National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

A Lazy Text-based Approach to Foundational Knowledge Acquisition

by

Louis Massey

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master in Computer Science

Ottawa-Carleton Institute for
Computer Science
and
Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada

January 1995

ISBN 0-612-04912-4

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Knowledge Acquisition (KA) from text requires that a large quantity of prior knowledge be made available to the Natural Language Processing (NLP) system. This prior knowledge is called foundational knowledge. The question of where foundational knowledge comes from in the first place is one of the biggest problem facing NLP (and all knowledge-based systems for that matter). Conventionally, foundational knowledge has been hand-crafted on a task- and domain-specific basis. However, it is difficult to determine *beforehand* exactly what knowledge will be required. It has been shown within the TANKA project that a potential solution to this problem is to use surface NLP. Indeed, surface NLP relies solely on syntax and on the help of a user to elicit knowledge from text, hence effectively eliminating the need for prior-hand crafting of foundational knowledge. However, the domain knowledge obtained in this manner from a text will be shallow and incomplete: we say that it contains gaps. The work presented in this thesis consisted in finding a better method than prior hand-crafting to acquire the knowledge needed to fill those gaps. The method presented, called Lazy KA, uses examples (short NL stories) and failures of an explanation mechanism such as EBL to find these gaps and to interactively and incrementally learn the required new knowledge. When the explanation of a particular example fails, the user is guided through a process that leads to the acquisition of the missing knowledge. Initially, the user is heavily involved, but as more examples are processed, the user becomes less and less involved. The convergence hypothesis, that is that the user interventions would decrease as examples are processed, was verified experimentally by using the prototype system FOKAS implementing these ideas. The EBL Domain Theory was acquired from the Canadian Individual Tax Guide. The short texts used as training examples were written by author's acquaintances.

Acknowledgments

I would like to acknowledge the support and direction of my research supervisor, Dr Stan Matwin, as well as the stimulating environment provided by the MaLTe and TANKA research groups and all their members.

I also thank the Canadian Forces for providing the opportunity and the financial support I needed to complete my Master degree.

Merci aux 23 volontaires qui ont accepté de donner quelques minutes de leur temps pour écrire les textes des exemples dont j'avais besoin pour mon expérience.

Finalement, j'aimerais remercier de tout mon coeur mon épouse Louise Allard et mes enfants Jonathan, Simon et Kateri pour leur support moral et psychologique, leur amour ainsi que leur patience au cours des deux dernières années.

Contents

Chapter 3 - Problem Statement

Chapter 4 - Solution to the Problem

Chapter 5 - Related Work

Chapter 6 - Empirical Evaluation

Chapter 7 - Concluding Remarks

List of Figures

List of Acronyms

CWA	Closed-World Assumption
CD	Conceptual Dependency
DT	Domain Theory
EBL	Explanation-Based Learning
ES	Expert System
FOKAS	FOundational Knowledge Acquisition System
GC	list of Ground Clauses
HC	Horn Clause
ILP	Inductive Logic Programming
IR	Information Retrieval
KA	Knowledge Acquisition
KB	Knowledge Base
KBS	Knowledge-Based System
LF	list of Local Failures
MALTE	MAchine Learning from TExt
ML	Machine Learning
NL	Natural Language
NLP	Natural Language Processing
PAC	Probably Approximately Correct
PAC-KB	Probably Acceptably Complete Knowledge Base
PtH	Proto-net To Horn clause
TANKA	Text ANalysis for Knowledge Acquisition
TR	Theory Revision

Index

The following is a list of important terms and the page number(s) where they are defined or where they first appear.

Chapter 1 - Introduction

> *"A characteristic of dumb systems is that they never have expectation failures. [They] never venture outside their limits, and never need to grow or change. Change is a product of failure. Learning depends upon a system venturing beyond known borders into places [...] where failure may occur."*
> *(Riesbeck and Schank 1989 , p. 381)*

1.1 Purpose

The purpose of this chapter, in addition to introducing the subject of this thesis, is to briefly summarize:

a. the general motivation behind this work;

b. the problem I aimed at solving, its actual solution, and the empirical evaluation of the solution; and

c. my contributions.

The organization of the thesis follows these points. A detailed account of each of the above items will be given in subsequent chapters.

1.2 General Introductory Remarks

Knowledge-based systems (KBS) are becoming increasingly present in all spheres of human activity as evidenced in the March 1994 issue of Communications of the ACM. The KBS ability to solve more and more complex problems depends on the availability of large quantities of knowledge. However, knowledge is not something which is easily available: much effort (in terms of time and money) must be expanded to transfer it from its source (for example, a human) to a computer. This difficulty of acquiring knowledge has become known as the *knowledge acquisition bottleneck problem* (Feigenbaum 1977).

The difficulty of acquiring knowledge arises first of all from the manner in which KA has traditionally been performed. Someone (known as a knowledge engineer) interviews an

expert in a particular field for which we want to build a KBS. The objective of the interview is to obtain the expert's knowledge about a given domain so that it can be *hand-coded* into a computer. We say that knowledge is *elicited* from the expert. The result of the KA process is a knowledge base (KB): the expert's knowledge now represented in computer memory. With this method of KA, the bottleneck effect is at its peak. The reasons why it is difficult to acquire knowledge in this manner include the following (after Jackson (1990)):

a. since experts often use a jargon specific to their area of expertise, it may be difficult for a knowledge engineer to understand what is really meant by an expert;

b. experts often base their decisions on personal experience. This kind of knowledge is implicit, which makes it difficult to state clearly and concisely compared to facts and general principles; and,

c. common sense knowledge is used on a regular basis by experts when they solve problems. This type of knowledge is generally taken for granted, and therefore often not captured by the elicitation process.

One of the solutions that has been explored to address these problems is the use of semi-automatic KA systems (for example: MOLE (Eshelman et al. 1987), PROTOS (Bareiss 1989) and ODYSSEUS (Wilkins 1990)). With these systems, the objective is to have the computer elicit the knowledge directly from the expert, effectively eliminating most of the requirements for the knowledge engineer.

1.3 Motivation

However, there are problems with the semi-automatic KA approach briefly mentioned above. We will study this approach and some sample systems as well as their limitations in more details in chapter 5, Related Work. For the time being, it suffices to say that one of the main problem this approach suffers from is that the expert, who now becomes the user of a KA system, must build the KB from scratch. It would be much better if the initial KB could be learned from data, or alternatively, if it could be extracted from existing texts about the domain.

Hence, machine learning (ML) and natural language processing (NLP) could greatly facilitate KA by providing the initial KB. This would be the next logical step in the evolution of KA techniques: after eliminating most of the knowledge engineer's work by having the KA system elicit the knowledge directly from the expert, we could also eliminate most of the requirement for the presence of an expert by either learning from data directly or by extracting knowledge from *existing* texts. However, there is a major problem that has prevented this from happening. This problem is that both NLP and ML systems need large amounts of *prior knowledge* to work. By prior knowledge I mean knowledge that must already be available *before* NLP and ML systems can respectively process text and learn in any useful manner. Therefore, to use ML or NLP as KA tools, we need to have been able to first acquire the knowledge they need to function: this is the *prior knowledge problem.*

1.4 Problem Overview

So, it appears that in order to be able to use NLP or ML to acquire knowledge, one needs some more fundamental knowledge: the prior knowledge we mentioned at the end of section 1.3 above. But where is this more fundamental knowledge coming from?

Let's consider the specific case of NLP as it relates to the need for prior knowledge. Traditionally, prior knowledge has been hand-crafted for specific tasks and provided as a given to NLP systems. However, this is far from practical for tasks that target non-toy problems. For example, it is difficult to determine beforehand exactly what knowledge will be required for a NLP system: one would have to think of all most plausible situations that might be encountered during the system's use. Consequently, there is always the risk of missing some knowledge. This eventual lack of knowledge in turn results in the NLP system being unable to cope with novel situations.

The Cyc project (Lenat and Guha 1990) was started to address this problem of prior knowledge acquisition. The idea governing Cyc's approach is that there is "no free lunch" (Lenat and Guha 1990 p. 24). That is, there is no easy way around the lack of prior knowledge: ML and NLP will not help. Therefore, the prior knowledge, called *foundational knowledge* in Guha and Lenat (1994) and also in the rest of this thesis[1], must be obtained

[1] A more formal definition of foundational knowledge will be given in chapter 3. Although I will generally use foundational knowledge in this thesis to refer to the prior knowledge required by a NLP system, sometimes the context will be more appropriate for other terms such as either prior knowledge or new knowledge.

from humans manually entering it in a computer. The hypothesis that the Cyc project wants to verify is that once most of the millions of foundational knowledge elements that we humans require to function in the real world have been made available to a computer, then and only then will it be capable of understanding text (and of learning, behaving intelligently, etc...).

The problem this thesis aims at solving is basically the same as Cyc's: the acquisition of foundational knowledge. However, the problem of acquiring foundational knowledge is viewed from the very different perspective that NLP and ML *can* help. In fact, NLP of a text *without prior knowledge* is seen as an ideal means of identifying what specific foundational knowledge is needed for a particular task. Then, ML techniques can be used to learn the foundational knowledge from an example set. From this point of view, Cyc's large scale prior hand-coding of knowledge is seen as unsatisfactory because it is just another open ended manual KA effort that must be conducted before any useful task can be performed by a KBS (in our case, a NLP system). Therefore, it would be useful if one could acquire foundational knowledge at least semi-automatically while a KBS, which is lacking this foundational knowledge, is being used for a task. This is the problem this thesis is tackling.

Specifically, the problem is as follow:

a. First, a *superficial KB* is elicited from a text by *surface NLP* (Delisle 1994). Surface NLP is NLP which relies solely on syntax (and on a user for disambiguation). We say the resulting knowledge is superficial because it is merely a literal translation of the text into a representation that computers can manipulate for problem solving (such as logic). Hence, it lacks all knowledge the text writer assumed the reader would contribute in order to understand the text. This missing knowledge is foundational knowledge, and we say that the KB contains *knowledge gaps*.

b. Then, a method to fill the knowledge gaps is required: this is the problem I intend to solve. Thus, the aim of this work is to acquire the foundational knowledge *after* the initial KB has been built, effectively allowing NLP to be used as a KA tool. This is a great advantage as explained in the last paragraph of section 1.3.

This problem moreover comes with two important restrictions: first, no knowledge can be hand-coded beforehand; and second, the solution has to be domain independent.

1.5 Solution Overview

1.5.1 General Approach

A novel method of KA for foundational knowledge called *lazy KA* was developed. The method works as follows:

(1) Given:
- a superficial KB obtained by surface NLP of a text
- a set of examples (NL stories) translated by surface NLP into the same representation as the KB
- an explanation mechanism (Explanation-based Learning (EBL), for example)

(2) The explanation mechanism is used to uncover the knowledge gaps by failing to *explain* an example (explanations as in the EBL sense (Mitchell et al. 1986, Dejong and Mooney 1986, Kedar-Cabelli & McCarty 1987)). A failure to explain an example is used as a learning opportunity.

(3) Foundational knowledge that allows to bridge the gap is then acquired and generalized with the help of a user. This new knowledge can then be used on future failures.

(4) As examples are processed, and knowledge gaps filled, the lazy acquisition system should become more and more autonomous. This is the *convergence hypothesis* that we will verify empirically.

1.5.2 Implementation and Experimental Work

I have implemented these ideas in a prototype system called FOKAS (FOundational Knowledge Acquisition System). FOKAS can be described as an intelligent agent that acquires, through machine-user cooperation, the foundational knowledge missing from a

KB built from surface NLP of a text. It is based on a theorem prover that handles failures within an "Open-World Assumption" (as opposed to the Closed World Assumption) framework. That is, it does not fail when it cannot prove, but instead assumes that it can learn something that will make the proof possible.

The representation is First Order Logic in the clausal form, restricted to Horn clauses. The KB built from surface NLP of a text is a domain theory (DT) consisting of Horn clauses. An example, after translation from text, is a conjunction of ground unit clauses. FOKAS attempts to explain the example similarly to what is done in EBL. When FOKAS cannot prove the example, it guides the user through a process that will lead to the acquisition of the foundational knowledge required for the example to be explained. This new knowledge is consulted when other failures occur, with the hope that the system will become less and less dependent on the user.

Indeed, as examples are processed, more and more foundational knowledge is accumulated. The hypothesis behind this work is that as the quantity of foundational knowledge increases, the system will be able to rely more on its KB of foundational knowledge than on the user. The convergence hypothesis, which states that the KB of foundational knowledge converges towards completeness as examples are processed, was verified experimentally. This was accomplished by measuring the level of user interventions as examples were being processed by the system. The examples originated from real world texts describing situations from the domain of childcare tax deductions. The system was trained and I was able to show that the level of user interaction decreased as expected.

1.6 Contributions

The work presented in this thesis can be seen as an alternative to Cyc's approach to the acquisition of foundational knowledge; a sort of "cheap lunch" by analogy to Cyc's rather "expensive lunch". As pointed out in Minsky (1994), there have been no such alternatives proposed up to now.

More specifically, the main contribution of this thesis is to apply the idea of learning missing knowledge interactively and incrementally when a performance task fails (Davis 1979) to the problem of learning foundational knowledge missing from a KB built by surface NLP. As a result, on one hand, this allows NLP to be used to acquire domain knowledge

semi-automatically from texts, with no prior KA effort, which is something that has not been possible up to now. On the other hand, the knowledge which is usually most difficult to acquire, that is common sense and general world knowledge (which are the main components of foundational knowledge), can be acquired incrementally and semi-automatically during problem solving, with convergence to an acceptably complete KB[2]. This provides realistic constraints on what knowledge has to be acquired, rather than facing an entirely open-ended manual KA effort such as in Cyc.

In short, the method I propose, Lazy KA, provides an attractive solution to:

a. the problem of KA in general by allowing the use of surface NLP (NLP with no prior knowledge) for the acquisition of an initial KB; and

b. the more specific problem of acquiring foundational knowledge, also addressed by Cyc, but for which I propose a more realistic alternative.

1.7 Thesis Organization

Chapter 2 will give an overview of the difficulties NLP research has encountered and how these difficulties lead to the work this thesis is about. In chapter 3, I will state the problem this thesis tackles and in chapter 4, I will present a solution to this problem, including the description of a simple implementation. Then, in chapter 5, while the problem and its solution will be fresh in our minds, I will give an overview of related work. An empirical evaluation of the approach I present in the thesis will follow in chapter 6. Finally, chapter 7 will list areas of future work, contributions and the conclusion.

[2]In the spirit of PAC learning: see section 3.3.

Chapter 2 - Background Information

2.1 Introduction

This chapter will give a brief overview of NLP research and how it leads to the TANKA and MALTE projects. These projects will then be described since it is from them that the work presented in this thesis originates.

2.2 Natural Language Processing

This thesis is not directly concerned with the study of NLP as an end: it is more interested in NLP as a means to acquire knowledge, and specifically where NLP fails at this task. Therefore, philosophical issues such as what a NL expression means, as well as technical ones such as the study of specific NLP techniques will not be discussed here. We will only look at how difficulties with NLP as a KA tool lead to the work presented in this thesis.

2.2.1 What is NLP?

In the context of this thesis, NLP is a process that maps an input representation in *written* natural language to an output representation which can be *used* by a computer. More specifically, the output representation must be such that an inference engine can reason with it for the purpose of solving problems. That is, given a problem, the inference engine will be able to solve the problem appropriately with the knowledge which was extracted[3] from the input text. In this thesis, I will refer to this ability of a NLP system to extract usable knowledge from a text as the NLP system's *task*.

[3]The word *extracted* refers to the transfer of information (or knowledge) from the NL representation to the computer's memory and representation. The words transfer, elicitation and extraction (of knowledge, of information) will be used throughout this thesis to refer to the mapping process of NLP.

2.2.2 The Difficulties Facing NLP

Before NLP techniques can be used to acquire knowledge from text, many problems must be solved. The problems most directly related to this work are as follows (Rich & Knight 1991, p. 378):

a. "English sentences are incomplete descriptions of the information that they are intended to convey" : when we speak or write, we often leave things implicit: we assume that our interlocutor will understand because we share knowledge and context. In fact, if we look at texts specifically (since they are the subject of this work), we realize that a text rarely contains the whole of the information it aims at conveying: the reader has an important part to play by contributing the missing pieces (Schank 1982). Adults can contribute these missing pieces of knowledge because they have acquired the relevant knowledge as they grew up. However, computers that aim at understanding NL do not have (yet) this capability of learning from their experience in the world. Hence, an outside agent must provide the knowledge a NLP system needs for its KA task.

b. "There are lots of ways to say the same thing": people use NL to describe events or objects in many different ways and yet, they understand each other. The knowledge (of the situation, of language and of the world) they share allows them to use a high degree of diversity in their NL exchanges. For computers to do the same, that is recognize expressions that "mean the same"[4], they must be provided with the necessary knowledge.

We can see that the two problems explained above have the same consequence: NLP systems rely on prior knowledge. Therefore, from now on I will refer to these two problems as a single one: the *prior knowledge problem*. This name comes from the fact that NLP requires that large quantities of knowledge be available *prior* to the task.

[4] We will see in section 2.2.3 an example of "equivalent meanings". However, it is not within the scope of this thesis to discuss the complex issues of meaning and of "equivalent meanings".

2.2.3 Solution to the Prior Knowledge Problem

The traditional approach to solving the prior knowledge problem has been to manually code the knowledge required by NLP systems. Many implementations of this *prior hand-coding paradigm* exist. In this section, I will only present a brief overview of two of them since ultimately, other implementations (for example Mellish 1985) are all equivalent: they all require that knowledge be hand-crafted before NLP can be performed.

2.2.3.1 Conceptual Dependency

One of the best known solution to the problem of prior knowledge is the use of *Conceptual Dependency* (CD) (Schank 1975). The idea behind CD is that the representation of a sentence on output from a NLP system should not be based on the words in the sentence, but rather on primitive concepts and relationships among these concepts. For instance, a set of primitive actions called ACTs as well as a limited number of allowable dependencies (semantic relations between concepts) are used to construct dependency structures that represent NL expressions describing events.

To illustrate the ideas of CD, we could use the following two sentences that are intended to represent the same event (this is an example of the second problem in section 2.2.2 - "There are lots of ways to say the same thing"):

(1) Peter sold his book to Mary
(2) Mary bought Peter's book.

If we represented these two sentences with the exact words that are used within the sentences, it would be rather difficult to see that they stand for the same situation. For instance, in (1) an agent named Peter *sells* an object named book while in (2) an agent named Mary *buys* an object book.

Within CD theory, both of these events would be represented identically on a conceptual level. Central to that representation would be the ACT "abstract transfer" (ATRANS). This transfer would specify a direction from Peter to Mary (i.e. that Mary is the recipient), and the presence of an object (book) in the transfer. CD would also imply other events, even if they are not explicitly stated in the text. For instance, a second ACT ATRANS in the opposite direction, involving an object money, could result from an ATRANS act originating from the words buying and selling. This is possible because CD

encodes semantic information about the concepts selling and buying, such as the fact that they imply an exchange of money.

2.2.3.2 Scripts, Plans and Goals

Conceptual Dependency allows a system to represent events and reason about them. Events, however, often occur in groups in which they are related to each other. For example, there are many events involved in the situation of eating at a restaurant: one first comes in, waits to be seated, is presented the menu by the waiter, then selects what to eat, etc. Such a sequence of events related among themselves form what is called a *script*. (Schank and Abelson1977). In NLP, scripts are very useful to understand situations such as the one that follows (which is an example of our first problem in section 2.2.2: "sentences are incomplete descriptions of the information that they are intended to convey."):

> "The man ordered a steak. He called the waiter back because the steak was overcooked and he had ordered it medium. The waiter gave him a bad look. The man did not leave any tip to the waiter".

In order to understand such a story, a NLP system would need first to realize that this is taking place in a restaurant, which is not mentioned anywhere in the text (but is very obvious to us). Then, it would have to use a script listing the events taking place in a restaurant, such as ordering food, eating it, paying for it and tipping the waiter. The system could also call upon knowledge about the purpose of tipping and about why we eat at restaurants (this kind of knowledge is not in the restaurant script: it is provided by *goals* and *plans,* also covered in Schank and Abelson1977). This being all provided beforehand, we could ask questions such as: did the man eat?, why did he not leave a tip?, etc. Without such knowledge, a NLP system would be unable to answer these questions because the information is not explicit in the text.

2.2.4. The Unresolved Issues

All prior hand-coding paradigm approaches, including CD and Scripts as well as those not described in section 2.2.3 above, leave two very important problems unresolved:

a. Where does prior knowledge come from?

b. How do we know what knowledge is required?

These two issues are discussed in the following two sub-sections.

2.2.4.1 Issue 1: where does prior knowledge come from?

Prior hand-coding approaches assume that prior knowledge is *somehow* available, with no indication as of its source. A possible method to acquire prior knowledge is with Machine Learning (ML) techniques. However, ML itself relies on prior knowledge to work (Minsky & Papert 1988), and consequently the same issue applies to ML. Therefore, it is reasonable to conclude that prior knowledge *must* be hand-coded by humans. In fact, this is exactly what has traditionally been done for NLP, because no better methods exist.

The real problem with this issue is thus not *where* the knowledge comes from but *when* should it be provided. Too often in Artificial Intelligence, assumptions about obscure sources of knowledge (and other forms of "cooking") are made. In general, we should simply accept that there has to be participation of a human knowledge provider in AI systems - the question is when, and how much.

So, the real problem of this issue is that by having people hand-code knowledge *beforehand*, we may defeat the purpose of acquiring some of that same knowledge from text later. Indeed, when the purpose of NLP is to acquire knowledge from text, the fact that some of the prior knowledge that requires to be hand-crafted is domain knowledge means that someone has to hand-code some of the knowledge which was aimed at being acquired from text in the first place. Proceeding this way therefore defeats the purpose of using text as a knowledge source. In conclusion, once the question of human involvement is clarified, the answer to this issue will be a matter of finding the *best time* to acquire prior knowledge.

2.2.4.2 Issue 2: How do we know what knowledge is required?

In order to decide what prior knowledge is required, someone must think of all most plausible situations or problems the knowledge extracted from the text will be used to solve. Then, from these situations, one must determine which exact prior knowledge will be required. Given a non-toy domain with almost unlimited situations and the quasi-infinite

ways in which NL can be used to express these situations, the initial KA effort becomes totally open-ended.

For example, in the tax domain, knowledge about the family, employment and finance will be required. There are many other areas about which we need knowledge. For example, we must think of all common sense knowledge which is needed. As we have seen in section 1.2, this type of knowledge is very difficult to acquire because we often take it for granted. For instance, we may need knowledge about movement such as: "when an object moves from a location A to a location B, it is not at location A anymore" (to address the situation of people who move from a city to another). Most probably, there are other situations which will require other knowledge (common sense or others), but it is not possible to think of all of them.

Then, we must ask ourselves *how much* knowledge must be hand-coded in each of the areas mentioned above? For instance, for the family-related knowledge: What about a situation in which someone claims a medical expenses deduction for its pet? Is it reasonable to expect that the person who hand-coded the prior knowledge should have thought about hand-coding knowledge that states that pets are not members of a family when comes time to make medical expenses tax deductions?

The complexity of this issue arises from the fact that it is not possible to really decide *beforehand* what knowledge will be required. Additionally, it is not realistically possible for the people hand-coding the prior knowledge to *think of all* the knowledge that will be required. This is why NLP systems are so brittle and can rarely process text from outside the very specific area for which they were given the required knowledge.

2.3 The TANKA and MALTE projects

The prior hand-coding of knowledge leaves the two issues seen in section 2.2.4 unresolved. The TANKA and MALTE projects looked at an original way to solve these issues. In this section, I will describe these two projects and show how they lead to the subject of this thesis.

2.3.1 TANKA: A General Description

The TANKA (Text Analysis for Knowledge Acquisition) project (Szpakowicz 1990, Copeck et al. 1992, Delisle 1994) looked at the issues discussed above in a rather unconventional way: by attempting KA from text *without* prior knowledge. The fundamental assumption behind the TANKA research is that structure is a good indication of meaning (Kieras 1985).

Specifically, TANKA's goal is to obtain domain knowledge from an *expository technical text*[5] by relying exclusively on syntactic clues and the guidance of a user. Hence, TANKA acquires a domain model from a text describing that domain with no prior knowledge other than syntactic and lexical knowledge. This approach is called *surface NLP* in this thesis. The word "surface" evokes the fact that neither deep knowledge of the domain nor of the world is used in mapping from the NL representation to the computer-usable representation. Surface NLP is described in detail in Delisle (1994).

Surface NLP allows us to acquire knowledge directly from text with none of the extensive and complex prior hand-crafting of knowledge described previously. This effectively seems to solve the problems we discussed in section 2.2.4.

However, surface NLP, by avoiding the issues involved in prior hand-crafting of knowledge neglects to consider the prior knowledge problem discussed in section 2.2.2. This problem consisted in fact of two difficulties:

a. "English sentences are incomplete descriptions of the information that they are intended to convey."

b. "There are lots of ways to say the same thing."

Consequently, a KB built with surface NLP of a text will suffer from many imperfections caused in part by these two difficulties. In other words, these difficulties will become very apparent in a KB built with surface NLP. We say that the resulting KB is *superficial*. Similarly as for the word "surface" in surface NLP, the word "superficial" in this

[5]This work is mainly concerned with what has been qualified of "expository technical" text (see Delisle (1994) pp. 7-8 for a justification of the choice of such texts). Examples of expository technical texts are: specifications, user manuals or user guides, and regulations (building code, tax regulations, etc).

case implies that the resulting KB is simply what we could call a *literal translation* of the NL text into a computer-usable representation. By literal translation it is meant that the words of the text are used as the computer representation symbols and that information implicit in the text will not show up in the computer representation.

