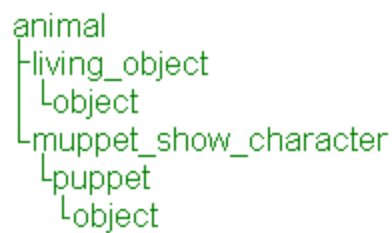


**figure 22**

When a user wants to know the superclass(es) of an animal, he/she writes:

```
?isa(animal,_);
```

and gets:



**figure 23**

If a user doesn't exactly remember the name of an object, he/she can use a wildcard '%':

```
?isa(hp880%,printer);
```

where LARiSSA answers:

```
isa(hp880cxi,printer) by user 2, rating 360
```

#### 4.6.2 Adding and retrieving property information

When adding a property to LARiSSA, the object and the property should be provided accompanied with its quantitative or non-quantitative relative value, or the absolute value.

The syntax to indicate the three different sort of values:

*In case of a quantitative relative value:*

has(object,property,pq:value);

e.g.: has(mouse,speed,pq:high);

*In case of a non-quantitative relative value:*

has(object,property,pnq:value) ;

e.g.: has(mouse,size,pnq:neat);

*In case of an absolute value*

has(object,property,pa:value [type]);    in case of an absolute value

e.g.: has(mouse,speed,pa:30[km/h[max]]);  
      has(mouse,age,pa:4[years[max]]);

When an object has not a unique name, the superclass should be provided, e.g.:

has((animal)mouse,age,pa:4[years[max]]);

The language should also provide means to make statements about parts of an object, e.g. the tail of a mouse has a long size. We have implemented this in the following syntax:

has(mouse.tail,size,pq:long);

We can see that parts are separated by a dot from the object. It is also possible to make statements about parts of parts, like 'the battery of the engine of a Ferrari has a weight of 1.4 kg'. The statement is expressed in the following way:

has(ferrari.engine.battery,weight,pa:1.4[kg]);

It is relevant to note that the calculation of the rating for statements is based on the object, and not on the eventual parts of the object.

The retrieval of property information has the same syntax like adding information, but only with a question mark in front of the statement. The same as with retrieval of objects, wildcards can be used.

```
?has(mouse,speed,%);
```

The system gives an answer like:

```
has(mouse,speed,pa:4[km/h[max]]); by user 3, rating 350  
has(mouse,speed,pq:fast); by user 2, rating 150  
has(mouse,speed,pnq:good); by user 3, rating 100
```

Another example with the use of wildcards:

```
?has((%)%.%.,%,pq:fast);
```

The above query gives the objects, parts inclusive, with 'animal' as superclass and 'fast' as quantitative value for a property. The answer of LARiSSA could be:

```
has((animal)mouse,speed,pq:fast); by user 3, rating 350  
has(ferrari,acceleration,pq:fast); by user 2, rating 250
```

### 4.6.3 Adding and retrieving functions

We show the syntax of the addition of a function with an example:

```
func(transport,car,human);  
func(drive,human,car);
```

The example shows that the first argument is the function name, the second argument is the object name and the third position is reserved for the name of the direct object. When the name of the object or the direct object has more than one meaning, the superclass should be given, e.g.:

```
func(eats,(animal)mouse,cheese);
```

The retrieval syntax is again the same as the adding syntax, only with a question mark in front, eventual with wildcards, e.g.:

```
?func(%,mouse,%);
```

## 4.7 Summary

In this chapter we introduced LARiSSA (Learning Analogous Reasoning in Simple Structured Areas). This system is an experimental tool to implement the evolving methods, as described in chapter 3. The name suggests that analogous reasoning is the only evolving method that should be implemented in the system, however this is not the case. Analogous reasoning can be viewed as the ‘end-goal’ of LARiSSA. It is also the last method that should be implemented, because it uses the other four methods. LARiSSA can be built in two different ways, namely in an existing environment with the usage of their tools (the three languages from chapter 2) or building the whole system from scratch. This chapter described the advantages and disadvantages of using an existing environment, and the choice felt on this option, mainly because of the time constraints to finish this project.

We chose On2broker [??] because the reasoning engine is easy to adapt, and is also very fast. The FaCT reasoner from the OIL language was too slow at that moment, and SHOE hasn’t a ready-to-use reasoning tool available. However, On2broker gave some installation problems and meanwhile we started to write the system from scratch. The progress with this system went very rapid, therefore we continued with it.

