THE UNIVERSITY OF CALGARY

An Intelligent Editing Tool for KL-ONE Knowledge Structures

by

Kwang Sim

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JANUARY, 1992

ISBN 0-315-75180-0

The University of Calgary

Faculty of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "An Intelligent Editing Tool for KL-ONE Knowledge Structures," submitted by Kwang Sim in partial fulfillment of the requirements for the degree Master of Science.

Supervisor Mildred L. G. Shaw
Computer Science

Brian R. Gaines
Computer Science

Bruce A. MacDonald
Computer Science

Nigel M. Waters
Geography

Date : January 22, 1992

Abstract

This thesis presents the design and development of an intelligent knowledge editing tool called KLOKE that supports the revision and maintenance of large scale knowledge bases using a KL-ONE based hybrid knowledge representation system. A KL-ONE based system consists of a terminological representation sub-system (called the T-BOX) to represent terminologies and their interrelationships, and an assertional representation sub-system (called the A-Box) to represent assertions about the real world. Much of the research that has been carried out for KL-ONE based systems has focused only on reasoning aspects, and the problem of revision has been neglected. The issues that are addressed in KLOKE are the revision of both terminological knowledge and assertional knowledge. The design objective for KLOKE is to provide incremental and reversible acquisition of both terminological and assertional knowledge.

In KLOKE, objects are represented by facts formed by predicates, and relations of facts are represented by inference rules. Each argument in a predicate is sorted and each sort defines the set of admissible terms for that argument. In KLOKE the definition of sorts and their relations represented in a sort taxonomy are automatically constructed by a sub-module called the sort classifier using the set of facts entered by the user. The acquisition of these terminological definitions or sorts is incremental and reversible. A justification based reasoning maintenance system is used as the assertional representational component of KLOKE. It records the dependencies of rules and facts by building a monotonic data dependency network. Revision of assertional knowledge is possible because one can identify the set of beliefs that lead to a contradicting belief by tracing the derivation paths recorded in the data dependency network.

KLOKE adopts a *sloppy modelling paradigm* which views knowledge acquisition as a cooperative process between the system and the user in a common problem solving activity. Thus the term *cooperative balanced modelling* is used to describe this class of system. The input to KLOKE is a domain model represented by a set of facts and rules. This initial domain model may be incomplete and/or incorrect. The system assists the user in building and restructuring a knowledge model by performing bookkeeping tasks, recognizing conflicts, repairing inconsistencies and hypothesizing about properties of facts to discover missing relations.

Acknowledgements

I am most grateful to my advisor Dr. Mildred Shaw who during the course of my research has provided much thoughtful comment and enouragement. I appreciate the professional guidance and assistance from Dr. Brian Gaines, which have made this thesis possible. My sincere thanks to Dr. Rob Holte, University of Ottawa who brought me into the field of artficial intelligence and to Dr. Lou Birta, Dean of Science, University of Ottawa for acting as my reference. Thanks also goes to Dr. Ian Witten who acted as my interim supervisor and to Dr. Bruce McDonald who provided me with some machine learning papers that were helpful to this thesis. Thanks also to Dr. Bruce Porter, University of Texas at Austin, for sending me some technical reports that are related to this research.

Contents

Approval....ii

Abstract....iii

Acknowledgements....iv

Contents....v

List of Table....viii

List of Figures....ix

Glossary....xi

1 Introduction....1

1.1 Knowledge Acquisition and Domain Modelling....1

1.2 Knowledge Representation....2

1.3 Solutions to the Problem of Knowledge Maintenance....3

1.4 Objectives....8

1.5 A Summary of this Thesis....9

2 Related Work....10

2.1 Hybrid Knowledge Representation Systems....10

2.1.1 An Overview of the KL-ONE System....11

2.2 Knowledge Base Revision....15

2.2.1 An Overview of KREME....15

2.3 Apprentice Learning and Cooperative Balanced Modelling....16

2.3.1 An Overview of LEAP....16

2.3.2 An Overview of DISCIPLE 19

2.3.3 An Overview of the BLIP System 22

2.4 Summary 30

3 System Design of KLOKE 32

3.1 The Architectural Design of KLOKE 34

3.2 Abductive Inference of Intension from Extension in KLOKE 36

3.3 Structural Components 36

3.3.1 Domain-level Knowledge 38

3.3.2 Meta-knowledge 38

3.4 Knowledge Representation Sub-system of KLOKE 40

3.4.1 The Classifier : Acquiring the Sort Taxonomy in KLOKE 41

3.4.2 The Design of the Reasoning Maintenance System 45

3.5 The Discovery Component of KLOKE 48

3.5.1 Module for Acquiring Meta-knowledge 48

3.5.2 The Rule Discovery Algorithm 52

3.6 Summary 57

4 The System Implementation of KLOKE 58

4.1 Implementing the Sort Classifier 58

4.1.1 The Sort Taxonomy 59

4.1.2 Acquiring the Sort Taxonomy 61

4.1.3 A Prolog Implementation of the Sort Classifier 65

4.2 Implementing the Reasoning Maintenance System 79

4.2.1 Recording Data Dependencies 79

4.2.2 A Prolog Implementation of the Reasoning Maintenance System 82

4.3 Meta-Knowledge Acquisition Algorithm 89

4.3.1 Implementing the Meta-knowledge Acquisition in Prolog 90

4.4 Summary 96

5 System Evaluation 97

5.1 An Example of Constructing a Sort Taxonomy 97

5.2 Deleting a Justification 101

5.3 An Example of Acquiring Meta-knowledge 104

5.4 Discussing the Sort Classifier 106

5.5 Discussing the Reasoning Maintenance System in KLOKE 107

5.6 Discussing the Meta-Knowledge Acquisition Module 108

5.7 Summary 113

6 Conclusion and Future work 114

6.1 Summary and Achievements 114

6.2 The User Interface of KLOKE 115

6.3 Discussion : Applying the KLOKE System 120

6.4 Summary 122

References 123

List of Tables

Table 2.1 Typical features of knowledge representation and learning systems.....31

List of Figures

Fig 1.1 A Conceptual Architecture of a Learning Apprentice System.....5

Fig 1.2 A Conceptual Architecture of a Cooperative Balanced Modelling System..... 7

Fig 2.1 A Simple Concept Taxonomy..... 13

Fig 2.2 Functional Components of BLIP..... 23

Fig 2.3 The Modeller of BLIP.....24

Fig 3.1 The System Architecture of KLOKE.....35

Fig 3.2 Model Knowledge in KLOKE.....37

Fig 3.3 Knowledge Representation Sub-system of KLOKE.....41

Fig 3.4 An Example Sort Taxonomy.....44

Fig 3.5 A Data Dependency Network.....47

Fig 3.6 Two Step Generalization Process for nth-level Inference Relation..... 50

Fig 3.7 Process of Generating Hypotheses..... 53

Fig 3.8 Checking Sort Correctness in Rules..... 55

Fig 4.1 An Extended BNF Definition of the Terminological Formalism in KLOKE.....59

Fig 4.2 The State of a Sort Taxonomy Before Deleting a Sort.....74

Fig 4.3 The State of a Sort Taxonomy After Deleting a Sort.....74

Fig 4.4 A Data Dependency Network of the Proposition Given Above..... 80

Fig 5.1 Resulting Sort Taxonomy After Step 1.....98

Fig 5.2 Resulting Sort Taxonomy after Step 2.....99

Fig 5.3 Resulting Sort Taxonomy After Step 3..... 100

Fig 5.4 Resulting Data Dependency Network After Step 1.....102

Fig 5.5 Resulting Data Dependency Network After Step 2.....102

Fig 5.6 Resulting Data Dependency Network After Step 2a.....103

Fig 5.7 Resulting Data Dependency Network After Step 2b.....103

Fig 5.8 Resulting Data Dependency Network After Step 5.....104

Fig 5.9 Generalization Taxonomies.....109

Fig 5.10 A Resolution Tree for Proving the Correctness of Meta-Knowledge.....112

Fig 6.1 Sort Taxonomy Window.....117

Fig 6.2 Lexicon Windows.....118

Fig 6.3 Data Dependency Network Windows..... 119

Fig 6.4 The System Architecture of BINAR.....121

Glossary

KL-ONE : A hybrid knowledge representation system which can represent definitions and assertions.

KREME : Knowledge Representation Editing and Modelling Environment is a versatile knowledge editing tool for developing and editing large scale knowledge bases.

LEAP : An explanation-based learning apprentice system.used in the domain of VLSI design. It consists of an expert system called VEXED and a learning sub-system.

DISCIPLE : A multi-strategy learning apprentice system used in the domain of manufacturing loudspeakers. It has an expert system and a learning component.

BLIP : Berlin Learning Instruction Program, is a cooperative balanced modelling system that interacts with the user to build or revise a knowledge base.

KLOKE : KL-ONE Knowledge Editor is a cooperative balance modelling system that can build or revise a knowledge base by interacting with the user.

BINAR : A proposed cooperative problem solving system which is capable of building an initial knowledge base and learn new knowledge during the process of problem solving.

Chapter 1
Introduction

This thesis presents an intelligent knowledge editing tool called KLOKE. In this chapter, the underlying background of KLOKE is described. This includes providing a unifying view of various areas of research in artificial intelligence. The principles and motivations for this research, the needs for designing such a system, and the solution that has been proposed are presented in this chapter.

Most research in knowledge representation addresses the static aspect. The dynamic aspects of knowledge representation concern reasoning with represented knowledge and revision of represented knowledge. This research investigates the dynamic aspects of knowledge representation in the context of hybrid knowledge representation systems based on KL-ONE. The design and the implementation of KLOKE is presented; it supports both the construction and revision of a domain model.

1.1 Knowledge Acquisition and Domain Modelling

Modelling is the process of constructing a domain model. Morik (1991) analyzes the term *modelling* based on four different perspectives on knowledge acquisition. First, from the perspective of performance, modelling can be described as constructing knowledge base systems. From a constructive perspective, modelling will include the lay-out of a representation language and the revision of all design decisions. One view of modelling from the knowledge-level, can be seen as a knowledge-level analysis of expert system building and applying terminology at that level. Finally, modelling can be described as the transfer of knowledge.

Knowledge Acquisition and the Transfer of Knowledge : Knowledge acquisition has long been considered a process of transferring knowledge from the knowledge engineer to the system. From this perspective, knowledge acquisition is just transferring a model from one representational media to another (eg, ideas conceived in the brain of the expert or on paper to the system). The knowledge acquisition bottleneck metaphor has been

cited in much of the literature in knowledge engineering. However, in more recent approaches and in the case of BLIP and KLOKE knowledge acquisition is an interactive approach (Compton & Jansen 1990) involving both the system and the expert.

Knowledge Elicitation : Knowledge flow from the expert to the system consists of two steps : elicitation and codification (formalization). Knowledge elicitation involves interactions between the expert and the system, including issues such as social and communication skills which are not present in today's computer. Today's technology only permits the construction of manual knowledge elicitation tools which are pervasive in knowledge acquisition. Knowledge elicitation presupposes that the model conceived in the brain of the expert is complete.

1.2 Knowledge Representation

Knowledge is made available after knowledge elicitation, and the next step is to formalize it or represent any changes to it. The system supports the encoding of a model, and the expert builds it. The knowledge representation formalism that the system provides is what the expert uses to construct the model, much like a programmer using a programming language to implement an algorithm.

A Hybrid Representation Formalism : Nebel (1990) states that one essential consideration that must be noted when designing knowledge-based system is to distinguish between different kinds of knowledge. Knowledge about technical vocabulary or knowledge that can be represented naturally using an object-centered knowledge representation formalism is called terminological knowledge. In some applications, having only a terminological formalism will suffice. However, if the application requires the system to reason about objects in the real world, an assertional formalism must also be employed.

A *hybrid representation* formalism will consist of both terminological and assertional formalisms. The assertional formalism describes the real world by making assertions. Terminological representation describes concepts and their interrelations. Each concept denotes a set of objects, and the term *extension* is used to describe this set of objects. Further discussions on hybrid representation systems are given in chapter 2.

Two Views of Knowledge Bases : There are two perspectives in knowledge representation : the static and the dynamic perspective (Nebel 1990). The static view addresses the kind of representation formalism to use for a given domain and the semantics of the knowledge representation language.

In most cases, the explicitly stored knowledge is only a fraction of the amount of implicitly represented knowledge in a knowledge base. As such one can distinguish the difference between knowledge representation and reasoning. This leads to one of the issues in the dynamic aspect of knowledge bases, giving rise to the question as to how can one derive implicitly represented knowledge and control computational resources allocated to this task.

Another aspect of the dynamic perspective of knowledge representation is the issue of changing the knowledge base. In this respect, one is concerned with the problem of revising and maintaining the knowledge in a knowledge base. This thesis only addresses the dynamic view of knowledge bases.

Knowledge Base Refinement : Tools such as knowledge editors, debuggers and menu interfaces have been developed which support the knowledge engineer in encoding and inputting knowledge to the knowledge base. An important issue that must be considered in knowledge engineering is the revision of the model. Another requirement that the system must satisfy is to handle representational deficiencies by supporting changing of the granularity or restructuring of the taxonomy of knowledge. This is needed because in some cases the expert may change his/her mind, or knowledge that comes from the knowledge engineer may not be correct and complete, or it may be contradictory.

1.3 Solutions to the Problem of Knowledge Maintenance

One approach of managing a large amount of knowledge in practical application would be to employ a knowledge base editor such as KREME (Abrett & Burstein, 1987), to directly manipulate the data-structures that are used in implementing the knowledge base. Nebel (1990) mentioned that a more principled solution is to separate the task of revising the knowledge base into two tasks:

- manipulation of formal expressions in the knowledge base
- providing an efficient interface for the user

Michalski (1991) describes learning as the process of creating and modifying knowledge representations. Thus, it follows intuitively that machine learning techniques such as learning by observation and learning by example can be applied to partially automate the first task. The following section describes the integration of some machine learning techniques for constructing and revising a knowledge base. In addition, knowledge editing tools such as KREME will provide the user with the capabilities of browsing and navigating through the knowledge base. The idea behind this research is therefore to design and develop a system that possesses these capabilities.

Integrating Machine Learning into Knowledge Acquisition : There are three approaches in applying machine learning techniques in building knowledge bases (Morik 1988) :

- serial learning
- apprenticeship learning
- cooperative balanced modelling

Serial Learning : In serial learning the system will induce rules independently from the knowledge that is already stored in the expert system's knowledge base. The discovery component does not obtain any feedback from the performance system, that is there are no interactions between the user and the system, unlike the next two classes of systems.

Learning Apprentice System : A learning apprentice system is defined as an interactive knowledge-based consultant that directly assimilates new knowledge by observing, analyzing and questioning about the problem solving steps contributed by their users through their normal use of the system (Mitchell 1990). It consists of a learning sub-system which interacts with a consultant (performance) sub-system as depicted in figure 1.1.

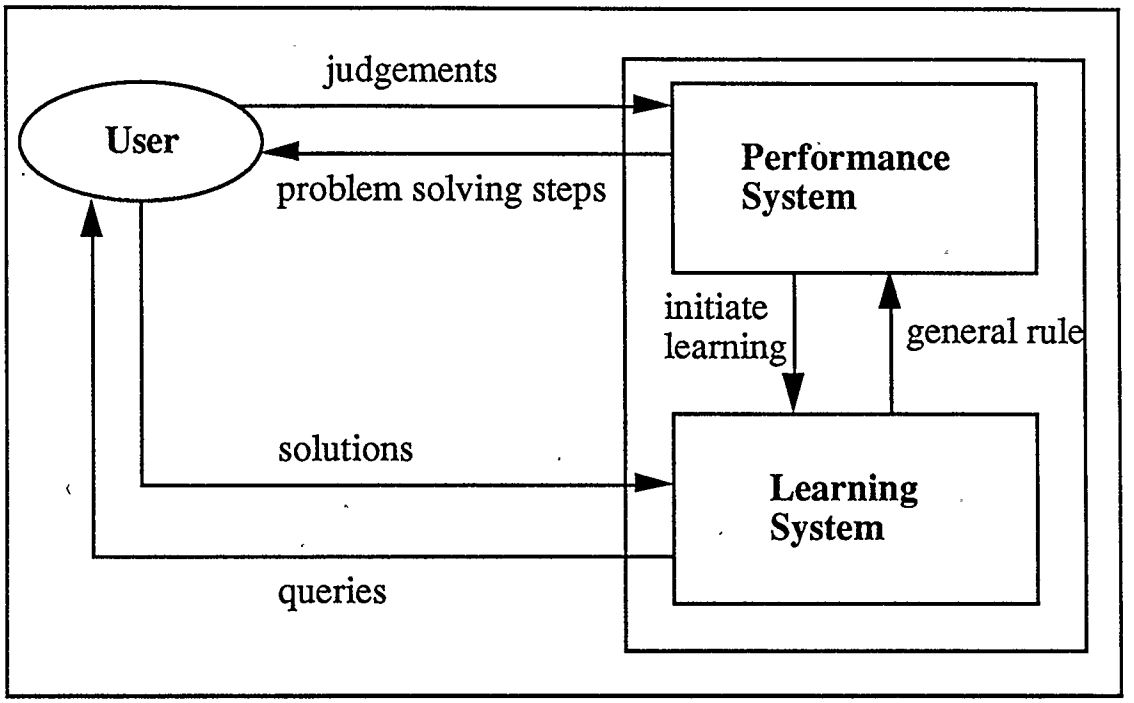

Fig 1.1 A Conceptual Architecture of a Learning Apprentice System

A learning apprentice system has two modes of operations : a problem solving mode and a learning mode.

Problem Solving : In the problem solving mode, the system simply functions as an expert system, where the user will consult the system regarding a particular problem and the system will try to provide solutions to the problem and explain the problem solving steps. Two possible situations can occur during this mode of operations :

- the current problem solving steps are accepted by the user. In this case, the current state of the knowledge base is considered satisfactory and no learning occurs.

- the system is unable to propose any solution or the solution is rejected by the user. In this case the system will prompt the user for a solution to the problem. As soon as the solution is given to the system, learning will take place.

The solutions that the teacher provides to the system will be used as a training example for the learning sub-system.

Learning Mode : During the learning mode, the learning sub-system accepts the teacher's solution as input and the following will take place :

- the system will try to explain the solution provided by the user by questioning the user about its features.

- the system will try to induce a general rule, so that it can propose a solution when it encounters a similar problem in the future.

At present several practical learning apprentice systems such as LEAP (Mitchell 1985, 1990) and DISCIPLE (Kodratoff and Tecuci 1987a, 1987b, 1987c, 1987d, 1988a, 1990), ODYSSEUS (Wilkins 1988, 1990) and PROTOS (Baresis 1989), (Baresis & Porter 1990), (Porter & Baresis & Holte 1990) have been developed. Both LEAP and DISCIPLE will be discussed in chapter 2.

Cooperative Balanced Modelling : Another class of systems in which there are user and system interactions is called *Cooperative Balanced Modelling* systems. The term cooperative balanced comes from the fact the user and the system are both working cooperatively towards a common goal, that is, to construct a domain model. The system is to act as an assistant to the user, and is charged with such responsibilities as bookkeeping, detecting inconsistencies and discovering properties of the givens sets of facts and of the knowledge in the existing knowledge base. The system supports the user both in building the model and enhancing the model by learning. Research has also focused on the types of assistance and how much assistance the system should provide to the user.

Since the system can deal with incomplete and incorrect domain theories, the user is not required to provide it with a perfect domain theory, rather a 'sloppy' domain model will suffice. Thus, the paradigm that is adopted by this class of system is termed *sloppy modelling*. The key point to take note of here is that in a cooperative balanced system, construction of the domain model and learning both occur simultaneously and the method that it adopts involves the participation of both the user and the system.

Conceptually, a Balanced Cooperative Modelling system as shown in figure 1.2 consists of three functional sub-units : a knowledge acquisition environment, a knowledge representation sub-system and a learning component.

BLIP (Morik 1987, 1988), (Wrobel 1987a, 1987c), (Thieme 1987), (Emde 1987b), (Emde & Morik 1987) is the first practical cooperative balanced system that was developed at the Berlin Technical University by a team that is led by Katharina Morik. BLIP will be discussed in Chapter 2. MOBAL, a successor to BLIP was due for completion in October 1991. Much of the design principles of KLOKE are also based on the BLIP system.

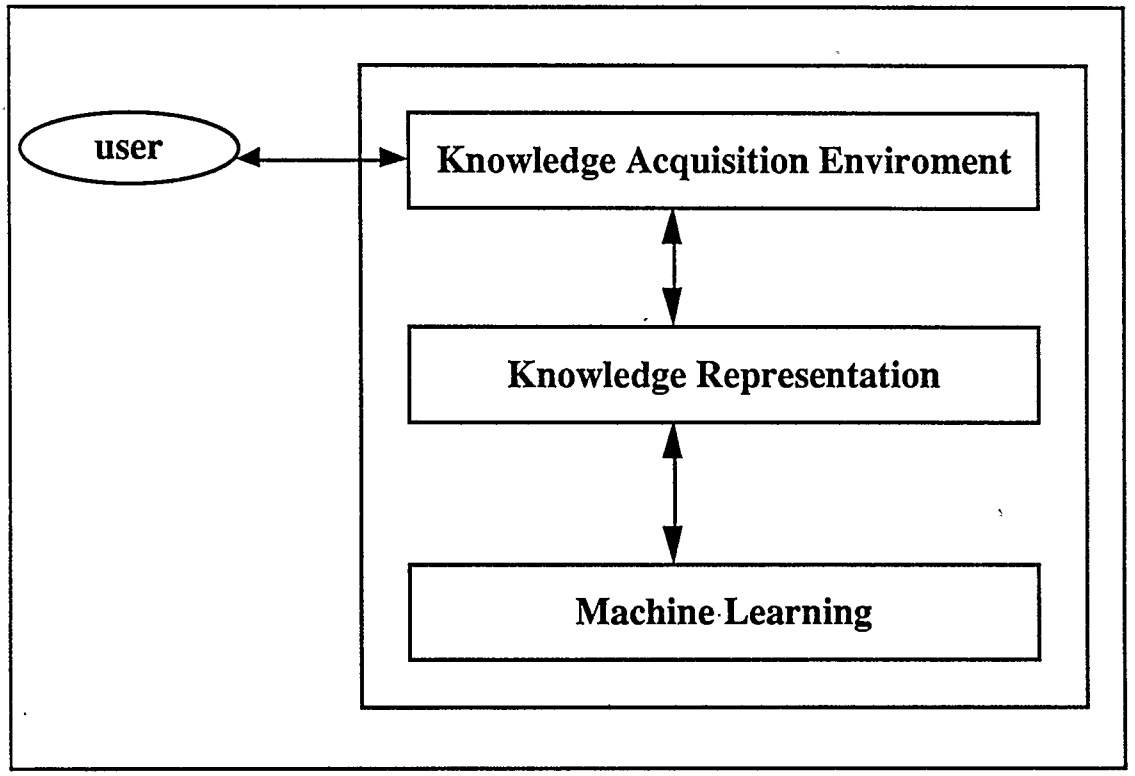

Fig 1.2 A Conceptual Architecture of a Cooperative Balanced Modelling System

1.4 Objectives

Unlike other well established sciences, artificial intelligence is not yet a mature discipline. One argument about the significance of a rational reconstruction is that artificial intelligence as a subject may need some internal repairs, to improve the standard of reproducibility and communicability of the results (Campbell 1990). Reconstruction of an existing system will filter out those parts that are irrelevant and make conspicuous those parts that have not been previously made explicit. The ideas behind the construction of BLIP are studied. Detailed analysis of the strengths and weaknesses of BLIP is given careful attention, while restructuring part of BLIP will also give a better understanding of the system. This research was initially carried out as rebuilding part of the BLIP system. However, in the process, I was inspired by Kodratoff's (Kodratoff & Tecuci 1990) idea of incorporating apprentice learning into a cooperative balanced modelling system. This led to a proposal of a system called BINAR, which will be described in chapter 6. The following summarizes the goal of this thesis:

- To analyse BLIP's principles and achievements
- To rationally reconstruct a substantial part of BLIP
- To develop clear algorithms for KLOKE based on BLIP
- To implement KLOKE in PROLOG
- To evaluate the plausibility of the system on solving problems from the literature
- To analyze the capabilities of KLOKE and to suggest possible enhancements
- To augment KLOKE by adding the capability to acquire knowledge during problem solving

1.5 A Summary of this Thesis

This thesis analyses the principles of BLIP, explains the rationale for reconstructing part of the BLIP system, gives a detailed design of KLOKE and suggests possible enhancements of the KLOKE system.

Chapter 2 surveys related work in the areas of automated knowledge acquisition and machine learning. The principles that KLOKE is built upon relate to many areas of research in artificial intelligence. Systems that are chosen for evaluation and comparison are : KL-ONE, KREME, LEAP, DISCIPLE and BLIP.

Chapter 3 discusses the design of KLOKE; the principles that the system is based upon and the detailed architecture are presented.

Chapter 4 discusses KLOKE's system implementation in BNR (Bell Northern Research's) Prolog version 3.0. This chapter provides some detail of the system implementation.

Chapter 5 gives a discussion of KLOKE. Its performance and limitations are discussed.

Chapter 6 provides a conclusion of this research and suggests future enhancements and augmentation to the KLOKE system. In particular, the design of the user interface is presented. Also presented in this chapter is the proposal for developing a system called BINAR, an augmentation of KLOKE.

Chapter 2
Related Work

The approach employed by KLOKE is based on a combination of techniques from various fields in artificial intelligence. It will be beyond the scope of this thesis to discuss all fields relating to this research. Three areas that are essential and primary to the underlying principles of KLOKE will be discussed in this chapter.

Knowledge Base Editing : On the one hand there is the manual knowledge acquisition environment that supports the knowledge engineer in building and restructuring the knowledge structures in the knowledge base. The area that is related to this aspect of KLOKE is knowledge base revision, specifically knowledge base editing. A prototypical knowledge base editor, KREME will be presented in this chapter.

Machine Learning : On the other hand, the learning component autonomously discovers properties of the knowledge given by the user. This aspect focuses on automating part of the knowledge acquisition and modelling process. Research areas relating to this aspect are machine learning and in particular apprentice learning and cooperative balanced modelling. Learning apprentice systems such as LEAP and DISCIPLE and a cooperative modelling system, BLIP are presented.

Knowledge Representation : Since the user and system are simultaneously working on the domain model a truth maintenance mechanism ensures consistencies in the domain model. In KLOKE this is supported by having a knowledge representation sub-system that includes a classifier for maintaining the *sort taxonomy* and a reasoning system for maintaining assertional knowledge. This aspect relates to hybrid reasoning and representation systems. The systems that are discussed here belong to the KL-ONE family of hybrid knowledge representation systems.

2.1 Hybrid Knowledge Representation Systems

KLOKE includes a knowledge representation sub-system that is based on an architecture that is similar to the KL-ONE class of knowledge representation systems. Recognizing

this relation, this section discusses the KL-ONE based family of hybrid knowledge representation and reasoning systems.

KL-ONE is an implementation of some ideas about the structure of descriptions and their applications in reasoning, which is a computational incarnation of what have been called *structural inheritance networks* (Brachman 1985). Its main feature is that it is capable of forming complex structured descriptions. The objective of KL-ONE is to provide an explicit representation of conceptual information based on the idea of structural inheritance networks.

2.1.1 An Overview of the KL-ONE System

KL-ONE is a knowledge representation system that implements the terminological and assertional distinction described in section 1.2. The assertional sub-system called the A-Box is concerned with representing the beliefs of the system, while the terminological sub-system called the T-Box served to represent the terminologies used to construct assertions and the interrelationships of these terminologies.

Structural Inheritance Network : The formalism in KL-ONE is developed from traditional semantic nets but with emphasis on 'object-centeredness'. In a semantic network, one can find links that represent linguistic relations, arbitrary conceptual relations, semantic relations, logical relations and implementation pointers. Brachman (1979) identified four levels on which semantic networks can be used : the implementation level, the logical level, the conceptual level and the linguistic level.

Implementation level : At the implementation level, networks are basically treated as data-structures with no semantics and having lists and pointers as primitives. That is, nodes represent destinations and links represent pointers.

Logical level : At the logical level, the semantic network is transformed into a notational variant of some logic using primitive logical operators and predicates. At this level a semantic network bears a strong relation to predicate calculus but with additional features of network topology. This provides a means of factoring and organizing the knowledge. Predicates and propositions are represented as nodes while logical

relationships between nodes such as 'and', 'subset' and 'there-exists' are represented by links. The issues of quantification and logical adequacy are dealt with at this level.

Conceptual level : This level focuses on semantic and conceptual relationships. The primitives present at this level are case relationships, primitive actions and other conceptual dependency-graphs. At this level, nodes will represent objects, actions, and events, while links represent case structures.

Linguistic level : This level uses arbitrary words and expressions as primitives and is highly language-dependent. The nodes represent words and the links represent real world relations.

Epistemological Level : All these four levels are self-contained network representations and are not dependent on the levels above or below. Brachman has added another level called the *epistemological* level, that is in between the conceptual and the logical level. At the epistemological level intensional descriptions and their interrelations are formed, this level will include the notion of inheritance and concept-specialization relations. Primitives at this level represent knowledge structures and their interrelationships as knowledge structures independent of the knowledge contained within them. On this level, one will experience the notion of inheritance and concept specialization. To represent the idea of an epistemologically adequate semantic network, Brachman proposed the formalism of *structural inheritance networks*. The main building blocks of this formalism are *concepts*, *roles* and *structural descriptions*.

Concepts : Concepts are terms that are intended to describe classes of individuals. They correspond to conceptually primitive pieces of domain knowledge. There are two kind of concepts in the KL-ONE system : primitive and defined. Primitive concepts are those which are atomic or concepts that cannot be exhaustively described. An example of a primitive concepts is *animal species*. In other words, primitive concepts represent concepts that cannot be specified in terms of necessary and sufficient conditions. Defined concepts are introduced by specifying their necessary and sufficient conditions. They are built from primitive concepts and previously defined concepts.