In TANKA, the superficial DT obtained from a text is not, strictly speaking, a literal translation of that text because additional information is added by a user who helps in the knowledge extraction process. However, such additions can be minimized - as well as the user's involvement - to give a DT which contains very little additions and which is as such very close to being a literal translation of the original text.

The following sub-section will give an example of a superficial KB. For historical reasons, examples in this thesis are all based on the tax domain[6].

2.3.1.1 An example of superficial KB obtained by surface NLP of a text

The two short texts in Figure 2.1 state respectively the regulations governing the deduction of child care expenses for tax purposes and an actual situation for a particular individual. Let us first look at how the difficulties inherent with NL (c.f. section 2.2.2) materialize with these texts.

Regulation

Child care expenses are the amounts you pay to someone to look after your child while you work or if you are a student. You can claim child care expenses if you are the supporting person of an eligible child. An eligible child is a child whose age is less than 14, or a child who is handicapped. A supporting person is the parent of the child or the parent's spouse.

Situation

Joe is the father of a pre-school child. Joe leaves his son at the daycare center when he goes to school. It cost him $2000 in child care services last year.

Figure 2.1: The first paragraph is a text describing a tax domain rule and concept definitions. The second paragraph describes a specific situation or example to which the rule applies.

The first difficulty discussed in Section 2.2.2 is that sentences carry implicit information. For example, when the situation states that Joe is the father of a pre-school child,

[6]The Tax Domain and the Canadian Personal Income Tax Guide for the year 1990 were respectively the domain and the main text investigated within the TANKA/MALTE projects. Since work presented in this thesis is closely tied to these projects, the same domain and text were used. Note that the tax regulations are not necessarily representative of Canada's Tax Law as they may have been modified for illustrative purposes.

there is information about the age of the child: we know that he is less than five. We are able to use this information because we have the necessary knowledge to relate the property pre-school to a certain age. Another example is when the situation text states that Joe leaves his son at the daycare center and the regulation text requires that someone looks after your child. From the former fact, we can infer that someone indeed looks after the child, because we know what is going on in daycare centers (by using script-like knowledge).

The second difficulty states that there are many ways to say the same thing. In Figure 2.1, this happens when in the situation text, the word father is used while in the regulation text, parent is used. We know that a parent is either a father or a mother, so we recognize these two words as meaning the same in this situation. Another example is that the situation text states that Joe is a student by using the clause: he goes to school. These are two ways to say the same thing, and again we can recognize them as such because we have the required knowledge.

In fact, it is obvious to most human beings, after reading both texts, that Joe can claim his child care expenses even though: (1) the situation text does not explicitly give all the information required by the regulation text; and (2), when it does, it often uses different words and expressions. We know that Joe can claim childcare expenses because we use our general world knowledge, common sense knowledge and knowledge of the language.

We can now look at what happens when surface NLP is used to translate the regulation text of Figure 2.1 into a KB usable by a computer. What we want are the rules that describe the tax regulations as well as the concept definitions and relations among concepts as stated in the text. Then, this knowledge can be used by computers to solve problems or answer queries such as : is Joe eligible for this deduction? The knowledge extracted from the regulation text as well as Joe's situation are shown in Figure 2.2 after surface NLP has been

---

Domain Theory:

```
claim(Person,Expense,Child):-
      eligible(Child),
      supporting(Person,Child),
     childcare_expense(Expense,Person,Child).

eligible(Child):-
      age(X,Child),
      less_than(14,X).
eligible(Child):-
```

```
    handicapped(Child).

supporting(Person,Child):-
      parent(Person,Child).
supporting(Person,Child):-
      parent(Person2,Child),
      spouse(Person2,Person).

childcare_expense(Expense,Person,Child):-
      work(Person),
      pay(Person,Expense,Someone),
      look_after(Someone,Child).
childcare_expense(Expense,Person,Child):-
      student(Person),
      pay(Person,Expense,Someone),
      look_after(Someone,Child).
```

Example:

```
father(joe,child).
pre_school(child).
leave(joe,son,daycare_center).
go(joe,school).
cost(joe,2000,childcare_services,last_year).
```

Figure 2.2: The Prolog rules obtained from surface NLP of the regulatory text of Figure 2.1 form a superficial DT. The Prolog facts result from applying surface NLP to the specific situation of Figure 2.1. We call the conjunction of facts an example.

used to translate the texts of Figure 2.1 into Prolog rules and facts. The Prolog rules correspond to the regulatory text and are called a *Domain Theory (DT)*. The Prolog facts correspond to the specific situation given in the second text in Figure 2.1 and are called an *example*.

We have seen previously that a human reader would have no problem recognizing Joe as eligible for child care expenses. However, once the two texts of Figure 2.1 have been converted by surface NLP into the Prolog rules and facts of Figure 2.2, it is not possible, by using an automatic theorem prover for example, to show that this is the case. Indeed, if we were to use the clauses of Figure 2.2 in a Prolog system and enter the query:

```
?- claim(joe,Expense,son).
```

this query would fail because Prolog lacks the knowledge to recognize father(joe,child) as a clause that carries the right information to prove parent(joe,child), that go(joe,school) means the same as student(joe), etc. We recognize in these missing knowledge elements the knowledge items mentioned previously and which were contributed by a human reader.

We can see that the KB built by surface NLP lacks the knowledge a human reader would normally contribute. The DT in Figure 2.2 is, from this point of view, an almost[7] literal translation of the knowledge contained in the original text, where the words of the text are used as Prolog symbols and no or minimal additional knowledge or information has been added to what is explicitly stated in the text. The DT is qualified as superficial for this reason. In contrast with a superficial DT, a non-superficial DT would also contain *foundational knowledge* such as the rules and facts listed in Figure 2.3. Foundational knowledge (Guha & Lenat 1994) is the general knowledge of the world and the common sense knowledge needed

```
age(less_than_5,A):-                    A less awkward rule would be: age(X,A), less_than(5,X):-
        pre_school(A).                                                        pre_school(A).
                                        but it would not be a HC, and HC are required as the representation
                                        for this work.
less_than(14,less_than_5).              Fact to use with above rule.

parent(A,B):-                           The father A of a person B is also his parent.
        father(A,B),
        isa(A,human),
        isa(B,human).

equiv(son,child).                       son and child are two constants used to refer to the same object.

student(A):-                            Someone is a student if he goes to an
        go(A,B),                        educational_institution.
        isa(B,educational_institution),
        isa(A,human).

pay(A,B,C):-                            A person A pays an amount of money B to
        cost(A,B,D,E),                  an organization C if it has cost that
        isa(A,human),                   person that amount of money for a
        isa(B,amount_of_money),         service D provided by the organization
        isa(C,organization),            at time E
        isa(D,service),                 (argument E is not really relevant)
        isa(E,time),
        provide(C,D).

look_after(D,B):-                       A person D looks after a child B if
        leave(A,B,C),                   another person A leaves the child at
        isa(A,human),                   a location C known to be a
        isa(B,human),                   childcare_provider location, and as long
        work(D,C),                      as the person D looking after the child
        isa(D,human),                   also works at the childcare_provider
        isa(C,childcare_provider).      location.
```

Figure 2.3 : Prior or foundational knowledge required to use the DT and example of Figure 2.2.

[7]Surface NLP is a semi-automatic process which allows for the introduction of information not present in the text. For instance, prepositional phrases attachment, pronoun resolution as well as the formation of some simple concepts (such as last_year, by linking words of the text with underscores) have all been done with the help of a user during the surface NLP process.

to understand a text (such as when reading an encyclopedia), or again, the knowledge we assume the others have when we interact with them.

These rules and facts allow for the above query to succeed (given the ability to use equiv(son,child) to actually match the constants son and child). They represent the knowledge which would be used by humans (as discussed previously) when reading the two texts in Figure 2.1. Hence, a KB obtained from text with surface NLP will need to be augmented with other knowledge such as what is given in Figure 2.3. However, Figure 2.3 lists only the specific knowledge required for the situation exemplified by Joe's case. Much more knowledge would be needed to support other situations. Note also that these rules relate DT literals which cannot be proven with the example ground clause that carries the right information for the proof to succeed. The rules have also been augmented of type information to constrain their application. These are all typical rules that the work in this thesis aims at acquiring.

2.3.2 MALTE: A General Description

The MALTE project (Matwin & Szpakowicz 1993, Delannoy et al. 1993), conducted in parallel with TANKA since 1992, aims at studying the synergistic effect of ML and NLP for the acquisition of knowledge from texts. MALTE's hypothesis is that by using ML methods, we may be able to improve the knowledge we have acquired by surface NLP. As such, MALTE is first of all a performance task for TANKA: it needs to use the knowledge from surface NLP as is, with all its shortcomings.

However, it was soon realized that there were just too many problems with the knowledge acquired by surface NLP to use ML techniques. For example, one of the ML techniques used by MALTE is Explanation-based Learning (EBL). EBL needs a complete DT, but because the DT is acquired by surface NLP, it is not complete. Therefore, EBL explanations were never going very far before failing because of missing knowledge. The next sub-sections will illustrate this situation and define the important concept of *knowledge gap*.

2.3.2.1 Knowledge Gaps

For the purpose of illustrating the problem that EBL faces when using a superficial DT, suppose we use the DT and example of Figure 2.2 for EBL. We could then ask for the EBL explanation by inputting the following query, partially instantiated with the constants joe and son from the example:

```
-? ebl(claim(joe,Amount,son), Explanation).
```

to explain why Joe can claim child care expenses for his son, and to determine what is the amount of that claim.

The result of the above query would be negative: EBL would fail to explain the example. Figure 2.4 shows the proof tree that would result from the above query if we allowed EBL to continue the explanation even after it failed. This allows us to see all failures that *should not* occur. These failures should not occur because the example carries all the necessary information for the explanation to continue, but this information is not explicit in the original texts, so it is not present either at the output of surface NLP.

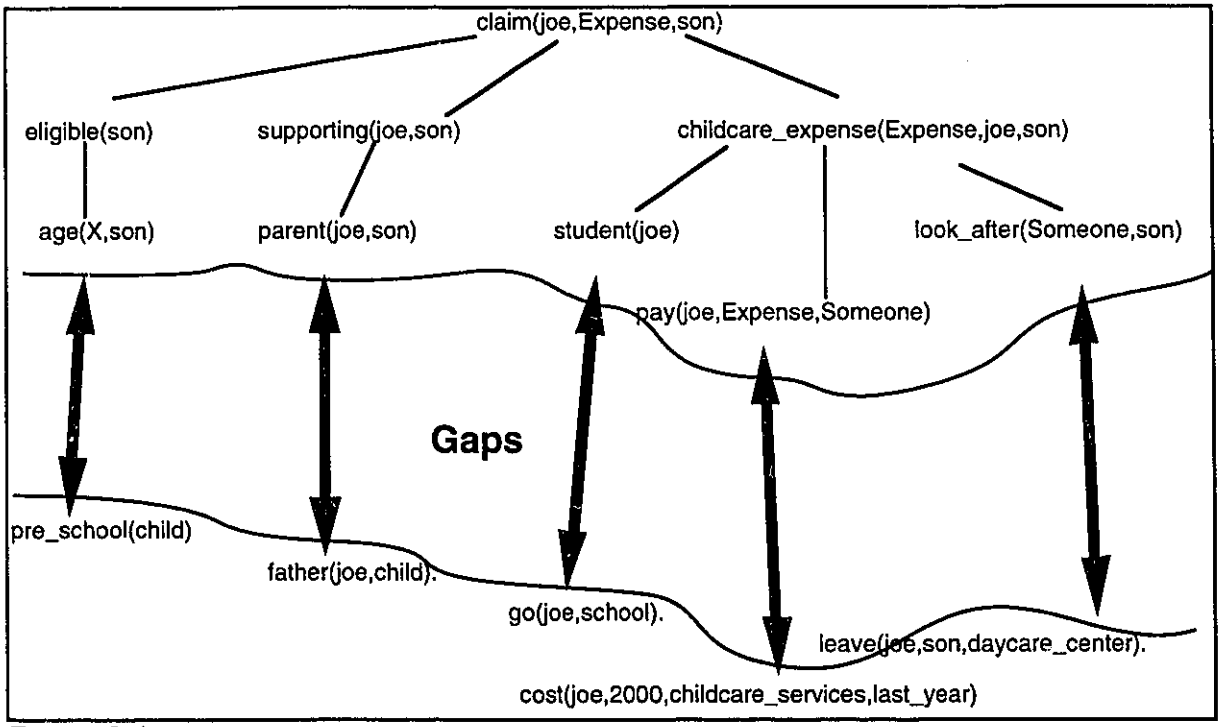

Figure 2.4: An EBL failing-proofs tree based on the example and DT of Figure 2.2. However, these failures are not real failures: they are merely caused by a lack of knowledge. This lack of knowledge results in knowledge gaps between the proof (in the DT) and the example clauses.

Thus, the only reason for EBL failures with this example is that we lack the foundational knowledge required to explain how the example features satisfy the DT concept definitions. In other words, EBL lacks the knowledge to recognize statements that mean the same or that carry the required implicit information: these are our NL difficulties of Section 2.2.2. More specifically, the proof cannot continue because the knowledge that allows EBL to go from a DT failing literal to the potentially proof-terminating ground clause of the example is not available. We call each occurrence of such missing knowledge element a *knowledge gap*. The foundational knowledge required to bridge these gaps was given in Figure 2.3. Once this knowledge is made available, Joe's situation can be explained by EBL.

2.3.2.2 Characterizing the knowledge gaps

There are three kinds of knowledge gaps, where the type of a gap is determined by the kind of knowledge required to fill it:

a. Domain Gaps. Since the final computer representation is a literal translation of the text, what is missing from the text in term of domain knowledge will also be missing from the DT. This problem is not apparent in the above example, but we can postulate its existence: suppose for instance that the author of the text forgot to mention some regulation. When EBL processes an example that needs the rules related to that regulation, the knowledge gap resulting from the failure to explain the example will be a domain gap; that is missing domain knowledge.

b. World and Common Sense Gaps. The DT resulting from surface NLP will not contain the general world knowledge and common sense knowledge that would normally be used by a human to read and understand the text. This type of knowledge is not in the text because the writer assumes (consciously or not) that the reader will contribute it. For instance, a text on moving claim deductions will most probably not state that once you have moved from city A to city B, you are not living in city A but in city B. The proof tree of Figure 2.4 abounds in such gaps: the gap between go(joe,school) and student(joe), and between leave(joe,son,daycare_center) and look after(daycare_center,son) are two examples.

c. Language Gaps. It is customary in text writing to use different words or phrases to refer to the same thing. This is often a question of style, and sometimes the result of inadvertence. The case of synonyms is an obvious example. A less obvious case is the use of words that are related by other relations such as *meronymy* (member_of, substance_of, part_of) and *hyponymy* (kind_of) (Miller 1990). Thus, it is possible to refer to an object by one of its part or by a more specific concept. This results in language gaps, because we are lacking the language knowledge to recognize the different words as referring to the same object. For instance, the words 'son' and 'child' are not synonyms: son is a kind of child, but they are nevertheless used to refer to the same person in the text given previously.

2.3.2.3 The Need for a Knowledge Gap Filler

We have seen that surface NLP results in a DT which lacks foundational knowledge. Consequently, such a DT cannot be used by EBL since EBL relies on a complete DT. In our context, a DT may be almost complete in term of its domain knowledge, but very incomplete in term of foundational knowledge.

Within MALTE, that meant that we needed a way to fill the knowledge gaps if we ever wanted to be able to use the knowledge obtained via surface NLP. In fact, this need to fill knowledge gaps can be generalized outside the MALTE context: if we want to be able to use knowledge acquired by surface NLP for any useful task, we need a way to acquire the missing foundational knowledge.

What we did not want to do though was to become involved in the major undertaking of prior knowledge hand-crafting. There were two reasons for not wanting to proceed this way. First, it was directly against the initial goal of the TANKA project, namely to acquire knowledge from text *without* prior knowledge hand-crafting. Second, it was rather difficult to delimit exactly what knowledge would be required (for the reasons discussed in section 2.2.4.2).

What is interesting to observe at this point is that the knowledge missing from a DT acquired by surface NLP is the same as the knowledge that should have been hand-crafted beforehand for non-surface NLP. Therefore, if we do not provide the NLP system with the

required knowledge at the beginning, the resulting KB will lack exactly that same knowledge.

This brings us exactly to the problem this thesis is tackling. Since the foundational knowledge needed for a NLP system is exactly the knowledge we end up missing if we use surface NLP methods to extract knowledge from a text; and since EBL is failing *where* this knowledge is missing (i.e. at knowledge gaps); therefore, we can use these failures as opportunities to learn the missing foundational knowledge on an *as needed basis*, only as it becomes required by failures of the EBL task. This effectively addresses the issue of when the missing knowledge should be acquired (c.f. section 2.2.4.1).

2.3.3 Overview of TANKA and MALTE

This section will show where the work presented in this thesis fits with respect to the TANKA and MALTE modules. For this purpose, I will summarize what we have seen about TANKA and MALTE as well as give additional details on how they interact.

2.3.3.1 Representation

The representation used for MALTE and for this work is First Order Logic in the clausal form, restricted to Horn clauses (HC). Moreover, a special HC representation is being used by MALTE. For example, instead of simply writing:

```
pay(joe, 2000, daycare_center, last_year).
```

to mean that Joe has paid $2000 to the daycare center last year, the following is used:

```
act(pay, [agt, obj, benf, tat], [joe, 2000, daycare_center, last_year]).
```

where the first argument is the verb; the second, a list of cases (agent, object, beneficiary and time at); and the last argument, a list of the objects involved in the roles given by the cases list.

However, in order to lighten the text, the former representation will be used in this thesis.

2.3.3.2 TANKA and MALTE Interaction

We have seen that the representation used by MALTE is Horn Clauses. However, TANKA's modules do not output HC, but parse trees augmented with semantic information. Therefore, a representation mapping module was required between TANKA and MALTE.

Figure 2.6 gives a graphical overview of where this representation mapping module fits with respect to TANKA, MALTE and my work. First, a textual NL input is processed sentence by sentence by the two TANKA modules: DIPETT and HAIKU. These modules look at the linguistic aspects such as parsing (DIPETT) and semi-automatic semantic analysis (HAIKU). The semantic analysis aspect is described in detail in Barker (1994) and parsing in Delisle (1994). I will simply mention here that semantic analysis involves assigning cases (Fillmore 1968) and establishing clause level and noun modifier relationships. Relationships as well as cases are assigned by a user from a limited list based on suggestions from the system.

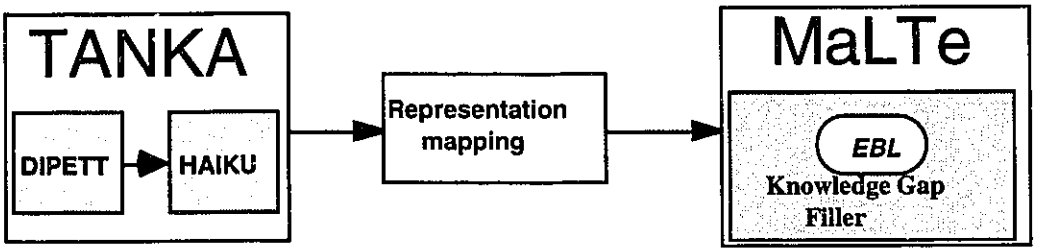

Figure 2.5: The three main modules of the TANKA/MALTE project: TANKA does the linguistic processing, then its output is mapped to a HC representation. MALTE uses ML techniques to improve the knowledge acquired. Within MALTE, a knowledge gap filler is required to catch EBL failures and learn missing knowledge.

The output of TANKA representing the current sentence is then passed to the representation mapping module for translation into Horn Clauses. This mapping is semi-automatic as it involves semantic interpretation. Once this is done, a HC representation of the sentence can be passed on to the MALTE module.

In the MALTE module, knowledge extracted from the text, now in a HC format, is accumulated as a domain theory. This DT is superficial because it lacks foundational knowledge (and may also be missing some domain knowledge). ML techniques are then

used to improve the DT. One of the ML techniques used is EBL. My work can be seen as a shell around EBL: a knowledge gap filler (see Figure 2.6). The DT, as we have seen, will exhibit knowledge gaps which are put in evidence by processing examples also originating from NL text. Then, the knowledge gap filler traps EBL failures and uses them as learning opportunities. When the required knowledge has been acquired, control is returned to EBL and the explanation of the example current continues.

Chapter 3 - Problem Statement

In this chapter, I will first define terms that are used throughout the remainder of this thesis. Then I will formally state the problem this thesis is tackling. Constraints that have a significant impact on the type of solutions possible will be discussed next. Finally, the working hypothesis as well as the assumptions will be presented.

3.1 Terminology

Rule: A rule is a Horn Clause (HC), that is, a logical statement with one consequent and one or more antecedent. A rule is used to define *concepts* (see below for a definition of concept) and to describe their relationships with other concepts. The rule which comprises the consequent C and the antecedents L_1 to L_n is expressed as:

$C{:-}\ L_1,L_2,...L_n.$

and reads: C is true if L_1 **and** L_2 **and**... **and** L_n are true.

Literals: The consequent C and the antecedents $L_1,L_2,...L_n$ of a rule are called literals.

Head: The head of a rule is its consequent C.

Body: The body of a rule is the conjunction of all antecedents $L_1,L_2,...L_n$.

Note: Although the head of a rule is also a literal, I will often use the term *literal* to refer specifically to a rule's antecedents, while I will use *head* to refer to the rule's head.

Predicate: Each component of a rule, whether its head or the literals in its body, are of the following form:

$p(A_1,A_2,...,A_m)$

where p is called a *predicate symbol* or for short, a predicate; and $A_1, A_2, ..., A_m$ are the predicate *arguments*. I will follow the usual Prolog convention of representing *variable* arguments by strings starting by a capital letter. *Constant* arguments will start by a lowercase letter.

Instantiation: The action of assigning an actual object to a variable argument. We say that a variable argument *has been instantiated* when this argument has become a constant.

Clause: In the thesis, I use the word *clause* to refer to either a HC or a *unit clause*. A unit clause is a HC with no antecedent and only one consequent.

Ground clause: A clause is *ground* if all its arguments are instantiated.

Fact: A fact is a ground unit clause.

Explanation Mechanism: An explanation mechanism is a Prolog meta-interpreter or EBL.

Task: The task of a NLP system is to acquire knowledge which will be *usable* by an explanation mechanism (c.f. section 2.2.1). The task of an explanation mechanism is to explain why an example is an instance of a DT concept (see the definition of "Explanation" below).

Prior knowledge: This is the knowledge which needs to be hand-coded prior to the NLP task. It includes general world knowledge, common sense knowledge, language knowledge as well as some domain-specific knowledge. It excludes however purely lexical or syntactic knowledge on which surface NLP relies. In Delisle (1994), the term *a priori knowledge* is used.

Foundational knowledge: Foundational knowledge is any knowledge that either humans or KBS (including NLP and ML systems) need to function properly. In our context, this knowledge can be available beforehand (it is then prior knowledge as well) or acquired as it is needed. As for prior knowledge, it includes general world knowledge, common sense knowledge and language knowledge. Domain-specific knowledge however is not foundational. According to Guha & Lenat (1994), from which the term is borrowed,

foundational knowledge is the general purpose knowledge we need to understand an encyclopedia text or that we assume others have when we interact with them.

Concept: A concept is defined by a HC. The definition of a concept is done in term of *sub-concepts*, which can themselves be defined, and so on. In this context, a rule (or HC) head becomes the concept being defined and its body the definition of the concept. Hence, the literals in the body represent sub-concepts in terms of which the concept represented by the rule's head is defined. A concept may have to be defined by more than one clause, in which case we say it is a *disjunctive concept.* Hence, clauses form an AND-OR tree of concepts (see Figure 3.1), where the root of the tree is called the *top concept.*

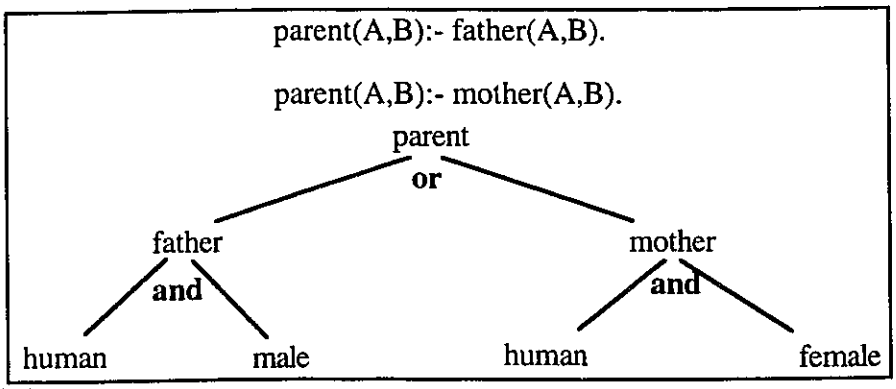

Figure 3.1: A disjunctive concept, its clause and its tree representation. The concept parent is defined as a relation between two variables A and B such that A is the parent of B if A is the father of B, OR if A is the mother of B.