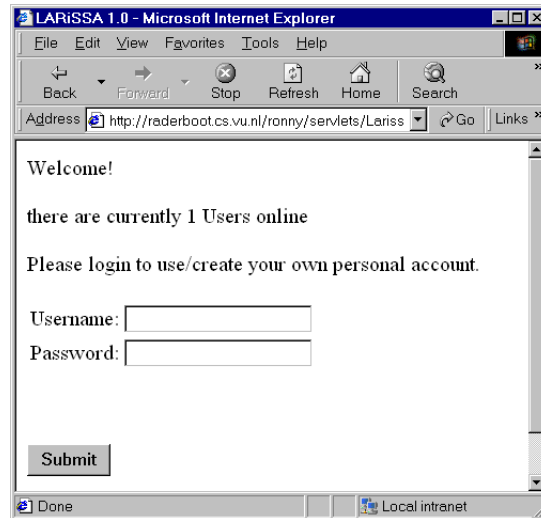
It is important to note that the language of the ontology is of secondary importance. The main focus lies on the evolving methods. However, these evolving methods make some requirements on the structure. The advantage of building this system from scratch is freedom to make fundamental choices on the structure of the ontology. Section 4.2 describes the properties of our ontology structure, where the fundamental choices are made in advantage of the evolving methods. Section 4.3 described the requirements of the system itself, like reasoning speed and accessibility. These requirements lead to software choices that are described in section 4.4. We saw that a Java Servlets in combination with a MySQL database, meet those requirements. The organisation of the ontology information in the database is described in section 4.5. There are several possibilities to organise the information, which can all be correct. We have chosen to discriminate four different kinds of information, namely: user information, class information, properties, and functions, which all have their own table. Each row in the class-table, property-table and function-table stands for a single statement from a user, where at least, each row contains the name of the object, the user id, the rating and a timestamp. The timestamp is needed for the devaluation function, to implement the decrease of ratings in time. Section 4.6 described the language to query and manipulate the ontology. The next chapter shows the implementation process of LARiSSA. It describes the implemented evolving methods, accompanied by some experiments to show its functionality.

## 5 Results

This chapter is a summary of the implementation details of LARiSSA. Some experiments are displayed to show the functionality of the system.

LARiSSA is implemented as a Java Servlet. Servlets are to servers what applets are to browsers. Servlet code can be downloaded into a running server to extend its behavior in order to provide new, or temporary, services to network clients [Voss 1997]. HTML forms, which can be viewed with a standard Internet browser (i.e. client), can provide the Servlet

with information based on standard form handling methods. There are two sides to form processing: specifying the input fields and widgets to be presented to the users in an HTML file, and processing the fields of data that have been filled in by the user once the submit button for the form has been pressed. An example form is showed in figure 24.



**Figure 24**

Once you press the submit button, the data in the form is passed as an argument to the Servlet specified in the form tag at the top of the HTML file:

```
<form action=http://raderboot.cs.vu.nl:8080/ronny/servlet/  
LarissaServlet method=POST>
```

When the Servlet receives the information from the client, it starts to process its information and responses to the user with an HTML document.

Now that we know how the communication between the browser and the system works, we take a closer look at the system itself. We have divided the program into three core segments:

### LarissaServlet

Responsible for:

- Initialising and maintaining the connection with the MySQL database
- Synchronising de different user requests
- Handling the different requests (HTTP get & HTTP post) and direct them to the LarissaManager.
- Handling the session with the client, concerning security issues and maintenance.

### LarissaManager

Responsible for:

- Generating the majority of HTML code, to communicate with the clients based on the incoming requests from LarissaServlet.
- forwarding reasoning requests to LarissaReasoner from LarissaServlet
- Handling reasoning results from LarissaReasoner, by adding HTML code and forwarding it to LarissaServlet.

### LarissaReasoner

Responsible for:

- Parsing and categorising the statements. This means:
  - Determine if the statement is a query or an addition
  - Determine the information type of the statement (object, property or function)
  - Check the syntax on errors. If anywhere in the parsing process a syntax error is detected, the system returns the error to LarissaManager, where it is translated into an HTML message.
  - When no errors did occur, the parsed information is send to the appropriate method, e.g. if the statement contains the addition of a function, this is send to the 'function\_addition' method.
- Provision of different methods to handle the parsed information. We describe it with an example:

Assume that the parser derives an addition of a function. First, the accompanied object and direct object are checked in the database on existence. If one of these don't exist, the system returns an error statement about the unknown object. If the object or direct object needs a superclass (in the case of more identical object names), also an appropriate error is returned. If both the objects exist, the rating method calculates the rating of the object from the user. Next, the information about the function and the rating are added to the database.