Roles : Roles are primitives that are used to represent the internal structure of a concept. They contain information about the intensional information of an attribute and also act as description of potential fillers or extensions of the attributes.

Structural descriptions : Structural descriptions account for relationships between roles.

Term-Subsumption : With descriptions in the KL-ONE system, an important issue to be considered is the relation between descriptions, determining if one description subsumes another. To say that a concept *subsumes* another concept, one would mean that every instance of a concept is also an instance of the concept that subsumes it.

Taxonomy : The term *taxonomy* refers to a network structure that is formed by the subsumption relationships between concepts. An example of a simple taxonomy describing the relationships of things in the real world is illustrated in figure 2.1. All ovals marked with '*' are primitive concepts. The concept 'Man' is a defined concept, while the shaded oval represents an individual concept 'Fred'.

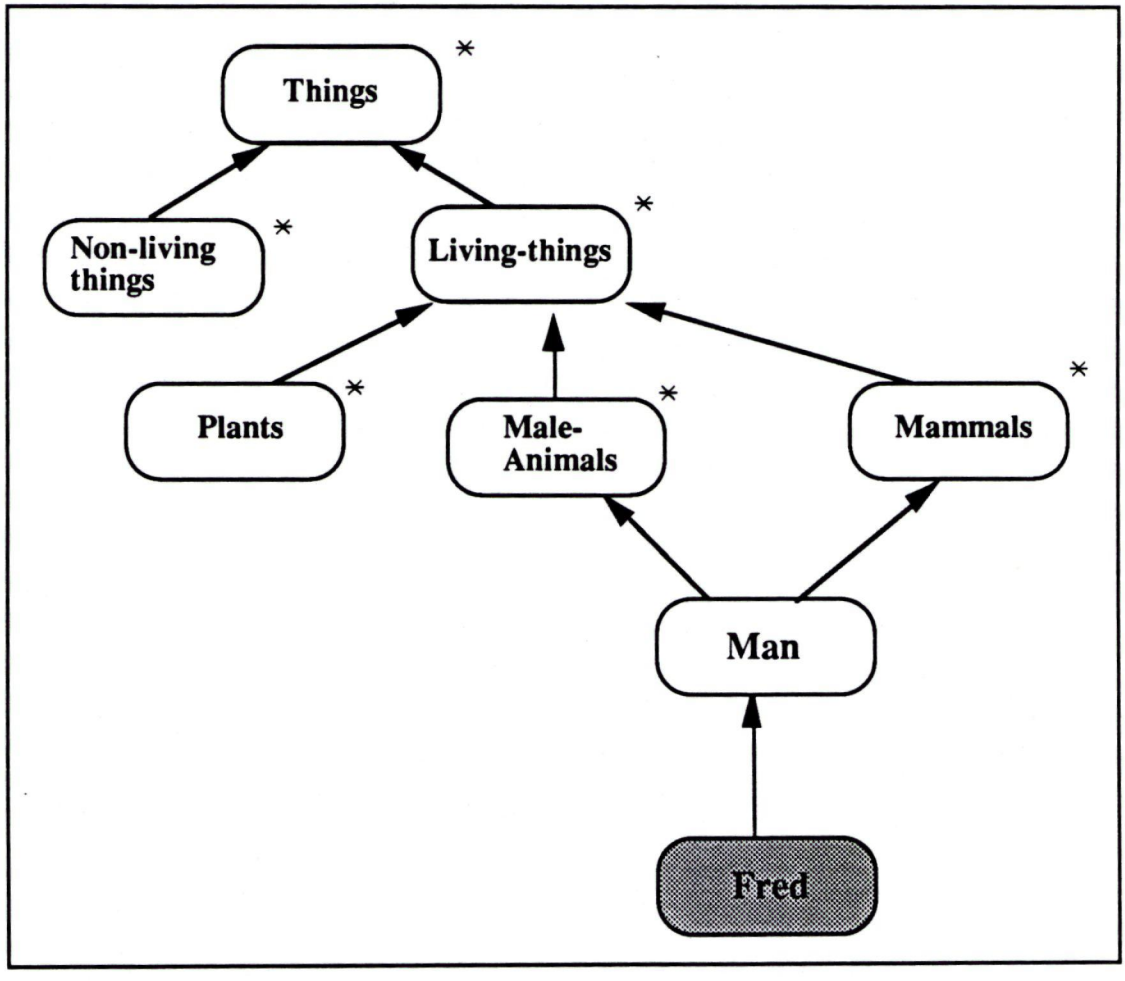

Fig 2.1 A Simple Concept Taxonomy

Classifier : The KL-ONE system includes a component called *classifier*, which automatically accepts a new concept and inserts it in an appropriate place in a KL-ONE taxonomy. This is done by placing the new concept below those concepts that subsume it and above those concepts that it subsumes. In this way, it simplifies the task of the user in creating static knowledge bases and it can dynamically create concepts (descriptions) during the execution of some task. The classifier in KL-ONE does not permit any incremental changes of concept descriptions, and there is only an operation to support the revision of a new, previously undefined and unused atomic concepts.

Weaknesses of the KL-ONE System : Much of the work done on KL-ONE focuses on description formation and little effort is spent on developing assertional representation capabilities. Also, major research in the KL-ONE system focused on the reasoning aspect only and the issue of revision has not been addressed. Some of the weaknesses of KL-ONE are overcome by two of its successors : KL-TWO (Vilian 1985) and KRYPTON (Brachman & Levesque 1983). The problem of revision is partially taken up in the KL-TWO system.

KRYPTON : The assertional sub-system used in KRYPTON is a fully-fledged theorem prover for first order logic. KRYPTON combines knowledge representation and theorem proving techniques. However, one weakness is that the terminological formalism used in KRYPTON is not very elaborate as some terminological constructs do not fit well into the first-order logic framework.

KL-TWO : Most reasoning systems based on KL-ONE do not support knowledge revision because it is computationally expensive to perform. One such research that attempts to incorporate knowledge base revision capabilities in the KL-ONE family of hybrid representation systems is the development of KL-TWO which used NIKL (Moser 1983) as its terminological component and a reasoning maintenance system, RUP (McAllester 1982) as its assertional component. This special architecture only facilitates assertional knowledge revision but does not support the revision of terminological knowledge.

2.2 Knowledge Base Revision

A knowledge engineer may enter incorrect terminologies into a knowledge base due to his/her own mistakes or wrong interpretation about the knowledge that is to be represented. This gives rise to the problem of knowledge base revision.

Knowledge Base Editing : A primary idea in realizing knowledge revision is to have a system that includes operations to manipulate the network that is used to implement the concepts in the knowledge base and their interrelationships. It provides two general functions : knowledge presentation, and modification of terminology. One example of a knowledge base editor for a terminological representation system is KREME.

2.2.1 An Overview of KREME

KREME (Abrett & Burstein 1987) provides functionalities to organize multiple representation formalisms and multiple knowledge editors, all within a coherent global environment. The key objective of KREME is to develop an environment to support the integration and organization of diverse types of knowledge in a coherent global representation system. One can think of the knowledge editor in KREME as an extensible set of globally coherent operations which can be applied just by applying several knowledge representation editors each tailored to a specific kind of knowledge. KREME's approach was to facilitate the extension of several existing representations languages by providing an open ended architecture to integrate these different formalisms. It also facilitates the addition of a new representation formalism. KREME has knowledge editors that support four different knowledge representation formalisms : frames, rules, procedures, and attached behaviours or methods, defined as functions.

These editors facilitate knowledge presentation by allowing the user to browse and navigate through the knowledge base. They also provide an extensive set of operations for manipulating and modifying the terminologies in the knowledge base. Basic modification functions include : an operation to add new concepts and roles, and a function to modify existing definitions of concepts and roles, a delete operation to purge concepts and roles, and an operation to rename concepts and roles.

In summary, KREME is a versatile and powerful tool for editing terminological knowledge bases. However, it does not support the addition or revision of assertions nor does it detect contradicting facts and rules.

2.3 Apprentice Learning and Cooperative Balanced Modelling

This section presents an overview of learning apprentice systems : DISCIPLE and LEAP and a cooperative balanced modelling system, BLIP. All these systems employ machine learning techniques to partially automate the process of knowledge acquisition.

Objectives of a Learning Apprentice System : Expert systems find their applications in many fields. However, one of their limitations is the inability to autonomously acquire and update knowledge in their knowledge bases to facilitate dynamic changes in the real world. This limitation is addressed by machine learning techniques. Learning apprentice systems such as LEAP and DISCIPLE are examples of expert systems that are capable of automatic knowledge acquisition and learning.

2.3.1 An Overview of LEAP

LEAP (Mitchell 1985, 1990) is an augmentation of VEXED, a knowledge based design consultant system that is applied in the domain of VLSI circuit design, to which has been added a learning sub-system. By itself, VEXED supports the user interactively by providing suggestions to circuit design and suggesting possible refinement, its knowledge base essentially consisting of implementation rules for the circuit design.

Problem Solving Mode : During consultation, LEAP maintains an agenda of sub-tasks and repeatedly selects a sub-task for examination, and suggests possible implementations for the corresponding circuit module. The user may either accept or reject the suggestion. If the user disregards the suggestion he/she may choose to manually refine a circuit module using the editor provided by VEXED. Whenever the teacher provides a solution to the system, the learning sub-systems will try to generalize the solution so that it can propose a similar solution in the future when a similar problem is encountered.

Learning Mode : During learning, the implementation steps contributed by the user are used as a training example for the learning sub-system. LEAP will generalize the

implementation steps and formulate a new rule to summarize a previously uncataloged implementation method.

Each training example consists of :

- a specification of the function to be implemented
- a description of the known characteristics of the input signals
- a circuit entered by the user to implement the given function for the given input signals

Learning Strategy : The learning sub-system of LEAP adopts an explanation-based generalization approach for inducing rules. It relies on complete and strong domain theories. A justifiable generalization is produced from a single training example. Incorrect training examples are rejected by the system. All knowledge that can be learned by the system is implicitly contained in the domain theory. The completeness of the domain theory also determines the relationships which can be compiled and identified by LEAP.

Details of LEAP's learning steps : Generalizations in LEAP consist of generalizing both the left and right hand sides of the example rule. The left hand side of the example rule consists of the specification of the function to be implemented and the characteristics of its input signals. The right hand side of the example rule is the user's solution (a circuit to implement the given function for the given input).

Generalizing the right hand side of an example rule : The steps for generalizing the right hand side of an example rule are as follows:

Step 1 : Compose the specification from the right hand side of the example rule

In this step, LEAP derives a description of the circuit's function from its structure by composing the functions of the sub-modules constituting the circuit.

Step 2 : Verify the Circuit Function

This step shows that the composed specification and the original specification (in the left hand side of the example rule) are equivalent by building a verification tree. This is done

by trying to determine a sequence of algebraic transformations from the composed specification to the original specification using a transformation operator. Each transformation operator has pre and post conditions much like a planning operator.

Step 3 : Generalize the composed specification

This step determines the the general class of expressions for which the verification sequence in the verification tree of step 2 can be applied. This is accomplished by determining the necessary preconditions of the transformation operator sequence. By back-propagating the precondition of each transform operator in the verification sequence of step 2, the necessary condition of the starting expression can be determined.

Step 4 : Generalize the original specification

This step reapplies similar verification steps as in step 2. In this case LEAP tries to determine a sequence of algebraic transformations from the composed generalized specification to the original generalized specification.

Step 5 : Form the new implementation rule

This final step simply combines the results from the previous steps.

Generalizing the left hand side of an example rule : By applying its knowledge about digital circuits, LEAP will analyze the single training example and compute a generalized precondition of the example rule.

Step 1 : Generalization of the precondition is carried out by propagating constraints derived from the operating conditions of each primitive component circuit along with the constraints on the global circuit output back to the global circuit input.

Step 2 : The resulting constraints on the global output is substituted with an appropriate variable name to form the precondition of a generalized rule.

Results : LEAP was successfully applied in the domain of manufacturing VLSI circuit designs. Many copies of LEAP were distributed to manufacturers all over the world. The strengths of LEAP are :

- it is able to reject noisy training examples
- it can produce a justifiable generalization from a single training example

However, LEAP presupposes a strong domain theory. Recognizing this problem, research has begun to combine explanation-based learning and similarity-based learning. The purpose is to develop learning methods that will utilize both a domain theory and multiple training examples to generalize concept definition and to justify the generalization. This kind of learning approach is essential in domains where only imperfect theories are available.

2.3.2 An Overview of DISCIPLE

DISCIPLE was developed in the domain of technology design for manufacturing loudspeakers.

Problem-solving Method : DISCIPLE adopts a *problem-reduction* approach to problem solving. It solves a problem by reducing the problem to simpler subproblems through successive decompositions and specializations. This reduction process continues until the given problem is reduced to a set of elementary problems, that is problems with known solutions. Whenever DISCIPLE fails to satisfactorily decompose or specialize a problem it will prompt the user for a solution. The solution that the teacher provides will be used as input to the learning sub-system.

Input : The input to the learning sub-system is a training example provided by user consisting of problem solving steps and a domain theory represented by a semantic network. An example of the problem solving steps provided by the user is:

In order to solve the problem.
CLEAN entrefer
  Solve the more specialized problem
    CLEAN entrefer with air-sucker

Output : The output from the learning sub-system is a generalized rule for problem solving having a set of necessary and sufficient conditions. An example of a generalized problem solving rule is :

IF

necessary condition : (x HAS z) & (y REMOVES z)

sufficient condition : (x HAS z) & (y ABSORBS z) &
(x PART_OF loudspeaker) & (y ISA cleaner) & (z ISA wastage)

THEN: achieve CLEAN z FROM x
by achieving CLEAN z FROM x WITH y

Learning Approach : DISCIPLE utilizes an integrated learning approach in that explanation-based learning and learning by analogy and similarity -based learning were used. As such an incomplete and/or weak domain theory is allowed. DISCIPLE uses an interactive approach for learning weak domain theories. It makes use of explanations drawn from an example to reduce the version space of the rules to be learned and also to generate new instances analogous to the training example. It asks the user intelligent questions in order to extract the missing knowledge in an incomplete domain theory. Learning in DISCIPLE is carried out in three stages.

Explanation-based learning mode : During this mode, explanation-based learning is applied. DISCIPLE will try to explain the solution given by the user in terms of the relations of objects present in the solutions. This is done by questioning the user in order to distinguish between relevant and irrelevant links about the concepts that are present in the solution given in the training example. The relevant links are considered to be positive explanations. All the concepts that appear in the positive explanations are generalized. Each generalized concept is replaced with a variable. This 'turning constants to variable' approach of generalization (Kodratoff 1987c) is also used in KLOKE which will be describe in chapter 3.

Learning by analogy mode : In this mode, DISCIPLE will try to generate instances (the generated instances could be positive or negative instances) of rules that are to be learned by specializing all the variables in the rule that were generalized during the first mode of learning. The heuristic that DISCIPLE used to find similar explanations is by searching for a mapping between the generalized explanation in the rule and some network relations in the domain theory. DISCIPLE will ask the user to classify all similar instances that are generated into negative or positive examples.

Similarity-based learning mode : The final mode of learning is to apply empirical learning techniques to characterize properties of the set of positive training instances, that is to find a generalization of the set of positive examples that exclude the negative examples. Each time DISCIPLE will try to find a general description of the current state of the rule to be learned and the instance under examination. When the system discovers a positive instance it will generalize the sufficient condition of the current state of the rule so that it includes the positive instance. When it discovers a negative instance it will 'particularize', that is an additional condition is added to both the sufficient and the necessary condition. The application of similarity-based learning in DISCIPLE is seen as a form of focusing the generalization.

Results : In contrast to LEAP, DISCIPLE does not require a strong domain theory, instead it can rely on incomplete and/or weak domain theories. The tradeoff is that it will rely on the teacher to reject incorrect training examples. Even though a single example rule is needed as input to DISCIPLE's learning module, during the second stage the system will actually generate numerous (both positive and negative) instances of the resulting generalized rule from the first stage. Thus, the generalization at the final stage is produced from several training examples.

Weakness of LEAP and DISCIPLE : LEAP and DISCIPLE adopt a *learning by doing* approach which requires that the initial background knowledge be present in the knowledge base before learning can actually takes place. Although DISCIPLE does not require a strong domain theory, it still depends heavily on the quality of the domain theory. The representation of entities and their properties in the domain theory will affect the explanation that is being drawn from it. Thus, even though the domain theory may be weak or incomplete, it must still contain a well structured hierarchy of relations describing the properties of the objects in the domain. However, neither LEAP nor DISCIPLE provide a mechanism for acquiring this initial knowledge.

Objectives of A Cooperative Balanced Modelling System : BLIP on the other hand is mainly concerned with building an initial domain model and can thus be applied to construct the knowledge needed for a learning apprentice such as DISCIPLE. This will be discussed in the section on future work in chapter 6. The other motivation of BLIP is to facilitate the revision of the domain model. Both the task of constructing and revising

the domain model emphasizes a high degree of interaction between the user and the system.

2.3.3 An Overview of the BLIP System

This section gives an overview of the BLIP (Morik 1988, 1990) system. KLOKE has adopted some of the design features of BLIP and those aspects of BLIP that are similar to KLOKE will not be discussed in this section but will be presented in Chapter 3. Both the functional and structural components of BLIP are described here.

Knowledge Representation : Knowledge in BLIP is categorized into (i) domain knowledge that describes the model that is to be constructed and (ii) meta-knowledge which guides the discovery component in hypothesizing about missing knowledge. Domain knowledge in BLIP consists of facts and rules. Each fact is represented by an n-place function, and an inference rule represents the interrelations and properties of facts. In BLIP concepts are represented by n-place functions whose arguments are constrained by functional concepts called *sorts*. Based on the set of example facts given by the user, BLIP will attempt to hypothesize the sorts of the arguments. Higher-order concepts are also adopted in BLIP (Wrobel 1987b). While first-order concepts express properties of base-level objects in the real world, higher-order concepts represent properties of properties.

Functional Components : BLIP has four main components ; the co-ordinator, the modeller, the inference engine and a user interface. The co-ordinator maintains the integrity of diverse knowledge sources, while the modeller automatically evolves domain knowledge and also acquires meta-knowledge for guiding the learning mechanism. The inference engine, IM-2, is responsible for representing, maintaining and making inferences about facts and rules at both the domain, and meta and meta-meta levels. The user interface provides immediate response by displaying to the user the consequences of an action. Figure 2.2 provides a system overview of BLIP's architecture.

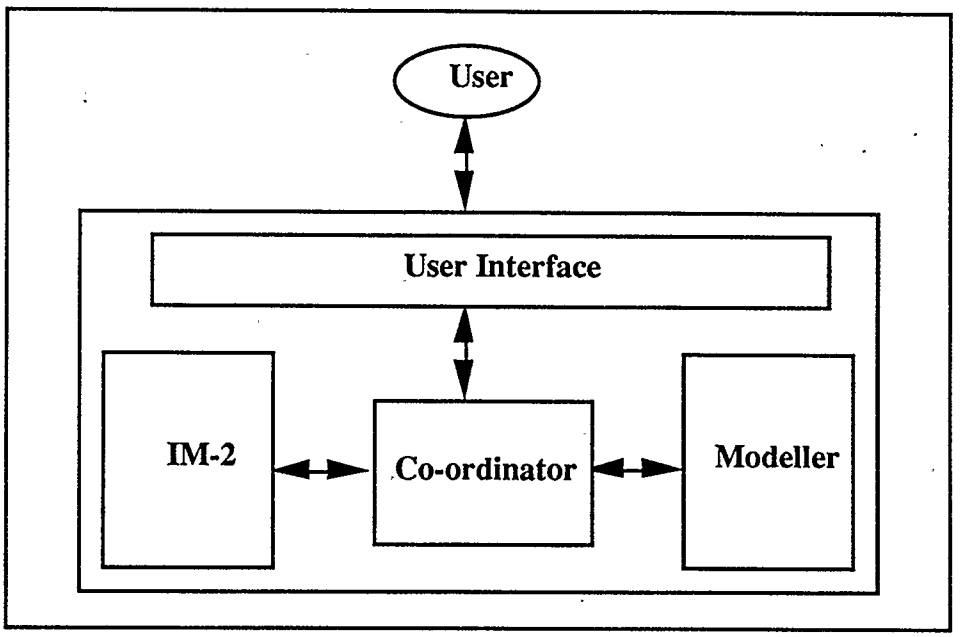

Fig 2.2 Functional Components of BLIP

The Co-ordinator : The co-ordinator maintains integrity among diverse knowledge sources and ensure correctness of sorts and terms in facts and rules. The various knowledge sources are knowledge which supports the acquisition process, definitional knowledge which represents the acquired knowledge, and background knowledge which guides the learning process. The co-ordinator can be viewed as the environment of the knowledge sources which represents the domain model. It also includes a program that acquires and (re)organizes the sorts and syntactic declarations of predicates and maintains the membership of constants to sorts.

The Modeller : The modeller is the learning component in BLIP and its architectural design is shown in the figure 2.3. Learning in BLIP consists of learning domain knowledge and acquiring meta-knowledge. Domain knowledge acquisition in BLIP includes learning domain rules and forming new concepts. Rule learning in BLIP is model directed and this allows the user to indicate the relations that he/she is interested in. Meta-knowledge acquisition consists of generalizing an example rule given by the user to an abstract structure of the type of rules to be learned. Details of the meta-knowledge acquisition process are described in chapters 3 and 4.

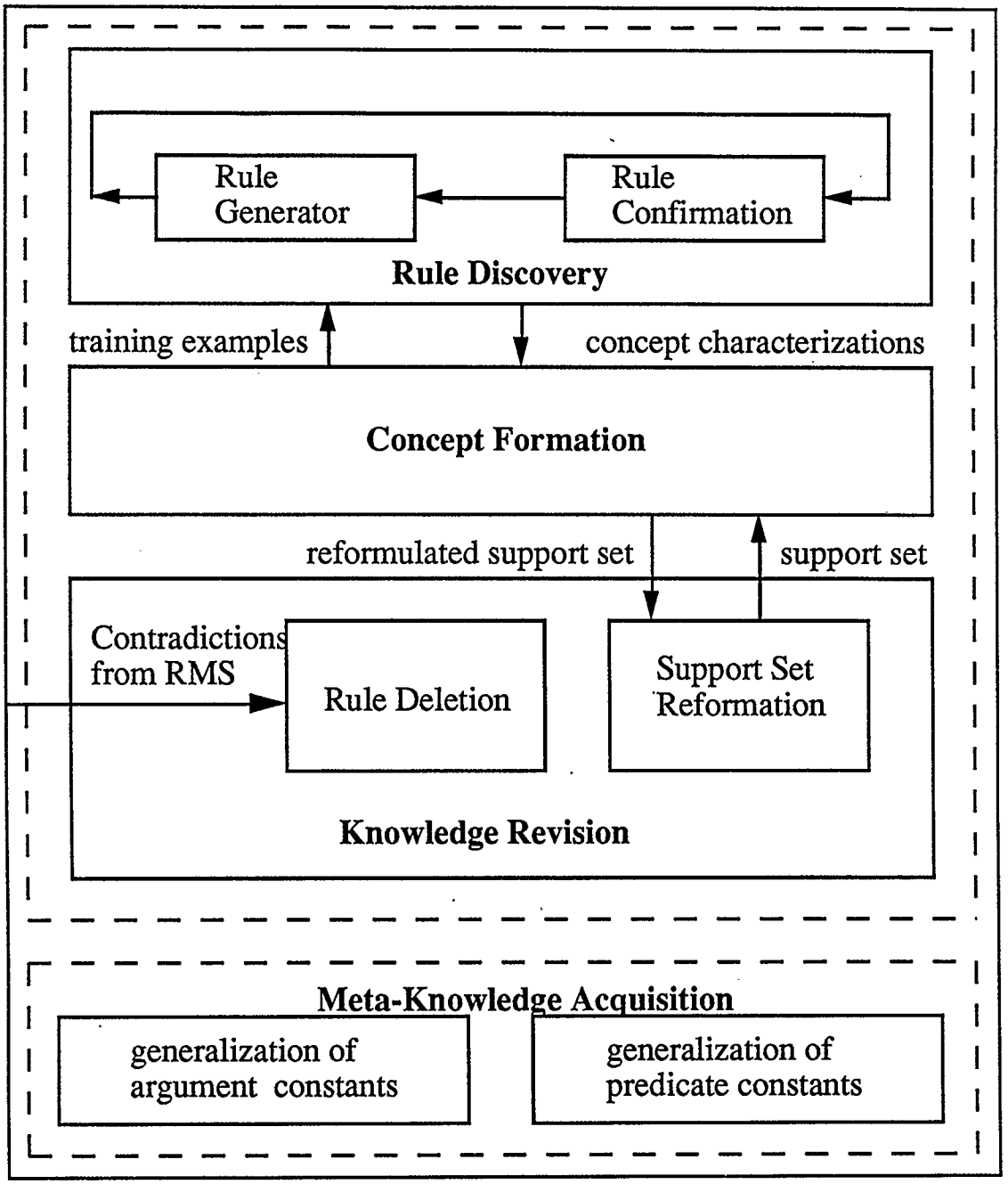

Fig 2.3 The Modeller of BLIP

Rule Discovery : BLIP's rule discovery algorithm is based on the approach used by the series of METAXA learning programs (Emde 1987a). It uses a model-driven, two step empirical learning approach. The first step of this discovery process is to use the system's existing rule scheme to instantiate facts in the knowledge base in order to generate one or more hypotheses about rules that might hold over the given data. In the

second step, those hypotheses generated are empirically tested against the knowledge base and those hypotheses which have been accepted are entered into the knowledge base. Hypotheses are tested with a search pattern called *characteristic situations.* Hypotheses that have been confirmed and validated will be added to the knowledge base and are used to infer new facts which are used with existing rule schemes to generate new hypotheses. This exhibits a closed-loop learning characteristic. This rule discovery approach is adopted by KLOKE and will be describe in detail in chapter 3.

Knowledge Revision : The knowledge revision module is used to revise a set of rules and/or facts when the system detects a contradiction resulting from the discovery of erroneous rules. The factors causing erroneous rules are :

- selection bias - the information entered may not represent the situation about the real world
- users incrementally enter information in a piece-meal fashion
- some incorrect information might have been entered and eventually retracted by the user when detected

There are two possible options to revise the knowledge base when a contradiction arises. The simplest way of handling a contradiction is to retract a set of facts and rules that lead to the contradiction. Another alternative is to prevent a rule from being applied in contradicting cases.

The conservatism heuristic : A strategy called *conservatism heuristic* (Salzberg 1985) was used to prune the area in the knowledge base that will have the least consequences. This parallels the idea with the revision problem addresses by the reason-maintenance technique (Nebel 1990) which resolves contradiction by the minimal mutilation of the set of beliefs. A complete derivation tree can be provided by the inference engine, since it maintains a record of those knowledge base elements that had been used in a derivation. The advantage of using the conservatism heuristic is that the system can preserve the knowledge that is already known instead of having to restructure the whole model.

Rule modifications : The method proposed by Hayes-Roth (1983) which is used for modifying rules in KLOKE is called the *exclusion method.* All contradicting cases are excluded from the rule's domain of applicability. Each rule in KLOKE is associated with a support set which specifies the instantiations of the rule's variables that are permitted.

This has a similar effect to Winston's *censored production rules* (Michalski & Winston 1986).

Support set : The support set of a rule is defined as the cartesian product of the permissible domains for its variables, minus the set of explicitly listed exception tuples.

Suppose X,Y,Z are variables in a rule and the domain of each variable is :

x = {x1, x2, ..,xn}
y = {y1, y2, ...,yn}
z = { z1, z2, ...,zn}

then an exception tuple is defined as T = {xi,yj,zk}, where

xi should be excluded from the domain of x,

yj should be excluded from the domain of y,

zk should be excluded from the domain of z,

and the list of exception tuples are :

{T1, T2,..,Tm}

The support set = x . y. z - {T1, T2,..,Tm}

For example, given the following rule and the set of facts :

bird(x) -> fly(x)
bird(hawk) bird(eagle) bird(dove)
bird(pigeon) bird(ostrich) bird(penguin)
not(fly(ostrich)) not(fly(penguin))

And the following inferences :

fly(eagle) fly(hawk) fly(pigeon)
fly(dove) fly(ostrich) fly(penguin)

The last two inferred facts lead to a contradiction. Thus 'ostrich' and 'penguin' should be excluded from the domain x in order that the above rule can be applied. The list of exceptions = {ostrich, penguin} and the support_set = {eagle, hawk, pigeon, dove}

Plausibility criterion : The justification of applying the exclusion method is not having to discard a well performing rule because of a few exceptions. However, there is a limit to this method, because keeping rules with an implausibly large list of exceptions in their

support set is undesirable as these rules will probably have fewer applications than exceptions. When the list of exceptions becomes implausibly large the support set must be reformulated.

Support set reformation : The idea is to find a concept to separate instances and exceptions. In concept formation, the set of exceptions in a rule are negative training examples and the set of instances are positive training examples. The system will first look for an already existing concept that can separate the instances from the exceptions. If this fails then it proceeds to form a new concept to describe the instances and exclude the exceptions from the new concept.

Concept Formation : Concept membership is represented by facts, where the predicate name is the name of the concept and the objects belonging to the concept class are represented by the arguments of the predicate. Thus, concepts are declared in the form concept1(x). For example, if one wishes to represent the concept 'minor_violation' and a traffic event, event_1 that is a member of this concept then the following is declared :

minor_violation(event_1)

The properties of concepts are represented by rules. There are both sufficient and necessary conditions for a concept membership.