Domain Theory (DT): A DT is a set of HC that define concepts of a given domain and establishes relationships between these concepts.

Example: An example is a conjunction of ground clauses which forms an *instance* of a DT concept. Examples in this thesis are therefore always positive examples. An example is usually an instance of the top concept. Examples are obtained by processing a short NL story via surface NLP. The short story states a particular *situation* to which the domain theory applies.

Explanation: The word explanation is used similarly to the EBL sense, with the difference that in our case we are neither interested in how this explanation proceeds, nor in how we can use it to learn an operational definition of the concept, nor in how we can generalize this new definition. We are rather interested in *where* and *why* the explanation fails. Hence, it is not EBL per se that is important, but the underlying inference mechanism

that allows for proofs to fail when knowledge is missing. That is, an explanation is the *proof* that an example is truly an instance of a DT concept. We say that the explanation *fails* when we cannot prove that an example is an instance of a DT concept.

Explanation failures: These are potentially caused by *missing knowledge* since examples are all positive examples. The missing knowledge is the knowledge we lack to prove that an example is indeed an instance of a DT concept.

Knowledge gap: A knowledge gap occurs at the point where the explanation fails and consequently, it corresponds to missing knowledge.

Type hierarchy: A type hierarchy is a tree of of types. The higher a type is in the hierarchy, the more general it is; and the lower it is, the more specialized it is.

3.2 Formal Problem Statement

We have seen in chapter 2 that the use of surface NLP resulted in a DT which lacked exactly the foundational knowledge which is usually hand-coded prior to doing traditional (i.e. non surface) NLP. Then, we saw that in order to be useful, a DT acquired by surface NLP was in need of a mechanism to acquire the missing foundational knowledge: in other word, a knowledge gap filler. Finally, within the MALTE project, it was realized that EBL failures, since they happen exactly were knowledge is missing, could be used as learning opportunities to acquire the required foundational knowledge.

3.2.1 The problem

Given:

- a superficial domain theory T obtained from surface NLP
- a set $E = \{e_1, e_2, \ldots e_n\}$ of examples
- an incomplete type hierarchy S
- an initially empty knowledge base FK
- an explanation mechanism M

Task:

- explain each $e_i \in \mathcal{E}$ with M and $\mathbf{T} \cup FK$.
- find all knowledge gaps

Incrementally learn:

- a foundational knowledge base FK such that all $e_i \in \mathcal{E}$ can be explained with the knowledge in $T \cup FK$.
- the rest of the type hierarchy S such that all objects in $\mathcal{E}$ can be appropriately typed.

3.2.2 The Sub-problems

The problem stated above can be decomposed into five sub-problems. They are listed below along with a brief description. A complete description will be given in section 4.2.

a. Find the *failing clause* and *failing literal*: These are respectively the Horn clause and the first literal within the Horn clause body on which EBL fails. The failing literal is where the knowledge gap occurs.
E.g.: supporting(joe,son):-parent(joe,son). is the failing clause and parent(joe,son) is the failing literal in Figure 3.2.

b. Find the *proof-continuation clause.* This is the DT or example clause which should have allowed for the continuation of the explanation. In other words, this clause contains the information required so that the explanation can proceed. The knowledge gap is between this clause and the failing literal.
E.g.: father(joe,son) is the proof continuation clause in Figure 3.2.

c. Learn new knowledge: The new knowledge allows to bridge the knowledge gap. It includes new rules and facts which are added to the KB of foundational knowledge and used on other failures.
E.g.: parent(A,B):- father(A,B).

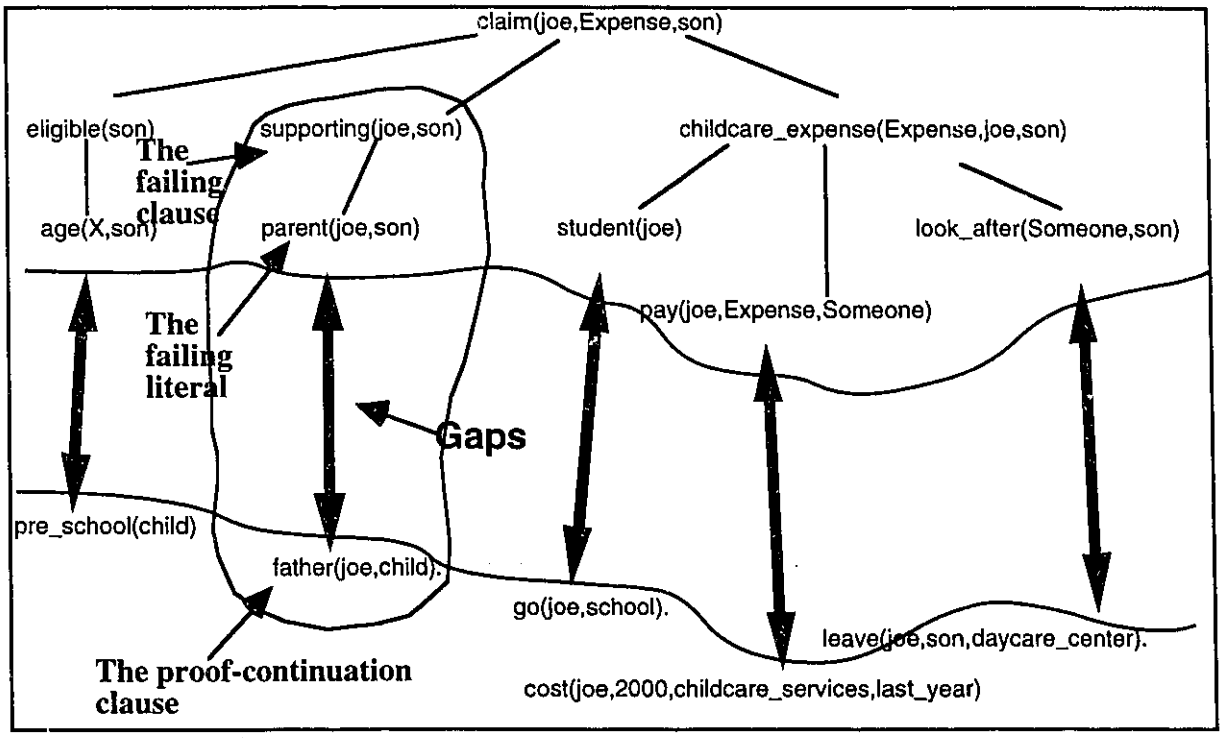

Figure 3.2: Failing clause, failing literal and proof-continuation clause shown in a proof tree.

d. Generalize/specialize the new rule: When a new rule has been learned, its use has to be appropriately constrained by assigning a type to each argument. The type must be chosen at the appropriate level of a type hierarchy.
E.g.: parent(A,B):- father(A,B), isa(A,human), isa(B,human).

e. Use the new knowledge: As new knowledge is acquired, it is used on other failures before other new knowledge is acquired.

3.2.3 Constraints

The problem came with a set of constraints that had a major impact on the solution. In this section, I will then talk about "the system" or "the solution" even though neither the solution nor an actual system implementing this solution have been presented yet. This will allow for a better explanation of the impact and rationale of the constraints.

C1: No knowledge can be hand-coded beforehand. The rationale behind this constraint is that the problem is about the acquisition of foundational knowledge, not

about the hand-coding of yet some more fundamental knowledge. This would simply be pushing the problem further away. This constraint has the consequence that the acquisition of the missing knowledge has to come either from a very elementary ability to learn from experience, or from a human. By elementary I do not mean that the learning process has to be rudimentary. On the contrary, the word elementary here refers to a quality of "innateness" where learning depends not on prior knowledge but on the general ability to recognize patterns, draw the right conclusion and generalize appropriately. Such a general purpose machine learning mechanism has yet to be invented. In fact, the only such good learner known is the human, and it is the human who also happens to be the best source of knowledge in our case. Therefore, this constraint implies a semi-automatic solution where the user must provide the knowledge when the system does not have it. However, relying on a human user to provide the missing knowledge should not mean that the user does all the work. The next two constraints ensure that this is taken care of.

C2: Human involvement must be kept simple and at a minimum. *Open ended questions* should be asked as seldomly as possible and a maximum use of *ranking/filtering* must be done when presenting choices to the user. An open ended question is a question which requires that the user comes up with the alternatives and then enters the one he selects. It is less demanding to a user if the system comes up with the alternatives and simply asks the user to select among them. Ranking/filtering is the process of ranking the alternatives from most plausible to less plausible, and then eliminating the most unlikely options. Thus, a limited set of prefered options is presented to the user first. Then, if the user rejects the first set of choices, the system proposes the next most likely set of options, and so on until no more options are left or until the user selects one. The rationale behind ranking/filtering is that it is thought to be better to ask a few questions with a limited number of choices than only one questions with an unmanageably large quantity of choices.

C3: The user involvement must decline with time. As the system handles new cases and fails on them, it will accumulate new knowledge. The system must be built to make maximum use of the currently available knowledge and thus constantly aim at reducing its dependence on the user. A system that would not become more autonomous with time would most likely exasperate the user with its numerous questions.

C4: The new knowledge must be correct. The ultimate aim of the system is to fill the knowledge gaps so that the KB acquired from text can be used for *real-world* applications. Therefore, we must ensure that the knowledge we acquire is as correct as it can be given a knowledgeable human user (who can still make mistakes). When the system does not know, it must ask for help rather than try to make a guess on its own. If the system takes a chance, it is only to tell so to the user and ask him to confirm or reject the system's choice.

C5: The solution has to be domain independent. This constraint comes from the requirement that we do not want to implicitly code domain specific knowledge in the solution: this would amount to disguising prior knowledge. Although on one hand, this would allow for the use of powerful domain-specific heuristics, on the other hand, it would simplify the problem in a way I find too restrictive. Domain independence results in increased flexibility and more scientific generality.

3.3 Hypothesis

Constraint C3 requires that the system becomes more autonomous with time. This implies that the sum of the knowledge acquired would converge toward a complete KB of foundational knowledge for the specific problem/domain for which examples are being processed.

Hypothesis: Given a sufficiently large number of examples, we can train a KA system in such a way that the resulting KB converges towards an *acceptably complete* KB of foundational knowledge.

By acceptably complete, I mean that the KB will be able to explain a new example with a certain probability. The idea behind acceptable completeness is similar to Probably Approximately Correct (PAC) learning (Valiant 1984). In our case however, it involves a Probably *Acceptably Complete KB* instead of a probably approximately correct single concept definition. Hence, we are dealing with *multiple* concepts definition, and are interested not in whether or not they are correct (since they are assumed to be correct: see constraint C4) but rather in the degree of completeness we have acheived with their acquisition. I refer to this idea as PAC-KB. In PAC-KB, the *error* is defined as the

probability of the KB not being sufficiently complete to allow for the explanation of a new example.

The aim is to set an error threshold that is acceptable (the *acceptability parameter*, similar to PAC's epsilon), as well as a *confidence parameter* (similar to PAC's delta). Then, by processing a sufficient number of examples, one has a guarantee to meet this acceptability parameter with the given confidence. In other words, if the KB is probably acceptably complete, a new, yet unseen example will be explainable by the KB in accordance with the given completeness and confidence parameters.

The basic assumptions behind PAC-KB are similar to the ones of conventional PAC. If a learner processes enough training examples that are representative of the situation space (i.e. the space of all possible situations that can occur for this task and domain), then, given any new example from a test set, where the test set was independently sampled according to the same probability distribution as the training set, the KB obtained from the training set will be a PAC-KB. The interesting property of this PAC-KB is that it will be able to explain any new example with a probability given by the acceptability parameter and a confidence given by the confidence parameter.

A PAC theory for KB is beyond the scope of this thesis. As pointed out by De Raedt and Bruynooghe (De Raedt and Bruynooghe 1992b), the study of learning complexity in the context of interactive and incremental learning of a whole KB is a difficult task which has usually been better described within the learning in the limit framework (Gold 1967). The introduction of the PAC-KB idea may prove useful in the study of learnability by merging the frameworks of PAC and in-the-limit.

3.4 Assumptions and Limitations

3.4.1 Simplifying Assumptions

The following assumptions were made for the time being, with the intention to explore the underlying issues in future work if these assumptions happen to be found too restrictive:

a. Information in examples is consolidated. For example, if an example involves the presence of many children and that information is given about each child individually (such as to the amount of child care paid for each and their respective age), only one child situation can be explained and missing knowledge learned for it. If we want to process the other children to learn about their specific case, each must be processed separately; therefore eliminating the "big picture" view of a particular situation/example.

b. All values are at the same scale and computations are not required. For example, an example may state that X dollars were paid monthly for child care while the DT may require a yearly amount. No attempt will be made to acquire the knowledge required to recognize this situation and to compute the right value.

c. Surface NLP input. When I started the design of the solution, the module converting between the TANKA representation and the MALTE representation was not fully implemented. I therefore made two assumptions as to the kind of input the knowledge gap filler would receive. The first one is that there would be no embedded lists or functions as predicate argument (flat structures). Secondly, I assumed that the examples would consist of ground unit clauses only.

d. No multiple explanation exists. The first valid explanation of the example is assumed to be the only one.

3.4.2 Limitations

The following points were not investigated in this thesis:

a. Linguistics aspects. This work is not about NLP as an end, but about acquiring the missing knowledge from a KB built by surface NLP. Surface NLP and the mapping to a logic representation themselves are not tackled as problems in this work. I merely use the current results of research in these areas along with their shortcomings and still unexplored areas as starting point for my work. For the time being, I must make assumptions as to the outcome of these research areas. When I make these assumptions (cf. section 3.4.1 above, assumption c.), I try to make them as little limiting as possible so that whatever the outcome of surface NLP research

and its logical representation is, the work I present here will still be valid (with minor modifications in the implementation, but hopefully not conceptually).

b. Software development. This work was not intended to be a software development or software engineering effort. Although some software was developed, it was done for the purpose of serving as a proof of concept that would support the ideas presented here.

c. Meaning and representation issues. It is not an objective of this work to look into the complex and philosophical issues surrounding these topics.

d. Knowledge Engineering. Issues other than the acquisition of knowledge are not covered in this work.

3.5 Motivation

Ultimately, what makes solving this problem worthwhile is that it allows us to acquire knowledge from text. Since most of the human knowledge is preserved in textual format, and since many such texts are now available on-line (or can become relatively easily available on-line with the advances in the areas of Document Processing and Optical Character Recognition), it would be a major advantage and a noteworthy advance if we could translate texts at least semi-automatically into the production rules of a KB.

For example, school textbooks contain a large amount of the knowledge required for humans to practice professions or trades in domains such as medicine, law, engineering, etc. Written regulations, procedures, specifications and many other documents in corporations and governments dictate the whats, hows and whens of employees and citizens.

Therefore, it would be very beneficial if organizations could turn the knowledge they already possess in textual format into a representation that computers can easily manipulate. For example, an organization could use its administrative regulations as the source of knowledge to build the KB for an expert system (ES). This ES could then assist the staff in their work. Such an ES would have many advantages that include more consistent application of regulations and increased efficiency of the staff. In our times of financial

constraints and global competitiveness, increased efficiency can be translated into improved service or sales even as staff is reduced.

Today, ES development may be seen as a complex and costly endeavour for which the cost-benefit is not obviously advantageous. By eliciting knowledge from existing text, we effectively eliminate the requirement for expert interviews, which is time- and money-consuming, and also often disruptive to a working environment (unless done off-premise with a specially hired expert). Hence, KA from text makes KA more affordable and more feasible.

Another motivating aspect is in the context of Information Retrieval (IR) technology. For example, the Canadian Forces administrative regulations (known as the CFAOs) contain around 7000 pages of regulations on various subjects from academic upgrading entitlements to sports. Although the CFAOs are supposed to help and guide managers and administrators in their daily work, it is sometimes difficult to find the relevant information from the enormous paper version. The CFAOs have recently been made available on CD, and with Information Retrieval (IR) technology, the retrieval of the right information is now an easier matter.

IR, however, is based on indexing of keywords and probability of occurrence of individual words and/or word pairs (Jacobs 1992). IR would be much more efficient in term of both *recall* (percentage of relevant texts found) and *precision* (percentage of text found that are relevant) if it could rely on a deeper understanding of the text. However, we have seen that up to now, that was only feasible at the expense of a major prior knowledge hand-crafting effort. Even once this is done, the success can only be measured on a relatively small corpus of texts on a specific domain. NLP-based IR is brittle and domain dependent, just as NLP in general.

KA from text and NLP-based IR are thus two challenging tasks that could benefit enormously from a better way (than prior hand-crafting) to acquire foundational knowledge.

A final motivating aspect has to do with the Cyc project (Lenat & Guha 1990). Cyc tackles the same general problem of foundational knowledge acquisition, but it aims at acquiring most foundational knowledge for all domains and problems while in this work I target problem specific foundational knowledge. Cyc is done with the aim of addressing the problem of brittleness in KBS, but also, more recently (Guha & Lenat 1994) with the aim of

building *intelligent agents*. As demonstrated in the Cyc literature, the acquisition of foundational knowledge is of primary importance for the construction of KBS and intelligent agents.

The hypothesis behind Cyc is that the availability of a large KB containing most of human foundational knowledge would enable intelligent behavior, and more specifically NL understanding and learning. Before such a large KB become a matter of reality, ML and NLP are not useful. Therefore ML and NLP techniques cannot be used to help acquire the knowledge Cyc aims at acquiring: the knowledge must be acquired manually. This is why researchers in the Cyc project have said that there is "no free lunch" (Lenat & Guha 1990 p. 24).

Cyc has addressed the NLP second issue (section 2.2.4.2) of the prior knowledge problem by claiming it will have acquired so much knowledge that:

a. when a new situation is encountered, enough intelligence will result from the Cyc KB that the system will be able to handle it; and

b. enough knowledge will be present to learn the required missing knowledge from experience.

The first issue (section 2.2.4.1) however is not addressed: Cyc is of the prior hand-coding paradigm like all the other approaches, except that it does it on a universal scale.

Hence, with reference to Cyc, it would be advantageous if we could find a better way (it has been ten years of work since the Cyc project started) to acquire foundational knowledge. To find an alternative to Cyc is therefore a motivating factor since, as Minsky (Minsky 1994) stated, there has been no alternative proposed to Cyc: most researchers are waiting to see the outcome of Cyc.

Chapter 4 - Solution to the Problem

We have seen in the previous chapters that a DT built by extracting knowledge from a text with surface NLP techniques will lack foundational knowledge. We have also seen that when an explanation mechanism (for example, EBL) is used to process examples with this DT, the explanation fails on knowledge gaps. These knowledge gaps correspond to the missing foundational knowledge.

In this chapter, I will describe how these failures can be used as opportunities to learn the missing foundational knowledge. Section 4.1 will present, from a conceptual point of view, an overview of the approach. Then, section 4.2 will give a detailed description of FOKAS, the foundational knowledge acquisition system I have developed and which implements the ideas presented in section 4.1.

4.1 Overview of the Approach

4.1.1 The Lazy KA Method

As we have seen in chapter 2, prior handcoding of foundational knowledge gives rise to two difficult and yet to be resolved issues. We therefore have to find a better way to acquire foundational knowledge. Additionally, our solution must stay within the constraints stated in section 3.2.2. The method I propose in this thesis acquires foundational knowledge semi-automatically and only when a performance task (in our case, the explanation of examples) requires it. I call this method *Lazy KA*[8].

Lazy KA is essentially based on the observation that an explanation mechanism will fail on knowledge gaps that correspond to missing foundational knowledge (see section 2.3.2.3). These failures are turned into learning opportunities. That is, when it is impossible to explain how an example satisfies one of the DT concept definitions, Lazy KA assumes that it lacks knowledge.

[8]The word *lazy* is used in the usual computer science sense of postponing until we really need to do something.

What is left to do at that point is to fill each knowledge gap as it is made evident by a failure to explain an example. However, an important question must first be answered: Where do we get foundational knowledge? As mentioned in section 3.2.2 with constraint C1, the only recourse in this case is to involve a human user. Moreover, based on constraint C2, the user must be able to do this with as little effort as possible. Such a system can be described as an intelligent agent that cooperates with a human user to acquire the foundational knowledge it lacks.

4.1.2 Basic Ideas Behind Lazy KA

Underlying Assumptions. The first assumption on which this approach rests is that a relationship must exist between one of the example's feature and the DT concept for which a failure occurred. If this does not occur, the example is considered noisy as it lacks any form of useful information that can be used to distinguish it from a negative example. (Note that there are ways around this situation. For example, the KA system can ask the user about the status of the example). The second assumption is that a cooperative human user must exist, and that this user will be able to recognize the relationship between one of the example's feature and a DT concept (as discussed in the first assumption above). If the user cannot recognize the meaning behind the symbols that stand for the example's features and the DT concepts and, as a consequence, the user cannot establish the required relationship, no knowledge can be acquired. A third assumption is that examples can be collected. Indeed, it is not an obvious matter to collect examples for Lazy KA because these examples must be provided *in writing, by people*. This first implies that enough people willing to write a short story relevant to the DT can be found; and second that the domain examples can be *verbalized*. The latter will not always be possible: for example, one cannot really verbalize outer-space infra-red background radiation data in the domain of astrophysics. A last assumption is that the text from which the DT originates is an almost complete description of the domain. A very incomplete description of the domain would result in many domain knowledge gaps rather than mainly foundational knowledge gaps. Although Lazy KA allows for the acquisition of missing domain knowledge, this work aims primarily at the acquisition of foundational knowledge, not of domain knowledge.

Failures as Missing Knowledge. A Lazy KA system cannot assume that the world is closed (Closed World Assumption: CWA) nor can it assume that different symbols have different meanings (unique-names assumption). Instead, it must assume that two different

symbols may be equivalent (representing the same object, action, situation...) and that a failure may in fact only be an indication that knowledge is missing. This method of KA must therefore rely on a human to handle failures and/or symbol mismatches. The effect of such a human dependent approach is that much of the power of conventional computer inferences (automatic theorem proving for example) in the context of the CWA and unique-names assumption is lost. This loss however is counter-balanced by the gain of a very interesting source of knowledge. Moreover, the CWA and unique-name assumption are often not realistic as they require careful hand-crafting and preparation when performing problem solving. Indeed, one must ensure that symbols are used consistently and that the KB is complete before applying these assumptions. However, in real world problem solving, these two criteria can rarely be met. Moreover, when they can be met - and there is never a guarantee of this - it is at the expense of long and tedious preparation. Lazy KA allows to avoid such preparation, and by avoiding preparation, it allows for the acquisition of useful foundational knowledge.

Constraining the KA Process. The new knowledge acquired is foundational knowledge that would have required prior handcoding if we had been using traditional knowledge-intensive NLP methods. With lazy KA though, this knowledge can be acquired when and only when it is made evident by the task. By proceeding in this manner, we provide a *realistic constraint* on what knowledge should be acquired, effectively eliminating the need to think of most situations beforehand (cf. issue 2, section 2.2.4.2).

Domain Independence. Lazy KA is totally domain independent and draws its power from the knowledge the system has of its own internal functioning. That is, the KA system knows how its explanation mechanism works and uses that knowledge to help acquire missing knowledge when the task fails.

Language First, Knowledge Afterwards. The DT originates from a *domain text*, and each example originates from a different *example text*. Based on the problems with NL seen in Section 2.2.2, it can be expected that an example text could use a different form and vocabulary than the domain text, and that it could also only state implicitly the information it aims at conveying. Due to surface NLP, these problems with the text will result in knowledge gaps between the DT concepts and the examples concepts. These gaps can then be used as learning opportunities to obtain foundational knowledge. Therefore, since an example aims at *illustrating* the domain concepts, rules and relations, it is the inherent differences between the two original set of NL expressions which form the domain text and an example text that

make the learning of foundational knowledge possible with Lazy KA. Hence, the richness of NL encodes foundational knowledge. This means that the difficulties with NL can be turned into advantages and that language itself (or rather the differences in the ways people use language) can be used to acquire foundational knowledge. This is partially in contradiction with the conventional idea that knowledge is required to understand NL. Now, we can say that *superficially understanding* language helps to acquire knowledge.

Knowledge Sources and Learnability. The example set is the primary source of knowledge. Indeed, the examples provide ground clauses that can be related to DT literals to create new concept definitions. Hence, the search for the missing knowledge often becomes a matter of finding the example ground clause[9] the failing DT literal must be linked to. This effectively establishes a relation between a DT concept and a new concept *contributed by the example*. Such a relation is foundational knowledge we aim at acquiring. The user helps in all semantically motivated decisions since there is no other assumed source of knowledge. As such, the user is the only other source of knowledge. The resulting new knowledge is accumulated in a foundational KB. As the amount of new knowledge grows, the system can rely more and more on itself rather than on the user. Such a learning system can normally be expected to learn in the limit (Gold 1967, De Raedt & Bruynooghe 1992b), but we have seen in Section 3.3 that learning in our context can also be characterized in a manner similar to Probably Approximately Correct (PAC) (Valiant 1984).

4.1.3 General Architecture

To conclude this presentation of the conceptual overview of the solution, I present a general architecture for KA from text and for the lazy acquisition of foundational knowledge (see Figure 4.1).

[9] I say "often" for the sake of generality because we may have a domain gap where the gap is between two non-ground HC of the DT.