- Implementation of the evolving methods.

Due to time constraints, only the rating method and the generalisation method are implemented in the system. We chose for these two, because the rating method is needed to let the ontology handle statements from different users, and the generalisation was not difficult to implement. The calculation of the relative aggregated rating is not the proposed method which involves the semantic distance. We have implemented the more plain method which gives all the ratings the same influence. The calculation of the individual ratings are as described in chapter three. At last, the structure of the software allows an easy insertion of the other methods (specialisation, analogous reasoning etc.). The

implemented evolving methods (except calculation of the rating) are independent from the moment of addition of the statement. These methods (generalisation, calculation of the time devaluation of the ratings) can be executed at a time when the server isn't busy. However, we directly start these methods at the time when a statement is provided to the system, because the systems load is not very high at this moment. This has the advantage, that the system ontology always is up-to-date, which makes experimenting a lot easier.

The plain division of tasks makes it possible to easily trace the process of execution and makes eventual debugging an uncomplicated task.

The evolving methods in 'LarissaReasoner' contain several parameters to implement different experimental scenarios.

The rating method contains the following parameters:

**INITIAL\_RATING** : the height of the rating which is provided to an unknown user. This is also the minimal amount of credits, i.e. if the calculation of the rating results in a lower amount, the initial rating is returned. The rating can be any positive integer. Our default setting is 150.

**SUPERCLASS\_INFLUENCE**: the influence of the average rating of the superclass level to the rating of the current level. This is different then weight function as described in chapter three. However, it doesn't effect the definition of semantic weights. The value could be any double value between zero and one, our default value is 0.5.

**SUBCLASS\_INFLUENCE**: the same as with **SUPERCLASS\_INFLUENCE** but now for the subclasses. The default value is also 0.5.

**MAX\_LEVEL**: the 'depth' of involving the super- and subclasses into the rating calculation. Just the nodes till depth 'max\_level' are involved into the calculation. This is done merely for efficiency reasons, because very 'far' nodes almost have no influence on the rating and costly reasoning time is saved. The value can be every positive integer from one. A lower level means a more local view. The default value is set on 4.

**ISA\_INFLUENCE**: the influence of the 'isa' type of statement into the rating calculation. We extended the way of the rating calculation by providing the possibility to determine the influence of the different sorts of statements. Normally spoken all the three different types of statements (superclass definitions, functions and properties) have the same influence. Now it is possible to indicate the influence of the type of the statement on the average calculation of the node. The value can is a double between zero and one. The higher the value, the higher the influence. Default is set on 1.

FUNC\_INFLUENCE: idem as ISA\_INFLUENCE, but now for functions. Default is set on 1.

HAS\_INFLUENCE: idem as ISA\_INFLUENCE, but now for functions. Default is set on 1.

ADD\_CREDITS: the amount of increase of an identical statement rating. The value can be any positive integer. Default is set on 200.

DEVALUATION\_CONSTANT: the constant 'c' from figure 15. The smaller the value, the longer the devaluation period. It can be any positive double between zero and one, but only small values make sense, e.g. a constant of 0.001 means a devaluation of the rating by 50% in 25 days. This is also the default setting.

REMOVE\_RATING\_TRESHOLD: the minimum rating for a statement to 'survive'. If the rating becomes below this value, the statement will be deleted from the ontology. To give an indication: when the DEVALUATION\_CONSTANT is 0.001, a statement with an initial rating (150), will be vanished after 60 days. The value can be any positive integer. The default is set on 10.

The generalisation method contains the following parameters:

MIN\_GENERALISATION\_RATIO: the ratio between the number of identical properties or functions and the total number of objects in the same level. For example: 14/16 means that in 14 of the 16 objects an identical property or function exists. When the ratio is above this minimum and the total number is above the MIN\_GENERALISATION\_NUMBER, the generalisation is executed. The value can be every double between zero and one. Default is set on 0.9.

MIN\_GENERALISATION\_NUMBER: The minimal number of identical properties or functions needed to do the generalisation step. The value can be any integer above 0. Default is set on 3.