Sufficient conditions : Sufficient conditions for a concept membership hold if the presence of all the premises of a rule deduce the concept membership. For example, the rule :

parking_violation(x) -> concept_1(x)

states that if parking_violation holds for x then x must be a member of concept_1.

Necessary conditions : A rule that has the concept predicate in its premises expresses a necessary condition on the concept membership. For example, the rule :

[concept_1(x), appeals(x,y)] -> court_citation(y)

states that if court_citation do not hold for y then x cannot possibly be a member of concept_1.

The concept formation process : This process attempts to find common characteristics among a set of objects.

Step 1 : Create a predicate, concept1(x) to represent the concept.

Step 2 : For every variable in the concept predicate there is a set of positive examples, and a set of negative examples.
Suppose the set of positive examples = {x1,x2,...,xn}
and the set of negative examples = {y1,y2,...,yn}

The following facts are entered to the knowledge base :
{concept1(x1), concept1(x2),..., concept1(xn),
not(concept1(y1)),not(concept1(y2)),...,not(concept1(yn))}.

Step 3 : Search for sufficient and necessary conditions for concept1 by invoking the rule discovery program, restricting the search space to only rules that include the concept predicate, concept1.

Learning and knowledge representation : The requirements that are imposed on knowledge representation formalism and knowledge base maintenance depend on the extent of the learning task of the system. Rote learning do not demand much of the knowledge representation formalism because it does not involve sophisticated problem solving activities. Many requirements and constraints are imposed on the knowledge representation formalism in an incremental learning system that performs complex tasks such as knowledge revision and maintenance, concept formation, generalization of description and applying induced knowledge. One requirement of incremental learning is that the original knowledge and the induced knowledge must be passed on to the next stage. In a closed-loop learning system such as the rule discovery module, the output of one learning stage is "looped back" as input to the next. This will impose some requirements for the knowledge representation formalism used in the learning program.

The Inference Engine, IM-2 : IM-2 (Emde 1987b) is the knowledge representation and maintenance component of BLIP. The knowledge representation system in KLOKE is not based on IM-2. IM-2 performs a series of tasks including :

(i) storing assertional knowledge (facts) and inferential knowledge (rules) at domain, meta and meta-meta levels.
(ii) making inferences about facts by applying forward and backward chaining on inference rules (domain rules).

(iii) maintaining assertional and inferential knowledge by recording dependency networks between pieces of knowledge or carrying out necessary reason maintenance operations.

Support sets are translated into additional premises. To some extent the inference engine performs forward-chaining from an input fact or meta-fact. Thus, contradicting facts and meta-facts can be detected. To sum up, the inference engine is charged with the responsibility of book keeping, inference and conflict recognition.

Knowledge representation in IM-2 : The knowledge representation used in IM-2, the inference engine of BLIP is briefly summarize here. A complete treatment of this aspect of IM-2 can be found in (Emde 1987b).

Rules that generate rules : "Rule generating rules" are inference rules with a rule schema as conclusion. Some examples of "rule generating rules" are :

p :: transitive(p) -> (x,y,z :: p(x,y) & p(y,z) -> p(x,z))
p,q :: inverse_2(p,q) -> (x,y :: p(x,y) -> q(y,x))

Support sets : A support set restricts the domains of variables that appears in a rule. In BLIP, the rule's domain of applicability is narrowed to exclude the exception. Support sets are used to specify the domain to which the rule is applicable.

Negation, uncertainty, and contradictions : A two-dimensional evidence rating is attached to formulae for representing negated statements, uncertain knowledge, and contradictory information. An evidence point, EP is determined by the positive and negative evidence entered and inferred for a position.

Worlds : Worlds are used to describe the organization of knowledge in BLIP. World-attribution is another attribution introduced in BLIP for organizing knowledge. It is used to gather assertions and inference rules which "belong together" in a larger entity called a 'world'.

Multiple theories : "Worlds" enable a learning system to represent competing models about a domain.

Competing theories : Each theory contains a rating with respect to a particular application. A competing theory may suggest to the system what area of a theory can be improved, it may also give pointers to weak spots and invalid assumptions in a prevailing theory.

The User Interface : The user interface immediately displays the consequences following the action by the user or the modeller. BLIP currently has three physical windows, each can be configured to display any one of the logical windows corresponding to the different knowledge sources. The interface facilitates addition, duplication, and modifications of an existing entry. Any changes made will be processed by both the modeller and the inference engine.

Results : BLIP is the first cooperative modelling system that has been developed and was applied to the domain about the side effects of painkillers. Continuation of this research has been taken up by a successor to BLIP called MOBAL that is due for completion at the end of 1991.

Apprentice Learning vs Balanced Cooperative Modelling : In learning apprentice systems such as LEAP and DISCIPLE there is no explicit learning mode. Learning only occurs when the system encounters a problem that it cannot solve correctly or for which it is unable to provide any solution. Both LEAP and DISCIPLE require an initial domain theory to start with. This issue will be disussed in chapter 6. BLIP and KLOKE on the other hand focus on the construction of an initial domain theory and learning takes place while the user and system cooperatively built the domain model.

2.4 Summary

A KL-ONE based system, consisting of a terminological sub-system and an assertional sub-system implements the idea of complex structure descriptions and their application in reasoning. Some weaknesses of KL-ONE are overcome in KRYTON and KL-TWO: KRYTON provides a facility to represent assertional knowledge while KL-TWO supports assertional knowledge revision but does not support terminological revision. KREME provides the facility to organize and manipulate knowledge structures through knowledge editor interfaces. LEAP and DISCIPLE explicitly support the incremental acquisition of new knowledge during the process of problem solving. BLIP facilitates the construction and revision of a domain model through a cooperative modelling process by assisting the

user to discover missing knowledge and detect inconsistencies. The relations between KLOKÈ and each of the system described in this chapter are summarized in table 2.1.

Systems : / Features :	KL-ONE	KREME	LEAP	DISCIPLE	BLIP	KLOKE
declaring terminologies	yes	yes	-	-	yes	yes
revising terminologies	-	yes	-	-	yes	yes
adding rules & facts	-	-	yes	yes	yes	yes
changing rules & facts	-	-	-	-	yes	yes
detect inconsistencies	-	-	-	-	yes	yes
presentation	-	yes	yes	-	yes	to be implemented
discovery	-	-	yes	yes	yes	yes

Table 2.1 Typical features of knowledge representation and learning systems.

Chapter 3
System Design of KLOKE

This chapter presents an overview of KLOKE, its system design and its system architecture. This project focuses on the dynamic perspective of knowledge representation, namely revision and refinement of knowledge bases. KLOKE is a intelligent knowledge editor that supports knowledge base maintenance in the context of a KL-ONE based knowledge representation system. It has been design to :

1. provide an environment for manual knowledge acquisition
2. build a domain model based on the knowledge given by the user
3. detect inconsistencies
4. support terminological knowledge revision
5. support assertional knowledge revision
6. hypothesize about sorts
7. hypothesize about terminological relations
8. hypothesize about properties of facts (inference rules)
9. infer facts from the existing set of facts and rules

Approach : The system adopts a sloppy modelling paradigm, which deals with imperfect domain knowledge. As such the user is not required to supply the system with a complete and correct domain model, instead a 'sloppy' model will suffice. The system will detect and correct any inconsistencies and attempt to discover any knowledge that is missing by hypothesizing about relations and properties of facts. The systems assists the user by performing bookkeeping tasks, correcting any mistakes and discovering new knowledge. Both the system and the user are working toward a common goal that is to construct a domain model. Thus the term cooperative balanced modelling is used to describe this kind of system.

Structural components : Two kinds of knowledge in KLOKE are : domain knowledge and meta-knowledge. Domain knowledge consists of facts and rules used to describe the domain model. Rules are used to describe properties of facts and the relations between facts. Meta-knowledge is used to guide the discovery component in hypothesizing about

the properties and relations of facts. The knowledge representation formalism used by the system is in essence of first order (higher order in the case of meta-knowledge) predicate calculus and inference rules.

Functional components : KLOKE has three major functional units : a knowledge base editing facility, a knowledge representation sub-system (consisting of a reasoning maintenance system and a sort classifier), and a discovery module.

Knowledge base editing environment : The knowledge acquisition component consist of a user interface to facilitate browsing and modifying the knowledge structures in the knowledge base.

Knowledge representation sub-system : This unit includes a sort classifier, a sort-checker and a reasoning maintenance system

Since the system and the user are both evolving the domain model, a reasoning maintenance system is needed to recognize any inconsistencies. The reasoning maintenance system performs the following tasks :

- inference : it has a forward chaining inference engine to infer facts from the already existing facts in the knowledge base

- bookkeeping : the reasoning maintenance system will keep a record of all facts and rules used in deriving the facts in the knowledge base. It performs this task by maintaining a data dependency network (DDN)

- conflict recognition : it will also check for any inconsistencies that may arise

The classifier performs automatic acquisition of a sort taxonomy. In order to construct a domain model, one has to provide a specification of a description language. In the case of BLIP/KLOKE, domain specific predicates must be declared along with their admissible arguments or sorts and the relationships between these sorts. During the initial phase of knowledge acquisition, the system will automatically build a taxonomy of sorts based on the set of facts given by the user. This automatic acquisition of sort taxonomy is incremental and reversible. During subsequent stages of knowledge

revision, when the user enters new facts, the system will automatically compute the necessary changes.

The sort checker filters out nonsensical facts. For example if the user makes a mistake in representing the facts, the system may detect it and report to the user. In checking the sort correctness of rules, the system restricts the hypotheses space during the discovery process. This is because only syntactically correct rules will be accepted by the system.

The discovery component : The discovery module has two sub-components : a module that discovers domain knowledge and another module to acquire meta-knowledge to guide the discovery of domain knowledge.

Rule discovery : Rule discovery is carried out in a two step process : generation of hypotheses and confirmation of hypotheses. Hypotheses are generated using rule schema (templates) and existing facts in the knowledge base. Even though checking the correctness of rules considerably restricts the hypotheses space, to search the entire hypotheses space would be combinatorially exponential. The heuristic that the system uses, is to prefer generating rules that contain facts that are used recently and frequently by the user. To include breadth of knowledge, the heuristic will also be biased towards facts that appear in very few rules.

Meta-knowledge acquisition : A two step generalization process is used to acquire meta-knowledge. The user enters an instance of a rule that contains no variables. During the first generalization step the argument constants are turned into argument variables. The second generalization step turns the predicate constants into predicate variables.

3.1 The Architectural Design of KLOKE

The functional and structural components of KLOKE are described in this section. The structural components in KLOKE are represented in the formalism shown in figure 3.2 on page 37. This formalism is based on BLIP's knowledge representation formalism. The overall design architecture of KLOKE is shown in figure 3.1.

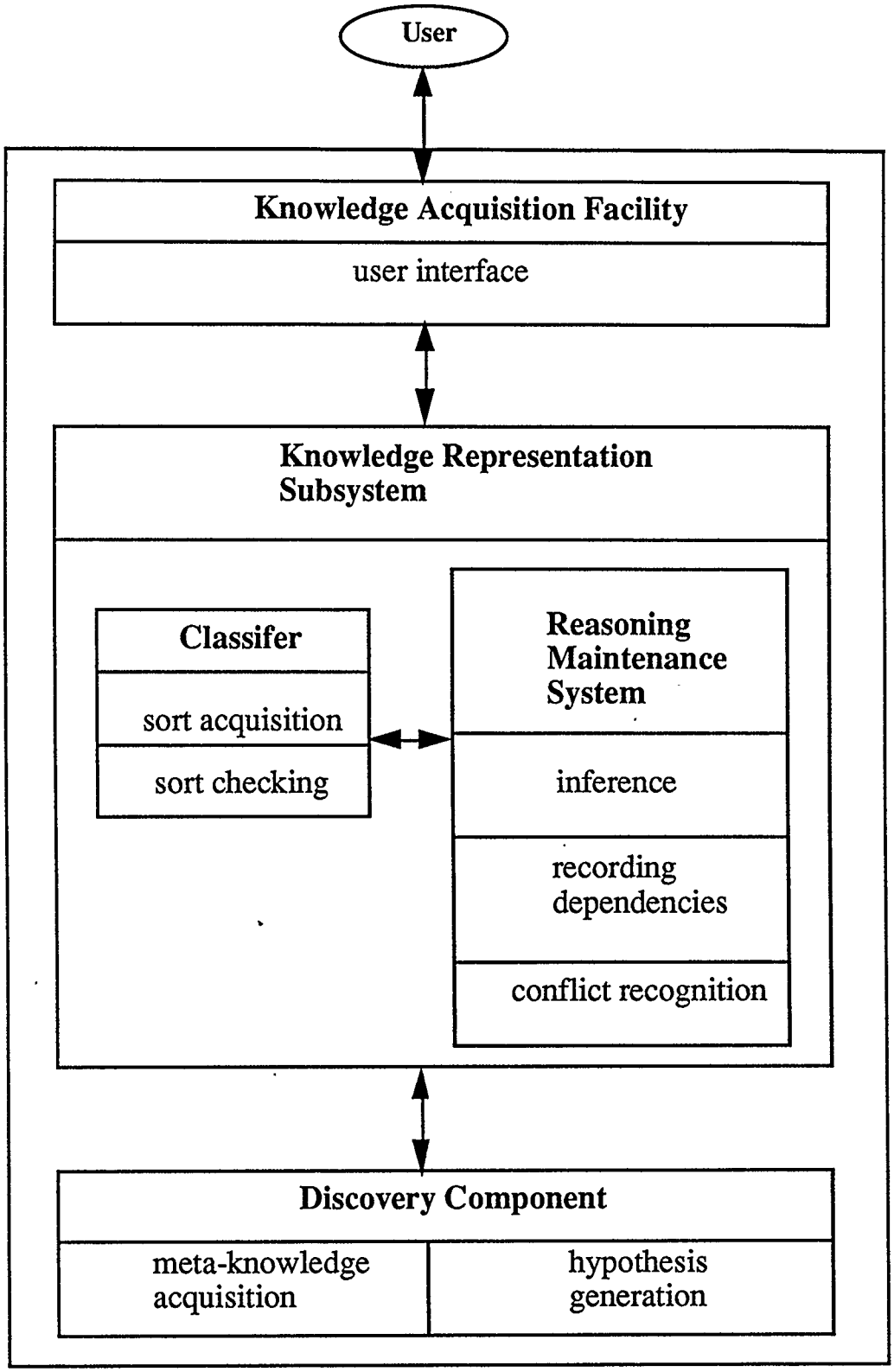

Fig 3.1 The System Architecture of KLOKE

3.2 Abductive Inference of Intension from Extension in KLOKE

A major inference technique used in KLOKE is to hypothesize constraints in concepts from facts entered by the user. This technique of hypothesis formation is an example of what Peirce (Shrager & Langley 1990) called *abduction*. Thagard (1988) defined abduction as a kind of inference having the form

> q is to be explained.
> If p then q
> ---
> Therefore, hypothetically p.

We can define an *intensional* relation between two concepts to be based on their structural definition and an *extensional* relation to be based on the set of entities that fall under the concepts. For example:

Intensional definition : Two concepts are intensionally equivalent iff they both imply the same constraints.
Extensional definition : Two concepts are extensionally equivalent iff they are based on the same extension set.

In hypothesizing about the sort relations in KLOKE, the system will first abduce a full extensional relation from the known extensional relation based on the place value of the facts entered by the user. From the full extensional relation the system can then abduce the intensional relation of sorts.

3.3 Structural Components

One of the most fascinating way to classify knowledge acquisition and machine learning systems is by their knowledge acquisition formalism (Salzberg 1990). Domain knowledge in a knowledge base can be expressed in many ways. Some systems acquire rules that are often expressed in logical form and try to generalize the left hand side of rules to increase the scope of applicability of the rule. Some systems such as LEAP also try to generalize the right hand side of a rule. LEAP learns about VLSI design by making

generalizations on the left and right hand sides of the rules. Another way to represent induced knowledge is by using decision trees such as Quinlan's ID3 and its successors. Some existing knowledge representation formalisms are first order logic, rules, frames, semantic networks, and hybrids.

Knowledge Representation in KLOKE : Domain knowledge in KLOKE is represented in first order logic. The formalism used to represent the domain model describes base level objects in the real world and rules describe properties of these objects. Meta-representation describes the properties of these properties (Wrobel 1987c). For example, if one makes the assertion that "Fred is strong" or strong(Fred), one must also be able to make the statement that "strong is the opposite of weak" or strong(x) -> not(weak(x)). Figure 3.2 shows the knowledge representation of KLOKE.

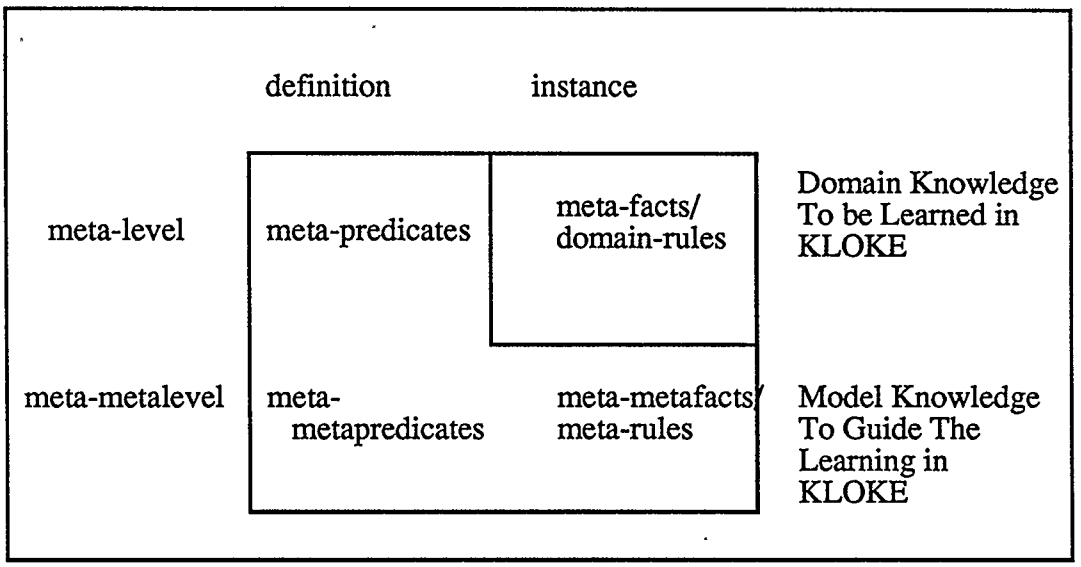

Fig 3.2 Model Knowledge in KLOKE (Taken from Thieme (1987))

The two kinds of knowledge in KLOKE : domain knowledge and model knowledge (meta-knowledge) are shown in figure 3.2. A domain model consists of facts and rules about a particular domain under study. Meta-knowledge consists of rule schemes that guide the system in acquiring domain knowledge. It uses predicates, meta-predicates and meta-metapredicates as its representation formalism. Relations between attributes of predicates are expressed using meta-predicates. Meta-knowledge serves the following purpose :-

- ensures consistency,
- inductively learns rules from facts,
- deduces rules from other rules.

3.3.1 Domain-level Knowledge

Predicates : Predicates are declared to have arguments of certain sorts (Wrobel 1987b). In KLOKE, *Sorted Logic* is used to represent facts. It plays an essential role in excluding semantically nonsensical hypotheses. Each predicate representing a fact is declared to have a definition specifying its *argument sort mask.* Each term in the argument place of a fact must conform to the argument sort that is defined. In other words, the argument sort describes the set of admissible terms in an argument place. The arguments of each fact must be constants, they denote real world objects. For example, the objects in the fact contains(aspirin, asa) are aspirin and asa. Composed expressions such as price(aspirin) are not permitted as arguments. The process for acquiring a sort taxonomy (the relations of sorts) and the declaration of sorts will be describe in chapter 4.

Domain rules : Domain rules in KLOKE are expressed using KLOKE's meta-predicates. They are inference rules describing the relations among the facts in a given domain.

example of a domain rule : [contained(x,y)] -> is_contained_in(y,x)

Meta-facts : Meta-facts are expressed by meta-predicates having predicates as arguments. Meta-facts can be transformed into domain level rules for meta-predicates. They are declarative representations of domain rules. That is why rule discovery in KLOKE is in fact, discovering meta-facts.

example of a meta-fact : inverse(contains, is_contained_in).

3.3.2 Meta-knowledge

Meta-predicates, meta-metapredicates and meta-metafacts build the meta-knowledge of KLOKE. They constitute the domain independent knowledge that KLOKE is equipped

with. Meta-facts are not considered part of the meta-knowledge as they result from the rule discovery algorithm.

Meta-predicates : Meta-predicates are used to express the relations between attributes of predicates. For every meta-predicate there exist a rule-model or rule schema, which is an abstract structure of a rule. The set of available meta-predicates constitute the rule model of KLOKE. It describes how a meta-fact can be matched to its corresponding domain rule.

example : inverse(p,q) where [p(x,y)] -> q(y,x).
opposite(p,q) where [p(x)] -> not(q(x)).

where *x* and *y* are the arguments of a predicates representing a fact, and *p* and *q* are the name of the predicate.

Meta-metafacts : Meta-metafacts are actually declarative instances of meta-metapredicate definitions. They are declarative representations of meta-rules. Just like meta-facts, meta-metafacts can be transformed into meta-rules using the corresponding meta-metapredicates.

example : symmetrical(opposite)

Meta-rules : Meta-rules are inference relations at the meta-level. The corresponding meta-rule of the meta-fact shown above is :

[opposite(p,q)] -> opposite(q,p)

Meta-metapredicates : Meta-metapredicates express the rule schemes of meta-predicates at the meta-metalevel. There is also a rule-model for every meta-predicate, which is an abstract structure of a meta-rule.

example : symmetrical(a) where [a(p,q)] -> a(q,p)

where *p* and *q* are names of predicates which represent facts, and *a* is a meta-fact.

3.4 Knowledge Representation Sub-system of KLOKE

Like most of the hybrid representation systems based on KL-ONE, the knowledge representation system in KLOKE consists of a classifier that represents the terms that are used to form arguments in facts and builds a taxonomy similar to that of the KL-ONE system, to describe the interrelationships between the sorts or types of the arguments. To support belief revisions, it also includes a reasoning maintenance sub-system that performs the task of inferring facts from the existing knowledge base, recognizing any contradiction that may arise during the process of building the domain model and discovering missing knowledge (since both these activities occur simultaneously in KLOKE). Additionally, the reasoning maintenance system will also keep a record of all the derivations that it makes during inference by maintaining a data dependency network. It also has a component to revise any belief in the system if a contradiction is detected. It is clear from figure 3.3 that the overall architectural design of the knowledge representation sub-system of KLOKE closely resembles that of the KL-ONE based systems, where the TBox or terminological representation sub-system corresponds to the classifier and the reasoning maintenance system in KLOKE can be likened to an ABox of the assertional representation sub-system.

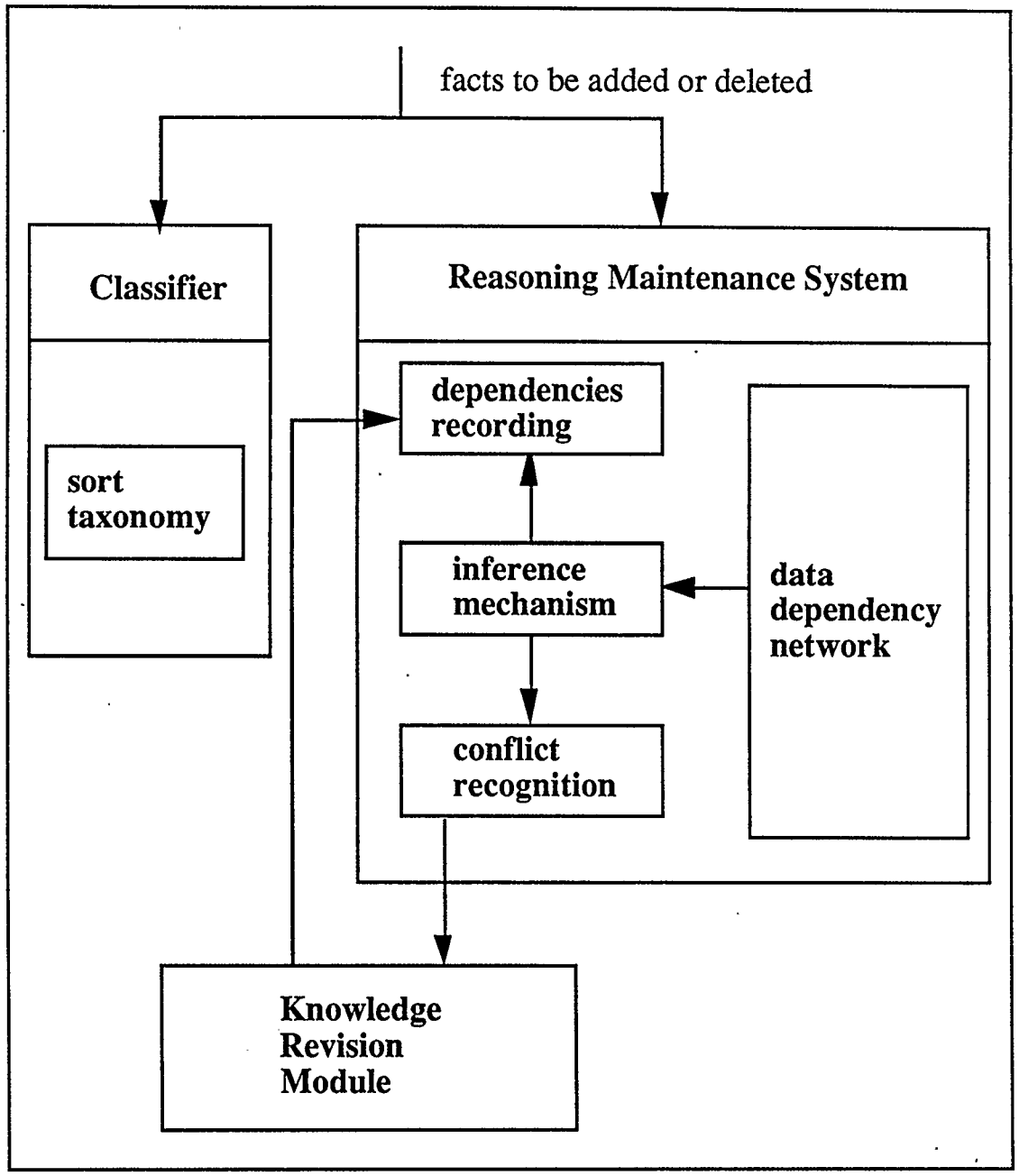

Fig 3.3 Knowledge Representation Sub-system of KLOKE

3.4.1 The Classifier : Acquiring the Sort Taxonomy in KLOKE

Specification of the description language is a primary prequisite for constructing a domain model. In a logic-oriented representation domain-specific predicates must be declared together with their admissible arguments. KLOKE uses a many-sorted logic

representation to model the domain that is to be constructed. With the sloppy modelling paradigm, the user is not expected to define the sort beforehand, it is the system's responsibilities to perform this task. The classifier sub-system will automatically acquire the sort taxonomy that is used to specify the well-sorted expressions of the formalism representing the domain model. The acquisition of the sort taxonomy in KLOKE, based on the approach proposed by Kietz (1988) is incremental and reversible. The order that the facts are entered does not affect the resulting sort taxonomy that is built. That is to say that given a set of facts, if they are entered in two different sequences, the taxonomy that is built each time is equivalent to the other.

Building the Sort Taxonomy : The classifier performs the same functions as that in the KL-ONE system. It maintains terminological knowledge and the interrelationships. A sort taxonomy or lattice that much resembles that of the structural inheritance network of the KL-ONE system is built based on the set of facts given by the user. Term-subsumption is also present here.

Revisions of facts in the knowledge base may requires that the sort taxonomy be revised. The acquisition of the taxonomy is both incremental and reversible. There are four types of essential relations between sorts :

- The equivalence classes of argument sorts
- The compatibility partial-order between classes
- The super-class and sub-class relations between classes that are built
- The intersection and disjointness of classes

Input and Output of the Classifier : The input to the classifier sub-system is a set of facts entered by the user such as:

indicate(sore_throat, flu)	cause(flu, sore_throat)
affect_pos(inspirol, sore_throat)	affect_pos(aspirin, flu)
affect_pos(bc, sore_throat)	affect_pos(asa, flu)
contains(inspirol, bc)	contains(aspirin, asa)
suck(willi, inspirol)	suck(uwe, vivil)

The output produced by the classifier is the extension set of sorts, declaration of sorts and a sort taxonomy (shown in figure 3.4).