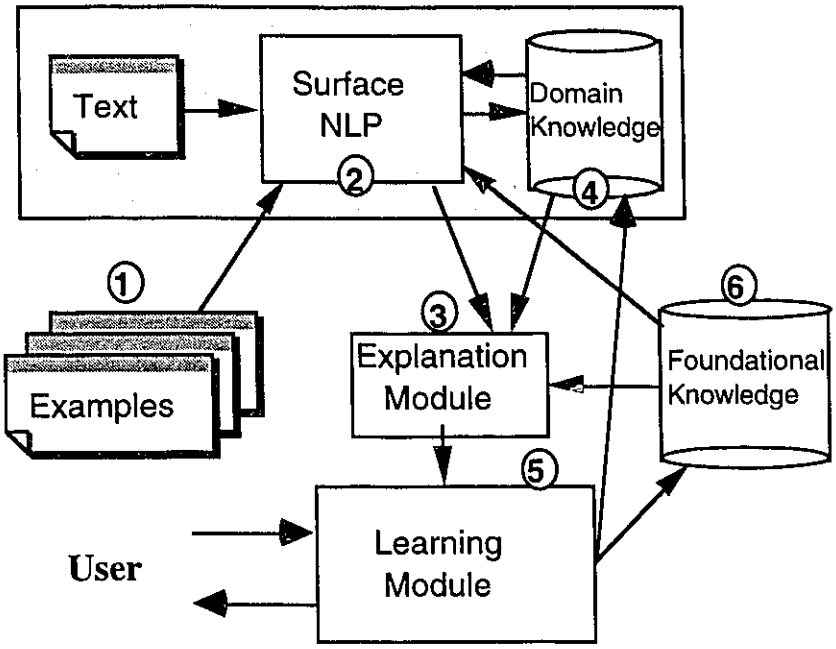

Figure 4.1: A general architecture for KA from text and for Lazy KA of foundational knowledge

In such an architecture, text examples (1) are processed by a surface NLP unit (2) and then submitted one by one to an explanation module (3) which attempts to explain the example in terms of the domain knowledge (4). This domain knowledge was obtained from surface NLP of a text (gray box on top). When an example cannot be explained, the task fails in a controlled manner. Failures are handed over to a learning module (5) which guides the user through the acquisition process of the missing knowledge. The resulting new knowledge can be either domain knowledge (4) or foundational knowledge (6). The new knowledge is then made available to the explanation mechanism and also to the surface NLP unit which can thus perform deeper and better NLP.

4.2 An Implementation of Lazy KA

This section will describe the specific implementation I did of the above ideas. The implementation is a prototype system called FOKAS (FOundational Knowledge Acquisition System) written in Quintus Prolog on a Sun workstation. The main purpose of the prototype was as a proof-of-concept for Lazy KA. In particular, it was used to demonstrate the validity of the approach in regards to the convergence hypothesis presented in section 3.3. The experimental work will be presented in chapter 6.

In section 3.2.1, I identified the following sub-problems:

a. Find the failing clause and literal;

b. Find the proof-continuation clause;

c. Learn new knowledge;

d. Generalize/specialize the new rule; and

e. Use the new knowledge[10].

In this section, we will see how each of these sub-problems are solved within FOKAS. For this purpose, section 4.2 is organized in sub-sections titled after the names of the sub-problems. Each sub-section describes how a specific sub-problem has been solved. I will use the word *module* to refer to each of the sub-problem solution because they correspond to different modules in the FOKAS implementation.

However, before we can look at the solution of the sub-problems, I must first clarify how the explanation mechanism handles failures and triggers learning.

4.2.1 Explanation Mechanism

The explanation mechanism is FOKAS' central component: it is the one that calls the other modules by trapping failures. It is also the one to which control is returned when learning has been completed, or when it has failed.

4.2.1.1 Algorithm Description

The explanation mechanism algorithm is given at Figure 4.2. What follows is a line by line explanation of the algorithm.

---

[10]Note that the first four sub-problems are related to learning the missing knowledge while the last sub-problem is not. The solution of this sub-problem is rather what allows the system to *avoid* learning when the system already knows how to handle the current failure.

0: The first input is a *goal clause* , that is a clause at least partially instantiated with constants of the example. This clause represents the DT concept we want to explain. The other two inputs are an example set as well as a DT both obtained from text by surface NLP.

1: Examples are presented serially to the explanation mechanism. FOKAS does not learn inductively from the example set as a whole. Rather, it aims at identifying the knowledge gaps one by one, and then filling them, with the help of a user, as they are made evident by the examples.

1.1: The list of local failures LF is set to empty for each new example because failures are relevant only to the example for which they occur.

0: Inputs: A goal clause
A set of examples E= $\{e_1, e_2, ..., e_m\}$
A domain theory T

1: For each example e_i in E
1.1: LF:= [ ] *(list of local failures set to empty)*
1.2: Until e_i is fully explained or learning fails
1.3: **explain** e_i
1.4: if fail on a literal L_k of a clause C: H:- L_1, L_2, ..., L_k, ..., L_m.
1.4.1: determine whether local or terminal failure
1.4.2: if local failure
use new knowledge to prove L_k *(sub-problem e.)*
if this fails
append [H, L_k] to the list LF of local failures
backtrack: try alternative explanations

1.4.3: if terminal failure
find the failing clause and literal in LF *(sub-problem a)*
find the proof-continuation clause in T U e_i *(sub-problem b)*
build the new rule and learn other kind of K *(sub-problem c)*
generalize/specialize the new rule *(sub-problem d)*
LF:= [ ] *(reset list of local failures)*

2: Output: A KB of foundational knowledge
A type hierarchy
Other various knowledge elements

Figure 4.2: Explanation Mechanism Algorithm and associated sub-problems.

1.2: An example may contain several knowledge gaps. Therefore, the explanation of an example must continue until all of them have been identified and filled. Once this is done, the example will be explainable because the required missing foundational knowledge will have been accumulated. On the other end, if the acquisition of new knowledge fails, the

explanation of the example is stopped and the user is advised that there was a problem with the example.

1.3: **explain** is the explanation mechanism itself. Originally, as we have seen in chapter 2, the explanation mechanism was supposed to be EBL. However, for the prototype FOKAS, the interest was not in EBL itself (that is learning new, operationalized and generalized concept definitions based on the explanation of an example), but rather in where an explanation would fail given an incomplete DT (mainly in term of foundational knowledge). Therefore, **explain** ended up being implemented as a simple Prolog meta-interpreter which explores a DT depth first, with the first explanation (or proof) found assumed to be the only valid one (see simplifying assumptions, section 3.4.2 paragraph c.). However, when this meta-interpreter fails to prove something, the failure must be trapped and used as a learning opportunity rather than resulting in an actual failure of the proof (step 1.4).

1.4: When **prover** cannot match a DT literal's predicate symbol or one of its constant argument with a ground clause, it fails. It fails on a literal L_k of a clause C. As soon as such a failure occurs, it is trapped by FOKAS before attempts to backtrack are made.

1.4.1: When a failure occurs, FOKAS must determine if there are alternative explanations. When there are, one of them *could* be successful, in which case no learning would be required. On the other hand, if there is no alternative, then FOKAS has identified a knowledge gap and it must learn new knowledge. The algorithm implements two levels of failures to determine whether it is time to learn: the *local failures* and the *terminal failures*. A local failure is a failure occurring before the prover has explored all alternatives. Such failures can therefore be followed by a successful proof by using other branches of the DT. A terminal failure however occurs when there are no more alternatives. The proof of the example is therefore impossible given the current state of knowledge, and learning must take place to continue the explanation. By looking ahead of the explanation, FOKAS determines what type of failure has occurred.

1.4.2: If it is determined that a local failure has occurred, the first thing the algorithm does is to use the current KB of foundational knowledge (the new knowledge FOKAS has acquired up to now from previous failures) to see if the current failure could be resolved by that knowledge[11]. If this does not work, then the pair (L_k, H) is added to the list LF of local failures (where L_k is the literal which could not be proved in the body of the clause C and H

[11] See section 4.2.6 for a description of how this is done.

the head of C). Then and only then is backtracking for alternative solutions permitted to occur (in a traditional explanation mechanism such as EBL, this backtracking would have occurred as soon as the failure was detected). By proceeding this way, FOKAS can keep track of all local failures. This is useful because it is not possible to know whether a terminal failure will indeed occur. If one does occur, then a list of all partial explanations is available: the list LF. From this list, one of the partial explanations can be selected as the one which should have completed. On the other hand, if a terminal failure does not happen for a particular concept of the DT (i.e. the concept *is* explained), the list LF can be reset so that the next concept for which an explanation is attempted can use it to store its local failures.

1.4.3: If all possible alternatives have been exhausted, a terminal failure has occurred. At this point, since FOKAS assumes the presence of positive examples only, it concludes that the failure is caused by lack of knowledge. FOKAS must therefore learn the missing knowledge, and the learning modules are called.

2: The output of FOKAS includes two main elements: a KB of foundational knowledge and a type hierarchy. The KB of foundational knowledge is made of HC (that is, Prolog rules) and of *equivalent constants relations* (these are actually Prolog facts of the form: equiv(a,b) where a is a constant which refers to the same object as b does). The type hierarchy is acquired during the generalization/specialization process (see section 4.2.5). A third output consists of various knowledge elements: these consist mainly of relations such as part_of, isa, etc. All the outputs will become clearer in subsequent sections.

4.2.1.2 Example: Explanation Mechanism

Given a DT (a simplified version of what has been used in chapters 2 and 3):

```
childcare_expense(Expense,Person,Child):-
      work(Person),
      pay(Person,Expense,Someone),
      look_after(Someone,Child).
childcare_expense(Expense,Person,Child):-
      student(Person),
      pay(Person,Expense,Someone),
      look_after(Someone,Child).
```

and the example:

```
go(joe,school).
```

```
cost(joe,2400,childcare).
leave(joe,son,daycare_center).
```

and the currently accumulated KB of foundational knowledge:

```
live(A,B) :- move(A,B), isa(A,person), isa(B,location).
work(A) :- receive(A,B), isa(A,person), isa(B, employment_revenue).
```

We then provide FOKAS with the goal clause:

```
?- childcare_expense(Expense,joe,son).
```

which means that we want to prove (or explain) that Joe can claim child care expenses for his son, and determine the amount of that expense. The goal clause is therefore partially instantiated with the constants joe and son, while the variable Expense is left uninstantiated because we want to determine the amount of the expense.

The explanation will first match the goal clause with the first rule head in the DT. Then, it will attempt to prove work(joe), the first literal in the body of the rule. Since it is unable to do so (there is no clause that can be matched), the KB of foundational knowledge will be consulted. There, a rule that could potentially explain work(joe) is found: work(A) :- receive(A,B), isa(A,person), isa(B, employment_revenue). However, the explanation with this knowledge is not possible because there is no clause with the predicate receive. We therefore need to add this local failure to the list LF (c.f. Figure 4.2):

LF = [[childcare_expense(Expense,joe,son) , work(joe)]]

The prover is then allowed to backtrack. The second clause for the concept childcare_expense is then selected, but failure occurs again, this time on the literal student(joe). In this case, there is no potentially useful new knowledge.

We then have two local failures right from the start, at the level of the DT top concept childcare_expense. Since there is no other alternative to prove the example, we come to a terminal failure.[12] The list of local failures then becomes:

[12]There will be more terminal failures in other parts of the DT, but the system will first address this one, fix it, and then continue the explanation and encounter the other failures. For the purpose of this example, we will only look at the first terminal failure.

```
LF = [ [childcare_expense(Expense,joe,son) , work(joe),
        childcare_expense(Expense,joe,son) , student(joe) ] ]
```

The rest of the processing for this example will be illustrated in the subsequent sections as each sub-problem is addressed.

4.2.2 Finding the Failing Clause and Literal

The aim of this module is to determine which local failure currently stored in the list LF should *not* have occurred. In other words, among all the possible partial explanations, one should be valid, but failed to be due to missing knowledge. Therefore, the algorithm must determine where the knowledge gap is. This is achieved by finding the literal which should have been proven given the required knowledge. This literal is called the *failing literal* and the clause it belongs to is called the *failing clause*.

4.2.2.1 Algorithm Description

The top level algorithmic view of a solution to this problem is given in Figure 4.3 a. The explanation of this algorithm follows below.

0: The list of local failures received as input contains all Horn clause (HC) heads and the *first* literal of their body for which the explanation failed. The presence of the HC head in the local failures list merely serves as a form of context for the user. In this manner, the literals for which there was a local failure are not presented alone. If they were presented alone, it could be difficult for the user to distinguish among them and to select the right one. On the other hand, although the whole clause could be presented, this was found to add unnecessary information for the user. Generally, all the required information for a good decision by the user is available with the HC head and the first failing literal.

1: At this point, using the currently accumulated foundational knowledge has failed (see the previous module, section 4.2.1). In order to find the failing clause, the system can thus only rely on the user since we assume no other source of knowledge (constraint C1, section 3.2.2). The user is therefore presented the list LF of local failures in a more readable format (for example, in a HC format: H_i :- L_j) and asked to select the clause that *should have been explained*. Alternatively, instead of the user, we could have used heuristics, based for

instance on the "level of completeness" of the proof, in order to select the failing clause. We could have assumed for example that a longer or a shorter proof was closer to completion, and selected the local failure associated with this proof as the one involving the failing literal. However, initial experiments with this approach showed that such a view leads to rather uncertain choices since they are based strictly on syntax while the task is highly semantic in nature. Based on constraint C4, such decisions would therefore have to be approved by the user anyway. The use of more semantically based, domain specific heuristics could have helped, but constraint C5 states that no domain specific heuristics can be used. Therefore, asking the user to select up front the right clause is the only choice left.

0: Input:
List of local failures LF = [[H_1, L_r], [H_2, L_s], ... , [H_p, L_t]]

1: Ask the user which L_k in the pairs (H_a,L_k) should have been proved
1.2: the selected pair (H_a,L_k) gives the failing literal L_f:
$L_f := L_k$
1.3: mark DT proof paths corresponding to other local failures as bad branches for this example

2: if none selected:
try other levels with [H_1, H_2,... , H_p]

Figure 4.3 a: Algorithm for "Finding the Failing Clause and Literal"

1.2: By selecting one of the pairs (H_i, L_j), the user recognizes that the literal L_j should not have failed. This means that the example is an instance of the DT concept represented by the clause in which H_i and L_j are found. This effectively identifies the failing clause and literal to which the knowledge gap corresponds. The failing literal is passed on to the next module for further processing. The other failures are not useful as learning opportunities.

1.3: The fact that the failing clause has been identified as the DT clause that should be used for the explanation effectively eliminates all the other clauses which were involved in local failures as valid paths in the DT. The sub-trees corresponding to these clauses are therefore marked so that the explanation mechanism does not try to use them again for this example's explanation[13]. Hence, the system will not ask questions about explanation paths that we know are not valid.

---

[13] Actually, the current implementation does go through these sub-trees, but it does not question the user about local failures in them. These local failures are not useful because a failing literal will never be selected among them since they correspond to parts of the DT that are not valid to prove the current example.

2: If the user does not select a failing clause, then **try other levels** is called. This allows for an escalation to the next higher level of the DT to possibly learn new knowledge at that level. As long as the user does not select a failing clause, the algorithm will go up in the DT. It will stop when the top concept is encountered or when the user selects a failing clause. The algorithm for **try other levels** is shown in Figure 3.3 b and is explained below.

```
try other levels

2.0: input: [ H1, H2,... , Hp ]

2.1: if no Hi:
        exit try other levels: bad example

2.2: for each Hi
        ask user if Hi is true
        if true
           Lf := Hi
           exit try other levels

2.3: if no Hi selected as true
           P := [ ]
           for each Hi
                 get parent in DT tree
                 if parent exists
                    if parent not already in P
                          append parent of Hi to P
           try other levels with P
```

Figure 4.3 b: Algorithm for "Finding the Failing Clause and Literal" at an other level of the DT

2.0: The input to **try other levels** is a list of DT literals. This list initially contains (when **try other levels** is called from the first time) the clause heads for which a local failure occurred. Since the user has not selected as the failing literal one of the literals associated with the heads, FOKAS will now ask if one of the heads would be the failing literal. Thereafter, **try other levels** is called recursively, so the input list will be a list of literals from higher and higher in the DT.

2.1: If the input list is empty, it means that the top level concept of the DT was reached and that no literal exists higher in the DT. This module and the learning of new knowledge for this example must be terminated.

2.2: Otherwise, for each literal in the list, FOKAS asks the user about its truth value. If the user says that one of the literal is true, then it should not have failed to be proved and as a

consequence must be the failing literal. The first such literal is assumed to be the only one, and this module is exited and learning continues with the next module.

2.3: if none of the literals were recognized as true by the user, we must look at the next higher level of the DT for the failing literal. This is done by adding the parent of each literal in the input list to a list P, when this parent exists and as long as it is not already in P (this happens when several literals have a common parent). **try other levels** is then called recursively with P. Hence, as we go up in the DT tree, some literals will not have parents because they correspond to the top concept and P will eventually become empty, which will terminate this module.

4.2.2.2 Example: Finding the Failing Clause and Literal

The above description of the algorithm is illustrated by the continuation of the example given in section 4.2.1. I remind the reader that at that point, we had a terminal failure and the explanation of the example could not continue. We also had a list LF of all local failures:

```
LF = [ [childcare_expense(Expense,joe,son) , work(joe),
        childcare_expense(Expense,joe,son) , student(joe) ] ]
```

Since missing knowledge is assumed, we must find out where the knowledge is indeed missing: is it because we do not know enough about the concept work, or about the concept student, or more generally about the concept childcare_expense?

The system presents all the collected failing clauses to the user and asks if one, and which one, should have been proved. By doing so, the user effectively selects the failing clause, which is the clause for which the explanation should *not* have failed if we had had the required knowledge. So, in our example the user would be asked:

```
Which of the following clauses should have been proved?

1. childcare_expense(Expense,joe,son):- work(joe) ...
2. childcare_expense(Expense,joe,son):- student(joe) ...
```

The user would thereby select student(joe) as the failing literal from which the proof should continue (the user *knows* that Joe is a student because of the presence of the example clause go(joe,school)). The literal work(joe) from the other local failure, and the clause that contains it,

would be marked so that the explanation does not use that clause again for that particular example proof. Therefore, as the system learns about what are the failing clauses (that should not fail), it also restricts its search space in the DT. In the case of our example, the first clause of the DT is effectively eliminated from future potential proofs of this example.

On the other hand, if the user says that none of these failing literals should have been proven (this can be done by selecting none of the above clauses), the prover asks if the parents of the failing literals should have been proved[14]. The **try other levels** part of the algorithm looks after this aspect. This allows to learn new concepts at any level of the proof tree rather than only at the leaves. Hence, if the user had selected none of the above clauses, the system would have asked:

is childcare_expense(Expense,joe,son) TRUE?

By doing so, FOKAS effectively backtracks one level up in the DT and now considers learning something for the head of the rule rather than one of the failing literals.

We can continue our illustration of this situation by supposing that the user answers **Y** to the above question "is childcare_expense(Expense,joe,son). TRUE?". This can happen on two occasions:

a. there may be a clause that states explicitly that joe is eligible for child care expenses, but for which constants could not match (e.g.: childcare_expense(2400,son,joe)) or for which the predicate was of a different arity (e.g.: childcare_expense(2400,joe,son,1990)); or

b. maybe there is some other information that does not implicitly state that joe is working or that he is a student, but that nevertheless make him eligible for that deduction . This happens sometimes when examples introduce situations which are not mentioned in the text describing the domain rules. We can then learn interesting new domain rules rather than foundational knowledge. For example, the text may neglect to mention that going on unemployment insurance sponsored training makes someone eligible for child care deduction. Then, supposing the presence of an

[14] In our case, both literals have a common parent: the rule head childcare_expense(Expense,joe,son). This is not necessarily always the case, however, since on backtracking another explanation that ends up also resulting in a local failure may involve a rule with a different head.

example that states that a person goes on unemployment insurance sponsored training, we could learn the new domain rule:

```
childcare_expense(Expense,Person,Child):-
    go(Person, Training),
    isa(Training,uic_sponsored_training),
    pay(Person,Expense,Someone),
    look_after(Someone,Child).
```

4.2.3 Finding the Proof-Continuation Clause

This sub-problem consists in finding the clause that should allow the explanation to continue: the *proof-continuation clause*. The knowledge gap is between the failing literal which was found in section 4.2.2 above and the proof-continuation clause. The missing knowledge we want to acquire will establish a relation between these two clauses, hence closing the gap.

4.2.3.1 Algorithm Description

1: The proof-continuation clause can be anything from a ground unit clause to a rule. Therefore, the search space is the whole set of clauses, either HC of the DT or ground clauses of the example. Since we assume no prior knowledge to discriminate among clauses, any clause can potentially be the proof-continuation clause. I remind the reader that at this point, FOKAS has unsuccessfully tried to use the accumulated new knowledge. Moreover, we have only one example available at a time, so large sets of examples cannot be used to help identify the proper clause either. Asking the user is again the only solution. However, to select among all clauses may be too demanding. As we have seen in section 3.2.2, constraint C2 requires that the human involvement be kept simple and minimal. We must therefore find a way to help the user in the search for the proof-continuation clause.

1.1: Although any clause can potentially be the proof-continuation clause, we can order clauses from most plausible to least plausible according to some domain independent heuristics. The algorithm implements an ordered list of *bridging heuristics*. These heuristics attempt to find the most plausible proof-continuation clause based on a preference for the simplest way to fill the knowledge gap. Hence, the user is presented only sub-sets of all the potential proof-continuation clauses, where the most plausible clauses are presented first.

These heuristics will be described in the next sub-section, immediately following the description of line 2 of the algorithm.

0: Input: the failing literal L_f

1: Present potential proof-continuation clauses in small groups to the user
1.1: where the preferred clauses are presented first
in accordance with the bridging heuristics

2: if none selected:
2.2: instantiate and assert L_f

3: Output: the proof-continuation clause C

Figure 4.4: Algorithm for "Finding the Proof-Continuation Clause"

2: It is possible that once all clauses have been presented to the user, it is realized that none of them is the proof-continuation clause. This can happen when the information is simply not present in the example. Such a situation may occur when something is so obvious that most people do not mention it. For instance, in the tax domain, to be able to claim a deduction, there may be a rule that states that someone must be resident of the country. However, few people will state this explicitly in their situation description. Most assume that anyone processing their claim ought to know that they are indeed resident, otherwise they wouldn't file a tax return. In general, this is a case of imperfect example data: that is examples that do not mention, even implicitly, a particular fact required by the DT.

2.2: In this case, the system simply asserts the failing literal after instantiating any variables. The effect of the assertion is that the explanation can now be completed. When asserting the failing literal, FOKAS assumes that the fact that could not be proved is indeed true. In such a situation, no further learning can occur (i.e.: we do not go to the next module), so we return control to the explanation mechanism and let the explanation of the example continue. When the same predicate occurs more than once in a situation such as this one where a clause is simply asserted, the system asks the user if clauses with this predicate should always be asserted without question: we call such predicates *fundamental.* If this is the case, the system will remember this fact and for any future failing of a clause with that same predicate, it will assert the clause.

4.2.3.2 Bridging Heuristics

The bridging heuristics come in two levels: the *top level heuristic* determines a global order of clause selection. It effectively splits all clauses in three disjoint sets, where the first set contains the most plausible clauses. The *secondary level heuristic* orders clauses within each of the second and third sets once the top level heuristic has been applied.

The top level heuristic builds the following three sets of clauses one after the other[15]:

a. The simplest knowledge gap fix must be tried first. The simplest fix is one that does not involve the learning of a new rule. Therefore, first look for ground clauses that use the same predicate as the failing literal.
Examples:

(1) if we failed on p(a) and there exists a clause p(a') where $a \neq a'$, then this last clause is preferred since maybe a and a' are actually two symbols that map to the same object in the interpretation domain.

(2) constants are simply not presented in the same order: we failed on p(a,b) but there exists a clause p(b,a) with the same intended meaning.

(3) predicates of different arity: we failed on p(a,b) but there is an assertion p(a,b,c) which means *at least the same* as p(a,b) (i.e., it carries *at least* the same information. Actually, some additional information is present).

b. If the simplest fix in a. does not work, try the next simplest. This involves the learning of new foundational knowledge under the form of a new rule that relates the failing literal with the proof-continuation clause. In this case, the proof-continuation clause is a ground clause (either unit or conjunctive, but not a rule) that does not use the same predicate as the failing literal. Find all such clauses. Apply secondary heuristic before presenting to the user (see below).

c. If the above does not work, get all other clauses (non-ground clauses). This kind of fix is considered the most complex. It allows for the learning of new domain

[15] In section 2.3.2.2, I have presented three kinds of knowledge gaps: language gaps, world and common sense gaps, and finally domain gaps. The top level bridging heuristic basically partitions the clauses according to these kinds of gaps. The language gaps correspond to the heuristic's first set (a), the world and common sense gaps to (b) and the domain gaps to (c).

rules, often by linking two trees or sub-trees of the DT. Apply secondary heuristic before presenting to the user (see below).

The secondary heuristic applies to the second and third set of clauses created by the top level heuristic. It is not worth applying to the first set of clauses because usually, there are very few asserted clauses with the same predicate. Hence, once all the relevant clauses have been found using the methods of paragraphs b. and c. above, before they can be presented to the user, apply the following heuristic to order them as three additional sub-sets:

a. find all clauses that use all the same constants as the failing literal. Present these first to the user;

b. If the proof-continuation clause is not selected from the above: among the clauses left, find all clauses that use at least one of the same constants as the failing literal; and

c. get all the other clauses (with none of the same constants of the failing literal).