To demonstrate the system, we examine an example scenario, where the different statements successively are executed. We report the event by a textual description, illustrated by several screenshots.

### Example scenario

In this scenario we have several users. User 1 and user 2 want to add information about their favourite topic: 'animals'. User 3, comes later, and adds information about clothing. The system doesn't know these users, thus they become both the initial rating.

- User 1 comes up with a new login john and password gorilla to get into the system

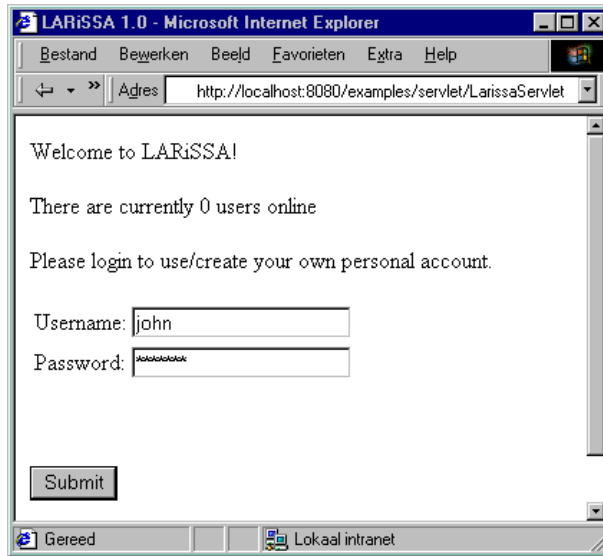


figure 25

- John adds 'isa(ob:gorilla,monkey);' (note that the syntax is slightly altered, ob: means that an object is meant).

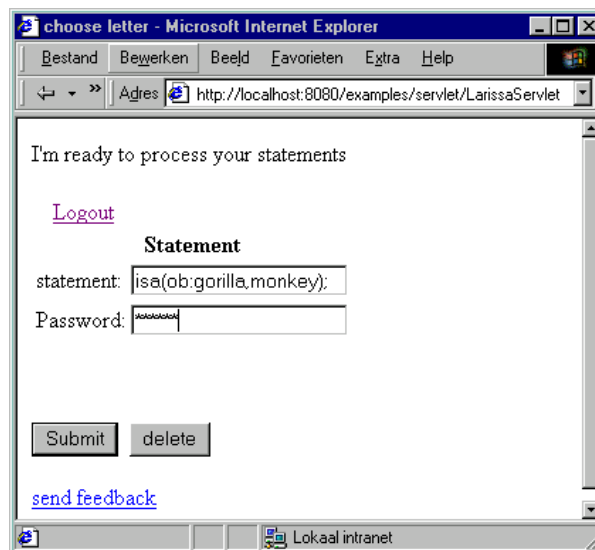


figure 26

- User 2 comes up with a new login mary and password bantam to get into the system
- mary adds `isa(ob:giraffe,animal);`
- mary adds `has(giraffe.neck,size,pq:long);`
- mary adds `func(eat,giraffe,plant,[]);` (note that the syntax is slightly altered, [] is a reserved slot for future extensions, currently it has no meaning at all)

The object 'plant' is never added to the system, therefore it answers:

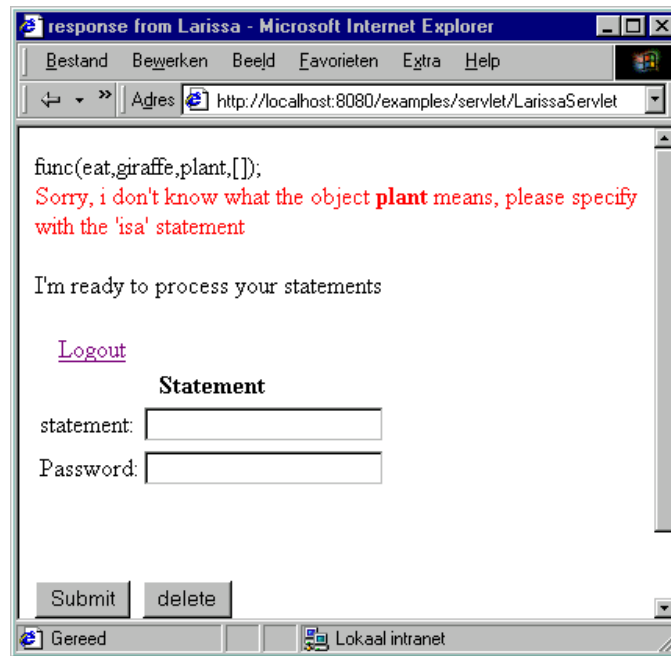


figure 27

- mary corrects the error by providing `isa(ob:plant,object);` and stating `func(eat,giraffe,plant,[]);` again.
- mary adds `isa(ob:monkey,animal);`
- john asks `?isa(ob:gorilla,\_);`