Extension set of Sorts :

irritation = {sore_throat, flu}

symptom = {sore_throat}

disease = {flu}

substance = {inspirol, aspirin, bc, asa}

act_agent = {bc, asa}

drug = {inspirol, aspirin}

person = {uwe, willi}

dragee = {vivil, inspirol}

Declaration of sorts:

indicate(<symptom>, <disease>)

cause(<disease>), <symptom>)

affect_pos(<substance >,<irritation >)

contains(<drug >,<act_agent >)

suck(<person>, <dragee>)

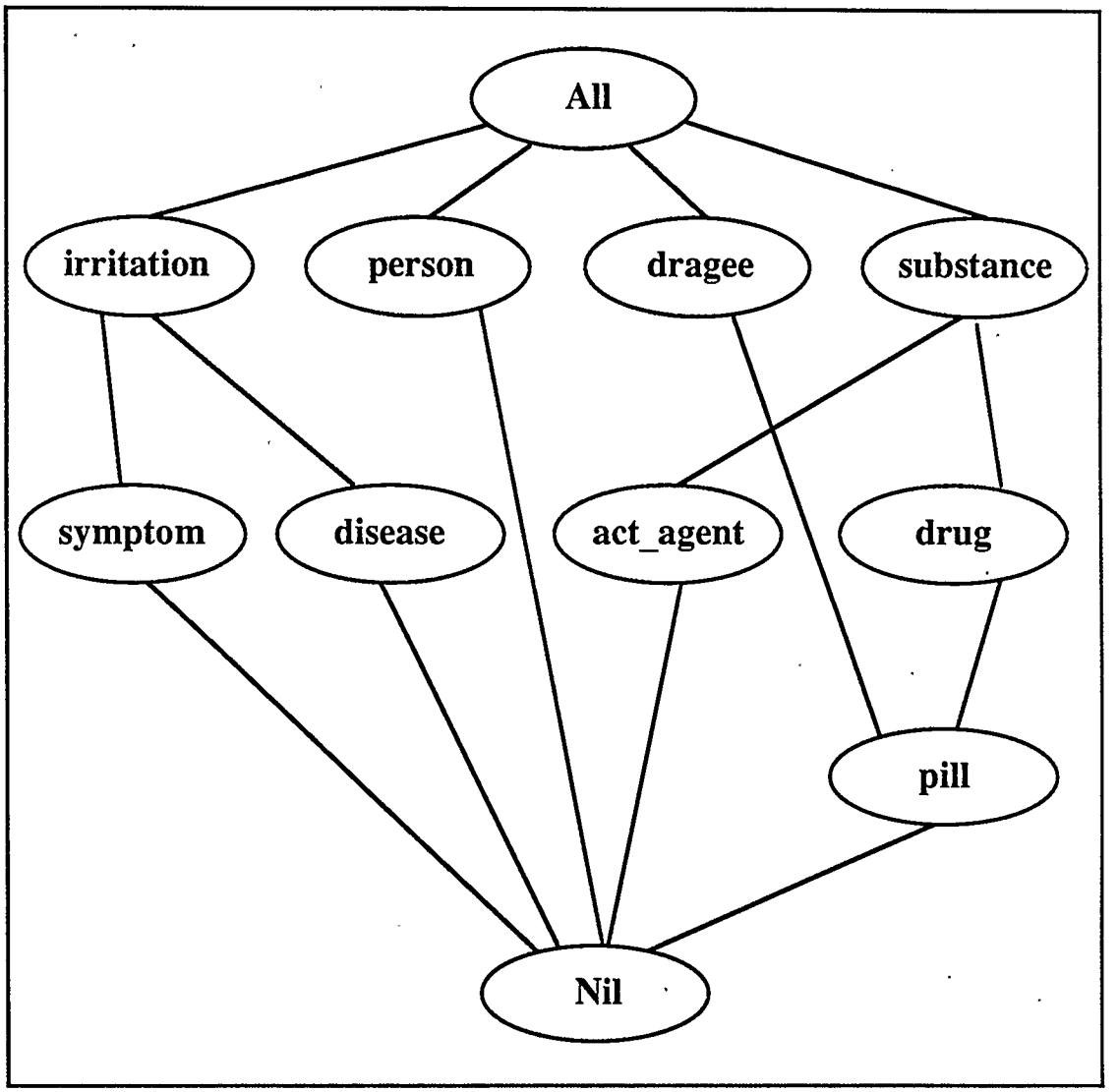

Fig 3.4 An Example Sort Taxonomy

Operations for Evolving the Sort Taxonomy : Two basic operations are required to restructure the sort taxonomy when the sorts and the relations between them change due to the assertions and retractions of facts by the users : an operation to restructure the taxonomy when facts are added and one to restructure the taxonomy when facts are deleted.

The Sort Checker : Included in the classifier is a module called the *sort checker.* In a cooperative balanced modelling system such as KLOKE, the system simplifies the user's task of eliciting knowledge by acting as an assistant to the user. This will include correcting some mistakes made by the user. Any nonsensical facts entered by the user

will be detected by the systems and the system will notify the user. To achieve this, an argument sort mask is defined for each predicate. To illustrate with an example, if the user has defined the following:

Given the following predicate declarations:

indicate(<symptom>, <disease>)
cause(<disease>), <symptom>)

When the following facts are entered :

indicate(disease1, symptom1)
cause(symptom1,disease1)

The system will warn the user that the arguments have been reversed. If the user insists that the inputs are correct, the sort taxonomy will have to be restructured.

3.4.2 The Design of the Reasoning Maintenance System

In a system such as KLOKE, where rules and facts are constantly added and deleted as a process of evolving and refining the domain model, derivation paths of all the inferences must be recorded to ensure that changes in the knowledge base due to deletion or addition of any facts or rules are propagated to the appropriate section of the knowledge base. Given such a requirement, the knowledge representation sub-system should possess capabilities that are similar to those of the assertional sub-component of a KL-ONE based knowledge representation system.

This section will present the design of the reasoning maintenance system of KLOKE, which has three functional components as shown in figure 3.3 on page 41. It consists of a forward reasoning inference mechanism, a sub-module to record the dependencies of rules and facts and a component to detect contradiction between new knowledge and the existing knowledge in the knowledge base. The reasoning maintenance system used in KLOKE is a justification-based reasoning maintenance system which records the dependencies of rules and facts by using a *monotonic data-dependency network* to keep track of all the derivations of all inferences and any beliefs that have been revised.

The Inference Mechanism : The inference mechanism is a forward reasoning engine. Thus, incoming facts can be either inferred from the existing set of domain rules and facts or asserted by the user. Beside facts, domain rules are also added to the knowledge base of KLOKE during the construction of the domain model. Rules that are induced by the rule discovery module are added to the knowledge base if they do not contradict any existing rules in the knowledge base. These induced rules will then be used to infer new facts from the existing facts in the knowledge base.

The Data Dependency Network : The data dependency network is the structural component of the reasoning maintenance system. It is essentially a directed graph G = (V, E), where V represents a set of two disjoint sets : N, which is the set of nodes that denotes believed propositions and justification nodes, J, which denote the set of propositions used in a derivation. E denotes the set of arcs that points from nodes to justification or from justification to nodes. A justification, j, supports a node, n, iff there is an arc from j to n. A node, n participates in a justification, j iff, there is a link from n to j. A premise justification is a justification that has no incoming arc. A premise is a node that is supported by a premise justification. An example of a data dependency network is shown in figure 3.5.

Given the following proposition :

fact0 : Man(Joe)
fact1 : Human(Joe)
fact2 : Person(Joe)
rule0 : Man(x) -> Human(x)
rule1 : Human(x) -> Person(x)

the reasoning maintenance system will build the data dependency network shown in figure 3.5.

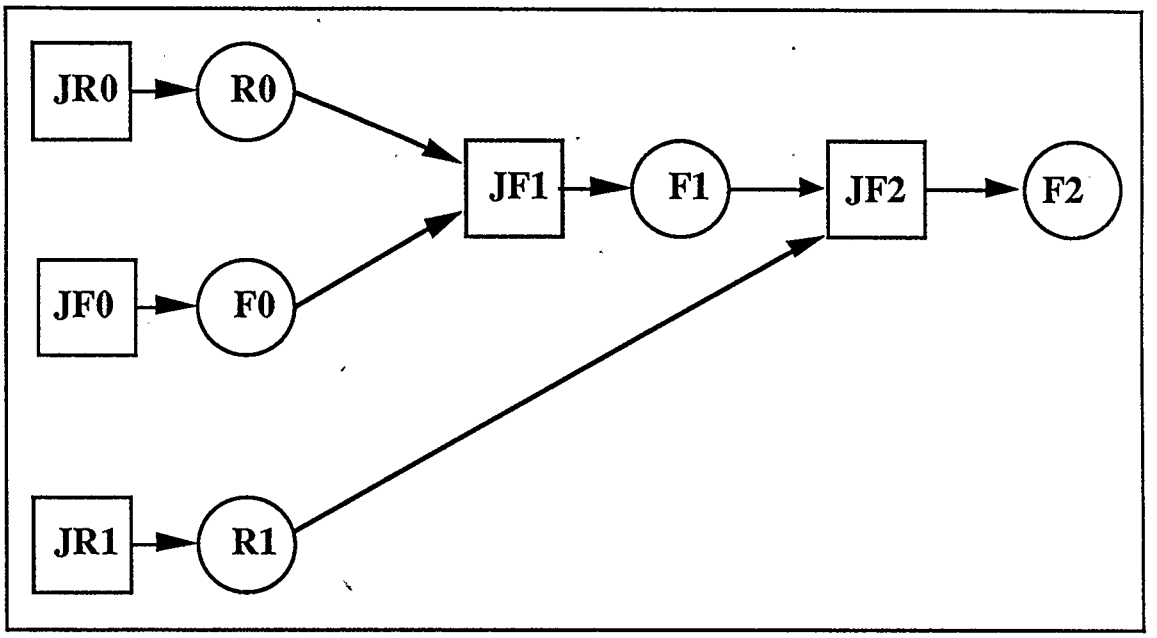

Fig 3.5 A Data Dependency Network

Recording Data Dependencies : Recording data dependencies is simply a kind of bookkeeping task that is termed data dependency network management (Charniak 1980). Two operations are used to evolve the data dependency network (DDN) : add derivation path and prune derivation path. The *build DDN* operation adds new derivations path when facts or induced rules are added to the knowledge base and the *prune DDN* operation deletes derivations paths when facts or rules are deleted. When an induced rule or a fact entered by the user is added to the knowledge base, the inferences performed by the inference mechanism will be recorded and added to the data dependency network. When a fact or rule is retracted from the knowledge base due to a contradiction, all dependencies on the deleted fact or rule are purged. The procedures for implementing the two operations will be describe in detail in chapter 4.

Conflict Recognition : The system needs to be able to recognize conflicting facts when an input or a deduction results in a contradiction. In KLOKE the conflict recognition task is performed by the reasoning maintenance system. Upon detection of a contradiction the knowledge revision module will be invoked to revise the knowledge base by changing or deleting a set of rules and facts.

3.5 The Discovery Component of KLOKE

The learning approach in KLOKE is based on the modeller module and meta-knowledge acquisition module in BLIP. Learning in KLOKE consists of acquiring domain knowledge and meta-knowledge. Domain rules are discovered from the set of already existing facts and meta-predicates that are available in the system. Beside adopting a set of heuristics, and having the sort checker to check the sort correctness of rules to restrict the hypotheses space, the system also uses meta-knowledge to guide the rule discovery module in searching for rules. The system can also evolve the sets of meta(meta)-predicates and meta-facts that are used to guide its learning process (Thieme 1987).

The discovery component of KLOKE is responsible for autonomously building part of the domain model. It has two major sub-programs : a program to evolve the domain model automatically and a program to acquire meta-knowledge. The rule discovery module which hypothesizes about domain rules has only been partially implemented.

3.5.1 Module for Acquiring Meta-knowledge

As described earlier, there are two kinds of knowledge in KLOKE. The knowledge that pertains to a particular domain is called domain knowledge, this is the knowledge that the system is supposed to discover or revise. The acquisition process is bottom up and domain independent and creates a generalization from an instance. Details of the acquisition process are described in chapter 4.

Acquiring Meta-predicates : Domain rules/meta-facts can either be entered by the user or they can be induced by the system using the appropriate facts in the domain. Rules which are entered by the user become background knowledge. Domain rules that the user entered will be automatically transformed into the corresponding meta-facts using the appropriate meta-predicate definition.

Input : In a meta-predicate acquisition process the user must provide an example rule, such as the one shown below :

[male(fred)] -> person(fred)

Output : The following is produced by generalization :

domain rule : male(x) -> person(x)

meta-fact : inclusive(male,person)

meta-predicate definition : inclusive(p,q) where p(x) ->q(x)

Approach : Meta-predicates are acquired by a generalization process. The generalization is based on a single inferential relation between statements given by the user, rather than on a set of statements (which are called facts in KLOKE). The generalization process is carried out iby applying the following generalization rules :

First Generalization Rule : Given a set of functions f(a), f(b), ..., f(n) where a,b,..n are constants, this rule will transform each of these functions into the form f(x) where x is a variable.

Second Generalization Rule : Given a set of function F(x), G(x), ..., Z(x) where F,G,...,Z are constants, this rule will transform each of these functions into the form p(x) where p is a variable.

The first step transforms the user's input into a general rule as a hypothesis and the second step transforms it to a rule scheme in a meta-predicate definition. During this process, the meta-predicate definition corresponding to the example rule given by the user is generated. The corresponding meta-fact and the domain rule are also generated as a side effect of the generalization process.

Redundancy : There is an algorithm in the system that will first attempt to instantiate the rule entered by the user to the known rule schemes offering the user possible choice between the meta-facts resulting from the matching process. This is to prevent redundancy by checking if the entered rule can be expressed by any of the already existing meta-predicates. The acquisition of a new rule scheme will enlarge the model knowledge in KLOKE, enabling the discovery algorithm to search for new instances as hypotheses in the current knowledge base.

Acquiring Meta-metapredicates and Meta-metafacts : The acquisition processes for meta-metapredicates, meta-metafacts and meta-rules are similar to the acquisition of meta-predicates. The generalization rules used are the same, but in this case the constants represent predicates and meta-predicates instead of objects and predicates.

Input : The user's input in this case must be at a higher level. The rule will contain meta-facts as predicates. For example :

[inclusive(bird, animal), inclusive(animal,living_thing)] ->
inclusive(bird, living_thing)

Output : In this case the result of the generalization process will consist of the following :

meta-rule : [inclusive(p,q), inclusive(q,r)] -> inclusive(p,r)

meta-metafact : m_transitive(inclusive)

meta-metapredicate definition : m_transitive(mp) where
[mp(p,q), mp(q,r)] -> mp(p,r)

Applying the Generalization Process at Different Inference Levels : The same generalization process used for acquiring meta-predicates, meta-metapredicates and meta-metafacts can in principle be applied at any inference level. The only requirement is that the input to the generalization process must be at the appropriate level of inference as illustrated in figure 3.6

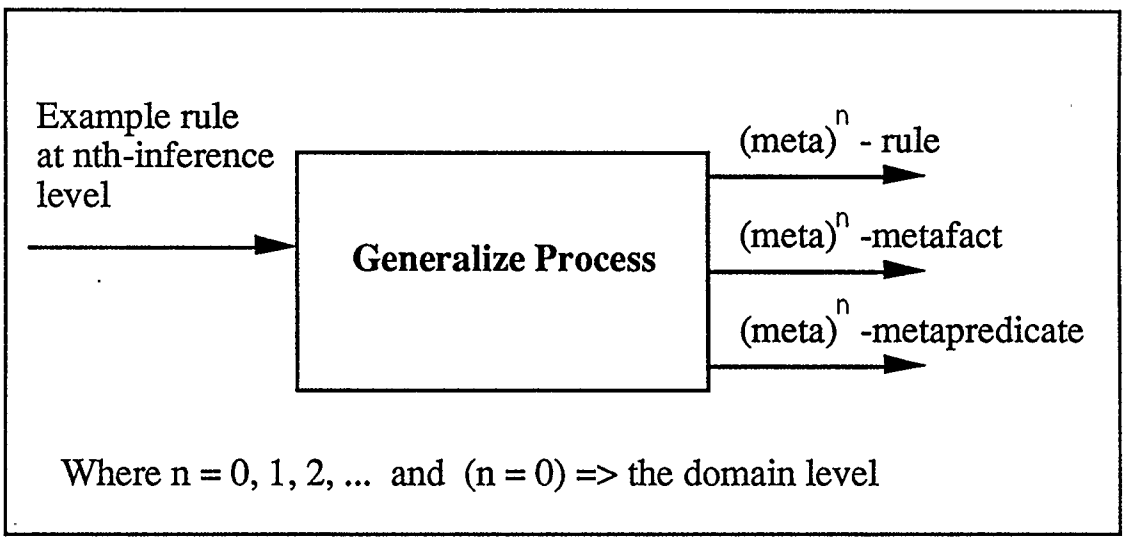

Fig 3.6 Two Step Generalization Process for nth-level Inference Relation

Applying Meta-Knowledge : This meta-knowledge acquisition process will also contribute to the generation of hypotheses and is also performed during the first step of

the rule discovery process. In this case the generation of hypotheses (domain rules/meta-facts) is a side effect of acquiring meta-predicates.

Meta-knowledge is also used to restrict the search space of the rule discovery algorithm. The possible structural dependencies between domain rules (which can be defined by a rule scheme or meta-predicate) are represented in meta-metapredicate definitions. Meta-metafacts and their corresponding meta-metapredicate definitions are classified into a constructive and restrictive category. Restrictive and constructive meta-metafacts guide the search in the learning module by restricting the search space.

Restrictive Meta-metafacts and Meta-metapredicates : Restrictive meta-metafacts are used to detect contradicting rules. The system checks for rules that have the same premises and contradicting conclusions. This function is similar to the conflict recognition task in the reasoning maintenance system except that the reasoning maintenance system in KLOKE will detect contradicting facts. To illustrate with an example, one cannot add the following rule :

(1) bird(tweety) -> fly(tweety)

if the the rule :

(2) bird(tweety) -> not(fly(tweety))

is already present in the knowledge base.

The first rule corresponds to the meta-fact :

inclusive(bird, fly)

with the meta-predicate definition :

inclusive(s,t) where s(x) -> t(x)

The second rule

bird(tweety) -> not(fly(tweety))

corresponds to the meta-fact :

opposite(bird, fly)

with the corresponding meta-predicate definition :

opposite(s, t) where s(x) -> not(t(x))

A meta-metafact :

m_opposite(opposite, inclusive)

with the corresponding meta-metapredicate definition :

m_opposite(ms,mt) where mp(s,t) -> not(mq(s,t))

instantiating ms with opposite and mt with inclusive, we obtain a meta-rule :

opposite(s,t) -> not(inclusive(s,t))

Thus the following meta-level inference relation :

opposite(bird, fly) -> not(inclusive(bird, fly))

can be used to express that both the rules (1) and (2) cannot exist in the knowledge base at the same time. Thus m_opposite is a restrictive meta-metafact.

Constructive Meta-metafacts and Meta-metapredicates : The other category of meta-metafacts is constructive; it is so called because it generates rules from rules. New meta-facts (domain rules) can be inferred from the existing set of meta-facts and meta-rules and thus there is this notion of deductive inference where a domain rule is deduced and added to the knowledge base without testing it. For example if one considers the meta-predicate definition inclusive, if there is a relation between two predicates s and t, and there is a relation between t and another predicate u, then the relation will also hold for s and u. Using the meta-metafact :

m_transitive(inclusive)

and its corresponding meta-metapredicate definition :

m_transitive(mp) where [mp(s,t), mp(t,u)]-> mp(s,u)

we have the meta-rule :

[inclusive(s,t), inclusive(t,u)] -> inclusive(s,u)

Thus if the meta-fact inclusive(bird, animal) and inclusive(animal, living_thing) are present in the knowledge base, then one can infer :

inclusive(bird, living_thing)

and the corresponding domain rule :

bird(x) -> living_thing(x)

can be added to the knowledge base.

3.5.2 The Rule Discovery Algorithm

A comparison of the design of the rule discovery program in KLOKE and the structure of the general rule induction (GRI) program proposed by Simon & Lea (1991) is presented here. The rule discovery program in KLOKE consists of two sub-modules : a hypothesis generator sub-program and a hypothesis testor module. This is a generate and test, model driven learning approach. In addition, heuristics described below are used to focus the search.

During the first stage the hypothesis (rule) generator will generate rules or meta-facts. In the second stage, positive and negative instances of the generated hypothesis are verified against the facts in the knowledge base. The criterion for accepting a hypothesis is governed by the number of positive and negative instances of the hypotheses. In the GRI program the test results of the second stage are made available to the input of the first stage, and the application of these results for the first stage of the discovery process is totally dependent upon the internal design of the rule generator. In the case of the hypothesis generator in KLOKE, induced rules are used to infer new facts from the existing facts in the knowledge base. These new facts are in turn used to instantiate with the existing set of meta-predicates to generate possible domain rules. The process of generating hypotheses is illustrated in figure 3.7 :

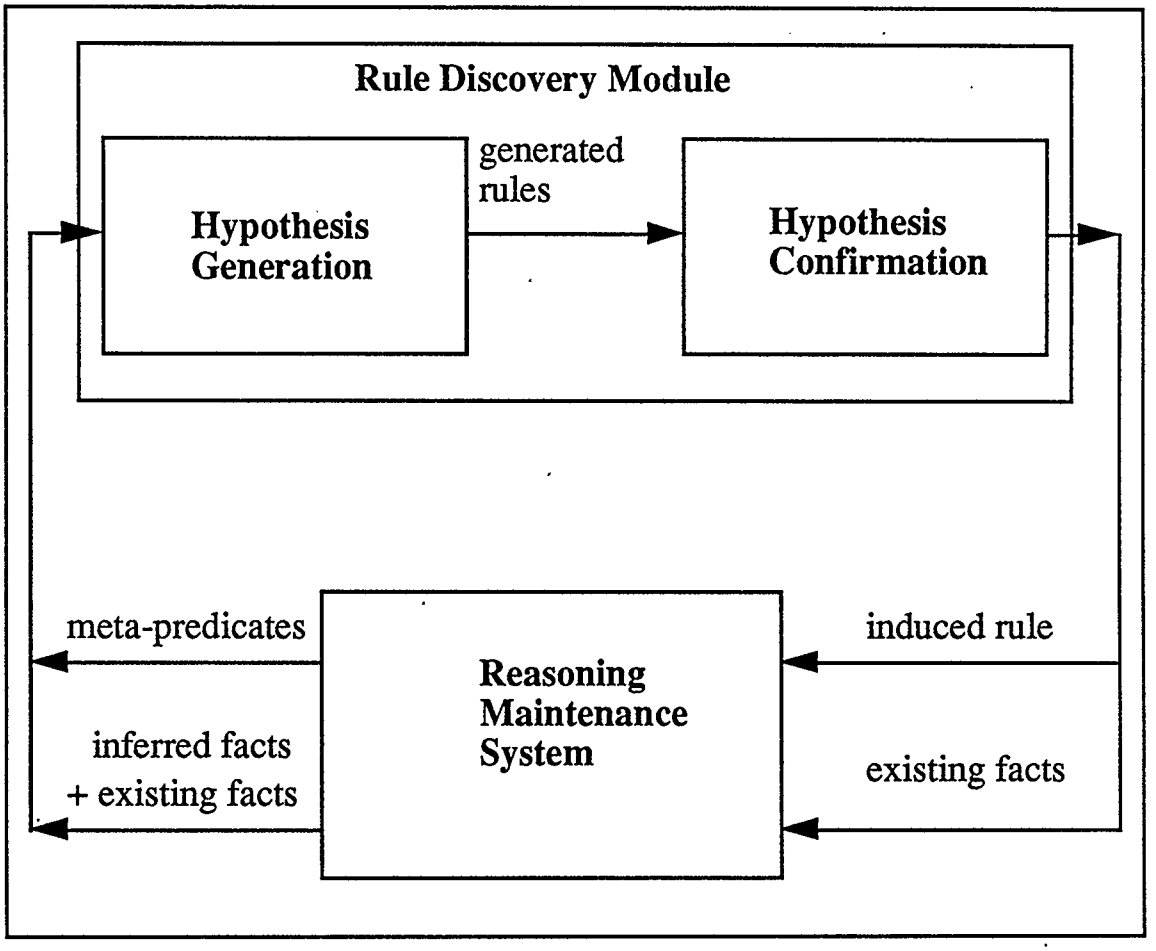

Fig 3.7 Process of Generating Hypotheses

Information Flow of the Rule Discovery Program : It is also interesting to note that a rule induction system may consist of two generators (Simon & Lea 1991). When there are two rule generators, the number of possible channels for the flow of information is much larger. In the hypothesis generator of KLOKE there is a closed-loop information flow. When a generated hypothesis has been verified and confirmed, it is entered into the knowledge base and can be used for inferring additional facts. The reasoning maintenance system uses all the facts and rules (original and induced) in the knowledge base to perform forward chaining inference. Those inferred facts can in turn be used in the next learning stage to confirm some other new hypotheses. In this respect, the rule learning module exhibits the characteristic of a closed-loop learning system.

The hypothesis space : KLOKE's approach to generating rules is model-directed in which instances of rule-models are hypotheses. Rule-models are abstract structures of rules to be learned, they are knowledge that is required for the learning task. As opposed to data-directed learning, model-directed learning permits the user to specify the relation in which he/she is interested.

Exhaustive search : The hypothesis space can be generated by exhaustively instantiating all meta-predicates with all the domain predicates in the knowledge base that are syntactically compatible. A hypothesis is a meta-fact, which corresponds to a domain rule. Sorts are used to exclude semantically incompatible hypotheses. However, the hypothesis space is astronomically large even if sort constraints are imposed when generating all possible points in the search space.

Guiding the search : The sort checker provides a sort correctness check in rules generated in the hypothesis generation module. This will reject syntactically illegal rules and thus reduce the hypothesis space. The example shown in figure 3.8 illustrates the checking of sort correctness in rules. Another way to guide the search is to used meta-knowledge to check for consistency of generated rules with those that already exist in the knowledge base.

Restricting the search space with sorts : Sorts have been previously applied in automatic deduction systems (Schmidt-Schaub 1989). The power of using a many-sorted resolution calculus to solve the Schubert's Steamroller problem was presented by Walther (1985). The difficulty of solving this problem is that its search space is intractable. In a many-sorted universe, the domain and ranges of functions, predicates and variables are

restricted to a certain subset of the universe. Each subset of the universe is formed by restrictions that are governed by inference rules. The restriction of unifications in a many-sorted resolution calculus is that a variable x is unifiable with a term t iff the sort of the term is subsumed by or equal to the sort of the variable x. Walther has also demonstrated that the result of using a many sorted resolution calculus has considerably reduced the initial search space. The same idea of reducing the search space with a many-sorted universe has also been applied in KLOKE to restrict the hypothesis space during the hypothesis generating stage, when the system is exploring possible hypotheses. Unfortunately, for realistic large scale knowledge bases, the restriction of sort will not be able to reduce the hypotheses space to a tractable size. In which case KLOKE has employed the heuristic mentioned below, beside using meta-knowledge to guide its discovery process.

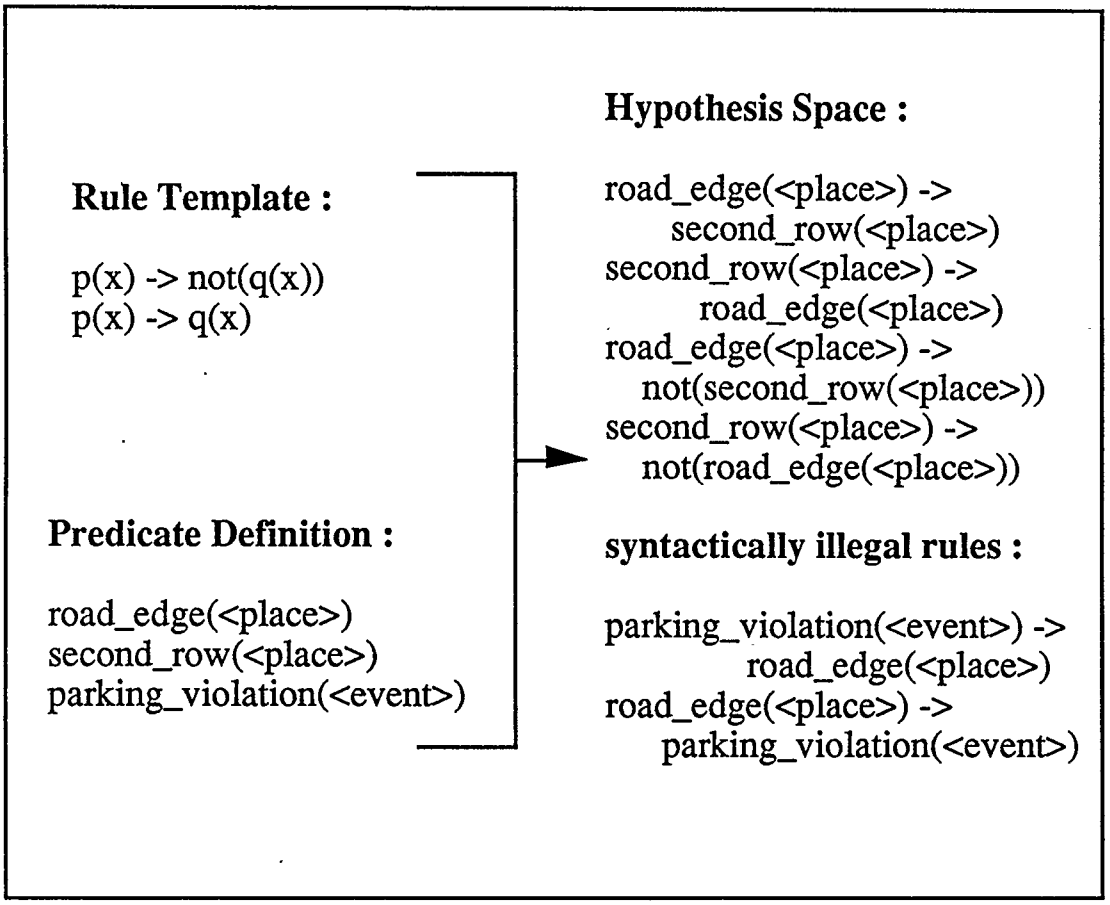

Fig 3.8 Checking Sort Correctness in Rules

Heuristics : Even though checking the sort correctness of rules and using meta-knowledge to filter out inconsistent rules considerably reduces the search space, the search will still be combinatorially exponential for large-scale practical knowledge bases. Some heuristics must be devised to allow the search to focus on some biased regions in the search space. Three heuristics are used in KLOKE for generating a hypothesis, and they select a predicate based on the following criteria :-

- prefer generating hypotheses about a predicate that has been used recently by the user
- prefer to generate hypotheses that involve predicate(s) which have been used in a large number of facts
- prefer to generate hypotheses about predicates about which few rules are known

Rating of hypotheses : Each rule is numerically rated with weighted addition distributed among the three criteria mentioned above.