The assumption here is that maybe clauses that contain more of the same constants are more closely related . This is merely a heuristic that aims at limiting the quantity of information presented to the user at one time. The aim is to ensure the user is not overwhelmed by information. We have to remember that FOKAS' objective is to guide the user and not have him do all the work.

4.2.3.3 Example: Finding the Proof-Continuation Clause

Once again, I will illustrate this algorithm with a continuation of the example from the previous sections. We have as input the failing literal L_f:

student(joe)

Based on the two bridging heuristics described above, the first clauses presented are those which use the same predicate as the failing literal. The system attempts to find ground clauses that use the predicate student. None are found, so the system must go to the next simplest fix. This is where all clauses that are both ground (top level heuristic) and that use all the constants used in the failing clause (secondary heuristic) must be found. In our case,

there is only one constant used in the failing literal: joe. So, all clauses that use that constant are presented to the user:

1. go(joe,school).
2. cost(joe,2400,childcare).
3. leave(joe,son,daycare_center).

The user then selects go(joe,school) as the proof-continuation clause. We now have an initial bridge over the gap:

student(joe):- go(joe,school)

The other parts of the bridging heuristic do not apply since we have found the proof-continuation clause. The initial bridge is passed to the next module so that the new rule can be finalized.

4.2.4 Learn new knowledge

At this point, we have two clauses: the failing clause and the proof-continuation clause. If both have the same predicate symbol, then we learn equivalent constants relations. Otherwise, we learn a rule where the failing literal is the head and the proof-continuation clause the first literal of the body. The top level algorithm is given at Figure 4.5a and is described below. More details about how each aspect is actually performed will follow in subsequent sub-sections.

4.2.4.1 Top Level Algorithm Description

1: The predicates of the failing literal and of the proof-continuation clause are checked to see if they are the same. The rationale behind this test is that if the proof-continuation clause has the same predicate symbol as the failing clause, then the only reason why the explanation could not continue was that one or several constant symbols could not be matched. Since the user has stated that the proof-continuation clause should allow for the explanation to continue, then it must be that the constants that cannot be matched actually refer to the same real-world object (or time, place, etc). I call these objects *equivalent*. Then, by learning a relation which states that indeed the two objects are equivalent, the explanation

mechanism will be able to explain the example. No further learning will therefore be required, so we exit the learning modules and return to the explanation of the example.

0: Input: the failing literal L_f and the proof-continuation clause C

1: if predicate of L_f is identical to a predicate of C
learn equivalent constants relations
exit , return to explanation mechanism
2: otherwise
learn a rule

3: Output:- a new rule L_f:- C, L_a, L_b, ..., L_z ; or
- equivalent constant relations and related matching lists

Figure 4.5a: Algorithm for "Learn New knowledge"

2: The learning of a rule allows for the creation of a relation between the two concepts represented by the failing clause and the proof-continuation clause. This rule is knowledge which can then be used to allow for the completion of the example explanation. The rule built by this module is passed to the next module so that its arguments can be generalized or specialized.

4.2.4.2 Learn Equivalent Constants Relations

Algorithm Description:

1: Equivalent constants relations are built by establishing *matching lists*. A matching list for two clauses is simply a list of pairs that shows the argument numbers that should match together. It also contains the predicate symbol for which the matching of arguments applies. It is possible to have more than one matching list for a particular predicate, but a matching list is predicate-specific: it applies for one predicate, not for another. This allows for the matching of constants that are not at the same argument position in their respective clause, given that they use the same predicate symbol. The initial matching list is built by finding identical constants in both clauses and storing their respective argument position as a pair.

2 -2.1: When no more identical constants can be found in both clauses, the user is asked to complete the matching of constants. This is done by presenting, one by one, each unused constant of the clause with the most constants to the user, along with a list of all

unused constants from the other clause. An unused constant is a constant which has not been involved in a matching yet. The user must then select the equivalent constant if there is one. FOKAS assumes that a constant occurs at most once in each clause. Therefore, once a constant has been involved in a matching, it will not be used again.

0: Input: the failing literal L_f and the proof-continuation clause C

1: for each constant that appear in both L_f and C
    append (position of constant in L_f, position of constant in C) to matching list
2: for each unused constant x
2.1: ask the user which of the unused constants y of the other clause is equivalent to x
2.2: assert an equivalent constant relation between x and y
2.3: append (argument position of x, argument position of y) to matching list

3: Output:- an equivalent constant relation
    - a matching list

Figure 4.5b: Algorithm for "Learn Equivalent Constant Relations"

2.2: Once the user has associated two constants, an equivalent relation is asserted. An equivalent constant relation is of the form: equiv(a,b). where a and b are two constants that refer to the same real world object, place, time, etc.

2.3: The matching list is updated so that the system remembers which arguments should match together. Matching different constant arguments without the matching list is still possible: such matches are called *direct matches* and they make use of the equivalent constants relations only. However, if constants cannot be matched directly because they are in a different order, then the system uses matching lists with equivalent constants relations to match them.

Example: Learn Equivalent Constants Relations

Given the failing literal p(a,b,c,d) and the proof-continuation clause p(x,b,y,c), an initial matching list is built by the system since the constants b and c are found in both clauses. We then have the initial matching list: [[2,2], [3,4]]. The constants a,d from the failing literal and x,y from the proof-continuation clause are still unused. The user is asked to select the matching constant for a by selecting among x and y. Then the last two constants can be associated. Assuming that the matching constant selected for a was x, we would obtain the matching list [[1,1], [2,2], [3,4], [4,3]]. This list tells that arguments 1 and 2 of both clauses

do match together while arguments 3 and 4 are interchanged. Finally, the predicate symbol p is added to the matching list.

From the matching list, the system can assert equivalent constants. Two different constants are equivalent if their respective argument should match (as specified by the matching list). In the case of our example, FOKAS asserts equiv(a,x) and equiv(d,y). Once this is done, the system returns control to the prover to continue the explanation. The next modules are not needed here because the only knowledge we need to acquire in this case is how to match the two predicates, and this has been done.

4.2.4.3 Learn a Rule

At this point, the rule consists only of the head (the failing literal) and of a one literal body (the proof-continuation clause). More literals will be added as the processing continues. This module involves two steps: constants linking and variable instantiation. The algorithm for each of these steps follows in the next sub-sections.

4.2.4.3.1 Algorithm Description: Constants linking:

1: By relating the proof-continuation clause C with the failing literal L_f, an initial rule L_f :- C is built. This rule allows for the bridging of the knowledge gap.

2: If a constant appears only in the head or only in the body, it is said to be *unlinked.* Unlinked constants in the head of the rule are assumed to be the result of language gaps (see section 2.3.2.2, sub-paragraph c.) Therefore, such constants may be equivalent to unlinked constants in the body. If they are equivalent, then the constants become linked.

2.1: The system helps the user to link unlinked constants of the head. It does so by presenting a list of potential equivalent constants from the body. These potential equivalent constants are the ones that appear only in the body, that is unlinked body constants.

2.2: For each equivalent constant found in this manner, an equivalent constant relation is asserted in the KB. The constant in the body is then replaced by the constant in the head for naming consistency and also to effectively link the two arguments.

```
0: Input: the failing literal Lf and the proof-continuation clause C

1: Rule Head:= Lf
   Rule Body := C

2: for each unlinked constant of the head

2.1:    ask the user if the constant is equivalent to one of the unlinked constant of the body
2.2:    if yes
            assert an equivalent constant relation between the two constants
            replace the constant in the body by the one in the head
2.3:    if no
            ask the user if the constant is somehow related to one of the unlinked constant of the body
            if yes
                    ask the user to select a relation from the list of relation
                    add the new relation between the two constants as a body literal

3: Output:- a rule for which constants that could be linked are now linked
```

Figure 4.5c Algorithm for "Constant Linking"

2.3: When two constants are found to be non-equivalent, the system inquires about a possible *more general relation* between them. A more general relation is one that is not as strong as equivalence but that nevertheless relates two constants. Examples of such relations are part_of, substance_of, etc (cf. section 2.3.2.2, sub-paragraph c.). In this case, an argument is linked by adding a new literal to the rule body. This new literal has two arguments: the two constants we wanted to link. Consequently, since a new literal may be added by this sub-module, it can be said to specialize the rule.

4.2.4.3.2 Example: Constant Linking

The example we used up to now does not need constant linking because all the constants of the head are also used in the body. I will therefore use another example to illustrate what this sub-module does. Suppose we have the following rule passed by the previous module:

```
pay(employer,salary,person):- work(person,office)
```

which bridges a gap between pay(employer,salary,person) and work(person,office). The constants employer and salary are unlinked head constants while office is the only unlinked body constant. The system will then question the user as follows in order to ensure that additional literals are added to link constants if need be (NB: user's inputs are in bold):

```
employer is not used in the body of the rule...
Is employer equivalent to one of these constants from the body?
1. office
N
Would employer be somehow related to office then? Y
What is the relation between employer and office?
```

Then the user will be presented a short list of generic relations such as part_of, substance_of, etc. If none of these are selected, a predicate will be invented by the system and the user will have the opportunity to rename it.

Continuing the above example, and supposing that the user has not selected one of the generic relations, the system would output the following:

```
Rename the predicate "relation_employer_office" (<CR> if no change) in:
pay(employer,salary,person):-
        work(person,office),
        relation_employer_office(employer,office).
```

The user can enter a new name such as belong(employer,office). This extra literal specializes the original rule. As it was initially, the rule implied that any employer could have paid a salary to a person as long as that person worked in an office, even an office that is not part of the employer's organization. Now, the extra literal specifies that the office must belong to the employer's organization[16].

Note that a constant does not need to be linked. The user can answer negatively to all system requests if it is believed that the rule is correct in its current form. It is up to the user to make this decision. For example, the user could decide that the constant salary does not need to be linked. The user would then keep the rule as it is after the relation between employer and office has been established.

4.2.4.3.3 Algorithm Description: Variable instantiation

0: At this point, we have an initial rule. The constants that could be linked were linked in the previous sub-module. The variables that could be instantiated were instantiated

[16] A literal involving the two unlinked constants may be more appropriate. For example: employ(employer,person,office), with the intended meaning that an employer "employs" a person in an office. The current implementation is limited to arity two predicates, which is a limitation on the expressiveness of the language. A better but more demanding solution would allow the user to specify relations among any number of constants.

by the explanation mechanism. However, some uninstantiated variable may be left in the head.

1: Each variable of the head must then be instantiated: the user cannot elect otherwise. This is a bias towards immediate completion of the proof, that is, filling the knowledge gap with only one rule.

0: Input: H :- L_1

1: for each variable V in H

1.1: ask the user which of the unlinked constants of L_1 should be used for the instantiation of V
1.2: if one selected
perform the instantiation
1.3 if none selected
ask the user to enter the constant c the variable V should be instantiated with
perform the instantiation
search DT U E for clauses that use c
if found
present the clauses to the user for selection
if none found or if user does not select any
invent a literal that uses c
add the instantiated literal the rule body

3: Output:- an equivalent constant relation
- a matching list

Figure 4.5d Algorithm for "Variable Instantiation"

1.1-1.2: Preference is given to instantiation with unlinked constants in the body, thus resulting in more argument linkages. The user is presented with a list of these constants and asked to select, if possible, the one which should be used for the instantiation.

1.3: If this is not possible or if the user refuses, a new constant must be entered. The system then searches for clauses that use that constant and attempts to add one of them to the body as a new literal. If nothing is found, or if the user rejects the system's attempts, the system invents a new literal that uses that constants. The user can then rename the predicate[17].

[17]The same general comment as for note 16 applies here. The current implementation is limited to learning arity one predicate in this case.

4.2.4.3.4 Example: Variable instantiation

In order to illustrate the variable instantiation process, we can take the same rule as presented above for constants linking, but let the second argument of the head be uninstantiated. We then have the following rule passed by the previous sub-module:

```
pay(employer,Salary,person):-
        work(person,office)
        belong(employer,office).
```

where Salary is not instantiated. The following dialogue between FOKAS and the user will result:

```
Should Salary be instantiated with one of the following constants?
1. office
N
Then, enter the constant with which it should be instantiated:
45000
Looking for clauses that use constant 45000...
Did you say that salary should be instantiated with 45000 because of one of these clauses?
1. earns(person,45000)
Y
```

Since office is the only constant in the body which is not also used in the head, it is presented to the user as the most probable constant for instantiation. The user will reject this proposal, which will prompt the system to ask for a new constant. The user enters 45000 because he knows from the original text that this is the case. Then the system looks for all clauses that use the constant 45000 and presents them to the user for selection. In the case of our example, only one such clause is found (earns(person,45000)) and the user selects it. Once this has been done, the variable is replaced by the constant and the clause added as an extra literal. The system then outputs this rule:

```
pay(employer,45000,person):-
        work(person,office),
        belong(employer,office),
        earns(person,45000).
```

The rule is passed to the next module so that types can be assigned to arguments and a specialization or generalization on the types performed.

4.2.5 Generalize/Specialize the New Knowledge

This module generalizes or specializes all the arguments of the rule by assigning a type to each of them. The algorithm is given in Figure 4.6 and an explanation of the algorithm follows the figure.

```
0: Input:
        - a fully instantiated rule
        - a type hierarchy
        - a selectional restriction list

1: Replace identical constants by identical variables
2: For each variable
2.1:    find the starting type
2.2:    until change of direction in type hierarchy
                generate one example
                ask user to classify
                if classification +
                        generalize: direction up in type hierarchy
                if classification -
                        specialize: direction down in type hierarchy
2.3:    add 'isa' literal to rule for the last type found acceptable
2.4:    check selectional restrictions

3: Output:
        - a generalized or specialized rule
        - a possibly augmented type hierarchy
        - a possibly augmented selectional restriction list
```

Figure 4.6: Algorithm for "generalize or specialize the New knowledge"

0: A type hierarchy is used to generalize or specialize arguments. Initially, the hierarchy consists only of a limited number (about 15) of *initialization types* such as living_organism and thing. These are used to present a short list of types to the user for selection when the type of an argument is unknown. An initialization type is one that is neither highly general nor highly specialized. As such, initialization types are types that *appear* as most appropriate to a human being to economically classify most objects and concepts. New (non-initialization) types are learned and added to the hierarchy as required. The input to this module also consists of a list of selectional restrictions (Allen 1987). This list is initially very short (less than 10), but it grows with time. Selectional restrictions are used to state what type of object can be used in what role. For instance, given an argument for which the role or case location_at was assigned, then only objects of type location and not objects of type human should occupy the position of that argument. We will see later how this list is used.

1: The first thing done is the replacement of constants by variables. All constants of the same name are consistently replaced by identical variables. The constants stay associated with the variables in a list so that we can later assign a type to each argument.

2: Each variable is then processed one at a time.

2.1: A starting type must first be assigned. If it is not known (that is, there is no isa assertion in the KB that tells the system what is the type of the constant associated with the variable), then the system proposes possible types based on case information (if it is available) and the current list of selectional restrictions. The case information is obtained by the TANKA linguistic module (see section 2.3.3). When the starting type of a variable cannot be determined by using cases and selectional restrictions, the system asks the user to select among the initialization types. It must be pointed out that FOKAS needs neither case information nor selectional restrictions to work. The only reason for the presence of selectional restrictions is to use them with cases; and the only reason for the presence of cases is because TANKA provides them. Moreover, as mentioned above, when cases and selectional restrictions are not available, the system simply asks the user.

2.2: Once the starting type has been assigned, the variable is generalized or specialized by going respectively up or down in the type hierarchy. For this purpose, the system generates an example and asks the user to classify it. An example is simply the rule itself where the current variable has been replaced by a *generic instance* of the type. A generic instance of a type is simply a symbol built by taking the type's symbol and prefixing it with "a_". This should generally be sufficient to evoke, in the user's mind, the ultimate meaning and purpose of the rule, and in turn result in a proper classification of the example.

2.3: The process continues until the user forces a change of direction. For example, the system was generalizing (i.e. the user was answering **yes** to classification queries) and the user answers **no** to the latest classification query. When this occurs, the last type which the user classified as acceptable (a **yes** answer to the query) is selected as the variable's type. The variable's type is added to the rule body as a new 'isa' literal, hence typing all linked arguments of the rule that are using that variable.

2.4 When a type has been assigned to a variable, and if a case is associated with this variable, the case-type pair is checked against the list of selectional restrictions. This operation of checking case-type pairs against a list of selectional restrictions allows for the

learning of useful general and common sense knowledge. This knowledge is similar to the new literals FOKAS can learn when building the rule (c.f. section 4.2.4.3). However, in this case, the new knowledge is not added as a literal to the rule but as a fact in the KB. This knowledge can later be used to restrict the number of items in a list of choices presented to the user. It can also be used by the system to present most probable answers first, such as with the selection of types seen at line 2.1 above. Here again, it is not necessary to have case and selectional restriction information for FOKAS to work: it is simply opportunistic and uses what is available when it is.

4.2.5.2 Example: Generalize/Specialize the New Knowledge

Example of the Generalization/Specialization Process: Lines 1 to 2.3

Given the rule obtained from the last module, where all constants have now been replaced by variables:

```
pay(Employer,Amount,Person):-
        work(Person,Office),
        belong(Employer,Office),
        earns(Person,Amount).
```

If we determined that the starting type for the variable Employer was living_organism, then the example would consist of:

```
Is this true for all living_organisms?
 pay(a_living_organism,Amount,Person):-
        work(Person,Office),
        belong(a_living_organism,Office),
        earns(Person,Amount).
```

where the variable Employer has been replaced with the generic instance a_living_organism of the type living_organism.

The system asks the user if this relation is true for all objects of type 'living_organism'. If the user answers **yes**, then generalization to the next higher level will take place. The process would then be repeated with the parent type of 'living_organism'. If the user answers **no**, then specialization occurs by going to the next level down in the type hierarchy. When multiple sub-types exist, the user must select the right one; and when the proper type does not exist, the user must add a new type to the type hierarchy.

Example of the Check Selectional Restrictions Process: Line 2.4

Suppose we are generalizing the last argument (D) of the following literal, and we come to the conclusion that D is of type "service_provider":

```
leave(C,B,D)
```

The literal is part of the body of the rule below, where all arguments have now been generalized or specialized, and the respective isa literals added to the rule:

```
look_after(A,B):-
        leave(C,B,D),
        work(A,D),
        isa(A,human),
        isa(B,human),
        isa(C,human),
        isa(D,service_provider).
```

This rule states that a person A will look after a child B if a person C leaves the child B at a service provider D, and if the person A works for the service provider D[18] .

We also assume the existence of two additional pieces of information: first, associated with argument D, we have the case location_at. This case was assigned during the linguistic processing in TANKA. Second, we have selectional restriction information stating that objects used in the role of location_at must be of type location. The system will then query the user as follows:

```
Usually, the role of "location at" cannot be held by objects of type service_provider.
How are objects of type service_provider and objects of type location related?
```

Then, a list of common relations is presented to the user (as seen previously in section 4.2.4.3) and the user can select one of them or enter another one. In our example, it could be appropriate to select the relation has_attribute (i.e., service_providers should all have an attribute location, otherwise, it could be appropriate to question their reliability or even their honesty). This new foundational knowledge item is then added to the KB for future use. A new selectional restriction is also added to the list of selectional restrictions since we now know that the case location_at is compatible with the type service_provider.

[18]This rule is not necessarily always correct. Leaving a child at the dry cleaner (also a service provider) does not imply that the employee there will look after her. It is possible that the user makes such mistakes.

At this point, learning is completed and control is returned to the explanation mechanism.

4.2.6 Use the New Knowledge

We have seen in section 4.2.1 that the knowledge acquired by the system is used on each local failure. This is done with the hope that the knowledge will allow the prover to continue the explanation of the example rather than having to interact with the user to learn yet other knowledge. In this section, we will see how the new knowledge is accumulated by the system and how it is used to fix failures. This module is very important since it is the one that reuses the knowledge acquired and thus allows the system to become more autonomous over time. [19]

The new knowledge is used on local failures, before backtracking occurs and before learning is attempted (see Figure 4.2, section 4.2.1). So, when there is a local failure, before the clause on which the prover failed is added to the list of local failures, the system searches for new knowledge that could allow for the proof completion. If this is possible, we say that the failure has been *fixed.*

The search for new knowledge proceeds in the same general order as when the system learns (cf. section 4.2.3.2: bridging heuristics): it first tries to find matching lists and relevant equivalent constants (**use new facts**). If this does not work, the system then tries using accumulated rules (**use new rules**) to fill the gap. If none of the above works, then and only then does an actual local failure occur. The algorithms for **use new facts** and **use new rules** are shown in Figures 4.7a and 4.7b.

4.2.6.1 Algorithm Description: use new facts

1 and 3: All ground clauses that use the same predicate as the failing literal are found. If at least one such clause exists, then it has the potential to allow for the explanation to complete. However, since the explanation did not complete, it must be that the constants in

[19] This module of the system implements a form of case-based reasoning (Kolodner 1984). A case is a failure and its "solution". Cases are indexed by failing literals. Hence, the matching lists and rules in the KB of foundational knowledge form a case base.

such a clause did not match the ones in the failing literal. Therefore, by using the accumulated equivalent constants relations and the matching lists, FOKAS may be able to fix this problem. If no ground clause is found, then this method of fixing the failure cannot work: FOKAS has to use new rules (line 3).

2-2.2: For each ground clause with the same predicate as the failing literal, the system tries to find an equivalent constant relation that would allow for a direct match (i.e. for constants in the same order) between the ground clause and the failing literal. This first attempt is made because FOKAS tries to find the simplest way to fix the current failure by assuming that constants may be in the same order. If they are not, another attempt will be made later (line 2.3) with matching lists. When the first acceptable equivalent constants relation is found, this module is terminated and control returned to the explanation mechanism. In such a case, FOKAS has successfully shown that this failure can be fixed by using previously acquired knowledge.

0: Input: - a potential local failure: a literal L_f of the DT that cannot be proven
- the currently accumulated new knowledge

1: Find all ground clauses using the same predicate as L_f . Store in list GC
2: for each ground clause found
2.1: find an equivalent constant relation that would allow for a direct match between L_f and any of the clauses in GC
2.2: if found
the KB contains the required knowledge to fill the gap
exit this module: continue the explanation

2.3: if not found,
find all matching lists that would allow for an indirect match between L_f and any of the clauses in GC
for each matching list found
find equivalent constant relations required
if found
the KB contains the required knowledge to fill the gap
exit this module: continue the explanation
if none found
use new rules
3: if not found,
use new rules

Figure 4.7a: Algorithm for "Use new facts"

2.3: If no equivalent constants relations work for direct matches, then FOKAS looks for matching lists for the failing literal's predicate. These matching lists, if found, would allow for the indirect matching of constants (i.e. constants that do not appear in the same argument positions) given that these constants, if they are not identical, are also related

by equivalent constants relations. If this does indeed happen, then the failure is fixed and control returned to the explanation mechanism. Otherwise, FOKAS will attempt to use foundational knowledge rules to fix the failure.

4.2.6.2 Example: use new facts

The prover failed on pay(richard, Expense, child).

We have in the KB (from previous examples failures and learning):

```
pay(richard,son,560)
equiv(son,child)
matching_list(pay,[ [1,1],[2,3],[3,2] ])
```

Since there is a clause in the KB that uses the same predicate as the failing literal, **use new facts** can be used. The clause pay(richard,son,560) is extracted as a potential proof-continuation clause and stored in the list of ground clauses GC (see Figure 4.7a). For this clause to work as a proof-continuation clause, we need to find knowledge that states either that child and 560 are equivalent; or a matching list for the predicate pay that allows for a matching with the failing literal. Since there is no equiv predicate for child and 560, the second possibility is tried. A matching list for pay is found that establishes a correspondence between son and child. The system then looks for equivalence between these two constants. Since they are found to be equivalent, the clause pay(richard,son,560) is accepted as a proof-continuation clause. The explanation can therefore continue: no terminal failure occurs.

4.2.6.3 Algorithm Description: use new rules

We now consider the case of reusing rule-type knowledge. First, I must point out that new rules are not asserted as active knowledge in the KB. They are kept in *passive memory* until they are needed, at which time they are brought into *active memory*. This is achieved by storing the new rules as lists. This allows the potential number of failures to stay constant as new rules are learned. Indeed, if new rules were asserted in the KB, the explanation mechanism would attempt to use them in explanations, and could fail on them too. As more new rules are added to the KB, each of them is potentially a new local failure point since each one adds an alternative to explanations. Therefore, as more rules are acquired, more local failures could occur, something we want to avoid because it would result in increased user interactions.

A rule only becomes active when the system determines that it can be useful (i.e., its head's predicate matches the failing clause's predicate). Since there may be more than one useful rule for a particular failure, the first one found in the KB is asserted and the others are asserted one after the other as the previous fails. Once a rule is found not to work, it is removed from active memory. A new rule may also cause another new rule to fire, thus allowing for backward chaining on the new knowledge.

The algorithm for this process is shown in Figure 4.7b and is explained below:

1-1.2: In order for the explanation to continue from the failing literal, a new knowledge rule must have a head literal that uses the same predicate as the failing literal. Passive memory is searched for such a rule and the first one is asserted in active memory and the continuation of the explanation is attempted with it. If ever this rule does not work, the next one will be made active and so on until one works or until none are left. If the retrieved rule works, this module is terminated and control returned to the explanation mechanism. In this case, the current failure has been fixed by the rule which was found, and the explanation of the example can continue. The user has not been involved and the system handled a failure by itself by using what it knew about the concept at which the local failure occurred.