The system answers:

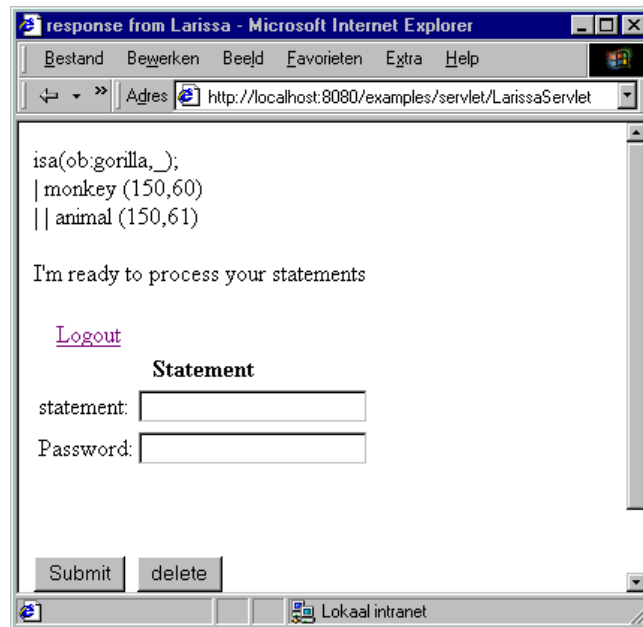


figure 28

The answer shows the tree structure (like figure 25), where a gorilla is a monkey, but also an animal. Both statements have a rating of 150.

- mary adds 'isa(ob:gorilla,monkey)'; This is the same statement which john added, therefore the rating of john's statement increases with the upgrade value (200). When mary wants to know the individual ratings and the aggregated relative rating of the statement she asks: '?isa(ob:gorilla,monkey);' Whereby the system answers:

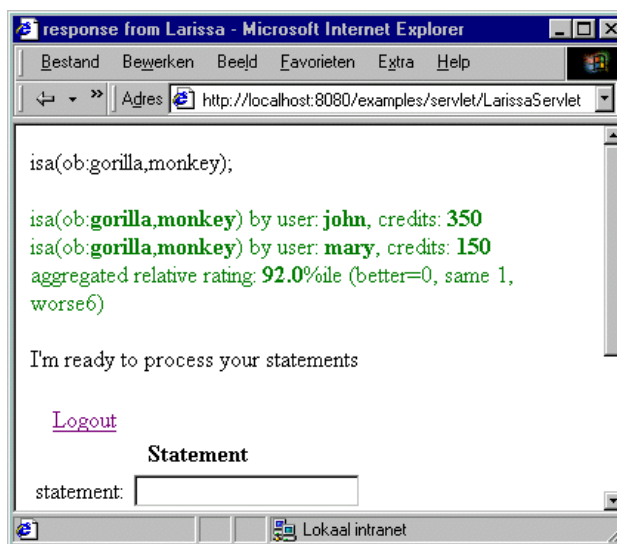


figure 29

- Next, john adds a function statement about a gorilla: `?func(eat,gorilla,plant,[]);?`. Now, john also wants to see the rating which is calculated, therefore he asks `?func(eat,gorilla,plant,[])?`. The answer of LARiSSA:

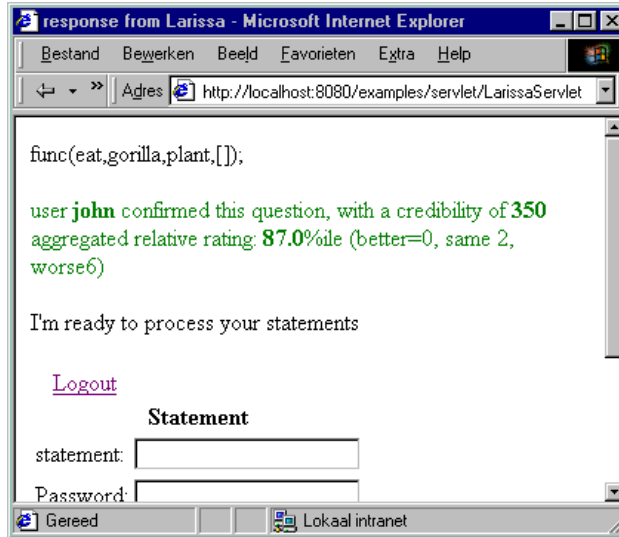


figure 30

As we can see the rating now is 350, because the single statement from john has a rating from 350 and remains in the same node.