Hypothesis Verification : The hypothesis testing module verifies all rules suggested by the hypotheses generation module against the knowledge base. A search pattern called the *characteristics situation* is used to count the number of positive and negative instances of the hypothesis. This is used for deciding whether to accept or reject a hypothesis. If a hypothesis is confirmed, it is entered into the knowledge base, and is used to infer additional facts. As an example, consider the meta-predicate w_trans(p,q,r) with the rule-model [p(x,y), q(z,y)] -> r(z,x). It has the following domain rule :

[involved_vehicle(event,car), owner(person, car)] -> responsible(person,event)
and a positive characteristic situation :

involved_vehicle(event, car) & owner(person, car) & responsible(person, event)

and a negative characteristic situation :

involved_vehicle(event, car) & owner(person, car) & not(responsible(person, event))

Thus, in general, each meta-predicate is supplied with a *pattern of verification* and a *pattern of falsification.* In the above example the pattern of verification is :

p(x,y), q(z,y), r(z,x)

and the pattern of falsification is :

p(x,y), q(z,y), not(r(z,x))

3.6 Summary

This chapter has shown the overall architectural design of KLOKE. Like the BLIP system, KLOKE has three major functional units : the knowledge acquisition environment, the knowledge representation sub-system, and the discovery component. The knowledge representation sub-system consists of a sort classifier which maintains a sort taxonomy and a reasoning maintenance system to maintain a data dependency network. The discovery component uses a two step generalization process to acquire meta-knowledge. It also includes a sub-program which discovers domain rules by using the set of existing rules and facts in the knowledge base as background knowledge. Unlike BLIP, KLOKE does not have a sub-module to learn concepts. The implementation of the sort classifier, the reasoning maintenance system and the module for acquiring meta-knowledge are presented in the next chapter. The knowledge acquisition environment will consist of the user interface for KLOKE which will be described in chapter 6.

Chapter 4
The System Implementation of KLOKE

This chapter will present the implementation detail of the sort classifier, the reasoning maintenance system and the meta-knowledge acquisition module.

4.1 Implementing the Sort Classifier

The approach used for constructing and restructuring the sort taxonomy and the formalism for representing the terminological knowledge and the Prolog implementation of the Sort Classifier are described here. Axioms and rules for finding a correct position to insert a new sort into the sort taxonomy are presented. The two operations for evolving the sort taxonomy mentioned in chapter 3 are illustrated here with greater details.

The hierarchy of each sort is computed by establishing set relations of its extension to the extensions of other sorts in the taxonomy. In KLOKE the set of admissible elements in an argument of a fact belong to a sort defined by the system or by the user. The system is capable of generating unique sort labels such as 'concept0' , 'concept1', and so on. The user can use more descriptive mnenomics such as 'symptoms,' 'disease', and so on as in the example given below. The classifier will maintain a lexicon of the predicate name and the list of arguments that are used to describe a fact.

Terminological Formalism in KLOKE : Figure 4.1 shows an extended BNF definition of the terminological formalism used in KLOKE. An extension is the set of terms which belongs to a sort. A fact is represented by a predicate. Each argument in a predicate is allocated an argument place. The universal set is the set of all terms used in the set of predicates that are used to represent the set of known facts.

fact ::= predicate

predicate :: = predicate_name (argument_sort..)

sort_taxonomy ::= sort_relation..

sort_relation ::= sort relation sort

relation ::= equivalence | subclass | superclass | intersection | disjoint

sort :: = sort_name, argument_sort.. ,extension..

argument_sort ::= argument_place_name, extension

extension ::= term..

term ::= identifier

predicate_name ::= identifier

sort_name ::= identifier

argument_place_name ::= identifier

identifier ::= character..

Fig 4.1 An Extended BNF Definition of the Terminological Formalism in KLOKE

4.1.1 The Sort Taxonomy

To built a sort taxonomy from a set of given facts, the system has to define the correspondence between argument places and sorts. Having established this correspondence (mapping) the extensions of the sorts can be computed. In the next few

sections, the extensional definitions of the relations of sorts that are generated from their intensional definition are discussed. These abduced extensional definitions are needed so that one can operationalize the hypotheses of sort relations. The extensions of the sorts are used to compute the following relations between sorts :

- the equivalence of sorts
- the subsumption of sorts
- the intersection of sorts
- the disjointness of sorts

The Equivalence Classes of Sorts : Establishing equality relations between argument sorts is achieved by decomposing the set of argument sorts into equivalence classes of argument sorts. Two or more argument sorts having the same extension are mapped into the same equivalence class. For example, if the following facts are entered :

indicate(sore_throat, flu)
cause(flu, sore_throat)

The system will generate a sort for each argument of each predicate so that the first argument sort of the predicate 'indicate' and the second argument sort of the predicate 'cause' both have the same extension sore_throat and will be mapped into a sort 'symptom' (a name which can be given by the user). This will result in an injective mapping between the set of equivalence classes and the set of extensions.

Intensional Definition : *Two concepts c1,c2 are equivalent iff they both imply the same constraints.*
Extensional Definition : *By abduction, two sorts s1,s2 are equivalent iff the extension set of s1 is equal to the extension set of s2.*

The Subsumption between Sorts : This relation is useful in determining the compatibility between sorts, because of its representative and definite properties. The compatibility between argument sorts can be induced from the compatibility between the classes that they are contained in. In the sort checker, the compatibility of sorts will be used to check the sort correctness of variable bindings in formulas used to represent facts and rules. The requirements for the subsumption relation is that there must exist a partial ordering between the extension sets of the classes.

Intensional Definition : *A concept, c1 is subsumed by another concept, c2 iff the constraints in c1 are necessarily implied in c2.*

Extensional Definition : *By abduction, a sort, s1 is subsume by another sort s2 iff the extension set of s1 is a subset of the extension set of s2, that is s1 is a subclass of s2 if all the terms belonging to s1 also belongs to s2.*

The Intersection and Disjointness between Sorts : Beside the equivalence and subsumption relations, the sorts in the taxonomy may have extension sets that overlap each other without any one of the extension set subsuming the other. Intuitively, this relation is used to differentiate between the overlapping and disjunction of classes. Two classes that are disjoint will have an infimum that is equal to the null set, while overlapping classes have an infimum that is greater than the null set. The system will generate a new sort to represent the intersection of the extension sets of the overlapping classes. This intersection sort does not represent admissible arguments of predicates but rather a possible sort of variables.

Intensional Definition : *Two concepts c1,c2 intersect each other iff it is possible to satisfy both their constraints together.*
Extensional Definition : *By abduction, two sorts s1,s2 intersect each other iff the intersection of the extension set of s1 and the extension set of s2 are not empty.*

Mapping the Set of Terms into Sorts : The classifier sub-system will initially define a unique sort for each argument of each of the predicate and the basic idea is to map the argument sort to a sort in the sort taxonomy. This is done by computing the equality of argument sorts by comparing their extension sets. Equivalence argument sorts are mapped into the same sort within the sort taxonomy and the extension of a sort in the sort taxonomy will be equal to the extension sets of its corresponding list of argument sorts. The set of terms or extensions set of an arguments sort is derived by collecting all the terms that are used in its corresponding argument place in the facts that it belongs to.

4.1.2 Acquiring the Sort Taxonomy

Initially, the system will start off with an empty taxonomy. As the user enters facts to the system, the classifier will invoke the *add* operation to restructure the sort taxonomy based on the set of facts that has been entered. Each state of the taxonomy corresponds to the current state of modelling.

The Operation for Building a Sort Taxonomy : The procedure used to implement the operation adding a sort to a given sort taxonomy is described below.

Step 1 : Start with a taxonomy consisting of only the empty set.

Step 2 : For every fact that is added to the knowledge base, there are two possible cases to be considered when a fact is added :

1. The predicate used to represent the fact has never been defined before.
2. The fact is represented by a predicate that has already been defined by the system.

Case (2.1)

2.1.1 Define the predicate name in the lexicon.

2.1.2. Generate a new argument class (sort) for each argument in the predicate representing the fact.

2.1.3. Collect the terms of each argument and place them in an extension set.

2.1.4. Try to map each argument sort in the predicate to a sort in the sort taxonomy by comparing their extension sets.

2.1.5. If an argument sort cannot be mapped to any existing sort in the sort taxonomy,

- Create a new sort in the sort taxonomy that maps to this argument sort
- Establish the relation of this new sort with all the other sorts in the taxonomy using the axioms described in the next section

Case (2.2)

2.2.1. Append the list of terms in the predicate representing the fact to their corresponding extension set.

2.2.2. Since the predicate representing the fact has been defined, all the arguments in this predicate have already been mapped to a sort in the sort taxonomy. There are two possible cases of the mapping for each argument sort and their corresponding sort in the sort taxonomy.

Case (2.2.2a)

The corresponding sort in the sort taxonomy is only mapped to the argument sort itself. In this case no new sort need be generated, but since the extension of the argument sort and the extension of the corresponding sort in the sort taxonomy changes due to an addition of a new term, the old relations of the sort with other sorts in the sort taxonomy no longer hold. The following steps are then carried out:

- Purge all the old relations of the sort with other sorts in the sort taxonomy
- Restablish the relations of the this sort with other sorts in the sort taxonomy

Case (2.2.2b)

There is a one to many mapping between the sort in the sort taxonomy and a list of argument sorts. In this case the following steps are carried out :

- Subtract the argument sort from the list of argument sorts that are mapped to the corresponding sort in the sort taxonomy
- Check if the argument sort can be mapped into any sort in the sort taxonomy
- Perform Step 2.1.5

The Operation to Restructure the Sort Taxonomy : Given a sort taxonomy that classifies the sort of the arguments of the facts in the knowledge base, there are three possible cases to be considered when a fact is to be retracted from the knowledge base :

1. When the fact to be deleted is represented by a predicate that is also used to represent other fact or facts. That is, we are deleting a fact that has a predicate name (defined in the lexicon) that is similar to that used by other facts in the knowledge base.

2. When a fact to be deleted is represented by a predicate name that is different from all the other facts in the knowledge base.

3. When the fact to be deleted does not exist in the knowledge base. This is an exception.

Case (1)

1.1 Remove the list of terms of the deleted fact from the corresponding list of extension sets in the list of argument sorts.

1.2 Following step 1.1 the corresponding list of extension sets for each of the argument sort of the deleted fact also changes and the mappings to their corresponding sorts in the sort taxonomy will have to be recomputed. Two possible cases are considered when the system is recomputing the mapping of the list of arguments sort for the deleted fact.

Case (1.2.1)

The corresponding sort in the sort taxonomy is only mapped to the argument sort of the deleted fact. Since the extension set of the argument sort changes, the extension set of the corresponding sort in the sort taxonomy must also be changed accordingly. Also the old relations of the sort with other sorts in the sort taxonomy no longer hold and the following steps are then carried out:

- Purge all the old relations of the sort with other sorts in the sort taxonomy
- Restablish the relations of the sort with other sorts in the sort taxonomy

Case (1.2.2)

The sort which is mapped to the argument sort of the deleted fact are also mapped to other argument sorts of other facts. In this case the following steps are carried out :

- Subtract the argument sort from the list of argument sorts that are mapped to the corresponding sort in the sort taxonomy
- Check if the argument sort can be mapped into any sort in the sort taxonomy
- Perform Step 2.1.5

Case (2)

2.1 Remove the predicate name of this deleted fact from the lexicon since it no longer exists.

2.2 Remove the mapping of the list of argument sorts of this fact with their corresponding sorts in the sort taxonomy.
2.3 If any of the corresponding sorts of the list of argument sorts of the deleted fact is not mapped to other argument sorts from other predicates, it will be removed (the deletion of a sort in the sort taxonomy is shown in figure 4.2 and 4.3). When a sort is deleted from a taxonomy, all its infimums become the infimums of all its supremums.

else

it will still maintain its mapping with other argument sorts of other facts

Case (3)

Simply report the error.
An example test case for the sort classier will be provided in Chapter 5.

4.1.3 A Prolog Implementation of the Sort Classifier

This section will present the implementation of the sort classifier in Prolog. The sort classifier is invoked whenever the sort taxonomy is to be restructured due to a fact addition or deletion. The sort classifier is invoked using the function update_knowledge_base which takes a fact as its input. The function restructure_sort_taxonomy is a top-level function of the sort classifier module.

```
update_knowledge_base(_operation, fact) :-

        (_operation = add ; _operation = delete)
                ->[get_symbol(_fact),
                _fact = _predName(_argList..),
        once(restructure_sort_taxonomy(_predName,_argList,_operation))];
                write('error').
```

When a fact is added to the knowledge base there are two possible cases to consider :
(i) when the fact is represented by a predicate that has not yet been defined
(ii) when the fact is represented by an existing predicate definition

```
restructure_sort_taxonomy(_predName,_argList, add) :-

    (lexicon(_predName,_sortList,_argPlaceList,
            _extensionList,_termListL,_numberOfFacts)) ->
    [add_a_defined_predicate(_predName,_argList,_sortList,_argPlaceList,
            _extensionList,_termListL,_numberOfFacts)];
    [add_an_undefined_predicate(_predName,_argList,_sortList,_argPlaceList,
```

```
_extensionList,_termListL,_numberOfFacts)].
```

The following function is used to compute the sort of all the arguments of an input fact that is represented by a predicate which is not in the system's lexicon, that is a fact whose predicate has not been previously defined. The system will first hypothesize that the sort of all the arguments of the input fact to be unique and different from all the sorts in the sort taxonomy by generating a new sort name for each argument sort. It will then compute the extension set of each argument by collecting the term that appears in the argument place holder of the predicate. Following that, the system will try to establish an equivalence relation between an argument sort and a sort in the taxonomy. If an argument sort cannot be mapped with any of the sorts in the taxonomy, a new sort is created and the system will insert it into the appropriate hierarchy in the taxonomy. Lastly, the classifier will also build the predicate definition of the fact that has been added.

```
add_an_undefined_predicate(_predName,_argList,_sortList,_argPlaceList,_extensionList
        , _termListL,_numberOfFacts):-

        [initialize_counter(arg_counter),
        generate_sortname(_predName,_argList, _argPlaceList, sortList, _emptyList),
        append_extension(_emptyList,_extensionList,_argList, _emptyList,_termListL),
        creat_sort(_sortList,_extensionList,_argPlaceList),
        once(mapping_sorts(_sortList,_extensionList)),
        map_argument_sort_to_lattice_sort(_argPlaceList,_newSortList),
        update_argument_sort(_sortList, _newSortList),
        assert(lexicon(_predName,_newSortList,_argPlaceList,_extensionList,
                _termListL,1)).
```

The following function is used to compute the sorts of the arguments in an input fact that is represented by an already defined predicate. It will collect the list of terms in the fact and append them to their respective extension set. Since the extension sets of each argument changes, their corresponding sorts will also change and must be re-computed.

```
add_a_defined_predicate(_predName,_argList,_sortList,_argPlaceList,
        _extensionList,_termListL,_numberOfFacts):-

         [retract(lexicon(_predName,_sortList,_argPlaceList,
                _extensionList,_termListL,_numberOfFacts)),
        _newNumberOfFacts is _numberOfFacts + 1,
        append_extension(_extensionList,_newExtensionList,
                _argList,_termListL,_newTermListL),
        update_extension_in_sort(_argList,_sortList,
                _newSortList,_argPlaceList, _newTermListL,add),
        once(mapping_sorts(_newSortList,_newExtensionList)),
        map_argument_sort_to_lattice_sort(_argPlaceList,_newSortList1),
```

```
        assert(lexicon(_predName,_newSortList1,_argPlaceList,
                _newExtensionList,_newTermListL,_newNumberOfFacts))].
```

The following segment of code is used to restructure the sort taxonomy when a fact is retracted from the knowledge base. There are three possible cases when a fact is deleted :
(i) When the deleted fact is the only fact of a predicate. In this case the predicate definition must be deleted.
(ii) When there are one or more facts represented by the same predicate definition.
(iii) When the deleted fact does not exist - this is an error.

```
restructure_sort_taxonomy(_predName,_argList, delete) :-

        [lexicon(_predName,_sortList,_argPlaceList,_extensionList,
                _termListL,_numberOfFacts) ] ->
                [retract(lexicon(_predName,_sortList,_argPlaceList,
                        _extensionList,_termListL,_numberOfFacts)),
                        _newNumberOfFacts is _numberOfFacts - 1,
                [_newNumberOfFacts > 0 ->
                [delete_a_predicate_with_more_than_one_fact(_predName,_argList,
                                _argPlaceList,_sortList, _newSortList,_newSortList1,
                                _extensionList,_newExtensionList,_termListL,
                                _newTermListL, _newNumberOfFacts)] ;

                [ delete_a_predicate_with_one_fact(_argList,_argPlaceList,_sortList,
                                _newSortList,_termListL)]
                ] ];
                write('error - facts do not exist').

delete_a_predicate_with_more_than_one_fact(_predName,_argList,_argPlaceList,
        _sortList, _newSortList,_newSortList1,_extensionList,_newExtensionList,
        _termListL,_newTermListL,_newNumberOfFacts):-

                        subtract_extension(_extensionList,_newExtensionList,
                                _argList,_termListL,_newTermListL),
                        update_extension_in_sort(_argList,_sortList,
                                _newSortList,_argPlaceList,_newTermListL,delete),
                        once(mapping_sorts(_newSortList,_newExtensionList)),
                        map_argument_sort_to_lattice_sort(_argPlaceList,_newSortList1),
                        assert(lexicon(_predName,_newSortList1,_argPlaceList,
                                _newExtensionList,_newTermListL,_newNumberOfFacts)).

delete_a_predicate_with_one_fact(_argList,_argPlaceList,_sortList,
        _newSortList,_termListL):-

        update_extension_in_sort(_argList,_sortList,_newSortList,
                _argPlaceList,_termListL,delete),
        purge_sort(_newSortList).
```

The following function will define the argument place holder of the input fact and generate a unique sort name for each argument. For example, if the fact affect_pos(inspirol, sore_throat) is entered, the place holders [affect_pos0, affect_pos1] and the corresponding names for the hypothesized sorts are [class0, class1] by default. The user can also enter sort names like [substance, irritations].

```
generate_sortname(_,[],[],[],[]).

generate_sortname(_predName,[_,_argList..],[_argPlace,_argPlaceList..],
        [_sort,_sortList..],[[],_dummy..]) :-

        gen_sym(_argPlace, _predName, arg_counter),
        hypothesize_a_new_sort(_sort),
        generate_sortname(_predName,_argList,_argPlaceList,_sortList,_dummy).
```

The append_extension function simply collects the set of terms that appear in each argument place holder of a fact. For example if the fact : affect_pos(inspirol, sore_throat) is entered, the extension of the two arguments of affect_pos are [inspirol], [sore_throat] and if the fact affect_pos(aspirin,flu) is entered the extension sets will changed to [inspirol, aspirin] and [sore_throat, flu]. If the fact affect_pos(aspirin, cough) is entered, the extension sets become [inspirol, aspirin] and [sore_throat, flu, cough]. The system will also maintain a list of terms that appear in each argument place. In the example the two lists are [inspirol, aspirin,aspirin] and [sore_throat, flu, cough]. The reason for maintaining these lists will become clear when the subtract_extension function is described.

```
append_extension([],[],[],[],[]).

append_extension([_extension,_extensionList..], [_extension,_newExtensionList..],
          [_arg,_argList..], [_termList,_termListL..],
           _newTermList,_newTermListL..]) :-

              member(_arg, _extension),
              append(_termList,[_arg], _newTermList),
              append_extension(_extensionList, _newExtensionList,_argList,
                        _termListL,_newTermListL),!.

append_extension([_extension,_extensionList..], [_newExtension,_newExtensionList..],
       [_arg,_argList..], [_termList,_termListL..],[_newTermList,_newTermListL..]) :-

              append(_extension,[_arg], _newExtension),
              append(_termList,[_arg], _newTermList),
              append_extension(_extensionList, _newExtensionList,_argList,
                        _termListL,_newTermListL).
```

This function creates a new sort for each argument in the input fact. Here the system establishes the correspondence of the sort names and argument place holder names generated in generate_sortname function and the extension sets built in append_extension function. For the fact affect_pos(aspirin, flu) the following classes are created : class(substance, [aspirin], [affect_pos0]) and class(irritation, [flu], [affect_pos1]).

```
creat_sort([],[],[]).

creat_sort([_sort,_sortList..],[_extension,_extensionList..],[_argPlace,_argPlaceList..]) :-

    assert(class(_sort,_extension,[_argPlace])),
    creat_sort(_sortList,_extensionList,_argPlaceList).
```

The sort of each argument is computed by comparing its extension set with the extension sets of all the sorts in the sort taxonomy and is inserted into the appropriate hierarchy in the taxonomy. Each sort in the sort taxonomy can be mapped to several arguments in different facts. The predicate class(sort,extensionSet, argumentPlaceList) contains information of all the arguments which are mapped into that sort. This function will record the mapping between an argument sort in the taxonomy.

```
map_argument_sort_to_lattice_sort([],[]).

map_argument_sort_to_lattice_sort([_argPlace,_argPlaceList..],
        [_newSort,_newSortList..]) :-

        class(_newSort,_,_argPlaceList1),
        member(_argPlace, _argPlaceList1),!,
        map_argument_sort_to_lattice_sort([_argPlaceList..],[_newSortList..]).
```

If the corresponding extension set of an argument is equal to the extension set of an existing sort in the sort taxonomy, an equivalence class is established and the argument sort is mapped to the sort in the taxonomy. For example, the knowledge base may contain the fact indicate(sore_throat,flu) with the sorts [symptom, disease]. When the fact affect_pos(inspirol, sore_throat) is entered, suppose that the system hypothesized the argument sorts to be [substance, class0], where class0 is just an arbitary class. After the mapping operations the system noticed that sore_throat belongs to the sort symptom and the function described here is used to update the argument sort.

```
update_argument_sort([], []).

update_argument_sort([_sort,_sortList..], [_sort,_newSortList..]) :-
```

```
    update_argument_sort([_sortList..], [_newSortList..]).

update_argument_sort([_sort,_sortList..], [_newSort,_newSortList..]) :-

    purge_sort_relation(_sort),
    update_argument_sort([_sortList..], [_newSortList..]).
```

When a fact is deleted, all the terms appearing as its arguments must also be deleted from their corresponding extension sets. For example, if the facts affect_pos(aspirin, flu) and affect_pos(aspirin, sore_throat) are currently in the knowledge base then the extension sets of the two arguments of affect_pos are [aspirin] and [flu, sore_throat]. If the system tries to delete the fact affect_pos(aspirin, flu) then we may end up having the extensions sets [] and [sore_throat] which is clearly incorrect. This is not the case if the system maintains a list to keep track of all the term appearing in each argument place holder of a fact as mentioned previously in the append_extension function. Maintaining the two lists [aspirin, aspirin] and [flu,sore_throat], when the fact affect_pos(aspirin, flu) is deleted, the two extensions will become [aspirin] and [sore_throat] because the system removes a term from an extension set only if this term can be totally removed from the corresponding term list.

```
subtract_extension([],[],[],[],[]).

subtract_extension([_extension,_extensionList..],[_newExtension,_newExtensionList..],
    [_arg,_argList..],[_termList,_termListL..],[_newTermList,_newTermListL..]) :-

    minus(_arg, _termList,_newTermList),
    [not(member(_arg,_newTermList)) ->
        [minus(_arg, _extension, _newExtension)];
        [_newExtension = _extension]],
    subtract_extension(_extensionList,_newExtensionList,
        _argList,_termListL,_newTermListL),!.
```

When a fact is added to or deleted from the knowledge base the extension sets of all its arguments must be updated. Consider the case when a fact is added. If a term is already in the extension set of an argument, it is not added to it. Otherwise, if the term is added to the extension set, the corresponding argument sort must be recomputed. There are two cases to consider here :

(i) when there is a one-to-one mapping between the sort in the taxonomy and the argument sort. For example the predicate class(irritation, [sore_throat], [affect_pos1]) indicates that the second argument of the fact affect_pos is mapped into the sort

'irritation' and no other argument sorts are mapped into the sort 'irritation'. So, if a fact affect_pos(aspirin, flu) is added, no new sort is generated but the system will only change the extension set. In our example, the class predicate becomes class(irritation, [sore_throat,flu], [affect_pos1]). Since its extension set changes, the relations between the sort 'irritation' and other sort in the taxonomy no longer holds and must be purged. The hierarchy of the sort 'irritation' is recomputed using the function mapping_sorts.

(ii) when there is a one-to-many mapping between the sort in the taxonomy and several other argument sorts. For example, if there is a sort 'symptom' represented by the class(symptom, [sore_throat], [indicate0, affect_pos1]) which is mapped to the first argument of the fact 'indicate' and the second argument of the fact 'affect_pos'. If the fact affect_pos(aspirin, flu) is added, the corresponding extension set of the second argument of 'affect_pos' is changed to [sore_throat, flu] and should no longer be mapped to the sort 'symptom'. The system will have to delete the mapping between the sort 'symptom' and the argument place holder affect_pos1 resulting in class(symptom, [sore_throat], [indicate0]). A new sort for the second argument of 'affect_pos' is then hypothesized, resulting in class(irritation, [sore_throat, flu], [affect_pos1]).

When a fact is deleted from the knowledge base, the extension set of all its arguments must also be changed. Again there are two cases to be considered :

(i) when a term is deleted from the extension set of an argument which has a one-to-one mapping with a sort in the taxonomy. In this case no new sort is generated but the relations of the sort with other sorts in the taxonomy are purged and the sort hierarchy is recomputed.