1.3: It is possible that the retrieved rule will not work. The reasons for such a failure are exactly the same as for the DT failures: a ground clause that matches one of the current DT clause's body literal cannot be found. This in itself either results from not finding a ground clause with the right predicate symbol (i.e. a predicate symbol identical to the failing literal's one) or from not being able to match constants given a ground clause with the right predicate.

1.3.1: At this point, the system has been unsuccessful in applying the current new rule to fix the initial local failure: we have a local failure within the new knowledge - a *new knowledge failure* (in opposition to a DT failure). Indeed, another new rule may exist that could fix the current failure, but the system does not know yet. The same general failure handling strategy is used for new knowledge failures as for DT local failures. For this reason, **use new facts** is called. This allows for the use of matching lists and equivalent constants relations *for failures in the new knowledge*. This process has been described in section 4.2.6.1. Then, if this fails, **use new rules** will be called and the process described here will be applied to the new knowledge failure. In this manner, FOKAS permits backward chaining on the new knowledge. If this process is successful, the DT local failure is fixed and the

explanation of the example can continue. Although there has been a failure, FOKAS has handled it by itself without the help of the user, based on previously accumulated knowledge.

0: Input: - a potential local failure: a literal L_f of the DT that cannot be proven
- the currently accumulated new knowledge

1: for each new knowledge rules in passive memory whose head predicate matches L_f predicate
1.1: assert the rule in active memory
1.2: try to apply it to the current failure
1.3: if fail
1.3.1: **use new facts** with the part that does not work
1.3.2: if fail:
learn simple things *(to make the rule work)*
if not suitable
retract the current rule from active memory

2: if not found or if none worked
fail this module

3: Output: - a possibly augmented KB of new knowledge *(when* **learn simple things** *has worked)*

Figure 4.7b: Algorithm for "Use new rules"

1.3.2: If using new knowledge to fix a new knowledge failure has not worked, then the system will try to learn simple facts (with **learn simple things**) to make the rule work in the current situation. The aim here is to find a way, with minimal user interaction, to make an existing rule work for this particular failure. In other words, FOKAS tries to avoid having to learn a new rule because the learning of a new rule is very expensive in terms of user interactions. For this purpose, **learn simple things** addresses three points: first, the current rule may just have its argument in a different order; second, it may contain equivalent constants we do not know about; and third, a type may not be recognized due to the use of a different symbol (ex: types human and person). FOKAS will then interact with the user to match arguments and learn equivalent constants relations, as described in section 4.2.4.2. So, FOKAS is improving on the learned knowledge, effectively reasoning not with it, but *about it* , at a meta level. The result of the rule improvement is not the assertion of a new, more general rule. Instead, the original rule is kept as is, and new knowledge that allows to use the rule is asserted. A rule requiring more complex changes than the ones mentioned above will be returned to passive memory, preference going to retrieving a different rule if one exists (line 1); or, otherwise, to learning a new rule (line 2).

2: If no rule head in passive memory were found to match the failing literal predicate, or if none of the rules found worked for this failure, this module is failed and control returned to the explanation mechanism. This means that the system does not possess

the required knowledge to fix this failure. The explanation mechanism now considers this failure as an actual local failure. It adds it to the lists of local failures for eventual selection as the failing clause and literal if a terminal failure happens. The explanation mechanism then looks for alternative explanations.

Before looking at an example for **use new rules**, I must mention that FOKAS behaves in a somewhat different manner with new knowledge failures than with DT local failures (see line 1.3.2). Indeed, with DT failures, learning is triggered only when a terminal failure happens, that is when it is known that no alternative explanation exists. In the case of new knowledge failures, FOKAS tries to learn simple things for each failure, even if there is a possibility for alternative explanations with other new rules. This may not be the best way for FOKAS to work since the user will be questioned for *every* retrieved rule that does not work, instead of waiting until the end to ensure that indeed, none of the rules are working. In other word, FOKAS is biased towards the belief that none of the rules will work *as is* but that several *could* work given minimal interaction with the user to acquire the necessary knowledge. In short, this area may need improvement by looking at a more intelligent way to retrieve rules from passive memory. For instance, rather than simply using the first rule found, some measure of similarity with the current failure may be used to select the rule that more closely match the failure.

Another major difference between DT and new knowledge failure handling is that the learning of new rules that could be attached to existing new rules to fix a new knowledge failure is not allowed by FOKAS. This is merely a bias for short bridges over knowledge gaps (i.e. one rule only is required to fill the gap). Although the idea of learning longer bridges may be attractive, it has not been implemented in FOKAS for two reasons: first, with the examples of the domain I studied, single rule bridges were always appropriate; and second, building a rule is very expensive in terms of user interaction. Therefore, since there seemed to be no requirement for longer bridge (at least in the case of the examples of the tax domain I have used), and since building longer bridges would result in higher user interaction levels, the bias for short bridges was deemed preferable.

4.2.6.4 Example: use new rules

The KB contains the clause (from the example):

```
give(employer,pay,richard).
```

and the currently accumulated KB of foundational knowledge[20]:

```
work(A) :-
        prop(A,B),              (object A has property B. For example, the sentence
        isa(A,person),          "Joe is a teacher" may result in the logic representation
        isa(B, employment).     prop(joe,teacher) after surface NLP has been applied)

work(A) :-
        receive(A,B),
        isa(A,person),
        isa(B, employment_revenue).

receive(A,B) :-
        give(C,B,A),
        isa(A,person),
        isa(C,person),
        isa(B,object).
```

The prover fails on work(richard). There is no other work predicate in the KB, so **use new facts** fails. The system must therefore find a rule that matches the failing literal predicate: work.

There are two such rules. The first one is asserted, with the argument binding resulting in the instantiation of the variable A:

```
work(richard) :-
        prop(richard,B),
        isa(richard,person),
        isa(B, employment).
```

The system then tries to use that rule to continue the proof of the example. For this, it must find a clause that matches prop(richard,B). Since this does not work, the system attempts to use the new knowledge on the new knowledge itself by calling back **use new facts.** This also fails because there is no equivalent clause, no matching list and no rule in the KB of foundational knowledge that satisfies prop(richard,B).

The system then attempts to find a simple way to make the rule work. There are three possibilities: equivalent constants, different order of the arguments, or equivalent types. The first two require the presence of clauses with the predicate prop in the KB. Since there are no such clauses in the KB, the system cannot make this rule work by learning additional equivalent constants relations and matching lists. The last possibility implies that the new

[20]These rules are initially not in active memory. They become active only when they match a failure.

rule does not work because the constant arguments of an isa literal do not match. The explanation has failed on a literal with the predicate prop, so the problem is not with matching constants of an isa literal. Since none of the simple fixes can work, this rule is retracted from active memory and the next one that matches the failing literal is made active (with the proper instantiation):

```
work(richard) :-
        receive(richard,B),
        isa(richard,person),
        isa(B, employment_revenue).
```

The literal receive(richard,B) must be proved but this fails. The system then tries to use the new knowledge on the new knowledge by calling **use new facts. use new facts** itself fails, resulting in a call to **use new rules**. This results in a search for a rule head that matches receive(richard,B). Such a rule is found:

```
receive(richard,B):-
        give(C,B,richard),
        isa(richard,person),
        isa(C,person),
        isa(B,object).
```

We therefore have backward chaining on the new knowledge: work(A)<- receive(A,B) <- give(C,B,A).

The example clause give(employer,pay,richard) then matches the first literal of the body of the rule. Variables are instantiated: C to employer and B to pay. The next literal of the rule is then processed. For the purpose of this example, let's suppose that the KB contains the following type facts:

```
isa(richard,human).
isa(employer, organization).
isa(pay, employment_revenue).
```

This means the second literal cannot be proved as is. We need either to learn some knowledge which will allow for the use of that rule, or retract that rule from active memory and try to find another one. If this was to happen with this example, the module "use the new knowledge" would fail since there is no other foundational knowledge rule that matches the predicate work. FOKAS would then have to learn a new rule (assuming a terminal failure happens). However, this is not the case: with our example, it is possible to learn simple things that will make the rule work. This should involve the user less than the learning of a new

rule. Indeed, since the system is failing to prove an isa literal within a new knowledge rule, FOKAS queries the user in an attempt to learn extra knowledge that will permit to fix this failure:

I know that richard is a human.
is richard also a person?
y
Are the types human and person equivalent?
y

The fact that the types human and person are equivalent will then be asserted. The dialogue with the user will continue for the other literals:

I know that employer is an organization.
is employer also a person?
y
Are the types organization and person equivalent?
n

As it is done in other modules, a list of generic relations is presented and the user can select among them or enter a new one. In this case, the user could select the relation part_of between person and organization. The system then continues the proof based on the fact that the user has said that an employer can be a person even if the two types are not equivalent. As a side effect, an additional "common sense" relation between person and organization has been acquired: part_of(person, organization).

The final dialogue for this rule will be:

I know that pay is an employment_revenue.
Is it also an object?
y
Are the types employment_revenue and object equivalent?
n
How are the types employment_revenue and object related?
Select from one of the following generic relations:
*...(user selects **a_kind_of**)*
asserting a_kind_of(employment_revenue,object).

In the end, although this rule could not be used as is to fix the failure, the system allowed its use by learning the required extra knowledge via simple queries to the user. The hope is that these queries will generally require less user involvement than the learning of a totally new rule.

The proof of the parent clause (where all variable are now instantiated due to bindings from the previous rule) is then completed with no problem:

```
work(richard) :-
        receive(richard,pay),
        isa(richard,person),
        isa(pay, employment_revenue).
```

Consequently, what was initially a potential local failure (because the literal work(richard) could not be proved) can now be proved thanks to the knowledge we had acquired on previous failures and minimal user interactions.

4.3 Summary

In this chapter, I have described both a conceptual solution to the missing foundational knowledge problem and an actual implementation called FOKAS.

The former resulted in the presentation of the general ideas behind the Lazy KA method, and of a general architecture for the acquisition of knowledge from text.

With the latter, we have seen how an explanation mechanism can trap its failures and use them to acquire new knowledge. The system uses knowledge of how its explanation mechanism works (for example: matching predicate symbols and constants) to guide the user through the acquisition process. The new knowledge is useful foundational knowledge. Finally, we have seen how the new knowledge is used on previously unseen situations of knowledge gaps and how it allows to fill them with minimal recourse to the user.

Chapter 5 - Related Work

This chapter will give a brief overview of five research areas related to the work presented in this thesis. However, my review of these fields is far from exhaustive: my aim was rather to illustrate points in common as well as differences with my work.

5.1 Theory Revision

Theory revision (TR) research (Nedellec 1991) shares the following points with the problem discussed in this thesis:

a. It acquires new knowledge for a deficient DT; and
b. It uses failures as learning opportunities

However, it is also quite different because:

a. TR usually applies to DT that only contains a few domain specific deficiencies. In our case, the domain related deficiencies are only secondary to the lack of a large quantity of foundational knowledge;

b. TR mainly uses automatic methods such as abduction and induction. Because TR assumes the presence of background knowledge and of large sets of examples, there is no necessity to involve the user. In FOKAS, there are no such assumptions; and

c. TR research has concentrated on learning one single rule at a time while FOKAS aims at the more complex task of multiple concepts learning.

We will now see a few examples of these common points and differences.

The field of TR abounds in systems that use failures as learning opportunities. In particular, EBL is often used as a mean to identify incomplete DT. For example, the system LISE (Genest et al. 1990) used a three-step approach which first resorts to abduction, then analogical inference and finally CBR to fix the DT. Abduction is based on the fact that knowing Q ⟵ P and given Q, it is plausible to conclude P.

Abduction however is not a method that can apply within the FOKAS framework. The reason for this is that although we may know that Q ‹— P and Q, we assume the presence of P' which is somehow related to P. It is the relation between P and P' which we aim at acquiring, not the fact P.

In LISE, a scoring mechanism has been implemented to select among the partial explanations which one should complete. The two criteria used are conciseness (shortest explanation) and coverage of the example (the number of common features the partial explanation has with the example). These criteria are syntactic and offer no real guarantee of success. We have seen in section 4.2.2 that the selection of partial explanations based on such criteria is not acceptable in FOKAS. For instance, there is no reason for a shorter explanation to be better: this is strictly a bias (Utgoff 1986). In our case, this bias has just too much chance to lead to a wrong choice.

Other TR systems use induction (for example: Duval and Kodratoff 1990, Feldman and Nedellec 1994) in conjunction with other methods such as abduction. Induction is guided by the current DT knowledge, which restricts the hypothesis search space. Feldman and Nedellec in particular stress the importance of incrementallity in TR, a property that FOKAS implements. They also generalize the new knowledge by going up in a type hierarchy as is done in FOKAS. The use of induction however is not suitable in this work, or at least not practical because we would need large sets of examples for each potential failure. This would require *ad hoc* hand-crafting of both examples and background knowledge, clearly against the aim of this work.

5.2 ML and Inductive Logic Programming (ILP)

The work in ML and ILP which is most relevant to this thesis is where a user is involved in the learning. Generally however, a good approximation of the target concept with high probability (the PAC framework: Valiant 1984) has been preferred in ML (Haussler 1988) and ILP. This approach has a tendency to eliminate the user from learning, actually *camouflaging his involvement* in the pre-classification of examples and in the provision of background knowledge. In this thesis however, I take the opposite position that *the user is an intrinsic part of learning*. I claim that for most tasks, some form of user involvement is required, so why try to hide this involvement into preparation of data, use of heuristics, etc?

Consequently, the approach taken by FOKAS is more in line with Angluin's work (Angluin 1988) who has shown that concept learning efficiency can be improved by using queries answered by a user during learning.

A few ML systems (mainly Learning Apprentice Systems and Tutorial Learning Systems) have involved a user to help learn the missing knowledge on explanation failures. The ROBO-SOAR system (Laird et al. 1990) for example has a user involved in selecting from a limited set of options when it realizes it misses knowledge. Also, many ideas in this work originated from CLINT (De Raedt et al. 1993, De Raedt 1992a, De Raedt 1992b). CLINT is an interactive and incremental learner that learns multiple new concepts. It also generates its own examples and increases its description language by predicate invention just as FOKAS does.

However, CLINT requires a minimal Herbrand model to generate its examples which are then used incrementally and inductively to learn new concept definitions. The user is involved by answering membership and existential queries. In FOKAS, there is no need for a minimal Herbrand model and multiple examples since the system learns multiple new concepts with only one example: the EBL example which could not be explained. Also, although FOKAS does generate examples, they are not used for the same purpose as in CLINT: they are used exclusively to generalize or specialize along a type hierarchy. As in CLINT, FOKAS needs to increase its description language by inventing new concepts which are not necessarily derived from existing ones (Rendell 1989), but tries to invent concepts that are of general interest to the user for building rules.

Most other work in ILP and ML is not really relevant since it concentrates on learning single concepts from a large set of examples while my work addresses the problem of learning an acceptably complete KB of foundational knowledge. Moreover, ILP and ML require prior (or background) knowledge to guide search, while I assume none. This is well illustrated by the Relational Clichés of FOCL (Silverstein and Pazzani 1991) and the rules models and predicate ontologies of MOBAL (Kietz & Wrobel 1992). For instance, MOBAL relies on rule models (a sort of abstracted rule structure) and on a domain topology (a definition of relationships between concepts in a domain). It is interesting to point out that the type of knowledge these systems need to restrict their hypothesis search space can be learned by FOKAS. Therefore, the method I present not only applies to NLP prior knowledge, but to the acquisition of any prior knowledge for any KBS (where ML can be seen as a KBS since it needs knowledge to work).

5.3 KA for Expert Systems

Compared to TR and ML in general, the work done in KA for ES on the contrary has many points in common with my work. For instance, the idea of using failures as learning opportunities has been used extensively in that field, but from a different perspective: the user is the expert, and the system tries to elicit domain knowledge from the expert in a semi-automatic manner. Such a point of view is close to the one I take, except that with FOKAS the interactions with the user result mainly in foundational knowledge, *the domain knowledge having been obtained from text.*

What is probably the oldest and one of the best known semi-automatic KA systems for ES is TEIRESIAS (Davis 1979). The basic idea behind this system and many of its descendants such as MOLE (Eshelman et al. 1987) and ODYSSEUS (Wilkins 1990) is a performance task (i.e., an inference engine and a KB) which explains cases. When the performance task uncovers a shortcoming in the KB, the user must provide new knowledge.

The power of the approach comes from the system's knowledge of the problem solving method of the performance task, thus making the overall approach independent of the domain. Moreover, the system guides the user, so that the user does not need to understand the technicalities of the approach. These are two characteristic that FOKAS also has.

An initial KB or raw model of the domain must be provided in TEIRESIAS. So there is a need for some initial KA effort to obtain that KB. In the case of FOKAS, this initial KA is done from an existing text. This is a major advantage since the role of the user is further restricted, and the overall KA process much improved. TEIRESIAS processes examples to find missing knowledge or bad inferences that will be used to correct or complete the initial KB. FOKAS proceeds similarly, but only considers valid inferences that do not complete: FOKAS makes no attempt at correcting the initial KB. In the case of ODYSSEUS, not only domain knowledge is acquired but also problem-state and meta knowledge, but still no foundational knowledge.

PROTOS (Bareiss 1989, Bareiss et al. 1989, Porter et al. 1990) applies the same technique to all steps of KB development, rather than just the refinement phase, as with the systems we have seen above. This results in not having to provide an initial KB as a whole and instead building it incrementally. Bareiss also introduces the idea that the system performance

should improve and less user involvement should be necessary as the system learns. We find the same idea with the Convergence Hypothesis in this thesis. In PROTOS, evaluating the system gain in autonomy was done by keeping track of the number of matches PROTOS discusses with the user as the number of cases increases. Since the number of matches discussed stayed relatively constant, a gain in autonomy was claimed. In the case of FOKAS, as the number of cases increased, the interaction actually *diminished.*

Both approaches to the acquisition of an initial KB (i.e., prior acquisition as with TEIRESIAS or incremental acquisition as with PROTOS) differ from the FOKAS approach where this initial KB comes from text. Since we can use texts that already exist, then there is no need to involve the expert in the initial elicitation phase. This means that the expert's work is essentially confined to refining the superficial KB obtained from text. More interestingly, the expert's work now results not only in additional domain knowledge, but also in useful common sense and general knowledge, something that no other KA system does. In fact, this type of knowledge is one of the most difficult to acquire (c.f. section 1.2) since it is so informal and implicit in experts' reasoning.

5.4 KA from Text

We have discussed the major ideas and problems of KA from text in chapter 2. I will therefore not repeat these here except for a brief summary in the next paragraph. Instead, I will present some complementary information on the subject.

We have seen in chapter 2 that KA from text has traditionally relied on the hand-crafting of large quantities of knowledge. This comes from the fact that a text (in the case of written NLP) contains only a fraction of the information it intends to convey. The reader must participate actively by using his own knowledge (Schank 1982). A well established way to deal with this problem is to use scripts, goals and plans (Schank and Alberson 1977). However, scripts, goals and plans are prior knowledge that requires hand-crafting. Hand-crafting results in long and tedious and even open ended preparation, something we would like to avoid.

Systems that do KA from text must therefore rely on some form or another of prior knowledge (recent examples are: Hauptman 1991, Kim and Moldovan 1993). Others have looked at restricted classes of problems by studying very specific forms of language (Moulin & Rousseau 1991,1992, Bozsahin & Findler 1992) in an attempt to avoid the prior knowledge

problem. Yet, some other research has been conducted to learn the required prior knowledge. For example, GENESIS (Mooney 1990) uses EBL to learn schemata that help text understanding. However, these new schemata can only be learned from other, already existing schemata. Therefore, given a new situation for which no such prior schemata exist, the system cannot learn (and cannot process the text either since it does not have the proper schemata).

Whereas I am taking the stand that no knowledge at all should be provided to a NLP system beforehand (except lexical and syntactic knowledge), and that the input must be a non-restricted form of NL[22], no KA from text system except TANKA (Delisle 1994) proceeds this way. Delisle's work however, did not go beyond the linguistic aspects, and the resulting knowledge is not of sufficient quality to be used by a performance task without continuously failing (Delannoy et al. 1993). This is where the approach I propose can help.

5.5 Universal KB and common sense reasoning

As we have seen previously, most, if not all work in ML and NLP that intended to acquire knowledge depended somehow on yet other, more fundamental knowledge. The question of where that knowledge is coming from is still open.

Cyc (Lenat & Guha 1990) aims at acquiring the general, more fundamental knowledge required for KBS, including ML and NLP. Cyc proposes the "no cheap lunch approach" where all such knowledge must be hand-coded for all possible tasks once and for all. Then and only then will machines be able to learn from reading text and learn as humans do in general. There has been no real alternatives proposed for the expensive and high risk venture that Cyc is (Minsky 94).

Actually, Cyc itself represents the prior hand-coding paradigm so common in NLP and ML. That is, the knowledge is acquired from people manually entering it in a computer-acceptable format *before* any task can be performed. Therefore, this leaves literally no investigation of how one can acquire such knowledge other than by trying to figure out beforehand what is the required knowledge and coding it manually. Cyc merely eliminates the figuring out of which knowledge is required by pretending it will have all of it. Someone still has to hand-craft it all.

---

[22]The use of expository technical text is not seen as a major restriction since a lot of the interesting texts from which we would like to perform KA fall in this category.

The work presented in this thesis can be seen as a restricted alternative to Cyc. It is restricted because there are many issues that Cyc and other research in common sense reasoning (see Davis 1990 for example) are addressing that this work is not even contemplating (such as representation issues, ontology building issues, computationality and complexity issues of inferences, and formalism). Still, from the strict "knowledge to be acquired" point of view, FOKAS learns the same common sense and general knowledge that Cyc aims at acquiring, but does so within the context of a specific task and in a lazy manner during problem solving.

As such, the work presented here is based on the same underlying assumption than Cyc: that is, prior knowledge is not created from nothing, it requires some heavy work *from people*. However, the important point is that my work clarifies the question of *when* and *how* prior knowledge should be acquired.

The advantages of the lazy KA approach versus Cyc are as follow:

a. Lazy KA allows NLP to be used with no preparation (but with the participation of a user) for the acquisition of domain knowledge. Cyc claimed that this would only be possible once most foundational knowledge would have been hand-coded;

b. The foundational knowledge can be acquired on a task-specific basis by using real-world examples. These examples constrain the knowledge that needs to be acquired. Therefore, KA of foundational knowledge can be done much more expediently than in Cyc where there is no constraint on what knowledge must be acquired;

c. Cyc's KB size will make inferences much slower than with a KB we acquire lazily. Moreover, storage requirements (and also hardware for speed) may make Cyc KB impractical or too expensive for many smaller applications. With Lazy KA, only the knowledge needed by a task is acquired.

Chapter 6 - Empirical Evaluation

6.1 General

In section 3.3, I stated the following hypothesis, which I called the convergence hypothesis:

> **Hypothesis:** Given a sufficiently large number of examples, we can train a Lazy Text-based KA system in such a way that the resulting KB converges towards an *acceptably complete* KB of foundational knowledge.

This chapter will present an experiment that aims at verifying or disproving this hypothesis. The empirical results were obtained by using FOKAS (the implementation presented in chapter 4) with real world examples. These examples are specific situations that FOKAS domain theory should be able to explain, but for which it may lack foundational knowledge. The examples were presented to FOKAS one by one. When FOKAS found itself unable to explain an example with its current state of foundational knowledge, it acquired, with the help of a user, the relevant new knowledge that allowed to complete the explanation.

As the quantity of new knowledge increases, the user should become less involved because FOKAS can now use that knowledge rather than the user to complete explanations. Therefore, if the KB converges towards completeness, user interactions should converge towards zero, or at least, towards a minimal level. Hence, by observing the user interaction level, we have a mean of observing convergence of the KB towards completeness. We will see that user interventions decrease significantly as examples are processed by the system, hence supporting the convergence hypothesis.

6.2 Methodology

6.2.1 The Domain Theory (DT)

For the purpose of the experiment, the DT will be limited to two Prolog rules for simplicity. The DT could contain any number of rules: the only reason why only two rules were used is to restrict the number of failures and as a consequence also decrease the number

of interactions, the quantity of new knowledge and the time required to conduct the experiment. This does not make the problem easier to solve: it simply makes it more practical for the experimenter.

The Prolog rules involved in the experiment are the rules defining the childcare_expenses concept in the DT shown at figure 2.2 in section 2.3.1.1. They are reproduced below for convenience, as well as the sentence from which the rules were extracted.

Original Sentence

Child care expenses are the amounts you pay to someone to look after your child while you work or if you are a student.

Domain Theory

```
childcare_expense(Expense,Person,Child):-
      work(Person),
      pay(Person,Expense,Someone),
      look_after(Someone,Child).
childcare_expense(Expense,Person,Child):-
      student(Person),
      pay(Person,Expense,Someone),
      look_after(Someone,Child).
```

Figure 6.1: The sentence (top) which was used to generate the Prolog rules (below) used as DT for the experiment.

These rules define the concept of childcare_expenses. Therefore, we will be interested in proving that an example is an instance of that concept. This rule is disjunctive and uses four leaf sub-concepts (i.e., concepts that are not further defined by the DT) to define the concept of childcare_expenses. The sub-concepts are pay, student, work and look_after.