- next, john adds `?func(eat,monkey,plant,[])?`, which means after asking the statement:

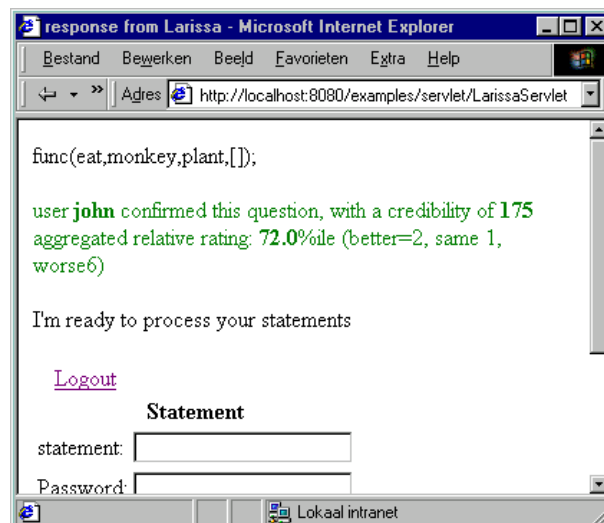


figure 31

- Next, a new user, named frenzy adds a new clothing-mark named 'gorilla' : 'isa(ob:gorilla,clothingmark)'. Now the word 'gorilla' has two different superclasses, which should be provided in the future.
- John doesn't know about the intervention of frenzy and forgets to provide the superclass.

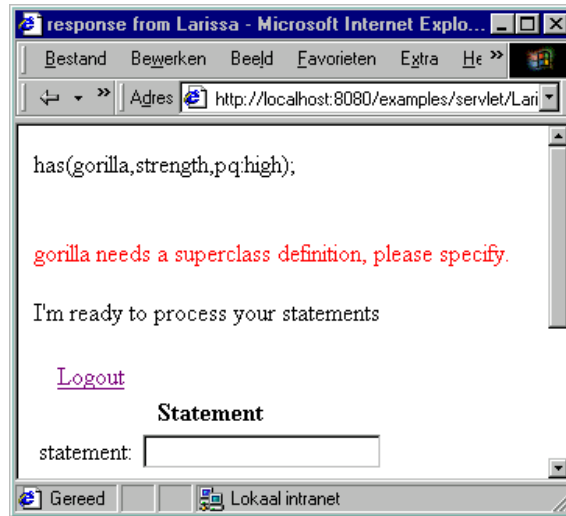


figure 32

- He adds the next statement 'has(gorilla,strength,pq:high);', which results in the next error statement:
- john wants to know which superclasses of a gorilla exist and asks '?isa(ob:gorilla,\_)'. After this he repeats the property statement accompanied by the superclass 'has((animal)gorilla,strength,pq:high);'
- To demonstrate the generalisation step, we introduce a new short scenario. Assume that john consecutively adds three different monkeys (gorilla, chimpanzee and orang-utan), which have all the function of eating plants, namely
  - isa(ob:plant,food);
  - isa(ob:gorilla,monkey);
  - isa(ob:orang-utan,monkey);
  - isa(ob:chimpanzee,monkey);
  - func(eat,gorilla,plant,[]);
  - func(eat,chimpanzee,plant,[]);
  - func(eat,orang-utan,plant,[]);

The generalisation threshold is set on three, therefore the generalisation process is executed. After adding the last statement about the orang-utan, LARiSSA answers:

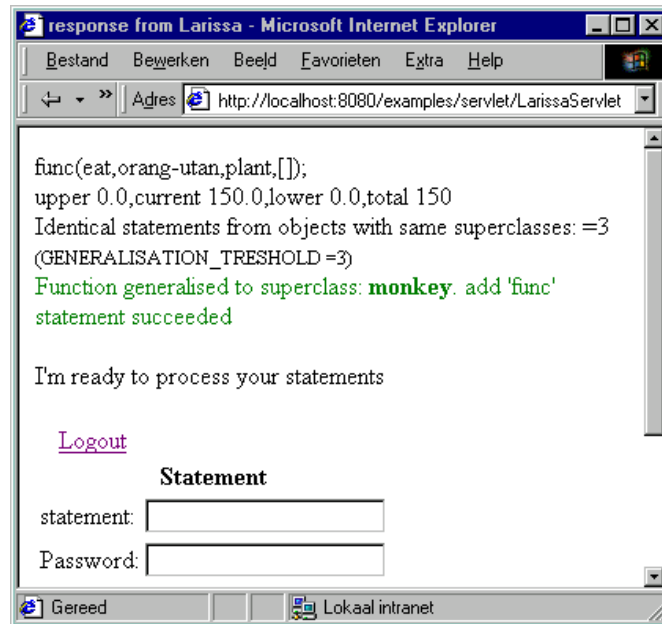


figure 33

The system now contains the information that monkeys eat plants.

The example scenario gave a brief overview of the possibilities, however we stress that the system shows its full strength when many users gave an important amount of information. Now that we know what the 2416 lines of Java code from LARiSSA are about, we come to the concluding part.

## 6 Conclusions

We stated in the introduction that ontologies remaining on the WWW have unique characteristics like the dynamic structure of the information and the uncertainty about the reliability of the source. The specification of both terms *dynamic* and *evolving* made a clear separation between different ontology environments. Both definitions can be used to categorise different ontology languages. A dynamic ontology provides efficient means to alter or extend its structure: when information is added or altered, the ontology remains a consistent and correct structure. Evolving ontology extends a dynamic ontology where the validity of the information, at least, stays the same. The literature study makes clear that the dynamic part of some different languages is sufficient, however the evolving aspects are almost always neglected. The different languages all depend on the assumption that

knowledge engineers develop reliable and semantic consistent ontologies. We think that this assumption is impossible to maintain on the WWW. Therefore we introduced a method to give a relative predicate on statements in the ontology. Another important issue is the implicit knowledge remaining in the different ontologies on the web. Most of the time information is not made explicit to relieve the knowledge engineer or for efficiency reasons like limited bandwidth. We examined several methods, like classic deduction, generalisation/specialisation, reasoning by analogy, which automatically frees this implicit knowledge and add it explicit into the ontology. When the information is made explicit, no reasoning effort has to be inserted to retrieve it, therefore the extended ontology can be seen as an evolved structure compared with the other ontology. LARiSSA is introduced as a prototype system with the objective to conceptualise and implement the different 'evolving methods'. Different requirements on the ontology structure are made to guarantee that the named methods can be inserted in the system. LARiSSA is implemented as a Java Servlet in combination with a MySQL database. This results in a fast and easy accessible system on the Internet, which is needed to reach and maintain a high amount of users that provide the information. Due to time constraints, only the rating method and a part of the generalisation/specialisation method are implemented. However, the architecture of the software makes it easy to extend the system with additional evolving methods. Adjustable parameters in the LARiSSA make it easy to write different scenarios where experiments should calibrate them.

This work should be viewed as a new way to look at handling ontologies on the Web, where the implemented prototype serves as an experimental platform for different evolving techniques. Future theoretical effort on this subject should lay a more solid fundament towards the evolving aspects introduced in this thesis.

## 7 References

[Bhatta, 1992]

Bhatta S. (1992). **A Model-Based Approach to Analogical Reasoning and Learning in Design**. Technical Report GIT-CC-92/60, Georgia Institute of Technology, College of Computing, Atlanta, GA, November 1992. Ph.D. Thesis Proposal.

[Budanitsky, 1999]

Budanitsky A. (1999). **Lexical Semantic Relatedness and Its Application in Natural Language Processing**. Technical report CSRG-390, Department of Computer Science, University of Toronto, August 1999.

[Budanitsky et al., 2001]

Budanitsky A., Hirst G. (2001). **Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures**. *Workshop on WordNet and Other Lexical Resources, Second meeting of the North American Chapter of the Association for Computational Linguistics*, Pittsburgh, June 2001.

[Carbonell, 1986]

Carbonell, J.G. (1986). **Derivational analogy: a theory of reconstructive problem solving and expertise acquisition**. In R.S. Michalsky, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371--392. Morgan Kaufmann Publ., Los Altos, 1986.