(ii) when a term is deleted from the extension set of an argument which is mapped to a sort in the taxonomy that has a one-to-many relation with other argument sorts. For example, if we have the facts indicate(sore_throat, flu), indicate(cough, flu), cause(flu, sore_throat), cause(flu, cough) and the sort 'symptom' represented by class(symptom, [sore_throat, cough], [indicate0, cause1]). If we deleted the fact cause(flu, sore_throat) the extension set of the second argument of 'cause' becomes [cough] and the mapping of the sort 'symptom' becomes class(symptom, [sore_throat, cough], [indicate0]), the second argument of the predicate 'cause' is no longer mapped to the sort 'symptom'. A new sort, say symptom1, needs to be created and is represented by class(symptom1, [cough], [cause0])

```
update_extension_in_sort([],[],[],[],[],_).

update_extension_in_sort([_arg,_argList..],[_sort,_sortList..],[_newSort,_newSortList..],
        [_argPlace1,_argPlaceList1..],[_newTermList,_newTermListL..],_operation) :-

        [[class(_sort,_oldExtension,_argPlaceList),
        [[_operation = add, not(member(_arg, _oldExtension))];[_operation = delete]]] ->
                [update_extension(_oldExtension,_newTermList, _arg,
                        _newExtension, _operation),
                retract(class(_sort,_oldExtension,_argPlaceList)),
                check_mapping_of_argument_sort(_argPlace1, _argPlaceList,
                        _newArgPlaceList,_sort,_oldExtension,_newSort,_newExtension)]
        ],
        update_extension_in_sort(_argList,_sortList,[_newSortList..],[_argPlaceList1..],
        _newTermListL,_operation).

update_extension(_oldExtension,_, _arg, _newExtension, add) :-

        append(_oldExtension, [_arg], _newExtension).

update_extension(_oldExtension,_newTermList, _arg, _newExtension, delete) :-

        [not(member(_arg,_newTermList)) -> [minus(_arg, _oldExtension,
                _newExtension)];
        [_newExtension = _oldExtension]].


check_mapping_of_argument_sort(_argPlace1, _argPlaceList, _newArgPlaceList,
        _sort,_oldExtension,_newSort,_newExtension) :-

        [set_compare([_argPlace1],_argPlaceList, subset) ->
                [subtract_argument_sort(_argPlace1, _argPlaceList, _newArgPlaceList,
                        _sort,_oldExtension,_newSort,_newExtension) ];
                [purge_old_relations(_sort,_newExtension,_argPlaceList,_newSort)]
        ].

subtract_argument_sort(_argPlace1, _argPlaceList, _newArgPlaceList,
        _sort,_oldExtension,_newSort,_newExtension) :-

        minus(_argPlace1, _argPlaceList, _newArgPlaceList),
        assert(class(_sort,_oldExtension,_newArgPlaceList)),
        hypothesize_a_new_sort(_newSort),
        assert(class(_newSort,_newExtension,[_argPlace1])).
```

When the extension set of a sort changes due to addition or deletion of facts, its relations with other sorts in the taxonomy must be purged. This is performed by the function purge_old_relations which consists of deleting all its superclass, subclass and intersection relations with other sorts. The deletion of a sort in the sort taxonomy is shown in figures

4.2 and 4.3. When a sort is deleted from the sort taxonomy all its infimums become the infimiums of all its supremums.

```
purge_old_relations(_sort,_newExtension,_argPlaceList,_sort) :-

        assert(class(_sort,_newExtension,_argPlaceList)),
        purge_sort_relation(_sort),
        purge_intersection_sort(_sort).

purge_sort_relation(_class) :-

        foreach(sub_class(_class, _superClass) do
                [retract(sub_class(_class, _superClass)),
                foreach(sub_class(_subClass, _class) do
                        [ [not(_subClass = nil); not(_superClass = $all)] ->
                        assert(sub_class(_subClass, _superClass))
                        ])
                ]),
         foreach(sub_class(_subClass,_class) do
                retract(sub_class(_subClass,_class))),
        foreach(int(_class, _intClass) do
                retract(int(_class, _intClass))),
        foreach(int(_intClass, _class) do
                retract(int(_intClass, _class))).

purge_intersection_sort(_class) :-

          foreach(class(_class1,_extension1, _argPlaceList1)
                do
                [member_int(_class, _argPlaceList1, int(_int..)) ->
                        [retract(class(_class1, _extension1, _argPlaceList1)),
                        set_compare([int(_int..)],_argPlaceList1, subset)] ->
                                [minus(int(_int..), _argPlaceList1, _newArgPlaceList1),
                                assert(class(_class1, _extension1, _newArgPlaceList1))];
                                [purge_sort_relation(_class1),
                                purge_intersection_sort(_class1)]
                ]).

purge_intersection_sort(_).

member_int(_class,_argPlaceList, int(_class, _class1)) :-

        member(int(_class,_class1),_argPlaceList), _class1 \= $all.

member_int(_class,_argPlaceList, int(_class1, _class)) :-

        member(int(_class1,_class),_argPlaceList), _class1 \= $all.

purge_sort([]).

purge_sort([_sort, _sortList..]) :- retract(class(_sort,_,_)),purge_sort([_sortList..]).
```

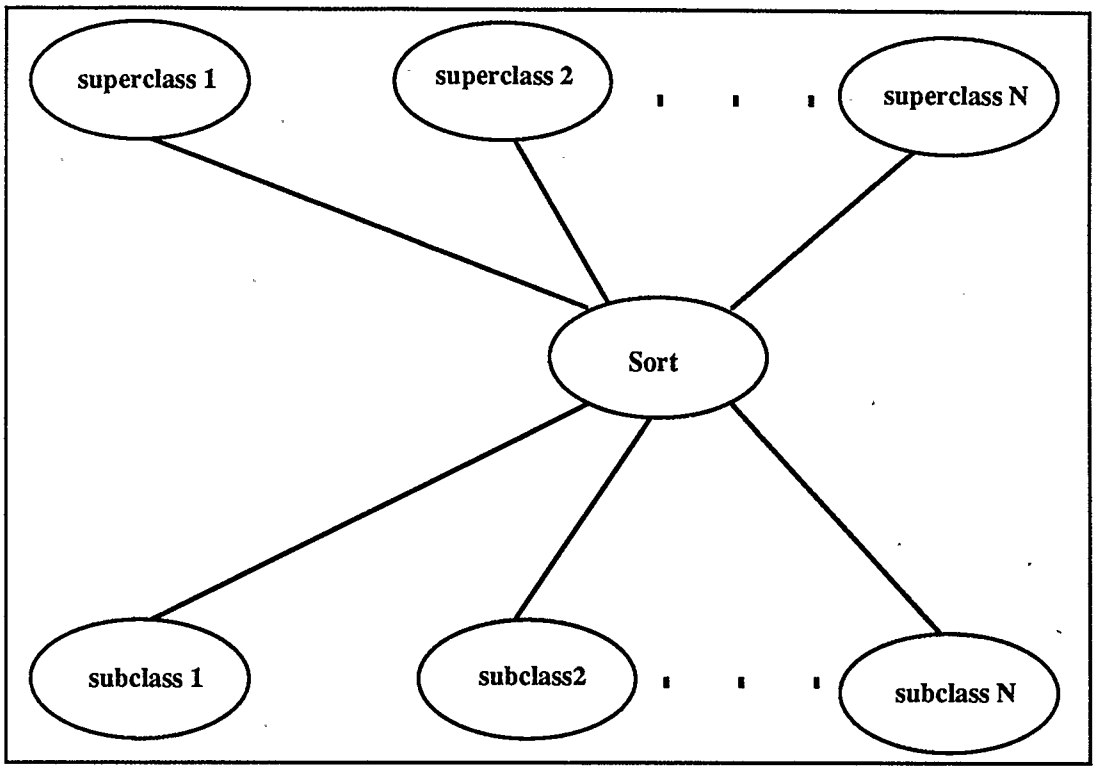

Fig 4.2 The State of a Sort Taxonomy Before Deleting a Sort

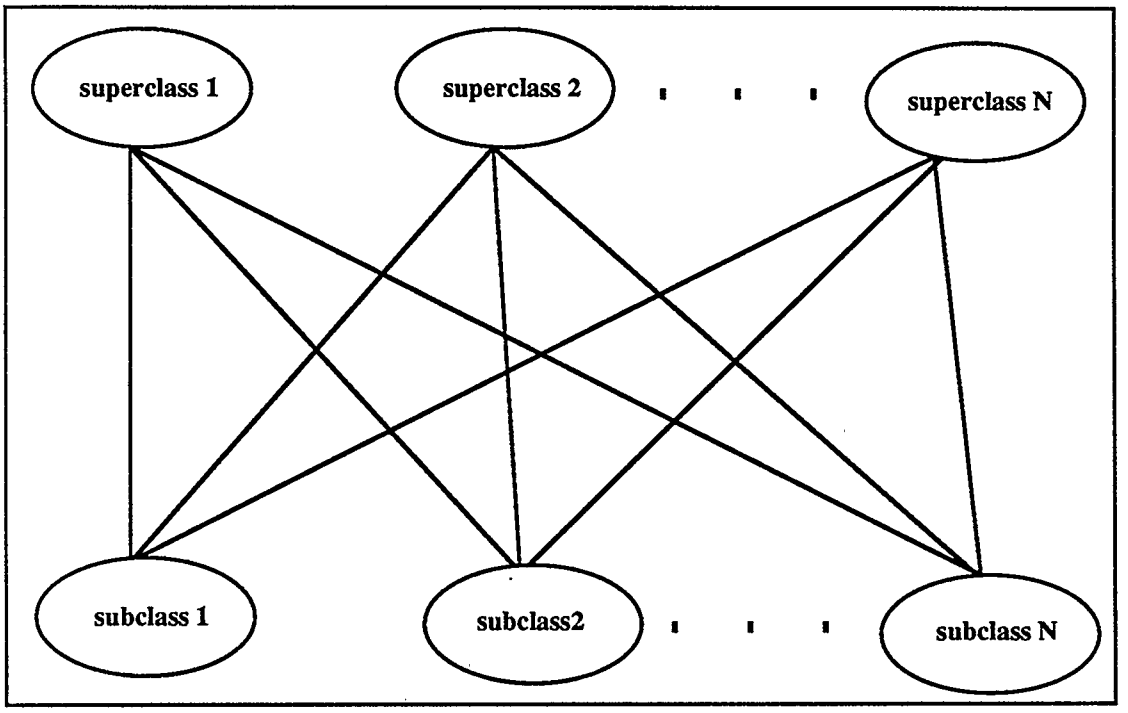

Fig 4.3 The State of a Sort Taxonomy After Deleting a Sort

This function is the core function of the sort classifier. It computes the relations among sorts in the sort taxonomy.

```
mapping_sorts([],[]).

mapping_sorts([_sort,_sortList..],[_extension,_extensionList..]) :-

    foreach( class(_class1,_extension1,_argPlaceList1)
    do
      [compute_sort_rel(_sort,_extension,_class1,_extension1,_argPlaceList1)] ),
    mapping_sorts(_sortList,_extensionList).

compute_sort_rel(_sort,_extension,_class1,_extension1,_argPlaceList1) :-

  [ _sort \= _class1 ->[set_compare(_extension, _extension1,_relation),
    sort_relation(_relation, _sort, _extension,_class1,_extension1,_argPlaceList1)]].
```

There are several kinds of relations between sorts in the taxonomy :

(i) equivalence relation : when two sorts have the same extension set, they are mapped into the same equivalence sort.

(ii) partial ordering relation : when the extension set of one sort is a subset of another sort.

(iii) intersection relation : when the extension of two sorts overlap.

(iv) disjoint relation : when the extension set of two sorts are disjoint.

```
sort_relation(equivalence,_sort, _extension,_class1,_extension1,_argPlaceList1) :-

    retract(class(_sort, _extension,_argPlace)),
    purge_sort_relation(_sort),
    append(_argPlaceList1, _argPlace, _newArgPlace1),
    retract(class(_class1,_extension1,_argPlaceList1)),
    assert(class(_class1,_extension1,_newArgPlace1)).

sort_relation(subset, _sort, _extension, _class1, _extension1, _argPlaceList1) :-

   [(sub_class(_subClass1, _class1))->
      [establish_partial_order_relation(_sort, _class1, _subClass1, subClassRelation)];
      [assert(sub_class(_sort, _class1))]].

sort_relation(superset, _sort, _extension, _class1, _extension1, _argPlaceList1) :-

    [(sub_class(_class1, _superClass1) ->
       [establish_partial_order_relation(_sort, _class1, _superClass1,
            superClassRelation)];
       [assert(sub_class(_class1, _sort))])].

sort_relation(intersect,_sort, _extension,_class1,_extension1,_argPlaceList1) :-
```

```
  assert(int(_sort,_class1)),
  intersection(_extension, _extension1, _intersection),
  hypothesize_a_new_sort(_newClass),
  assert(class(_newClass, _intersection, [int(_sort,_class1)])),
  (mapping_sorts([_newClass], [_intersection])).

sort_relation(disjoint,_sort, _extension,_class1,_extension1,_argPlaceList1).
```

Establishing the Partial Ordering between Classes : The following section describes the function used for establishing the sub-class and super-class relations between sorts.

```
establish_partial_order_relation(_sort, _class1, _subClass1, subClassRelation) :-

        [(class(_subClass1, _extension1,_)),
        (class(_sort, _extension2,_)),
        build_subclass_relation(_sort, _class1, _subClass1,_extension1,_extension2)].

establish_partial_order_relation(_sort, _class1, _superClass1, superClassRelation) :-

        [(class(_superClass1, _extension1,_)),
         (class(_sort, _extension2,_)),
        build_superclass_relation(_sort, _class1, _superClass1,_extension1,_extension2)].
```

SubClass Relation : To establish a subset relation between two sorts of class A and class B, the following axioms are used:

1. Every class in the taxonomy is a subset of the universal class.
2. The empty class is a subset of all the classes in the sort taxonomy.
3. Every class in the taxonomy except the empty class has at least one subset.
4. If a class, class A, is a subset of another class, class B, and if there exists another class in the taxonomy, class C, such that class C is a subset of class B then the following must be checked:

- If class C is a subset of class A then class C is no longer the infimum of class B. class A now becomes the infimum of class B and class C becomes the infimum of class A
- If class C is a superset of class A then clearly, class A is not an *immediate* subset of class B
- If class C intersects class A, then both class A and class C can be immediate subsets of class B
- If class A and class C are disjoint, both classes can be immediate subsets of class B

• If class A and class C are equivalent, an equivalence relation is established between them and the equivalence class is the subset set of class B

```
build_subclass_relation(_sort, _class1, _subClass1,_extension1,_extension2) :-

    set_compare(_extension1,_extension2, _relation),
    insert_subclass_link(_sort, _class1, _subClass1, _relation).

insert_subclass_link(_sort, _class1, _subClass1, subset) :-

        retract(sub_class(_subClass1, _class1)),
        [not(sub_class(_subClass1, _sort)) ->
                assert(sub_class(_subClass1, _sort))],
        [not(sub_class(_sort, _class1)) ->
                assert(sub_class(_sort, _class1))].

insert_subclass_link(_sort, _class1, _subClass1, superset).

insert_subclass_link(_sort, _class1, _subClass1, disjoint) :-

    [no_transitive_rel(_sort, _class1),not(sub_class(_sort, _class1))] ->
                                assert(sub_class(_sort, _class1)).

insert_subclass_link(_sort, _class1, _subClass1, intersect) :-

        not(sub_class(_sort, _class1)) ->
                assert(sub_class(_sort, _class1)).

insert_subclass_link(_sort, _class1, _subClass1, equivalence).
```

Super-Class Relation : To establish a superset relation between two sorts of class A and class B, the following axioms are used:

1. The universal set is a superset of all the classes in the sort taxonomy.
2. Every class in the taxonomy except the universal class, has at least one superset.
3. Every class in the taxonomy is a superset of the empty class.
4. If a class, class A, is a superset of another class, class B, and if there exists another class in the sort taxonomy, class C such that class C is a superset of class B then the following must be checked:

• If class C is a subset of class A then clearly, class A is not an *immediate* superset of class B

- If class C is a superset of class A, then class C is no longer the immediate superset of class B and class C now becomes the supremum of class A and class B becomes the infimum of class A
- If both class A and class C are disjoint, they can both be immediate supersets of class B
- If class A and class C are equivalent, an equivalence relation is established between them and the equivalence class is the superset set of class B
- If class A intersects class C, both the classes can be immediate supersets of class B.

```
build_superclass_relation(_sort, _class1, _superClass1,_extension1,_extension2) :-

    set_compare(_extension1,_extension2, _relation),
    insert_superclass_link(_sort, _class1, _superClass1, _relation).

insert_superclass_link(_sort, _class1, _superClass1, subset).

insert_superclass_link(_sort, nil, _superClass1, disjoint) :-

    [no_transitive_rel(nil,_sort),not(sub_class(nil, _sort))] ->
            assert(sub_class(nil, _sort)).

insert_superclass_link(_sort, _class1, _superClass1, equivalence).

insert_superclass_link(_sort, _class1, _superClass1, superset) :-

    retract(sub_class(_class1, _superClass1)),
    [not(sub_class(_sort, _superClass1)) ->
            assert(sub_class(_sort, _superClass1))],
    [not(sub_class(_class1,_sort)) ->
            assert(sub_class(_class1,_sort))].

insert_superclass_link(_sort, _class1, _superClass1, intersect) :-

    not(sub_class(_class1, _sort)) ->
            assert(sub_class(_class1, _sort)).

no_transitive_rel(_sort, _class1) :-

    class(_sort, _sortExt, _), class(_class1, _class1Ext,_),
    foreach([class(_class2, _class2Ext,_),
            _class2 \= _class1,_class2 \= _sort,] do
            [[set_compare(_class2Ext,_sortExt,superset),
            set_compare(_class1Ext,_class2Ext,superset)]->!,fail]).
```

4.2 Implementing the Reasoning Maintenance System

The implementation details of the reasoning maintenance system of KLOKE is described here. A high level pseudo-code for implementing the operation for pruning the data dependency network and the operation for building the data dependency network are presented.

4.2.1 Recording Data Dependencies

In the design of the reasoning maintenance system of KLOKE, the inference mechanism is incorporated into the reasoning maintenance system itself. This allows each inference step to be recorded. The idea is that whenever a rule 'fires', the system will create a new justification node that points to the inferred fact, and establishes a link from all beliefs that participate in this justification to the justification node.

Building a Data Dependency Network : The procedure for building the data dependency network is summarized below.

For every rule that fires, the system will do the following :

1. Create a justification node that points to the belief inferred from the rule. If this justification is the first justification of the belief that it points to, then the arc pointing from the justification to the belief is called its *current support*. The idea of a current support will be illustrated in the following section.

2. Establish an arc which points from a node representing the rule to the justification that it participates in.

3. For every proposition in the premises of the rule, establish an arc pointing from the proposition to the justification that it points to.

Given the following proposition :

fact0 : Man(Joe) fact1 : Human(Joe) fact2 : Person(Joe)
rule0 : Man(x) -> Human(x) rule1 : Human(x) -> Person(x)

rule2 : Person(x) -> Human(x)

The Reasoning Maintenance System will build the data dependency network shown in figure 4.4.

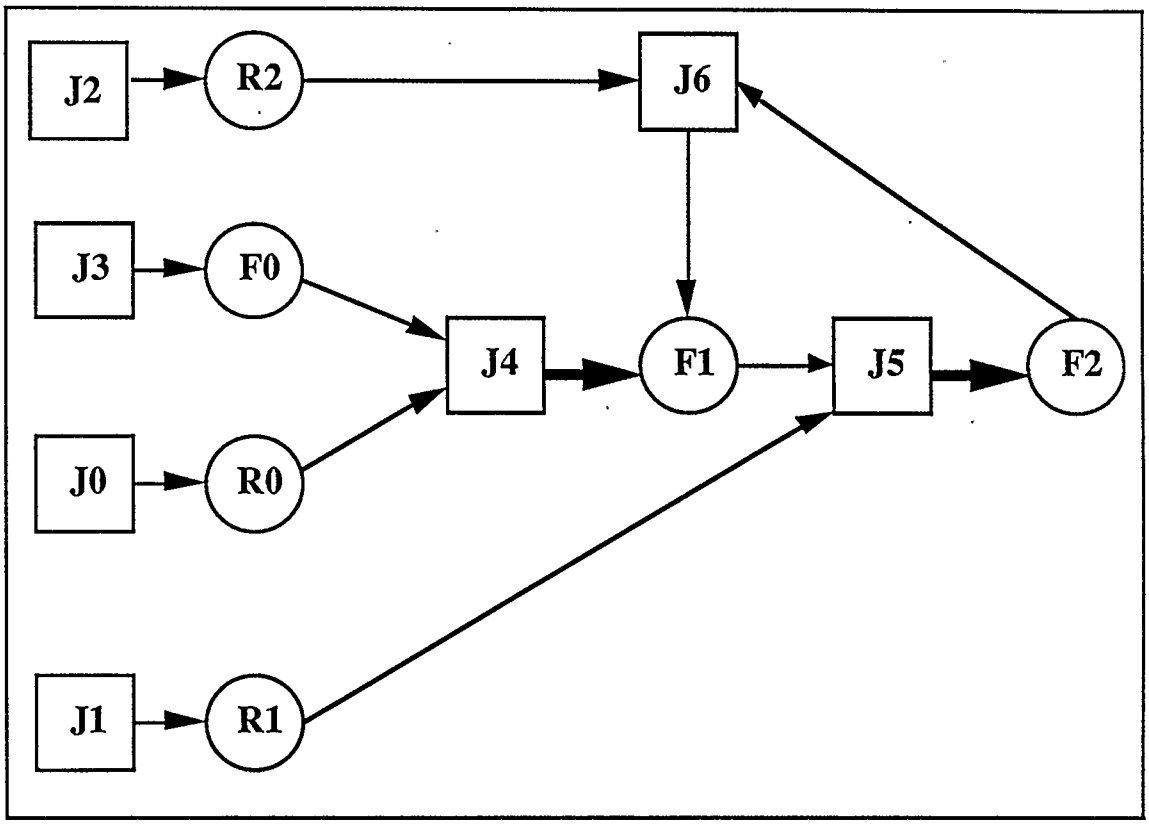

Fig 4.4 A Data Dependency Network of the Proposition Given Above

Pruning the Data Dependency Network : To delete a belief, the system will first delete the justification to the belief. An intuitive method for deleting a justification is to simply remove it and remove all the nodes it supports which do not have an alternate justification and remove all the justifications that are pointed to be any removed nodes. This process is applied recursively until no justification is removed. However, this method does not handle data dependency networks that contains not *well-founded* beliefs. To illustrate this point, the case of deleting the justification, J3 (in the data dependency network shown in figure 4.4) is considered. Fact0 (represented by F0) and justification, J4 will be remove but facts that are derived fact0 , fact1 and fact2 (represented by F1 and F2) still remain.

A method for handling *unfounded beliefs* is proposed by Doyle (1979) and McAllester (1982). This method requires that for all the nodes, one of its justifications must be

maintained and guaranteed to be well-founded. This justification of the node is called the *current support* of the node. When the data dependency network is first constructed, the first justification that supports a node when added to the network will be used as the current support for that node. The current supports are represented by bold arrows in figure 4.4. An example for deleting a data dependency network using the following method will be given in chapter 5.

Step 1 : If a justification to be remove is not the current support of any node, simply remove it and exit

else perform steps 2 to 5.

Step 2 : If a justification is the current support of some node, n, remove it and mark node n and all justifications that n participates in. If any justification that have been marked is the current support of some node n1, recursively invoke step 2.

Step 3 : For all nodes that have been marked, check if there is any unmarked justification that supports it. If there is an unmarked justification that supports a marked node perform steps 4 and 5

else perform step 5.

Step 4 : The following steps are performed in step 4.

4.1 Make the unmarked justification the current support of the marked node and unmark the node.

4.2 Unmark all the justifications that the unmarked node participates in and check if any of the unmarked justifications can be used as a new current support for some other marked node.

4.3 If any unmarked justification can be used as a new current support of some other marked node then recursively perform step 4.

Step 5 : Delete all the justification and nodes that are marked.

4.2.2 A Prolog Implementation of the Reasoning Maintenance System

This section will present a detail implementation of the reasoning maintenance system in Prolog. The reasoning maintenance system is invoked each time when a fact or a rule is added to or deleted from the knowledge base. The main idea is to record the current state of dependencies of the existing set of rules and facts by using a directed graph to keep track of each step of the previous reasoning process.

When a fact or a rule is to be added to the knowledge base, the system will have to ensure that it does not contradict any existing beliefs in the knowledge base. If there is conflict between an incoming belief and some belief in the knowledge base, the user will have to decide whether to have the system revise the knowledge base and add the incoming fact or to ignore the new fact and keep the existing set of beliefs.

```
build_DDN(fact) :-

        get_symbol(_fact),
        add_fact(_fact).

build_DDN(rule) :-

        get_symbol(_premise -> _conclusion),
        add_rule(_premise -> _conclusion).

add_fact(_fact) :-

        conflict_recognition(fact, _fact),
        [status(contradiction, _) ->
                [retract(status(contradiction, _oldFact)),
                getln(_response,'would you like to revise the knowledge base?(y/n)'),
                [_response = y  -> [ restructure_DDN(fact,_oldFact),
                        perform_inference_with_new_fact(_fact)];
                        [nl,nl,write('fact not added to knowledge base')]]
                ];
                [perform_inference_with_new_fact(_fact)]
        ].

add_rule(_premise -> _conclusion) :-

        conflict_recognition(rule, _premise -> _conclusion),
        [status(contradiction, _) ->
                [retract(status(contradiction,_premise1 -> _conclusion1)),
                getln(_response,'would you like to revise the knowledge base?(y/n)'),
                [_response = y -> [restructure_DDN(rule,_premise1 -> _conclusion1),
                perform_inference_with_new_rule(_premise -> _conclusion)];
                [nl,nl,write('rule not added to knowledge base')] ]
```

```
            ];
            [perform_inference_with_new_rule(_premise -> _conclusion)]
    ].
```

When a new belief is added to the system, it will be used together with the existing set of beliefs to reason about new beliefs. Each time when a rule or fact is added to the knowledge base, the system will create a node representing the belief, a justification for this belief and an arc pointing from the justification node to the belief node.

```
perform_inference_with_new_fact(_fact):-

    [not(fact(_factId1, _fact)) ->
            [gen_sym(_justificationId, justification, justification_counter),
            gen_sym(_factId, fact, fact_counter),
            assert(justification(_justificationId, _factId)),
             assert(fact(_factId, _fact))]
    ],
    assert(status(fired)),
    inference.

perform_inference_with_new_rule(_premise -> _conclusion):-

    [not(domain_rule(_ruleId1, _premise -> _conclusion)) ->
            [gen_sym(_justificationId, justification, justification_counter),
            gen_sym(_ruleId, rule, rule_counter),
            assert(justification(_justificationId, _ruleId)),
            assert(domain_rule(_ruleId, _premise -> _conclusion))]
    ],
    assert(status(fired)),
    inference.
```

When a belief is removed from the knowledge base due to a contradiction or due to the change of constraints in the real world, the data dependency network must be restructured and all the derivation paths that are dependent on the deleted belief must be pruned. Also, the set of beliefs that has been used to derive the contradicting belief must be pruned.

```
restructure_DDN(fact,_fact) :-

    retract(fact(_factId,_fact)),
    prune_DDN(_factId),
    delete_derivation_leading_to_this_belief(_factId),!.

restructure_DDN(fact, _) :- nl,nl,write('fact do not exist').

restructure_DDN(rule, _premise -> _conclusion) :-

    retract(domain_rule(_ruleId,_premise -> _conclusion)),
```

```
        retract(justification(_justificationId, _ruleId)),
        prune_DDN(_ruleId),!.

restructure_DDN(rule, _) :- nl,nl,write('rule do not exist').
```

Pruning the data dependency network will consist of deleting the justification node of a belief, the belief node and the set of justification nodes which are pointed to by the belief node. The functions described here, operationalize the procedure of pruning a data dependency network as describe in the previous section.

```
prune_DDN(_factId) :-

        foreach(edge(_factId, _justificationId) do
                [remove_justification(_justificationId)]).
```

If the justification to be removed is the current support of some belief node , remove the justification, and mark the belief node and all other justification that the belief node participates in.

```
remove_justification(_justificationId) :-

        current_support(_justificationId, _factId) ->
                [retract(justification(_justificationId, _factId)),
                mark(_factId),
                check_for_unmark_justification,
                retract(current_support(_justificationId, _factId)),
                foreach(edge(_factOrRule, _justificationId) do
                        retract(edge(_factOrRule, _justificationId))),
                delete_all_mark_nodes_justifications

                ].
```

If a justification is to be removed and it is not the current support of any node, just remove it and exit.

```
remove_justification(_justificationId) :-

        foreach(edge(_factOrRule, _justificationId) do
                retract(edge(_factOrRule, _justificationId))),
                retract(justification(_justificationId, _Id)).

mark(_factOrRule) :-

        assert(mark_node(_factOrRule)),
        foreach(edge(_factOrRule, _justificationId) do
                [assert(mark_justification(_justificationId)),
                [current_support(_justificationId, _otherFactId) ->
                        mark(_otherFactId) ]
```

```
        ]).

check_for_unmark_justification :-

        foreach(mark_node(_nodeId) do
                [[justification(_justificationId,_nodeId),
                not(mark_justification(_justificationId))] ->
                unmark_node(_nodeId, _justificationId)
        ]).


unmark_node(_nodeId, _justificationId) :-

        assert(current_support(_justificationId, _nodeId)),
        retract(mark_node(_nodeId)),
        foreach(edge(_nodeId, _otherJustificationId) do
                [retract(mark_justification(_otherJustificationId)),
                [[justification(_otherJustification, _otherFactId),
                mark_node(_otherFactId),
                not(current_support(_otherJustification1, _otherFactId))]
                        -> unmark_node(_otherFactId, _otherJustificationId)]
        ]).

delete_all_mark_nodes_justifications :-

        foreach(mark_node(_nodeId) do
                [retract(mark_node(_nodeId)),
                 revise_knowledge_base(_nodeId, belief)
        ]),

        foreach(mark_justification(_justificationId) do
                [retract(mark_justification(_justificationId)),
                retract(justification(_justificationId,_node)),
                retract(edge(_factOrRuleId, _justificationId)),
                retract(current_support(_justificationId, _factOrRuleId1))
        ]).

delete_derivation_leading_to_this_belief(_factId) :-

        foreach(justification(_justificationId, _factId) do
                [ [current_support(_justificationId, _factId) ->
                        retract(current_support(_justificationId, _factId))],
        foreach(edge(_factOrRule, _justificationId) do
                [retract(edge(_factOrRule, _justificationId)),
                revise_knowledge_base(_factOrRule, derive)] ),
                [not(exist(_factId))->
                        retract(justification(_justificationId, _factId))]
        ]).
```

Inference : The following segment of code is used to derive new facts using the existing set of rules and facts in the knowledge base and any newly user-entered or system hypothesized rules or facts. When a new fact is derived from a particular rule and a set of

facts, the system will first ensure that the newly derived facts do not contradict any existing fact in the knowledge base. If a derived fact leads to a contradiction, the data dependency network will have to be revised.