6.2.2 The examples

Since most people on the project were quite familiar with the original text and the problems we were having with it in term of missing knowledge, volunteers from outside the project were solicited for the provision of real-life examples. In this manner, I ensured unbiased examples were provided. A total of 23 examples were collected from 22 people (family members, friends and neighbors). Each person was asked to write a short story[23] describing their employment and family situation in regard to tax child care deductions. One

[23] The experiment was conducted in French. However, the same general results can be expected for the English language.

person provided two examples. Three more examples were inspired by the tax guide itself. These 26 examples were then translated into ground clauses by the same surface-NLP system used to build the DT.

An important assumption of PAC-KB is that examples should be representative of the situation space (c.f. section 3.3). Since the examples were not obtained in a random manner from the entire population (i.e. they were selected non-randomly from acquaintances), the example set used in this experiment is not representative of the situation space. The only difference this should make however is that we will miss many cases or situations which are not particular to people I know. Hence, the resulting KB of foundational knowledge will not be as complete as it should be given a truly random sample from a whole population.

6.2.3 The surface NLP subsystem

As mentioned in section 2.3, this work evolved from the surface NLP project TANKA. Since the TANKA NLP system and the representation mapping module (the module which provides the HC representation) were not fully operational at the time of the evaluation, and also because the examples I collected were in French and TANKA was designed for the English language, I have implemented my own, basic surface NLP subsystem. This surface-NLP subsystem was used to translate the NL inputs (domain text and examples text) into a Horn clause representation. It consisted of a very simple semi-automatic parser, case assignment unit and prepositional-phrase attachment module. The purpose of this work being to look at the foundational knowledge acquisition aspect, the linguistic modules were kept extremely simple. Actually, as discussed in the limiting assumptions, linguistic considerations were not part of this work. Therefore, the surface NLP module will not be described in detail. A final note on that subject however: due to its simplicity, the surface NLP module outputs required from time to time some final manual fixes to make them usable by FOKAS. However, these fixes could very well be done as part of the semi-automatic NLP process.[24]

[24] The TANKA modules already perform several of these final fixes (for example, pronoun resolution). Others, such as a capability to re-attach prepositional phrases, are being developed at the time of writing this thesis.

6.2.4 The experimental procedure

Out of the 26 examples, two thirds (18) were randomly selected to train the system. The one-third left (7) were kept to test the coverage of the acquired knowledge. The training and testing sets were then used separately to generate interaction data so that the level of interaction of both could be compared. The experiment was conducted in the following manner:

For the training set

1. assert the current example ground clauses.
2. initiate the proving mechanism.
3. the user answers the system's questions and the system keeps track of the total number of interactions for the current example (see section 6.2.5 for how this is done).
4. upon completion of the explanation of the current example (once the system has successfully explained the example by acquiring all relevant knowledge):
   - 4a. retract the example clauses
   - 4b. reset the interaction counter
   - 4c. process the next example starting at step 1. above.

For the testing set

Follow steps 1 to 3 above. For the failures, rely on all the knowledge acquired during training. If required, learn still more knowledge when the current new knowledge does not suffice. Then perform step 4 below (where only 4c. was added):

4. upon completion of the explanation of the current example (once the system has successfully explained the example by acquiring all relevant knowledge):
   - 4a. retract the example clauses
   - 4b. reset the interaction counter
   - 4c. retract any new knowledge that may have been acquired by answering the system's questions for the current example (we do not want to accumulate knowledge incrementally within testing: we want to rely solely on knowledge acquired during training)
   - 4d. process the next example starting at step 1.

The sole difference between testing and training is that the system "forgets" what it has learned on a test set example. We do this to ensure that during testing, the system relies uniquely on the knowledge accumulated during training. The interaction data is collected automatically and accumulated in a file.

The experiment was repeated 7 times. The total number of interactions for each example in each experiment were averaged[25]. For each repetition, the 26 examples were split randomly into a training set and a testing set.

6.2.5 Observations Required

FOKAS learns by being told, and in accordance with Dietterich's (1990) taxonomy of learners, it should be evaluated by looking at how well it exploits the new knowledge it has acquired. In our case, this can be achieved by looking at whether the system becomes more and more autonomous with time. Therefore, the primary aspect we will be interested in observing is the user interaction level as the system sees more examples. What we are hoping to see is convergence towards minimal user interaction such as shown in figure 6.2 below. Note that the curve we will obtain empirically will most likely have a much different look. For instance, interactions may decrease more slowly and less smoothly: the idea here is to show the overall expected results.

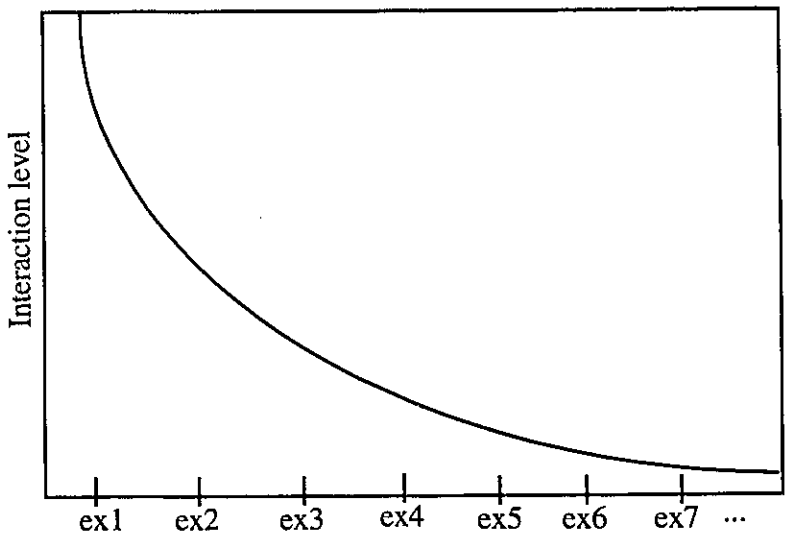

Figure 6.2: Expected curve of interactions with respect to number of examples seen.

[25] This is a rather user-intensive experiment and consequently it was neither realistic nor practical to repeat it more than about a half-dozen times.

There are two metrics used to measure user interaction:

a. the total number of questions: This is simply the total number of questions asked by the system for each example; and

b. the total number of choices: Most questions involve choosing among multiple alternatives (including choosing between yes or no). The total number of alternatives presented to the user is collected for each example.

The rationale behind the second metric is that having to choose among more alternatives is assumed to be more difficult, and thus should take more time. Therefore, a question with two alternatives and a question with 10 should not have the same weight. The first metric does not distinguish between two such questions, while the second does. On one hand, we want to ensure that when the number of questions diminishes, the number of choices does not increase so much as to eliminate the gain. On the other hand, we want to ensure that a decreasing total number of alternatives is not counter-balanced by an increasing number of questions. By observing these two metrics, we can catch such situations.

Finally, I was interested in observing the following three types of behaviors in the interaction data:

a. *Convergence*. Any example e_i will be able to use the knowledge acquired from explaining examples e_1 to e_{i-1}. Therefore, the level of interaction should decrease as examples are processed. This is a measure of the incrementallity of learning *during training*;

b. *Predictive power*. This shows how well the learner behaves when faced with new situations. It also serves as a measure of completeness of the foundational knowledge base we have acquired. It tests learning done during training with a yet unseen set of examples.

c. *Base comparison*. This shows the actual gain made by memorizing the new knowledge and using it on new cases compared to not memorizing it and having to constantly question the user on the same matters. This is the experimental control group.

6.3 Results

Experimental results are shown in the next two graphs. The first graph shows the average total number of questions that were asked by the system while processing each example. The second graph shows the average total number of choices. The testing phase curve is shown as a continuation of the training interaction curve so that we can easily compare the interaction levels. Also shown on each graph are two curves: the top one, a straight line, shows the interaction level given no learning (i.e., no knowledge being accumulated or memorized), while the other curve is the level of interaction with learning.

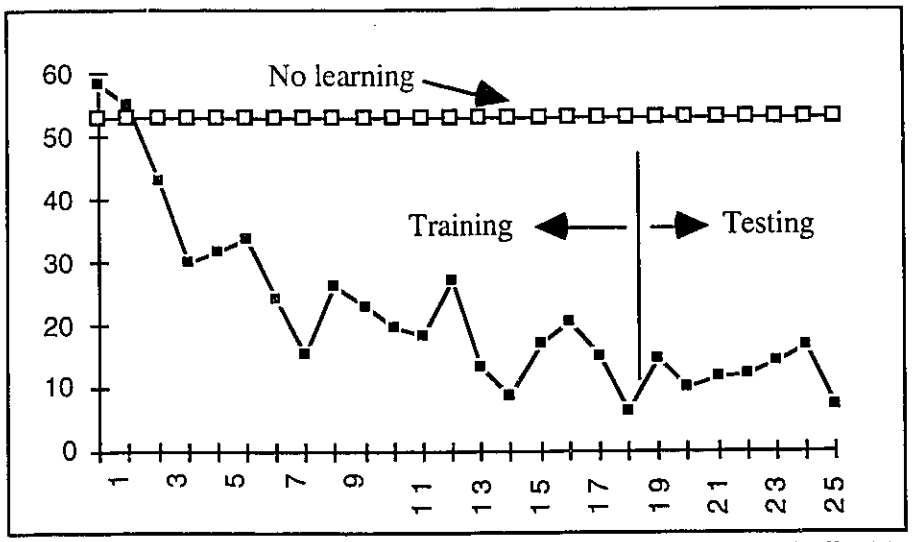

Figure 6.3: Total number of questions generated by the system (Y-axis) to explain each example (X-axis).

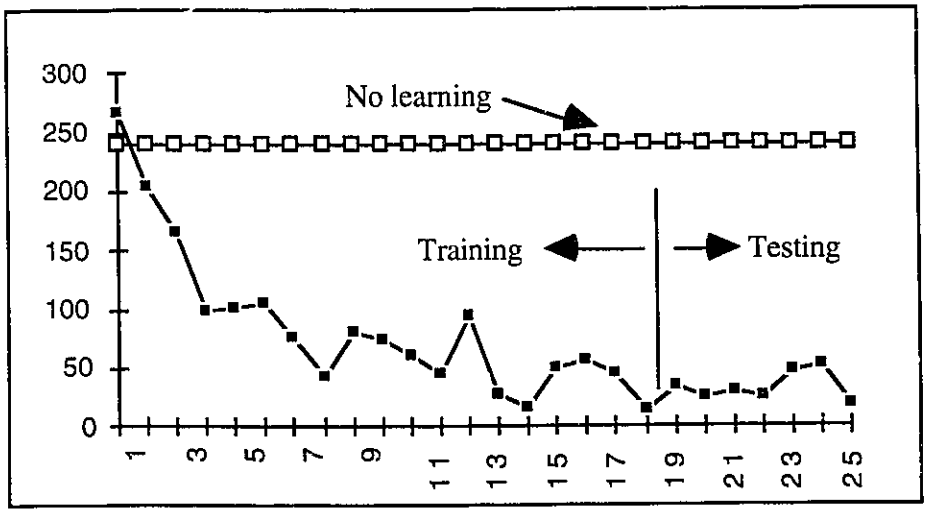

Figure 6.4: Total number of choices the user had to select from (Y-axis) to explain each example (X-axis).

There are four main observations we can make on the results depicted in Figures 6.3 and 6.4. The first three follow from the characteristics I wanted to observe and which I listed at the end of the previous section. The fourth one is an additional observation on the data.

a. *Convergence*. First, we can observe a definite decrease in user interactions (bottom curve of each graph) in the training phase. Both the number of questions and number of choices curves drop significantly, on average, between the first training example and the last one, example 18 (the fluctuations will be discussed below). This illustrates the incremental learning that takes place: each k^{th} example can and does make use of the knowledge acquired from processing the k-1 previous examples.

b. *Predictive power*. Starting at example 19, there was no more learning taking place, so all failures had to rely on knowledge acquired between example 1 and 18 to be solved. If the required knowledge was not available, the user had to provide it on an individual example basis since no memorizing was done. This allows to measure the degree of completeness in the KB we have acquired. We can observe that the user interaction stays relatively low compared to the training part. There is however a small increase of interactions followed by an important decrease on the last example of both graphs. We will see later (section 6.4) how this can be explained.

c. *Base comparison*. The third observation is the user interaction level when there is no learning at all (top straight line of both graphs). This serves as control group, that is what would have happened if we had not acquired and used the new knowledge. The average of all examples interactions after five repetitions was taken as the single interaction level (this explains the perfectly straight line). We can see by comparing this curve with the learning one that there is a definite advantage in acquiring knowledge and using it on new cases.

d. *Fluctuations*. Finally, the graphs still exhibit some major fluctuations. Some are due to the average on only seven repetitions and would most probably disappear if more repetitions were made. Other fluctuations are caused by the distribution of examples, and as such could never be eliminated: they are probabilistically-caused artifacts. In fact, as we repeat the experiment with different draws from the same examples set, these artifacts would become more and more visible. We will see in the next section how this can happen.

6.4 Discussion

6.4.1 Do we have convergence?

To answer this question, we must first explain the phenomena of fluctuations and mainly that of increase of interaction in the test phase as mentioned in section 6.3 above. First, I must say that examples tend to aggregate into clusters, or classes *for each concept that we need to explain.* All examples in one class will be covered by the same (or very similar) new knowledge. That is, once an example in one class has been processed and the new knowledge acquired, the other examples that belong to that class will be explained by the system with minimal interaction. This is illustrated as follow:

Example:

We have seen in Section 6.2.1 Figure 6.1 that the rule used in the experiment involved four leaf sub-concepts to define the concept of childcare_expenses. The sub-concepts are pay, student, work and look_after. Each of these concepts must be explainable, either with or without foundational knowledge, for the examples proofs to be successful. Therefore, for each of these concepts, examples will aggregate into classes for new concepts they introduce.

For instance, for the concept pay, the example set introduces a total of eight new concepts that are related to pay and for which we must learn some knowledge that establishes that relation. These new concepts form classes. Each example belongs to one of the eight classes for the concept pay.

Some of the new concepts related to pay are (represented as ground unit clauses):

have(receipt,expense):	having a receipt for a child care expense
cost(childcare, amount):	child care has cost a certain amount
total(amount,expense):	the total amount of a child care expense

When people want to say that they have paid child care expenses, they will use expressions that introduce these concepts. For instance, of the 23 people who provided examples, three used the concept have(receipt,expense), six the concept cost(childcare, amount), and two the concept total(amount,expense). Examples that used the same new concept to mean pay belong to the same class. FOKAS must establish the correct

relation between pay and these new concepts. This is the knowledge which must be acquired.

Once knowledge has been learned for a class, all examples in this class can be processed with minimal interaction for the DT concept pay. For another concept, such as work, other classes exist, and examples again belong to one of the classes. And so on for each concept of the DT for which a failure can occur.

Probabilistic effects that result in fluctuations and increase of interaction in the test phase arise from the fact that some classes are more populous (i.e. they contain more examples) than others. This is explained in the next paragraphs.

Classes with only a few members will on average tend to be picked towards the end. Once an example is drawn from such a class, the interaction will peak since it has never been seen before. Then the probability of drawing from an unknown class will go down momentarily, since we have acquired new knowledge that covers a few more of the examples (i.e. the other examples in that same class). But as more examples are drawn, and fewer examples are left to be seen by the learner, the probability of not having the proper knowledge will go up steadily until it meets again a new case. Figure 6.5 shows a simplified example of this phenomenon.

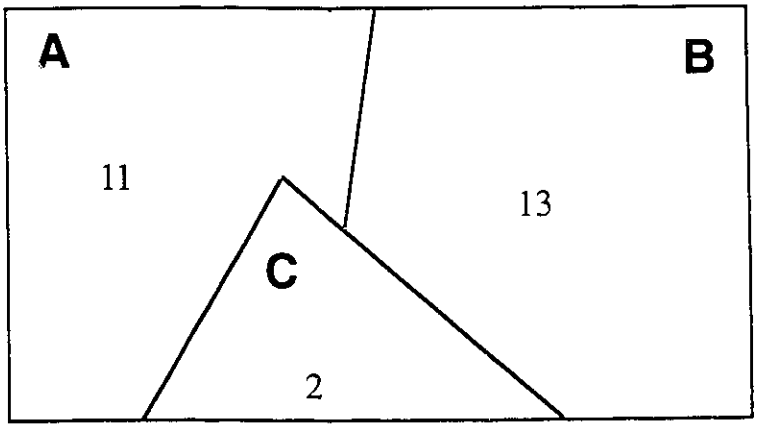

Figure 6.5: The examples may aggregate into 3 classes A,B and C for a particular DT rule or concept. Classes A and B being much larger (i.e., they contain more examples) than class C, examples will be picked from these two classes first with a higher probability. Once an example has been drawn from A and B, the system will have learned most of the knowledge required to process any other example from A and B, therefore bringing interactions with the user to a minimal level. As more examples are drawn from classes A and B, the probability to draw an example from class C increases, and as such the probability of not having the required knowledge since examples from class C have not been seen yet.

These probabilistic effects become more and more evident as the experiment is repeated. Averaging results will hence only smooth out non- probabilistically caused irregularities and tend towards the theoretical probability curve of drawing an example from a new class. Figure 6.6 shows this theoretical probability curve in the case of our example set.

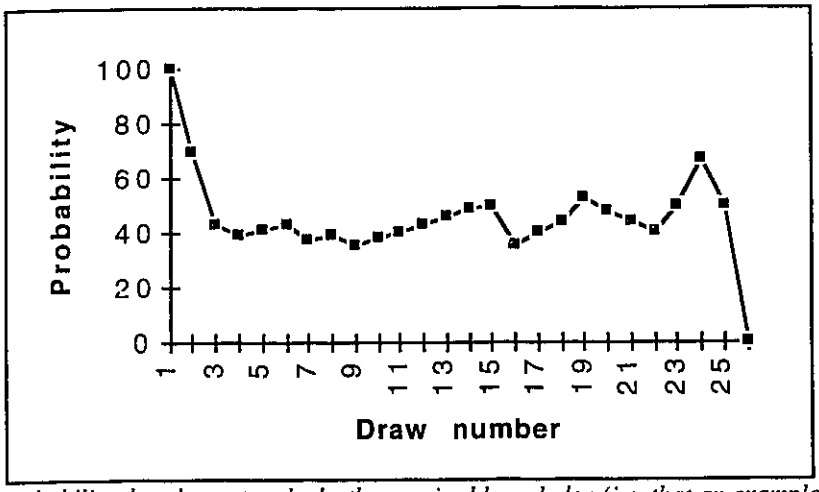

Figure 6.6: The probability that the system lacks the required knowledge (i.e. that an example be drawn from a new, yet unseen class) varies in accordance with the examples distribution in classes (Average probability for all rules, most probable drawing scheme).

The graph of Figure 6.6 shows the variations in the probability of drawing an example from a new class (in other words, the probability of not having the required knowledge to explain the example). This means that where the graph curve is higher, there are more chances of having more interactions. Initially, the probability of not having the knowledge is 1, and it then decreases as examples from the most populous classes are drawn. However, as the population of the most populous classes decreases, there is more and more probability that an example from a smaller class be drawn and that we thus not have the required knowledge. Statistically, examples from the smallest classes will tend to be drawn last, resulting in an ascending curve as we approach the second last example. Then, since no class contains one single example, the last example drawn will always belong to a class that has already been seen. Hence, the probability of not having the required knowledge at that point will be zero.

The probabilities themselves were computed by assuming that an example will always be drawn from the currently most populous class. Therefore, the graph is a most probable approximation of the actual theoretical probability graph. Also, since classes only exist relative to a rule or a concept that we want to explain, we would have a total of four such graphs in our case (because we have four concepts that we need to explain and for which we can fail). This graph is the average of the four curves: this way we can compare it with the experimental

results of Figures 6.3 and 6.4. What follows is an example of how the probability of drawing from a new class is computed:

Example of probability computation:

We have seen in the previous example that the following distribution exists for some of the classes[26] for pay:

Class 1: having a receipt for an expense	3
Class 2: something has cost a certain amount	6
Class 3: the total amount of an expense	2

When randomly drawing the first example, we therefore have the following probabilities (in percent, rounded to the first decimal) to draw from each class:

Class 1: $3/26 \approx 11.5\%$
Class 2: $6/26 \approx 23.1\%$
Class 3: $2/26 \approx 7.7\%$

For the purpose of drawing the graph in Figure 6.6, we assumed that an example is *always* drawn from the most populous class[27]. Actually, this is not what happens: on average, examples tend to be drawn from classes in accordance with the probability of drawing from those classes. The graph of Figure 6.6 is therefore a kind of "best case" approximation, with best case being used in the sense that as long as an event has more chances of happening than others, then it will keep happening.

So, an example will be drawn from class 2. Since it is an example from a new class, knowledge is learned. Since examples are not replaced, the probability to draw from this class will decrease, but will still be higher than the others. We will keep on drawing from class 2, and interactions will be minimal since we have already acquired the bulk of the required knowledge to explain examples of that class. Therefore, the probability to draw an example from an unknown concept class will steadily increase

---

[26] There are a total of eight classes for pay: to keep this example of the probability computation short and simple, I show the computation for only three of them.

[27] When two classes have equal membership, we alternate: for the first such event, we keep on drawing from the same class we were drawing from before the event occurred; then the next time we have two classes with equal membership, we select the other class than the one we were drawing from; and so on.

until we draw from that class. This will happen when the population in class 2 has decreased to 2. Class 1 will then become the most populous and an example drawn from it. Since it is the first time we see an example with this concept, we learn new knowledge and the interaction level increases.

After drawing the fourth example, always assuming a draw from the most populous class, we have the following probabilities to draw from each class:

Class 1: 3/22 ≈ 13.6%
Class 2: 2/22 ≈ 9.0%
Class 3: 2/22 ≈ 9.0%

Once we have drawn an example from class 1, probabilities to draw from any class will be equal. We then select a new class, which causes interactions to go up again. We therefore get a saw-like graph for interactions.

By comparing the probability curve with the empirical results, we can see that an increase of interaction towards the second last example is highly probable and will be followed by a high probability (100%) of low interaction on the last example. Moreover, we can observe fluctuations in the probability curve, which means for us that we have to expect fluctuations such as these as we average our results. Theoretically, our result curve should still smooth out as we repeat the experiment, but in the end, it should nevertheless exhibit a major increase at the end and some "waves" corresponding to higher or lower probabilities to see an example from a new class.

In light of the above discussion on the example distribution among classes and the subsequent probabilistic effect of drawing from smaller classes last, we can say that convergence occurs in an acceptable manner (i.e. the slight increase of interactions in the testing phase can be accounted for). However, in order to really determine what is acceptable as convergence, we would have to look at specific tasks and applications. In some cases, a constant but low intervention level by a user may be acceptable, while in other situations we may want a guarantee that the interactions will be very few and very seldom, and actually quantify the "few" and "seldom" precisely. This aspect was explored briefly in the PAC discussion in section 3.3.

There are two more aspects of the learning we must consider before concluding this section: utility and usefulness.

6.4.2 Is there a danger of suffering from the utility problem?

During the experiment, there were instances where the new knowledge was resulting in many useless inferences and questions to the user. This is a form of the utility problem (Minton et al. 1989). Most of these cases could be attributed to over-general rules. These rules in turn were almost always originating from generic verbs such as "have".

For instance, in some examples, the verb "have" may have been used for three or four different facts such as "I have a husband", "I have children", "I have a salary of $40000". Surface NLP would translate these into facts such as:

```
have(i,husband).
have(i,children).
have(i,salary, 40000).
```

Suppose we have acquired the following foundational knowledge from previous examples, and these two rules are present in the KB in the order they are listed here:

```
pay(S,A):-
        have(S,R,A),
        isa(S,human),
        isa(A,amount_of_money),
        isa(R,proof_of_payment).

pay(S,A):-
        cost(S,A,P),
        isa(S,human),
        isa(A,amount_of_money),
        isa(P,service).
```

Also, suppose we have the following example:

```
have(i,husband).
have(i,children).
have(i,salary, 40000).
cost(it,me,2500,childcare)
```

Let's assume that FOKAS is processing this example and that the explanation mechanism of FOKAS is now looking at the DT clause for the concept pay. Since there is no predicate pay in the example, a local failure will occur. FOKAS will then try the first of the above foundational rule in an attempt to avoid questioning the user. In order for this rule to succeed, FOKAS has to find a ground clause that uses the predicate have so that it matches the first literal of the

foundational knowledge rule. FOKAS will try all "have" clauses from the example and query the user every time something does not work (trying to find equivalent types, incompatible arities, etc). However, none of the "have" clause is accepted by the user. The system must then retract the first rule from active memory and assert the second one, which uses the predicate cost instead of have. This rule will work as it can match with the clause cost(it,me,2500,childcare) of the example (note that FOKAS will have to learn the required matching list because the two literals do not have the same arity).

FOKAS does not know which rule to use first, and will try everything to make the current rule work because it assumes a failure is an indication of missing knowledge. FOKAS goes in the order it finds the rules, so if the n^{th} rule is the right one, the n-1 rules before that one will result in numerous queries.

In general, however, this utility effect seems outweighed by convergence, as evidenced by the graphs. Since we have considered only a small training set, further experiments with larger sets are needed to demonstrate without doubt that some utility problem effect indeed does not appear in the long term as more rules are learned.