[Decker et al. 1999]

Decker S., Erdmann M., Fensel D., Studer R. (1999). **Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information**. In *Semantic Issues in Multimedia Systems. Proceedings of DS-8*. Kluwer Academic Publisher, Boston, 1999, 351-369.

[Farquhar et al., 1997]

A. Farquhar A., Fikes R., Rice J. (1997). **The ontolingua server: A tool for collaborative ontology construction**. In *Journal of Human-Computer Studies*, 46:707-728.

[Fensel et al. 1998]

Dieter Fensel, Stefan Decker, Michael Erdmann, and Rudi Studer (1998). **Ontobroker: The Very High Idea**. In *Proceedings of the 11th International Flairs Conference (FLAIRS-98)*, Sanibal Island, Florida, May 1998.

[Fensel et al. 1999]

Fensel D., Angele J., Decker S., Erdmann M., Schnurr H., Staab S., Studer R., Witt A. (1999). **On2broker: Semantic-Based Access to Information Sources at the WWW**. In *Proceedings of the World Conference on the WWW and Internet (WebNet 99)*, Honolulu, Hawaii, USA, October 25-30, 1999.

[Fensel et. al, 2000]

Fensel D., Horrocks I., Harmelen van F., Decker S., Erdmann M., Klein M. (Oct 2000). **OIL in a nutshell**. In *Knowledge Acquisition, Modelling, and Management, Proceedings of the European Knowledge Acquisition Conference (EKAW-2000)*, Lecture Notes in Artificial Intelligence, LNAI, Springer-Verlag.

[Gruber, 1992]

Gruber T., (Nov 1992). **Ontolingua: A Mechanism to Support Portable Ontologies**. Knowledge Systems Laboratory.

[[http://ksl-web.stanford.edu/KSL\\_Abstracts/KSL-91-66.html](http://ksl-web.stanford.edu/KSL_Abstracts/KSL-91-66.html)]

[Heflin, 1998]

Heflin J., Hendler J., and Luke S. (1998). **Reading Between the Lines: Using SHOE to Discover Implicit Knowledge from the Web**. In *AI and Information Integration. Papers from the 1998 Workshop*. WS-98-14. AAAI Press, 1998. pp. 51-57.

[Heflin, 1999a]

Heflin J., Hendler J., Luke S. (1999). **SHOE: A Knowledge Representation Language for Internet Applications**. *Technical Report CS-TR-4078 (UMIACS TR-99-71)* Institute of Advanced Computer Studies, University of Maryland, October 1999.

[<http://www.cs.umd.edu/projects/plus/SHOE/pubs/techrpt99.ps>]

[Heflin, 1999b]

Heflin J., Hendler J., Luke S. (1999). **Coping with Changing Ontologies in a Distributed Environment**. In *AAAI-99 Workshop on Ontology Management*. Department of Computer Science, University of Maryland.

[<http://www.cs.umd.edu/projects/plus/SHOE/pubs/shoe-aaai99.ps>]

[Heflin, 2000]

Heflin J., Hendler, J. (2000). **Dynamic ontologies on the web**. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449. AAAI/MIT Press, Menlo Park, CA, 2000.

[Klein, 2001]

Klein M., Fensel D. (Aug 2001), **Ontology versioning for the Semantic Web**. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, Stanford University, California, USA.

[Owen, 1990]

S. Owen. (1990). **Analogy for Automated Reasoning**. Academic Press Ltd, 1990.

[Schmid-Isler, 1998]

Schmid-Isler S., Selz D., Wittig D. (1998). **Retrieval, Rating and Reliability - How to establish RRR Standards on the Internet?** Contribution for the

Conference: *Information Quality and Knowledge. First European Half-Day Conference on Information Quality and Knowledge*, 3rd of December 1998, MCM institute, St. Gallen, Switzerland, 09/98  
[\[http://www.mediamanagement.org/netacademy/publications.nsf/all\\_pk/1053\]](http://www.mediamanagement.org/netacademy/publications.nsf/all_pk/1053)

[Voss 1997]

Voss G. (1997), **JavaServer Technologies , Part I**

[\[http://developer.java.sun.com/developer/technicalArticles/Servlets/JavaServerTech1/index.html\]](http://developer.java.sun.com/developer/technicalArticles/Servlets/JavaServerTech1/index.html)