Each derivation step in the inference process is recorded by creating a derivation path from the rule and the set of facts used to derived a fact. The system will create a new node to represent the justification for the derived fact and a node to represent the fact itself, and each fact and justification will be given an identification. The system will also establish the edges pointing from the rule and the set of facts that participate in the justification.

```
inference :-

        status(fired) -> eval.

eval() :-

        retract(status(fired)),
        foreach( [domain_rule(_ruleId, _antecedent ->_consequence),
             not(previously_derived(_ruleId, _consequence))]
        do
                [[verify(_antecedent),
                get_consequence(_ruleId,_antecedent,_conclusion)] ->
                [[conflict_recognition(fact, _conclusion),
                status(contradiction,_)] ->
                        [restructure_DDN(rule, _antecedent -> _consequence)];
                          [assert(status(fired)),
                          gen_sym(_justificationId, justification, justification_counter),
                          [not(fact(_oldFactId,_conclusion)) ->
                          [gen_sym(_factId, fact, fact_counter),
                          assert(fact(_factId, _conclusion)),
                          assert(justification(_justificationId, _factId)),
                          assert(current_support(_justificationId, _factId))];
                          [fact(_oldFactId,_conclusion),
                          assert(justification(_justificationId, _oldFactId))]],
                          assert(edge(_ruleId, _justificationId)),
                           build_link(_antecedent, _justificationId)
                         ]]
           ]).

verify([]).

verify([_antecedent,_antecedentList..]) :-

        fact(_factId,_antecedent),
        verify([_antecedentList..]).
```

```
build_link([], _).

build_link([_antecedent,_antecedentList..], _justificationId) :-

        fact(_factId,_antecedent),
        assert(edge(_factId, _justificationId)),
        build_link([_antecedentList..], _justificationId).

previously_derived(_ruleId, _consequence) :-

        fact(_factId, _consequence),
        justification(_justificationId, _factId),
        edge(_ruleId, _justificationId).

get_consequence(_ruleId,_antecedent,_pred(_argList1..)) :-

        domain_rule(_ruleId, _cond -> _pred(_argList..)),
        find_and_get_arg(_cond, _antecedent,[_argList..],[_argList1..]),!.

get_consequence(_ruleId,_antecedent,not(_pred(_argList1..))) :-

        domain_rule(_ruleId, _cond -> not(_pred(_argList..))),
        find_and_get_arg(_cond, _antecedent,[_argList..],[_argList1..]).

find_and_get_arg(_cond,_antecedent,[],[]).

find_and_get_arg(_cond, _antecedent, [_arg,_argList..], [_arg1,_argList1..]):-

        bind_vars(_arg),
        find_Var_arg(_arg, _cond, _nthPred, _nthArg),
        get_nth_arg(_arg1, _antecedent, _nthPred, _nthArg),
        find_and_get_arg(_cond, _antecedent, [_argList..], [_argList1..]).
```

Conflict recognition : The following segment of code is used to detect any contradiction between the beliefs in the knowledge base and new beliefs that are either derived from the existing set of beliefs or from beliefs that are asserted by the user or hypothesized by the system.

```
conflict_recognition(fact, _newFact) :-

        foreach(fact(_id, _fact) do
                [contradiction(_newFact, _fact) ->
                        [assert(status(contradiction, _fact)),
        nl,nl,
        write('contradicting facts : ',_newFact),nl,
        write('                      ',_fact)]]).

conflict_recognition(rule, _premise -> _conclusion) :-
```

```
        foreach(domain_rule(_id, _premise1 -> _conclusion1) do
                [ [same_premises(_premise, _premise1),
                contradiction(_conclusion, _conclusion1)] ->
                [assert(status(contradiction, _premise1 -> _conclusion1)),
                nl,nl,
                write('contradicting rules:',_premise -> _conclusion),nl,
                write('                    ',_premise1 -> _conclusion1)]]).

contradiction(_fact(_argList..), not(_fact(_argList..))).

contradiction(not(_fact(_argList..)), _fact(_argList..)).

same_premises(_premise1, _premise2) :-

        subset(_premise1, _premise2),
        subset(_premise2, _premise1).
```

Knowledge revision : When a contradiction arises due to an attempt to add a conflicting belief, the user will have to decide whether to discard the new belief or to revise the knowledge base and add the new belief.

```
revise_knowledge_base(_id, _beliefOrDereive) :-

        name(_id, _idCharList),
        reverse(_idCharList, [_char, _charList..]),
        reverse(_charList, _charList1),
        name(_id1, _charList1),
        purge_fact_or_rule(_id1, _id, _beliefOrDereive).

purge_fact_or_rule(_id1, _id, belief) :-

        _id1 = fact,
        retract(fact(_id, _fact)).

purge_fact_or_rule(_id1, _id, derive) :-

        _id1 = fact,
        retract(fact(_id, _fact)),
        delete_derivation_leading_to_this_belief(_Id).

purge_fact_or_rule(_id1, _id, _beliefOrDereive) :-

        _id1 = rule,
        retract(domain_rule(_id, _rule)).
```

4.3 Meta-Knowledge Acquisition Algorithm

The procedure for acquiring meta-predicates is presented here. An example of acquiring meta-predicates, meta-metapredicates, meta-facts and meta-metafacts is provided in chapter 5.

Step 1 : If the example rule can be generalized to an existing rule schema then we are done,

else

Perform step 1.1 to 1.4 to and step 2.1 to step 2.4 to generalize the example rule into a rule-template or rule schema.

Step 1.1 : Concatenate the conclusion part and condition part of the rules into a list.

Step 1.2 : Scan the list to collect all the argument constants. Each of the argument constants in the list, (except specially tagged constants which are not to be generalized) is turned into a unique variable.

Step 1.3 : Scan the same list (which now consists of unique argument variables) again. This time the corresponding constant term in each argument place (except specially tagged constants) is matched with all other terms in the other argument places. All equal terms are generalized to the same argument variable.

Step 1.4 : The resulting list of facts consisting of generalized arguments are decomposed to obtain the generalized conclusion part and the generalized condition part. This step will also compose a new domain rule consisting of only predicate constants and argument variables.

Step 2 : Generalize the predicate constants in the generated domain rule.

Step 2.1 : Step 2.1 is similar to step 1.1, that is to form a list consisting of the conclusion and condition part.

Step 2.2 : Each predicate constant in this list, except arithmetic predicates is generalized to a unique predicate variable.

Step 2.3 : Using similar approach as in step 1.3, each predicate is matched with all other predicates in the list and predicates having the same predicate name are generalized to the same predicate variable.

Step 2.4 : This step is quite similar to step 1.4, except that in this case a rule schema and its corresponding meta-fact is generated. The user will be prompted to give a name to describe the relation that is defined in this newly acquire meta-predicate definition.

4.3.1 Implementing the Meta-knowledge Acquisition in Prolog

The following segment of code operationalizes the generalization process for acquiring meta-knowledge. To prevent redundant meta-knowledge, the system will first try to instantiate the user entered example rule with existing meta-predicates. If the example rule cannot be describe by any meta-predicate definition the system will then generalize this rule to hypothesize about a new predicate definition.

```
acquire_meta_knowledge(_meta_knowledge) :-

        get_symbol(_exampleRule),
        [not(exist_meta_predicate(_exampleRule)) ->
        generalize(_meta_knowledge,_exampleRule,_generalizedRule);
        [nl,write('rulescheme already exist')]].
```

This segment of code performs the generalization of the example in two steps, by turning all the argument constants to argument variables and by the turning the predicate constants to predicate variables.

```
generalize(_meta_knowledge, _Conditions -> _Conclusion,
   _Var_Pred_conditions->_Var_Pred_conclusion) :-

        generalize_argument(_Conditions -> _Conclusion,
                _Var_Arg_conditions- > _Var_Arg_conclusion),
        generalize_predicate(_Var_Arg_conditions ->_Var_Arg_conclusion,
                _Var_Pred_conditions ->_Var_Pred_conclusion,
                [_Var_Pred_Set..],[_Pred_Set..]),
        findset(_X, special_const(_X), [_spec_const_list..]),
        add_rule(_meta_knowledge, _Var_Arg_conditions ->_Var_Arg_conclusion),
        get_predicate_definition(_meta_knowledge,[_Var_Pred_Set..],
                [_spec_const_list..], [_Pred_Set..],
                _Var_Pred_conditions ->_Var_Pred_conclusion).
```

```
add_rule(meta_pred, _Var_Arg_conditions->_Var_Arg_conclusion) :-

        assert(domain_rule(_Var_Arg_conditions->_Var_Arg_conclusion)),
        nl,nl,nl,
        write('the following domain rule is added to the knowledge base'),
        nl,nl,
        listing(domain_rule).

add_rule(meta_metapred, _Var_Arg_conditions->_Var_Arg_conclusion) :-

        assert(meta_rule(_Var_Arg_conditions->_Var_Arg_conclusion)),
        nl,nl,nl,
        write('the following meta-rule is added to the knowledge base'),
        nl,nl,
        listing(meta_rule).

get_predicate_definition(_meta_knowledge,[_Var_Pred_Set..],[_spec_const_list..],
        [_Pred_Set..],_Var_Pred_conditions ->_Var_Pred_conclusion) :-

        getln(_meta_pred_name,'enter meta-predicate name. <enter>'),
        append(_Var_Pred_Set , _spec_const_list, [_Var_Predicate..]),
        [special_const(_c) ->
                [append(_Pred_Set, [_c],[_m_argument..]),
                assert(_meta_pred_name(_m_argument..))];
                assert(_meta_pred_name(_Pred_Set..))],
                assert(_meta_pred_name(_Var_Predicate..)
                where rulescheme(_Var_Pred_conditions->_Var_Pred_conclusion)),
        nl,nl,nl,
        write('the following meta-predicate definition has been hypothesize'),
        nl,nl,
        listing(where),
        write('the following meta-fact has been hypothesize'),
        nl,nl,
        listing(_meta_pred_name),
        nl,nl,
        add_metapredicate_definition(_meta_knowledge, _meta_pred_name).

add_metapredicate_definition(meta_pred, _meta_pred_name):-

        assert(meta_predicate(_meta_pred_name)).

add_metapredicate_definition(meta_metapred, _meta_pred_name):-

        assert(meta_metapredicate(_meta_pred_name)).
```

The generalization of the argument constants in an example rule is carried out in two passes. In the first pass each constant is generalized to a unique variable and in the second pass the argument place holder having the same value will be generalized to the same variable.

```
generalize_argument(_Conditions -> _Conclusion,
        _Var_conditions->_Var_conclusion) :-

        generalize_arg_list([_Conclusion,_Conditions..],
                [_Var_conclusion,_Var_conditions..]).

generalize_arg_list(_Pred_List, _Var_Pred_List) :-

        generalize_arg_list_1st_pass(_Pred_List, _Var_Pred_List),
        generalize_arg_list_2nd_pass(_Pred_List, _Var_Pred_List).

generalize_arg_list_1st_pass([], []).

generalize_arg_list_1st_pass([not(_Pred(_Arg1..)),_Cond_List..],
        [not(_Pred(_Var1..)),_Arg_var_list..]) :-

         var_list([_Arg1..],[_Var1..]),
        generalize_arg_list_1st_pass(_Cond_List, _Arg_var_list),!.

generalize_arg_list_1st_pass([_Pred(_Arg1..),_Cond_List..],
        [_Pred(_Var1..),_Arg_var_list..]) :-

         var_list([_Arg1..],[_Var1..]),
        generalize_arg_list_1st_pass(_Cond_List, _Arg_var_list).
```

The following function will generalize each argument constant to a unique variable and scan the argument list to detect a special constant that is not to be generalized. An argument is not generalized if it is preceded by a '$' sign.

```
var_list([],[]).

var_list([_Arg,_Arg_List..],[_Var,_Var_List..]) :-

        [name(_Arg, [36,_Rest..]) -> [name(_Var,[_Rest..]),
        assertz(special_const(_Var))]],
        var_list(_Arg_List, _Var_List).

generalize_arg_conditions_2nd_pass([], []).

generalize_arg_conditions_2nd_pass([not(_Pred(_Arg1..)),_Cond_List..],
        [not(_Pred(_Var1..)),_Arg_var_list..]) :-

        var_list_2nd_pass([_Arg1..],[_Var1..], _Cond_List, _Arg_var_list),
        generalize_arg_conditions_2nd_pass(_Cond_List, _Arg_var_list),!.

generalize_arg_conditions_2nd_pass([_Pred(_Arg1..),_Cond_List..],
        [_Pred(_Var1..),_Arg_var_list..]) :-

        var_list_2nd_pass([_Arg1..],[_Var1..], _Cond_List, _Arg_var_list),
        generalize_arg_conditions_2nd_pass(_Cond_List, _Arg_var_list).
```

```
var_list_2nd_pass([],[],_,_).

var_list_2nd_pass([_Arg,_Arg_List..],[_Var,_Var_List..],_Cons_Arg_List,
        _Var_Arg_List) :-

        [find_arg(_Arg,_Cons_Arg_List, _Nth1, _Nth2)] ->
        [once(get_nth_arg(_Var, _Var_Arg_List, _Nth1, _Nth2))],
        var_list_2nd_pass(_Arg_List, _Var_List,_Cons_Arg_List, _Var_Arg_List).

find_arg(_A, [not(_(_Arg_List..)), _..],1, _Nth) :-

        find_ele(_A,[_Arg_List..],_Nth),!.

find_arg(_A, [_(_Arg_List..), _..],1, _Nth) :-

        find_ele(_A,[_Arg_List..],_Nth).

find_arg(_A, [_, Ys..],_Nth, _Nth2) :-

        find_arg(_A, Ys,_Nth_1, _Nth2),
        _Nth = _Nth_1 + 1.

get_nth_arg(_V, [not(_(_Var_List..)), _..],1, _Nth) :-

        get_nth_ele(_V, [_Var_List..], _Nth),!.

get_nth_arg(_V, [_(_Var_List..), _..],1, _Nth) :-

         get_nth_ele(_V, [_Var_List..], _Nth).

get_nth_arg(_X2, [X1, Ys..],_Nth,_Nth2) :-

        _Nth_1 is _Nth -1,
        get_nth_arg(_X2, Ys,_Nth_1,_Nth2).

find_ele(X, [X, _..],1).

find_ele(X, [_, Ys..],_Nth) :-
        find_ele(X, Ys,_Nth_1),
        _Nth = _Nth_1 + 1.

get_nth_ele(X, [X, _..],1).

get_nth_ele(_X2, [X1, Ys..],_Nth) :-
        _Nth_1 is _Nth -1,
        get_nth_ele(_X2, Ys,_Nth_1).
```

The following segment of code performs the generalization of the predicate constants to predicate variables in two passes just as in the generalization of the arguments. The same technique used for generalizing the arguments described above is also used for

generalizing predicate constants. Special arithmetic predicates such as less-than, and equal are not generalized.

```
generalize_predicate(_Conditions -> _Conclusion, _Var_conditions->_Var_conclusion,
            [_Var_Pred_Set..],[_Pred_Set..]) :-

        generalize_condition_predicates([_Conclusion,_Conditions..],
              [_Var_conclusion, _Var_conditions..],[_Var_Pred_Set..],[_Pred_Set..]).

generalize_condition_predicates(_Pred_List, _Var_Pred_List,
        [_Var_Pred_Set..],[_Pred_Set..]) :-

        generalize_predicate_1st_pass(_Pred_List, _Var_Pred_List, _Pred_Name),
        generalize_predicate_2nd_pass(_Pred_List, _Var_Pred_List, _Var_Pred_Name),
        find_predicate_set(_Pred_Name, [_Pred_Set..]),
        find_var_pred_set([_Pred_Set..], [_Var_Pred_Set..],_Pred_Name,
        _Var_Pred_Name).

generalize_predicate_1st_pass([], [], []).

generalize_predicate_1st_pass([not(_Pred(_Arg1..)),_Cond_List..],
                    [not(_Var_Pred(_Arg1..)),_Arg_var_list..], [_Pred,_Pred_Name..]) :-

        [exception_predicate(_Pred) -> _Var_Pred = _Pred],
        generalize_predicate_1st_pass(_Cond_List,_Arg_var_list,_Pred_Name),!.

generalize_predicate_1st_pass([_Pred(_Arg1..),_Cond_List..],
        [_Var_Pred(_Arg1..),_Arg_var_list..],[_Pred,_Pred_Name..]) :-

        [special_predicate(_Pred) -> _Var_Pred = _Pred],
        generalize_predicate_1st_pass(_Cond_List,_Arg_var_list,_Pred_Name).

generalize_predicate_2nd_pass([],[],[]).

generalize_predicate_2nd_pass([not(_Pred(_Arg1..)),_Cond_List..],
        [not(_Var_Pred(_Arg1..)),_Arg_var_list..],[_Var_Pred,_Var_Pred_Name..]) :-

        [(find_element(_Pred(_Arg1..), _Cond_List,_Nth)) ->
         once(get_nth_element(_Var_Pred(_Arg1..), _Arg_var_list,_Nth))],
        generalize_predicate_2nd_pass(_Cond_List,_Arg_var_list,_Var_Pred_Name),!.

generalize_predicate_2nd_pass([_Pred(_Arg1..),_Cond_List..],
              [_Var_Pred(_Arg1..),_Arg_var_list..],[_Var_Pred,_Var_Pred_Name..]) :-

        [(find_element(_Pred(_Arg1..), _Cond_List,_Nth)) ->
         once(get_nth_element(_Var_Pred(_Arg1..), _Arg_var_list,_Nth))],
        generalize_predicate_2nd_pass(_Cond_List,_Arg_var_list,_Var_Pred_Name).
```

```
find_predicate_set(_Pred_List, _Pred_Set) :-

   reverse(_Pred_List, _Rev_Plist),
   find_pred_set(_Rev_Plist,[],_Rev_Pset),
   reverse(_Rev_Pset, _Pred_Set).

find_pred_set([],L,L).

find_pred_set([_Pred,_Pred_List..],_L,_Pred_Set) :-

        [not(not([bind_vars(_Pred),
        bind_vars_list(_Pred_List),
        member(_Pred,_Pred_List)]));
        exception_predicate(_Pred)],
        find_pred_set(_Pred_List,_L,_Pred_Set),!.

find_pred_set([_Pred,_Pred_List..],_L,[_Pred,_Pred_Set..]) :-
        find_pred_set([_Pred_List..],_L,[_Pred_Set..]).

bind_vars_list([]).

bind_vars_list([_V,_Var_list..]) :-
   bind_vars(_V),
   bind_vars_list(_Var_list).

find_var_pred_set([],[], _,_).

find_var_pred_set([_Pred,_Cond_List..],
        [_Var_Pred,_Arg_var_list..], _Pred_Name, _Var_Pred_Name) :-

        exception_predicate(_Pred),
        find_var_pred_set(_Cond_List,_Arg_var_list,_Pred_Name, _Var_Pred_Name).

find_var_pred_set([_Pred,_Cond_List..],
        [_Var_Pred,_Arg_var_list..], _Pred_Name, _Var_Pred_Name) :-

        [(find_ele(_Pred, _Pred_Name,_Nth)) ->
        once(get_nth_ele(_Var_Pred, _Var_Pred_Name, _Nth))],
        find_var_pred_set(_Cond_List,_Arg_var_list,_Pred_Name, _Var_Pred_Name).

find_element(X(_..), [not(X(_..)), _..],1) :- !.

find_element(X(_..), [X(_..), _..],1) :-!.

find_element(X(_..), [_, Ys..],_Nth) :-

        find_element(X(_..), Ys,_Nth_1),
        _Nth = _Nth_1 + 1.

get_nth_element(_X(_..), [_X1(_..), _..],1) :-

        [_X1 @= not] -> fail;[_X1 = _X,!].
```

```
get_nth_element(X(_V..), [not(X(_V1..)), _..],1) :-!.

get_nth_element(_X2(_..), [X1(_..), Ys..],_Nth) :-

        _Nth_1 is _Nth -1,
        get_nth_element(_X2(_..), Ys,_Nth_1).
```

4.4 Summary

In this chapter, the procedures for implementing the sort classifier, the reasoning maintenance system and the meta-knowledge acquisition module have been discussed. The Prolog codes for this module were also presented. The purpose of this chapter has been to present detail of operationalizing the design principles of KLOKE which were presented in chapter 3. In the next chapter, the testing and discussion of each of the three modules mentioned above will be presented.

Chapter 5
System Evaluation

This chapter will provide a test example for building a sort taxonomy, an example to delete a node from a data dependency network and an example for acquiring meta-knowledge. A discussion on evaluating the sort classifier, the reasoning maintenance system and the meta-knowledge acquisition module are also presented.

5.1 An Example of Constructing a Sort Taxonomy

Using the set of facts shown below, this section will trace through the construction of the sort taxonomy at selected stages shown in figure 3.4 in chapter 3, page 44:

indicate(sore_throat, flu)	cause(flu, sore_throat)
affect_pos(inspirol, sore_throat)	affect_pos(aspirin, flu)
affect_pos(bc, sore_throat)	affect_pos(asa, flu)
contains(inspirol, bc)	contains(aspirin, asa)
suck(willi, inspirol)	suck(uwe, vivil)

The user interface for KLOKE has not been implemented, so the sort taxonomy at each stage is in the form of PROLOG predicates describing the relations (subclass, superclass and intersection) between sorts and the extension of sorts. However, diagrams have been included to show the state of the sort taxonomy at each stage.

Step 1. When the fact indicate(sore_throat, flu) is entered, the sort taxonomy shown in figure 5.1 is built. Its corresponding predicate representation is :

class($all, [$all], _).	class(disease, [flu], [indicate1]).
class(symptom,[sore_throat],[indicate0])	class($nil, [], []).
sub_class(symptom, $all)	sub_class(disease, $all).
sub_class(symptom, $nil)	sub_class(disease, $nil).

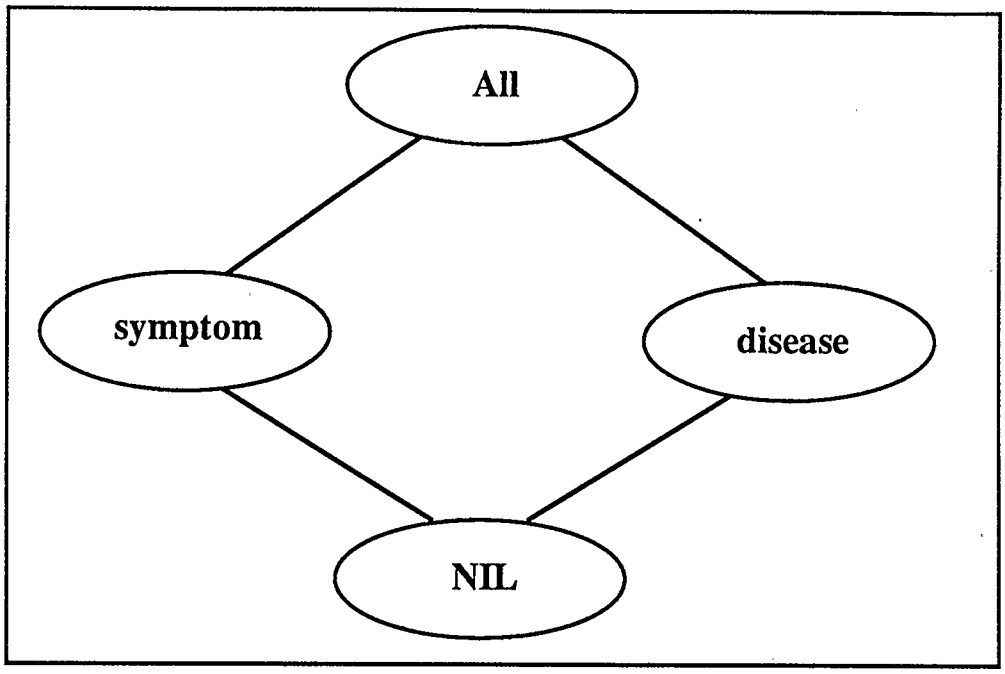

Fig 5.1 Resulting Sort Taxonomy After Step 1

When the fact cause(flu, sore_throat) is entered the state of the taxonomy still remain unchanged, but there are changes to the list of argument sorts 'symptom' and 'disease' represented as:

class(symptom, [sore_throat], [indicate0, cause1]).
class(disease, [flu], [indicate1, cause0]).

Step 2. When the following set of facts are entered :

affect_pos(inspirol, sore_throat) affect_pos(aspirin, flu)
affect_pos(bc, sore_throat) affect_pos(asa, flu)

the resulting sort taxonomy is shown in figure 5.2 and the predicate representations are :

class($all, [$all], _).
class(substance, [inspirol, aspirin, bc, asa], [affect_pos0]).
class(irritation, [sore_throat, flu], [affect_pos1]).
class(symptom, [sore_throat], [indicate0, cause1]).
class(disease, [flu], [indicate1, cause0]).
class($nil, [], []).

sub_class(substance, $all). sub_class(irritation, $all).
sub_class(symptom, irritation). sub_class(disease, irritation).

sub_class(nil, disease). sub_class(nil, symptom).
sub_class(nil, substance).

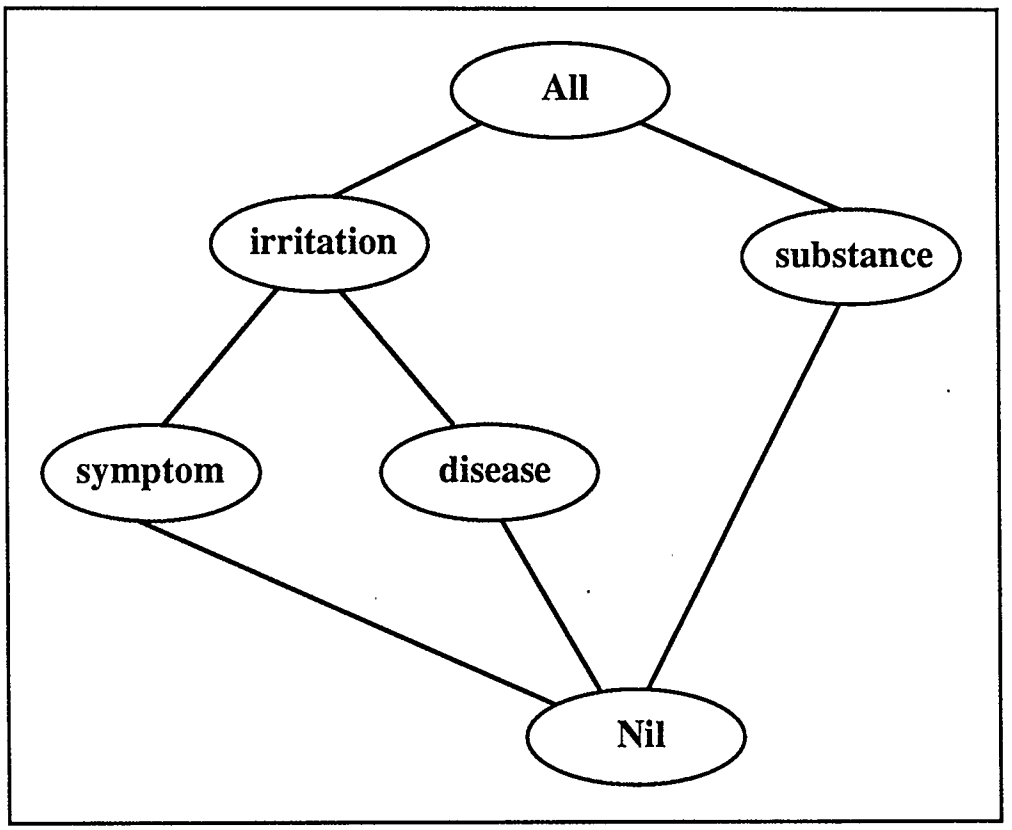

Fig 5.2 Resulting Sort Taxonomy After Step 2

Step 3. When the following facts are entered :

contains(inspirol, bc)
contains(aspirin, asa)

the resulting taxonomy are shown in figure 5.3 and the corresponding predicate representations are shown below:

class(act_agent, [bc, asa], [contains1]).

class(drug, [inspirol, aspirin], [contains0]).

class(substance, [inspirol, aspirin, bc, asa], [affect_pos0]).

class(irritation, [sore_throat, flu], [affect_pos1]).

class(symptom, [sore_throat], [indicate0, cause1]).

class(disease, [flu], [indicate1, cause0]).

class(nil, [], []).

class($all, [$all], _).

sub_class(irritation, $all).

sub_class(substance, $all)

sub_class(disease, irritation)

sub_class(symptom, irritation).

sub_class(act_agent, substance).

sub_class(drug, substance).

sub_class(nil, disease)

sub_class(nil, symptom)

sub_class(nil, act_agent).

sub_class(nil, drug)

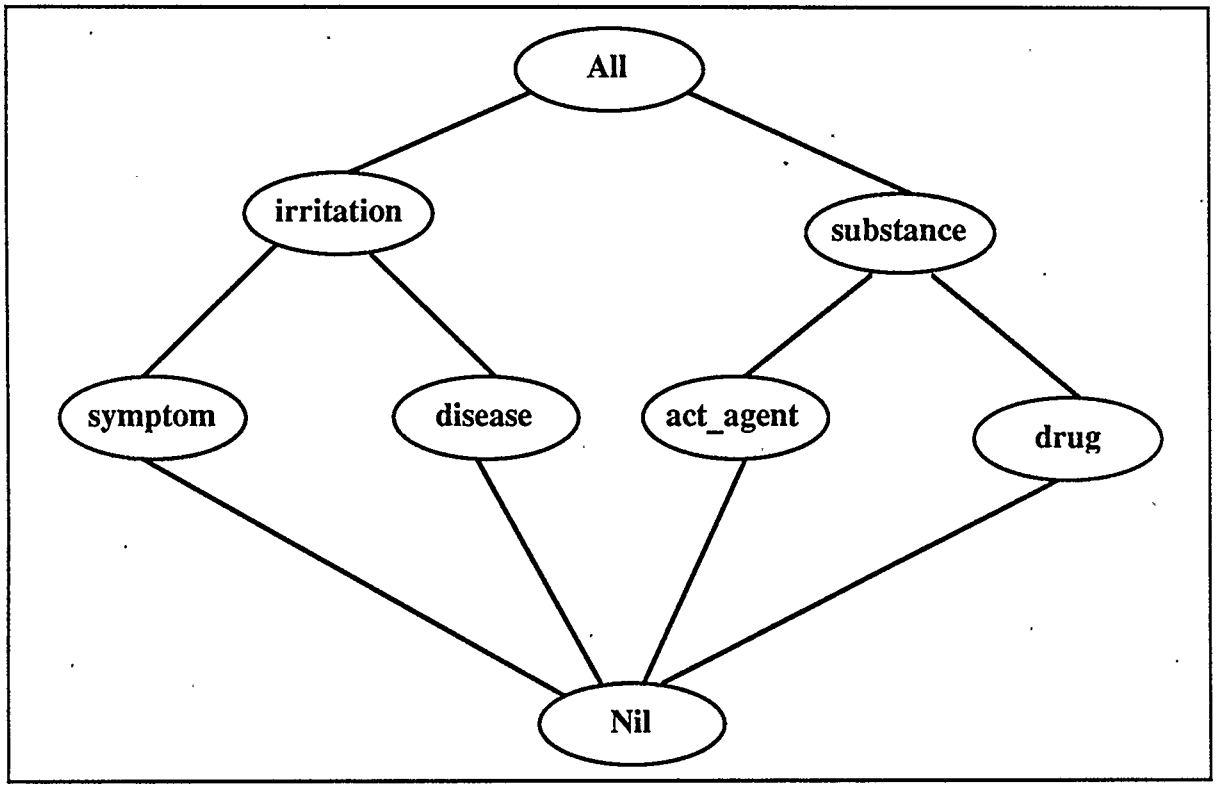

Fig. 5.3 Resulting Sort Taxonomy After Step 3

Step 4. When the following facts are entered :

suck(willi, inspirol)

suck(uwe, vivil)

the resulting sort taxonomy is shown in figure 3.4 in chapter 3, page 44. The corresponding predicate representation of the sort taxonomy is shown below.

class($all, [$all], _).

class(pill, [inspirol], [int(class10, class7), int(class10, class4)]).

class(dragee, [vivil, inspirol], [suck1]).

class(person, [uwe, willi], [suck0]).

class(act_agent, [bc, asa], [contains1]).

class(drug, [inspirol, aspirin], [contains0]).

class(substance, [inspirol, aspirin, bc, asa], [affect_pos0]).

class(irritation, [sore_throat, flu], [affect_pos1])

class(symptom, [sore_throat], [indicate0, cause1])

class(disease, [flu], [indicate1, cause0]).

class($nil, [], []).

sub_class(dragee, $all). sub_class(person, $all).

sub_class(substance, $all). sub_class(irritation, $all).

sub_class(pill, drug). sub_class(pill, dragee).

sub_class(act_agent, substance). sub_class(drug, substance).

sub_class(disease, irritation). sub_class(symptom, irritation).

sub_class($nil, disease). sub_class($nil, symptom)

sub_class($nil, person). sub_class($nil, act_agent).

sub_class($nil, pill).

int(dragee, substance). int(dragee, drug).