One may also question the fact that there is an increase of interaction as we go towards the last example. Such an increase could be a sign that as we accumulate more knowledge, FOKAS becomes more "entangled" in it, and cannot differentiate among multiple rules for the same concept. However, I previously demonstrated how the interactions increase in the end of the testing set can be accounted for by draws from small classes. A point in favour of such a probabilistic effect rather than a utility one is that during testing (where the most important increase of interaction occurs), no new knowledge is accumulated. Therefore, an increase in interactions caused by an increase in the quantity of knowledge is illogical at that point since there is no such increase of knowledge in testing. Therefore, an increase of interaction can only be a probabilistic artifact.

6.4.3 Is the knowledge acquired useful?

Usefulness is a rather subjective quality, but in general we can say that the rules the system has learned are useful in the sense that they are true common sense and general world knowledge that could be used by a KBS. Rules shown below are typical rules learned

by FOKAS (where the isa literals have been removed and replaced by more legible argument names):

```
parent(Person1, Person2):-
        father(Person1, Person2).

look_after(Employee, Child):-
        leave(Parent, Child, Daycare_center),
        work(Employee, Daycare_center)

live(Person, Location2):-
        move(Person, Location1, Location2).
```

The first rule gives us a partial definition of the concept parent. By seeing an example that introduces the concept mother, the full parent definition would be acquired. The second rule can be seen as representing a short script about daycare centers. Indeed, it states events that take place in a daycare center: an employee of the daycare center looks after a child if one of the parents of the child leave her at the daycare center and the employee does indeed work at the daycare center. The last rule is common sense knowledge stating that when a person is moving from a location1 to a location2, then this person now lives at location2.

We can see from the above the wide range of useful knowledge FOKAS can learn: concept definitions, scripts and common sense knowledge. Moreover, this knowledge is general enough to be used in other domains than the tax domain.

Other rules are less useful, as the ones that cause useless inferences (c.f. section 6.4.2). Yet others are more or less relations between synonyms such as:

```
look_after(S,C):-
        take_care_of(S,C),
        isa(S,human),
        isa(C,child).
```

Such rules however encode linguistic knowledge that may be quite specific to a domain, and that would be difficult to come up with a priori. As such, they have a certain usefulness. Others are more commonly available linguistic knowledge which could be made available via existing on-line dictionaries rather than learning.

6.5 Summary

In this section, we have seen that FOKAS allows us to learn new, useful foundational knowledge. Moreover, this new knowledge can be used in novel situations to fill knowledge gaps with minimal user interventions where without that new knowledge, it would have required intensive user participation. We can observe a definite convergence towards minimal user involvement, which means the acquired KB becomes closer and closer to completeness as examples are processed.

Based on the study of probability effects in Section 6.4.1, we can also postulate the form of a general curve of interactions given a larger and possibly infinite example set (see Figure 6.7). In this curve, user interactions converge asymptotically towards zero (or some acceptable minimal level), with occasional peaks when new concepts are encountered. We can expect these peaks to become more and more distant as more examples are processed. However, due to the continually changing nature of the world and of language, peaks can be expected to occur infinitely as examples illustrating new situations (or old situations expressed with a new vocabulary) are processed.

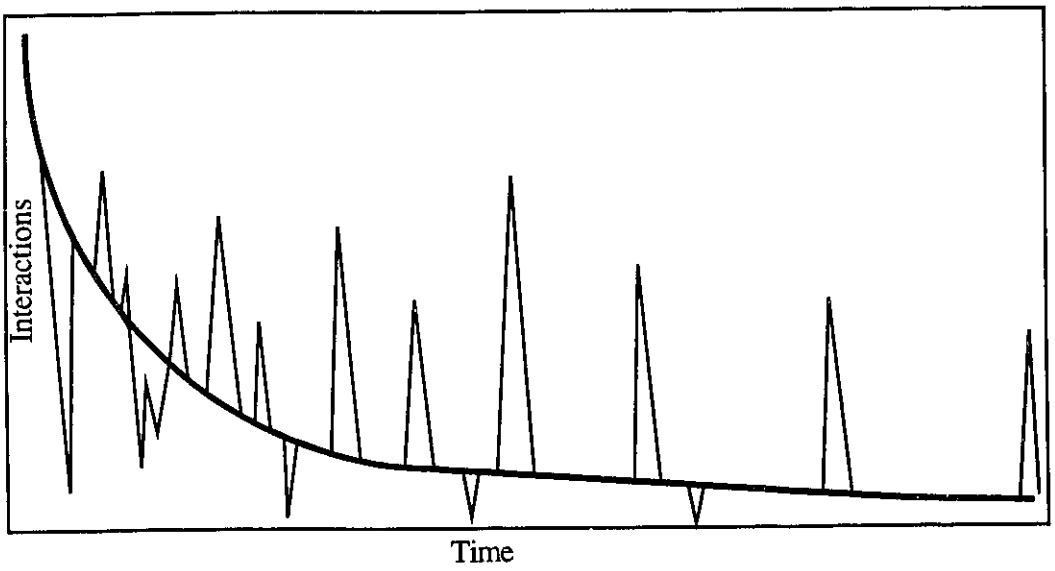

Figure 6.7: Long term interaction curve. High interaction peaks become more and more distant.

Chapter 7 - Concluding Remarks

7.1 Future Work

In this section, areas in need of additional research as well as possible enhancements to FOKAS will be presented. This section has no intention to present solutions; rather it aims at identifying problems that still need to be solved.

1. *Inconsistencies* The system as implemented can only handle positive examples, and as such can only learn knowledge involved in the non-coverage of a positive instance of a concept by a DT (incompleteness problem). It would be interesting to investigate how the system could be extended to handle negative examples that end up being proved (inconsistency problem).

2. *Dictionaries.* FOKAS would greatly benefit from the use of public domain dictionaries and word ontologies such as WordNet (Miller 1991). This would eliminate many of the most mundane interactions and avoid the learning of some of the more useless rules (c.f. section 6.4.3). Finally, it would be a very good tool to help generalizing both the predicates and the arguments. On the other hand, this could be considered prior knowledge, which goes against our initial assumption of having no such knowledge. This is why dictionaries or ontologies would have to be publicly available (or at least already in existence) and not specially built or tailored for a particular task the system performs. A first step has already been done in that direction within the TANKA/MALTE project (Li et al. 1995).

3. *Common sense reasoning.* Presently, FOKAS uses the new knowledge in a rather limited manner. When a failure is encountered, it attempts to find a rule, built from a previously processed knowledge gap, which will also apply or closely match the current situation of missing knowledge. However, when learning the new rules, other knowledge is acquired "on the side". For example, the system learns about equivalent and related constants, and also other predicates that relate or qualify objects (see sections 4.2.4.3 and 4.2.5.1). This knowledge is not used presently. It would be interesting to see if FOKAS could further decrease the user involvement if it were used.

4. *Wider scale training*: The experiment described in this thesis suffered from two problems: it was not feasible (in terms of resources) to obtain both a larger set of examples (for

training and testing) and to repeat the experiment more than about seven times (to smooth out the learning curves). On the other hand, although a rather small training set of 26 was used, it resulted in a significant decrease of user interaction. The next logical step is to verify how much farther down user interaction can be pushed, and how much more knowledge could be acquired given more examples. Additionally, as discussed in section 6.4.2, more training examples may result in more knowledge and therefore in more apparent utility effect, which could potentially increase user interaction. The convergence hypothesis would be better confirmed if we could observe a graph like the one of Figure 6.7 with long term training. However, a wider scale training may be a difficult endeavor because: first it involves sampling from a large population of people; and second, each individual must be willing to write a text describing a particular situation, and write this text without forgetting important details.

5. *Refine the implementation.* The current implementation was built with a simple aim in mind: it was to be a discardable prototype to be used as a proof of concept. There are therefore many refinements that could be added to a future implementation. For instance, a better user interface (graphics-based) and NL interactions (rather than presenting clauses) are possible improvements. Also, adding a KB editing facility to correct user errors would be required. Finally, full integration with TANKA when the representation mapping module is completed (c.f. section 2.3.3) is required to demonstrate the full capability of the approach.

6. *Reuse the new knowledge.* Since the knowledge FOKAS acquires is foundational knowledge, it should be usable in other applications and even in other domains. For example, it would be interesting to feed back the new knowledge to the NLP module so that it can be used for linguistic analysis. We could then obtain a less superficial representation of the text's knowledge which would decrease user interactions within the linguistic module.

7. *Real world use.* There are still at least two outstanding questions that need to be resolved before the approach I presented can be used for real world KA. First, can a surface NLP system (including the mapping to HC) that provides a usable output for FOKAS with decreasing user interactions be produced? Second, can declarative knowledge be made procedural? (example, actually *multiply* when we have the predicate symbol multiply extracted from the text). The latter is not always important in advisory systems. For instance, a sentence stating to give medication X to a patient should not result in an actual action by a medical expert system (in 1995 anyway). On the other hand though, a text stating: "if some quantity X is smaller than Y, then a medication Z1 should be given; while if the contrary is true, then a

medication Z2 should be administered", then the operation of comparing X and Y should be performed by the KBS and the proper advice given based on the result if this operation.

8. *Develop the idea of PAC-KB.* I have merely outlined what a PAC-KB theory might be by presenting the basic ideas behind it. An in-depth study of these ideas and of how they can be applied in practice is required.

7.2 Contributions

We have seen that what used to be a major problem in NLP (the prior knowledge problem) can be turned into an advantage. Indeed, by using surface NLP on both a domain text and stories illustrating that text, we have shown that *not* hand-crafting the prior knowledge results in knowledge gaps which are themselves great learning opportunities. Hence, by superficially understanding a domain text, one can semi-automatically acquire both useful domain and foundational knowledge.

In this thesis, I have:

a. presented a novel method that addresses the problem of acquisition of foundational knowledge in a setting where the initial DT is acquired from text;

b. shown that foundational knowledge can be acquired in a relatively easy manner (compared to conventional prior hand-crafting) by guiding a user through the process of filling knowledge gaps; and,

c. demonstrated empirically that the acquired knowledge converges towards acceptable completeness and defined acceptable completeness within the framework of PAC-KB.

The main contribution of this work is the demonstration that knowledge gaps and examples processed by an explanation mechanism allow for the acquisition of foundational knowledge on an as-needed basis, or lazily. User's cooperation is necessary, but I have demonstrated empirically that the volume of this interaction decreases with time.

This has two important consequences for KA in general: on one hand, it allows NLP to be used to acquire domain knowledge from texts, with no initial KA effort, something that has been impossible up to now. On the other hand, it allows for the acquisition of foundational knowledge incrementally and semi-automatically during problem solving, with convergence to an acceptably complete KB of foundational knowledge. The examples provide realistic constraints on what knowledge has to be acquired, rather than facing an entirely open-ended manual KA effort prior to the task. This is a very important point since foundational knowledge consists in great part of common sense and general world knowledge, knowledge which is very difficult to acquire.

In short, elicitation of domain knowledge can be done from existing texts, effectively confining the role of the expert to the refinement of the KB. This refinement takes place by answering specific queries from a system that processes examples. This is very similar to what has been done with semi-automatic KA systems such as TEIRESIAS (Davis 1979) and PROTOS (Bareiss 1989) except that in our case, the initial KB comes from text and the knowledge acquired by the system is mainly foundational knowledge.

Another contribution of this work is that it can be seen as a potential alternative to Cyc. As pointed out by Minsky (Minsky 1994 p. 27-28) no such alternatives have been presented up to now. FOKAS acquires foundational knowledge just as Cyc does, but it does so lazily, only for the knowledge which is required for a given task. Additionally, it does it in the context of a synergistic integration of NLP and ML, hence presenting a kind of "cheap lunch" (c.f. the "free lunches" in Lenat & Guha 1990 p. 24-26).

7.3 Conclusion

To conclude, I first summarize the advantages of the lazy KA approach to foundational knowledge:

a. it allows NLP to be used for the acquisition of domain knowledge with no foundational knowledge having to be hand-crafted beforehand;

b. it allows for the acquisition of foundational knowledge required for a particular domain and task from specific situations, during problem solving;

c. the acquisition is done incrementally so that any new situation will result in new knowledge being acquired;

d. convergence to an acceptably complete KB of foundational knowledge can be done by supervised training;

e. the knowledge acquired is useful world and common sense knowledge that could be fed back to the NLP system for more in-depth and meaningful analysis of text or reused by other KBS; and,

f. because it is targeting specific missing knowledge as constrained by the examples, the resulting KB of foundational knowledge will be smaller (compared to Cyc's for instance), less of an overhead, and will also make inferences faster.

Secondly, it seems right to finish this thesis with remarks that are not necessarily so much down to earth as the rest of this work. It allows one to open up to what is next; to see what the future might bring if we successfully pushed this research farther ahead.

The work presented in this thesis can ultimately be seen as *compiling natural language*.[28] The input is a natural language textual specification, and the output a program executable for problem solving.

We could assume the following scenario for software development. An incomplete and only partially correct text specification is compiled into an initial program. The program can then be tested and fixed via training with examples of problems it is supposed to be able to solve. The resulting program can be guaranteed to work properly (acceptability parameter) most of the time (confidence parameter) given a training set of the right size and sampling.

There is however much work left to arrive at this ultimate goal. For instance, words must somehow be linked to actual internal machine operations. We have studied declarative knowledge, but not operational knowledge. If a specification states "draw a circle", the program must indeed draw a circle and not merely declare that it must draw a circle. In a way, this eventually links this work with software reuse, where libraries of objects can be called upon when specific operations are required. In turn, objects could be built from less complex ones in the same manner: by compiling textual specifications. In the end, some basic objects

[28] I make a parallel here with compiling the source code of a programming language.

only may have to be hand-crafted. These might very well already exist under the form of machine instructions since it is on these instructions that everything today's computers do is based. A PAC theory for Logic Programs Learning must also be developed. I have merely outlined what such a theory might be specifically for KB completeness.

As such, what I have proposed in this thesis with the Lazy KA method is more of a *first step* towards KA from text than an all-encompassing and exhaustive study of KA from text (and eventually of program development from text). Specifically, I have presented a proof of concept that shows that the idea of lazily acquiring foundational knowledge missing from a DT obtained by using surface NLP is workable in its simplest form. I have listed in section 7.1 many of the future work areas. I think that the most important ones, the ones that will make or break this approach, are the thorough investigation of surface NLP, wider-scale training and the real world use of the resulting KB.

The question of whether surface NLP with lazy KA of foundational knowledge is faster and better as a KA tool than other approaches must also be answered. Therefore, it would be extremely valuable to compare the acquisition times and quality across different KA methods. For this purpose, I would propose, as a final comment, the following general framework for an experiment:

Given:

- a text and experts on a particular domain
- the required personnel and KA systems they are experienced with

Build the following KBs in parallel[29]:

KB1: Use conventional, foundational knowledge-intensive, NLP methods. Preparation is done on site, with no prior warning as to the domain.
KB2: Use conventional knowledge engineering with an expert
KB3: Use a state-of-the-art semi-automatic KA tool such as PROTOS with preparation done on site if required
KB4: Use a surface NLP and lazy KA system.

[29]There is always a problem with this type of experiment: if the same people work serially on the same problem, but with different methods, they may gain experience which will bias the results. On the other hand, using different people, and in particular different experts, may result in not having people of the same strength or quality. For instance, an expert may be very collaborative with the knowledge engineer while the other will not be that collaborative with a semi-automatic KA system. However, I think using different people who are experienced with a particular method is the most appropriate since it gives an equal chance to all approaches.

Collect the total time required to build a KB of a certain number of rules. Then evaluate empirically the quality of each KB with the same performance task (in the case of this thesis experiment, it was the task of explaining examples). A method that acquires the required number of rules rapidly, but for which performance is poor should be allowed to go back to acquire more knowledge until performances becomes acceptable. Time to acquire that new knowledge will be added to the initial time. Hence, all systems could be compared on an equal basis of at least acceptable performance. Then, systems that perform better get extra points.

A final validation of the approach presented in this thesis will only be made when such a comparison of methods shows conclusively that surface NLP and lazy KA work better and faster than existing alternatives.

References

Allen, J. (1987). Natural Language Understanding, Benjamin/Cummings, 1987.

Angluin, D. (1988). "Queries and Concept Learning", Machine Learning, 2, pp. 319-342, 1988.

Bareiss, R., Porter, B. W. and Murray, W. (1989). "Start-to-finish Knowledge-base Developement", Machine Learning, 4, pp. 259-283, 1989.

Bareiss, R. (1989). Examplar-Based Knowledge Acquisition: A Unified Approach to Concept Representation, Classification and Learning, Academic Press, 1989.

Barker, K. (1994) "Clause-Level Relationship Analysis in the TANKA System", Department of Computer Science,University of Ottawa, TR-94-07, 1994.

Bozsahin, H. C. and Findler, N. V. (1992). "Memory-based hypothesis formation: heuristic learning of commonsense causal relations from text", Cognitive Science, 16, pp. 431-454, 1992.

Copeck, T., Delisle, S. and Szpakowicz, S. (1992) "Parsing and Case Analysis in TANKA", Procs. of 15th Int Conf on Computational Linguistics COLING-92, pp. 1008-1012, 1992.

Davis, R. (1979). "Interactive Transfer of Expertise: Acquisition of New Inference Rules", Artificial Intelligence, 12, pp. 121-157, 1979.

Davis, E. (1990). Representation of Common Sense Knowledge, Morgan Kaufmann, 1990.

De Raedt, L. and Bruynooghe, M. (1992a). "An Overview of the Interactive Concept-Learner and Theory Revisor CLINT", in Inductive Logic Programming, S. Muggleton, ed., Academic Press, pp. 164-191, 1992a.

De Raedt, L. and Bruynooghe, M. (1992b). "Interactive Concept-Learning and Constructive Induction by Analogy", Machine Learning, 8, pp. 107-150, 1992b.

De Raedt, L., Lavrac, N. and Dzeroski, S. (1993) "Multiple Predicate Learning", Procs. of ILP 93 Workshop, 1993.

DeJong, G. F. and Mooney, R. (1986). "Explanation-Based Learning: An Alternative View", Machine Learning, 1, pp. 145-176, 1986.

Delannoy, J. F., Feng, C., Matwin, S. and Szpakowicz, S. (1993) "Knowledge Extraction from text: Machine Learning for Text-to-rule Translation", Procs. of ECML 93, Notes From the Machine Learning Techniques and Text Analysis Workshop, pp. 7-14, 1993.

Delisle, S. (1994) "Text Processing without a priori Domain Knowledge: Semi-automatic Linguistic Analysis for Incremental Knowledge Acquisition", PhD Thesis, University of Ottawa, 1994.

Dietterich, T. G. (1990). "Machine Learning", Annual Review of Computer Science, 4, pp. 255-306, 1990.

Duval, B. and Kodratoff, Y. (1990). "A Tool for the Management of Incomplete Theories: Reasonning about Explanations", in Machine Learning, Meta-Reasoning and Logic, P. B. Brazdil and K. Konolige, ed., Kluwer Academic Press, pp. 135-158, 1990.

Eshelman, L., Ehret, D., McDermott, J. and Tan, M. (1987). "MOLE: A tenacious KA Tool", International J of Man-Machine Studies, 26, pp. 41-54, 1987.

Feigenbaum, E. A. (1977) "The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering", Procs. of 5th IJCAI Conference, pp. 1014-29, 1977.

Feldman, R. and Nedellec, C. (1994) "A Framework for Specifying Explicit Bias for Revision of Approximate Knowledge Bases", Procs of Knowledge Acquisition for KBS Workshop, 1994.

Fillmore, C. (1968). "The Case for Case", in Universals in Linguistic Theory, B. E. and R. T. Harms, ed., Holt, Rinehart and Winston, pp. 1968.

Genest, J., Matwin, S. and Plante, B. (1990) "Explanation-Based Learning With Incomplete Theories: A Three Step Approach", Procs. of ML 90, 1990.

Gold, E. M. (1967). "Language Identification in the Limit", Information and Control, 10, pp. 447-474, 1967.

Guha, R. V. and Lenat, D. B. (1994). "Enabling Agents to Work Together", Communications of the ACM, 37, pp. 127-142, 1994.

Hauptmann, A. G. (1991) "From Syntax to Meaning in Natural Language Processing", Procs. of AAAI-91, pp. 125-130, 1991.

Haussler, D. (1988). "New Theorical Directions in Machine Learning", Machine Learning, 2, pp. 281-284, 1988.

Jackson, P. (1990). Introduction to Expert Systems, Addison-Wisley, 1990.

Jacobs, P. ed. (1992). Text-Based Intelligent Systems, Lawrence Erlbaum, Hillsdale, NJ, 1992.

Kedar-Cabelli, S. T. and McCarty, L. T. (1987) "Explanation-Based Generalization as Resolution Theorem Proving", Procs. of 4th International Workshop on Machine Learning, pp. 383-389., 1987.

Kieras, D. E. (1985). "Thematic Processes in the Comprehension of Technical Prose", in Understanding Expository Text - A Theoritical and Practical Handbook for Analysing Explanatory text, B. &. Black, ed., LEA, pp. 89-105, 1985.

Kietz, J.-U. and Wrobel, S. (1992). "Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models", in Inductive Logic Programming, S. Muggleton, ed., Academic Press, pp. 335-359, 1992.

Kim, J.-T. and Moldovan, D. I. (1993) "Acquisition of semantic patterns for information extraction from corpora.", Procs. of 9th IEEE Conference on AI Applications, 1993.

Kolodner, J. (1984). Retrieval and Organizational Strategies in Conceptual Memory: A computer Model, Erlbaum, Hillsdale, NJ, 1984.

Laird, J. E., Hucka, M., Yager, E. S. and Tuck, C. M. (1990) "Correcting and Extending Domain Knowledge Using Outside Guidance", Procs. of Machine Learning, 1990.

Lenat, D. B. and Guha, R. V. (1990). Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project, Addison-Wesley, 1990.

Li, X., Szpakowicz, S. and Matwin, S. (1995) "A WordNet-based Algorithm for Word Sense Disambiguation", Submitted to IJCAI-95.

Matwin, S. and Szpakowicz. (1993) "Text Analysis: How Can Machine Learning Help?", Procs. of PACLING-93, pp. 33-42, 1993.

Mellish, C. S. (1985). Computer Interpretation of Natural Language Descriptions, E. Horwood, 1985.

Miller, G. A. (1990). "WordNet: An On-Line Lexical Database", International J of Lexicography, 3, 1990.

Minsky, M. and Papert, S. (1988). Perceptrons (An Introduction to Computational Geometry), Expanded Edition, MIT Press, 1988.

Minsky, M. (1994). "A Conversation with Marvin Minsky About Agents", Communications of the ACM, 37, pp. 23-29, 1994.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O. and Gil, Y. (1989). "Explanation-based Learning: A Problem Solving Perspective", Artificial Intelligence, 40, pp. 63-118, 1989.

Mitchell, T., Keller, R. M. and Kedar-Cabelli, S. T. (1986). "Explanation-Based Generalization: A Unifying View", Machine Learning, 1, pp. 47-80, 1986.

Mooney, R. J. (1990). "Learning Plan Schemata from Observation: Explanation-Based Learning for Plan Recognition", Cognitive Science, 14, pp. 483-509, 1990.

Moulin, B. and Rousseau, D. (1991) "Extracting Logical Knowledge from Prescriptive Texts in Order to Build Deontic Knowledge Bases", Procs. of 6th AAAI-Sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop, pp. 17.1-17.20., 1991.

Moulin, B. and Rousseau, D. (1992). "Automated Knowledge Aquisition From Regulatory Texts", IEEE Expert, pp. 27-35, October 1992.

Nedellec, C. (1991) "A Smallest Generalization Step Strategy", Procs. of 8th International Workshop on ML, pp. 529-533, 1991.

Porter, W. B., Bareiss, R. and Holte, R. C. (1990). "Concept Learning and Heuristic Classification in Weak-Theory Domains", Artificial Intelligence, 45, pp. 710-746, 1990.

Rendell, L. (1989) "Comparing Systems and Analysing Functions to Improve Constructive Induction", Procs. of 6th International Workshop on Machine Learning, pp. 461-464, 1989.

Rich, E. and Knight, K. (1991). Artificial Intelligence, McGraw Hill, 1991.

Riesbeck, C. K. and Schank, R. C. (1989). Inside case-Based Reasoning, Lawrence Erlbaum, 1989.

Schank, R. C. (1975). Conceptual Information Processing, North-Holland, Amsterdam, 1975.

Schank, R. C. and Abelson, R. P. (1977). Scripts, Plans, Goals, and Understanding, Erlbaum, 1977.

Schank, R. C. (1982). Reading and Understanding - Teaching from the Perspective of AI, Erlbaum, 1982.

Silverstein, G. and Pazzani, M. (1991) "Relational Cliches: Constraining Constructive Induction During Relational Learning", Procs. of 8th International Workshop on Machine Learning, 1991.

Szpakowicz, S. (1990). "Semi-automatic Acquisition of Conceptual Structure from Technical Texts", J. Man-Machine Studies, 33, pp. 385-397, 1990.

Utgoff, P. E. (1986). Machine Learning of Induction Bias, Kluwer Academic press, 1986.

Valiant, L. (1984). "A theory of the learnable", Communications of the ACM, 27, pp. 1134-1142, 1984.

Wilkins, D. C. (1990). "Knowledge-base Refinement as Improving an Incorrect and Incomplete Domain Theory", in Machine Learning: An AI Approach, Vol. III, Y. Kodratoff and R. S. Michalski, ed., Morgan-Kaufman, pp. 493-514, 1990.