The predicate 'int' represents intersection of two classes.

5.2 Deleting a Justification

Using the data dependency network given in figure 4.4 of chapter 4, page 80, this section will trace through the deletion steps.

Step 1 : If we were to delete the justification, J3, fact, F0 will also be deleted. The resulting data dependency network after step 1 is shown in figure 5.4.

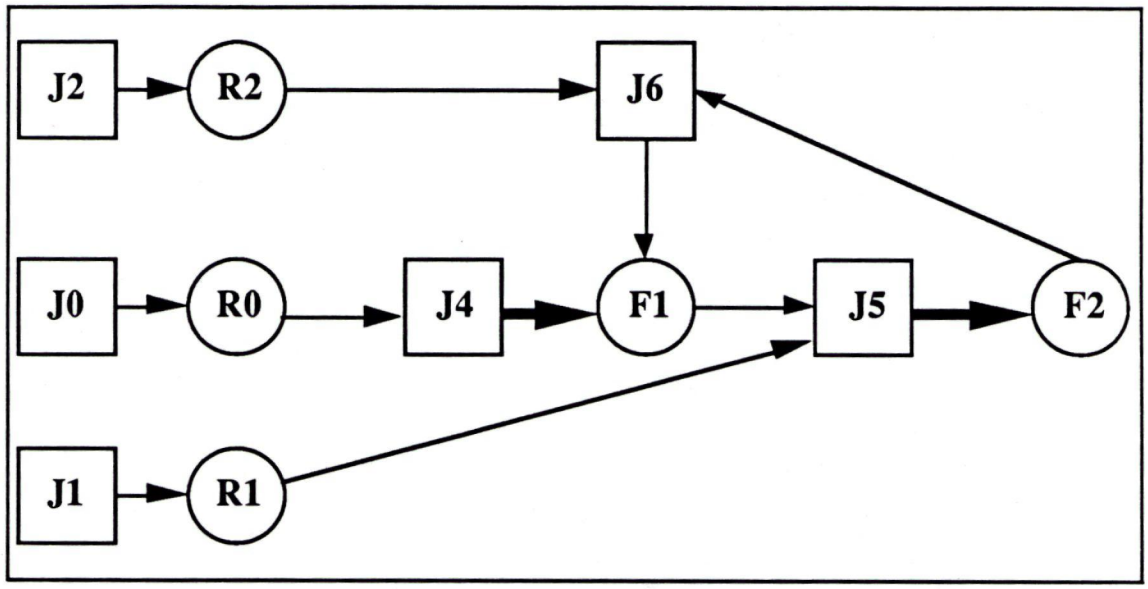

Fig 5.4 Resulting Data Dependency Network After Step 1

Step 2 : Since F0 and R0 participate in J4 , following step 1, J4 is no longer justified and must be marked, this will cause step 2 to be applied recursively. The resulting data dependency network after the first call to step 2 is shown in figure 5.5.

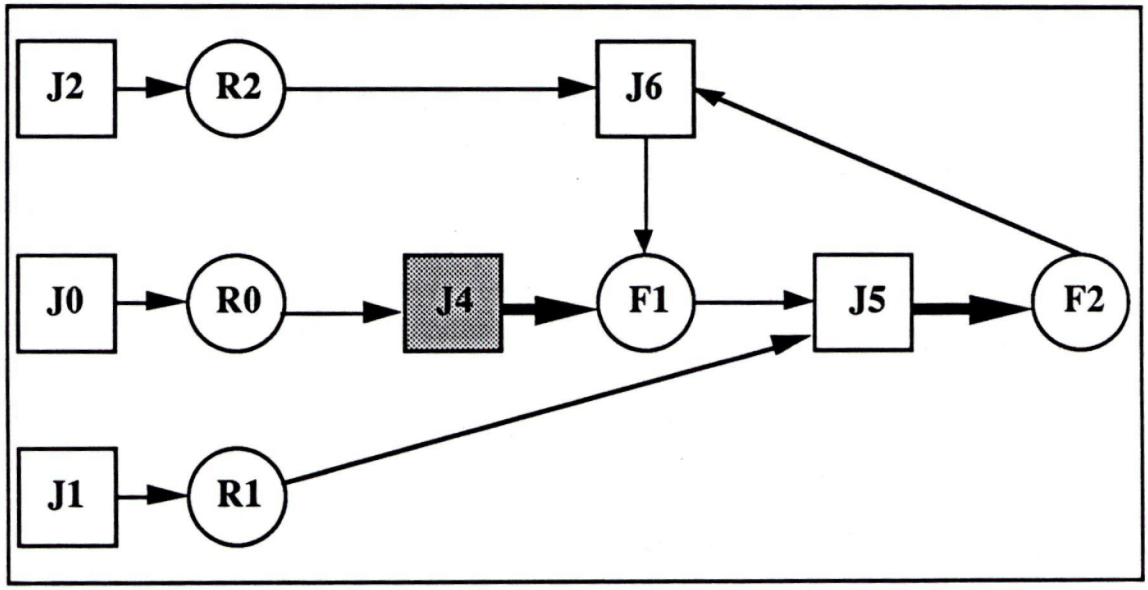

Fig 5.5 Resulting Data Dependency Network After Step 2

Step 2a (first recursive call) : The first recursive call to step 2 will cause F1 and J5 to be marked. The result is shown in figure 5.6. This will also lead to another recursive call of step 2.

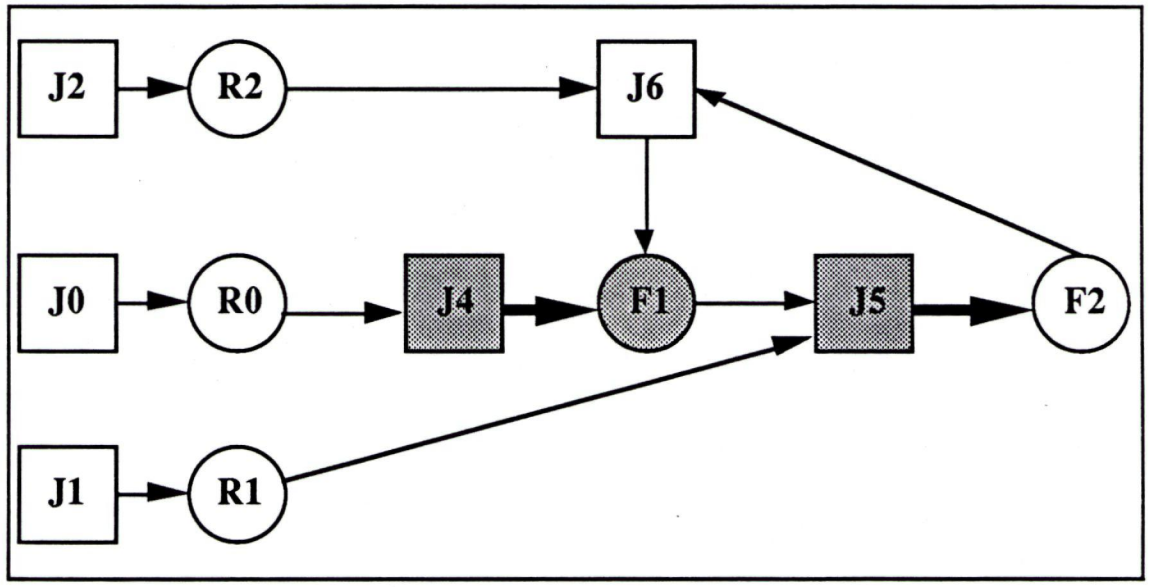

Fig 5.6 Resulting Data Dependency Network After Step 2a

Step 2b (second recursive call) : In the second recursive call to step two, F2 and J6 are marked. The result are shown in Figure 5.7. Since J6 is not the current support of any node, the recursion will terminate here.

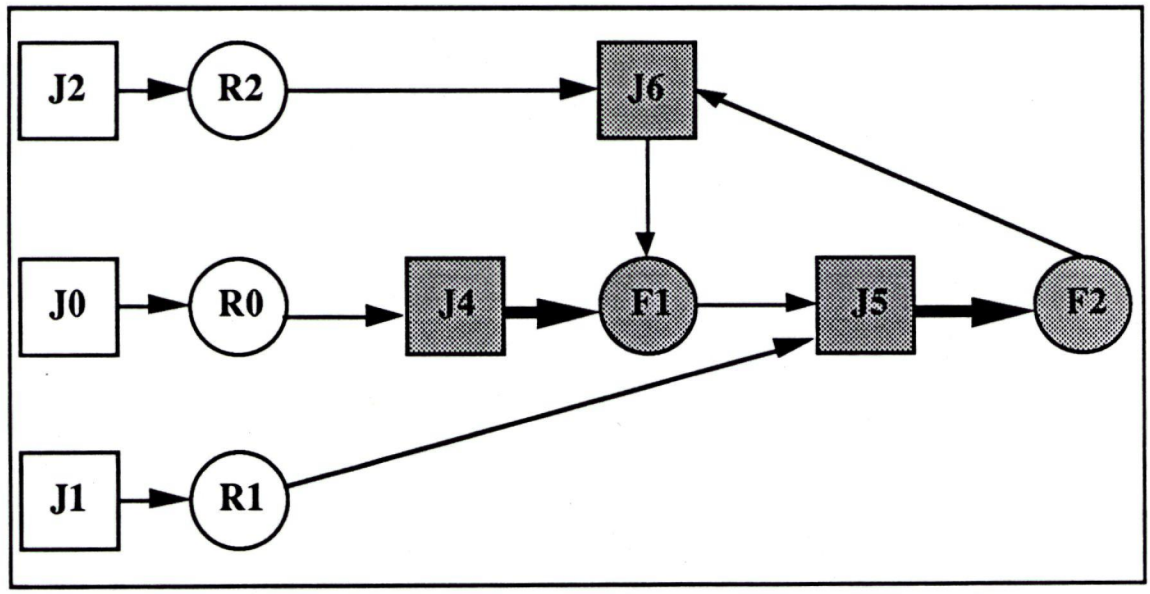

Fig 5.7 Resulting Data Dependency Network After Step 2b

Step 3 : Attempt to restablish current support fails.

Step 4 : All the marked nodes and justifications remain marked.

Step 5 : All the marked nodes and justifications are purged resulting in the data dependency network of figure 5.8 leaving only rule R2, R0 and R1.

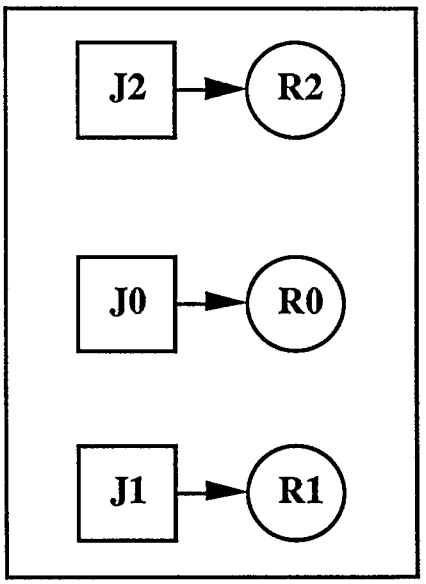

Fig 5.8 Resulting Data Dependency Network After Step 5

5.3 An Example of Acquiring Meta-knowledge

An example for acquiring meta-predicates is illustrated here. Given the training example shown below, the '$' sign is used to tag a special constant which describes a threshold.

[age(fred,19), eq(19,$19)] -> adult(fred)

Step 1.1 : The result of step 1 is the following list :

[adult(fred), age(fred,19), eq(19,$19)]

Step 1.2 : The result of this step is : [adult(_x1), age(_x2,_x3), eq(_x4,$19)], where an argument variable is preceded by an underscore. The constant '$19' is left ungeneralized.

Step 1.3 : The result of this step is [adult(_x1), age(_x1,_x3), eq(_x3,$19)]

Step 1.4 : The result of step 1.4 is a domain rule :

domain_rule([age(_x1, _x3), eq(_x3, '19')] -> adult(_x1))

Step 2.1 : The list that is obtained in step 1.3 is used for processing in the new few steps.

Step 2.2 : The system maintains a list of arithmetic predicate names such as lt (less than), greater (gt than) and eq (equal). In this step except for the arithmetic predicate 'eq', the other predicates are generalized. Again an underscore precedes a generalize predicate variable. The result of this step is :

[_p1(_x1), _p2(_x1,_x3), eq(_x3,$19)]

Step 2.3 : Since there is no identical predicate name in the list the result remains the same as in the previous step.

Step 2.4 : If the user enters 'threshold' as the relation name then the following is formed:

meta-fact : threshold(adult, age, '19').

meta-predicate definition : threshold(_p1, _p2, '19') where
rule scheme(_p2(_x1 _x3), eq(_x3, '19')] -> _p1(_x1)).

5.4 Discussing the Sort Classifier

This section will discuss the sort classifier, and make comparisons to the classifier in the KL-ONE system.

Deriving the Properties of the Sort Taxonomy : It has been proven that the sort lattice (taxonomy) which is constructed from the sort information contained within the set of given facts will represent the sort relations mentioned above since there exists a bijective extension mapping between the constructed lattice and a *complete infimum-homomorph lattice* of the lattice of the *power set* of the universal set (the set of all terms). From this mapping it is reasonable to transport the valid laws of boolean algebra (union, intersection, complement, disjunction) with partial ordering into the constructed lattice. The partial ordering (subset relation), the infimum operation (intersection) and the supremum operation (union), all can be transported from the extension into the sort lattice (taxonomy).

Comparing the Sort Taxonomy in KLOKE and KL-ONE : Kietz (1988) mentioned that it is possible to translate a sort taxonomy acquired using this method into a primitive KL-ONE taxonomy. The sort taxonomy in KLOKE is essentially a simplification of the KL-ONE concept taxonomy. The sort taxonomy in KLOKE and the concept taxonomy in KL-ONE are lattices that are based on subsumptions or inclusion partial ordering of sorts. Each of the sorts in the KLOKE taxonomy corresponds to a primitive concept in the KL-ONE concept taxonomy. In both KL-ONE and KLOKE all the terms that belong to a primitive concept or sort must be given explicitly, that is there is no intensional characterization in either system. In the classifier sub-system of KLOKE, the declaration of argument sorts will impose a restricted set of admissible terms for the corresponding argument place. This resembles a similar restriction in KL-ONE, where the domain and range definition of concepts will restrict the set of permissible role fillers for roles in the KL-ONE system.

Design Features of the Sort Classifier : The other design features of this approach are that of reversibility and incrementality. The system will automatically revise the taxonomy at all stages of the modelling when new facts are added to the knowledge base. At any stage during the modelling the user can revise the set of facts in the knowledge base, the classifier will automatically compute any changes in the sort taxonomy due to

the revision. Another point to note is that the sequence of entering the set of facts into the knowledge base does not affect the eventual result of the taxonomy that is constructed. Using the example given above, if we reverse steps 1 and 2 the resulting taxonomy that is constructed is the same as that shown in figure 5.1.

Semantic Limitation of the Classifier : One of the requirements that is imposed by the classifier is that the user is expected to enter facts that are represented by well-formed expressions. Secondly, using this approach for acquiring taxonomies, the system has no notion as to whether the facts that are entered are true or false, that is there is no recognition of the semantics of these facts. The following example will illustrate this point. Given the following set of facts :

temperature(37) age(42)
age(37) temperature(42)

The arguments appearing in both the predicates 'temperature' and 'age' are supposed to represent the measure of heat energy and the chronological age of a person respectively, that is we expect the system to declare the two predicates as temperature(<degrees>) and predicate age(<years>). However, as the classifications of sorts is done by mapping the extension in the arguments in the predicates, the system is not aware that these same extension sets are supposed to be categorized with different sorts. Since both the extension sets of the argument place of 'temperature' and 'age' are the same, that is [37,42], the system will establishe an equivalence class relation between the two, even though in reality the user does not want both the sorts to be equivalent.

5.5 Discussing the Reasoning Maintenance System in KLOKE

This section will discuss the reasoning maintenance system and provide some justification for the design of the system.

Truth Maintenance and Knowledge Revision : Fundamental practice in machine learning mainly focused on the problem of acquiring 'correct' rules (Kodratoff 1988b). This is insufficient in cooperative balanced modelling systems such as KLOKE where consistency is an important factor. The issue of ensuring global coherence of the system, that is *truth maintenance,* must also be considered. Consideration must also be given to additional issues such as 'intelligent' structuring of knowledge and filtering out

'dangerous' rules. This is the problem of detecting contradictions in the knowledge base and rectifying them by restructuring the knowledge base with the minimum changes from various options using meta-knowledge or heuristics.

Incorporating The Inference Mechanism : The problem of propagating beliefs is to be able to identify beliefs that become questionable when the premises from which they have been derived are disbelieved. This will require the reasoning maintenance system to keep track of all derivations that have taken place. That is, to solve the belief problem the reasoning maintenance system has to know how the belief has been derived. Shanahan (1989) mentioned two principle solutions, the first is to integrate the inference mechanism into the reasoning maintenance system so that each step of the reasoning process can be recorded. This method is used in the reasoning maintenance system of KLOKE. The second method is to provide some form of communication between an external inference system and the reasoning maintenance system. This method was used in RUP (McAllester 1982).

Justification-Based RMS vs Assumption-Based RMS : The assumption based reasoning maintenance system offers an improvement over a conventional reasoning maintenance system for search problems where all or many solutions are required. Since the assumption based reasoning maintenance system explores all paths in parallel, backtracking is eliminated. When only one or a few solutions are required, the conventional justification reasoning maintenance system is more efficient (Shanahan 1988). For a system where only a small part of the search space is explored, the assumption based reasoning maintenance system will be an inefficient solution as it will be computationally expensive to explore all possible paths for practical large scale knowledge bases. Thus, a justification based reasoning maintenance system will suffice in the case of KLOKE. In contrast to the inference engine IM-2 of the BLIP system, the reasoning maintenance system of KLOKE builds a monotonic data dependency network and it does not support the storing of contradicting knowledge in the system.

5.6 Discussing the Meta-Knowledge Acquisition Module

This section will discuss the module for acquiring meta-knowledge. The approach of generalizing rules and the proving of the correctness of meta-knowledge will be briefly discussed here.

Generalization Approach : There are two ways to generalize a rule (Kodratoff 1988b). One method is to turn all its constant terms into variables, which is used to learn predicate logic (first order and higher order). This method is employed in KLOKE for acquiring meta-knowledge from an example rule given by the by user. The other method is to climb the taxonomy or the generalization tree. DISCIPLE uses a combination of the two approaches. During its first stage of learning the first method is used to turn all the constants in a training example to generate an explanation. The third stage employs the method of climbing up the generalization tree.

As a discussion, I will present a negative example of using the climbing taxonomy taken from Kodratoff (1988b). For example, given the taxonomies and rules shown in figure 5.9, P cannot be inferred from Q by using the rules and climbing the taxonomies. If P is substituted with suppressor, P1 with suppressor-1, P2 with suppressor-2, Q with cable and so on, a problem will arise.

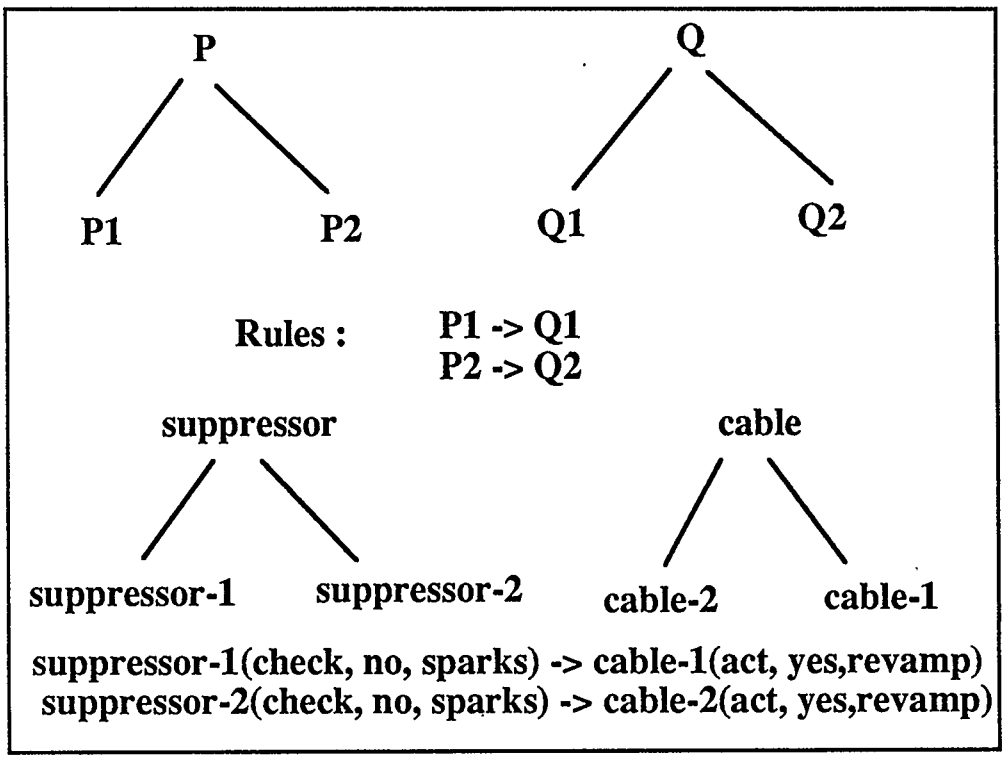

Fig 5.9 Generalization Taxonomies

Although the knowledge representation of these rules appears to be inappropriate for practical application, they are merely used here for illustration purposes. Semantically,

they mean that cable-1 should be revamped if suppressor-1 has no spark and the same is true for cable and supressor-2. If one generalized from the taxonomies to obtain :

suppressor(check, no, sparks) -> cable(act, yes,revamp)

The rule mean that if a suppressor has no spark then one of the cable needs to be revamped which is clearly incorrect. The method of turning constants to variable will solve this problem. It will treat each of the implications as a non-commutative function, f(P,Q). Thus for the two rules :

P1 -> Q1

P2 -> Q2

P1 and P2 must be generalized with Q1 and Q2 simultaneously. In this case, the solution is to add an extra arity to each predicate to represent each instance of the object :

(1) suppressor(check, no, sparks, 1) -> cable(act, yes,revamp, 1)

(2) suppressor(check, no, sparks, 2) -> cable(act, yes,revamp, 2)

Thus, the result of generalizing (1) and (2) will be :

(3) suppressor(check, no, sparks, x) -> cable(act, yes,revamp, x)

where x is a place holder for value 1 and value 2.

The Correctness of Meta-knowledge in KLOKE : A knowledge representation formalism is expected to have the following features :

- a well-defined semantic
- a well-defined and natural set of inferences
- representational and inferential efficiency

Formal treatments of this subject can be found in (Wrobel 1987b). A formal proof of the fact-completeness and tractability of meta-knowledge used in KLOKE/BLIP will not be provided here but an example of proving the correctness of a meta-metafact is presented below.

To provide the user with the possibility to enter new meta-metafacts and to ensure correctness of meta-metafacts, pose a serious problem. Entering meta-metafacts is a quite difficult task for the user because of its abstract representation. According to Wrobel (1987c) only about 38 percent of the meta-metafacts are provable by a theorem prover.

Ensuring the Correctness of Meta-Knowledge : The meta-metafacts can be classified into sets and each set will influence the system differently. The system will only generate some syntactically possible meta-metafacts. When a rule is entered and a new predicate defined by the system, the system will try to instantiate new meta-predicates with known meta-metapredicate definitions to generate meta-metafacts or meta-rules. The user will have to decide which meta-metafacts fits his/her representation language design. It is the system's responsibility to acquire and represent new parts of this model. However, KLOKE will rely on the user to ensure the correctness on meta-metafacts that have been entered.

Proving the Correctness of Meta-metafacts : An example of proving a meta-metafact shown below is taken from (Wrobel 1987c) :

Given the the meta-metafact :

(1) m_inclusive(symmetrical, symmetrical_neg)

its corresponding meta-metapredicate is :

(2) m_inclusive(mp, mq) where [mp(p)] -> mq(q)

symmetrical and symmetrical_neg are meta-facts having meta-predicate definitions as follow :

(3) symmetrical(p) where [p(x,y)] -> p(y,x)

(4) symmetrical_neg(p) where [not(p(x,y)] -> not(y,x)

Following this, the corresponding meta-rule of (1) is :

(5) symmetrical(p) -> symmetrical_neg(p)

(5) can be better illustrated by :

(6) [[p(x,y)] -> p(y,x)] -> [[not(p(x,y)] -> not(y,x)]

To prove (1) and (5), one has to prove that the rule scheme of (4) can be inferred from the rule scheme of (3). Transforming them to logical form, and letting P represent the rule scheme of (3) and Q represent the rule scheme of (4), results in the following :

(7) P : all x,y : (p(x,y) -> p(y,x))

(8) Q : all x,y : (~p(x,y) -> ~p(y,x))

Problem : Given the premises (7) and (8) we want to prove P -> Q (or ~P v Q) by Refutation.

1. Negating the goal we have P & ~Q
2. Transforming (7) to clausal form we have :

   (i) all x,y : ~p(x,y) v p(y,x)

3. Transforming (8) to clausal from we have :

~(all x,y : ~p(x,y) -> ~p(y,x))

~(all x,y : p(x,y) v ~p(y,x))

exist xs,ys : ~(p(xs,ys) v ~p(ys,xs))

exist xs,ys : ~p(xs, ys) & p(ys,xs)

and the resulting clauses are :

(ii) exist xs,ys : ~p(xs, ys)

and (iii) exist xs,ys : p(ys,xs)

4. The resolution tree is shown figure 5.10 :

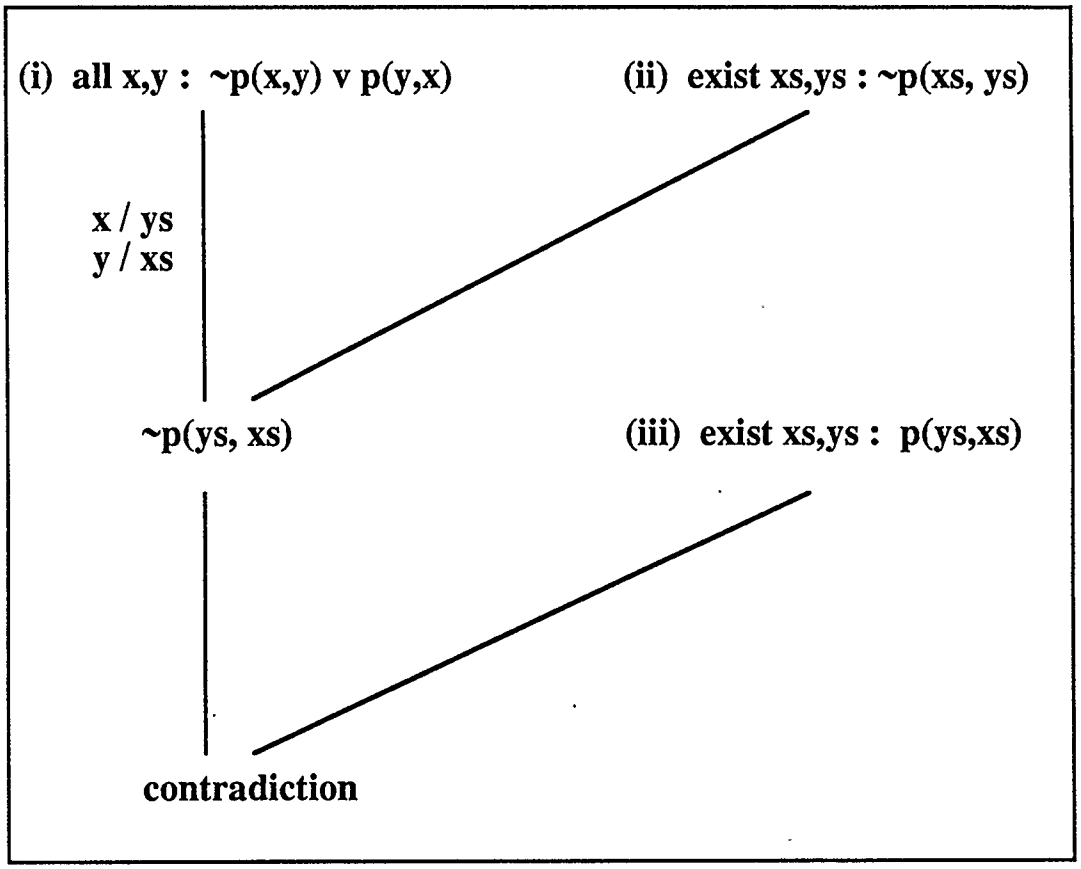

Fig 5.10 A Resolution Tree for Proving the Correctness of Meta-Knowledge

Defining Logical Relations from Existing Meta-Knowledge : Logical relations that are not implicitly included in KLOKE's knowledge representation formalism can be defined using some of the existing meta-knowledge. One such example is the logical equivalence relation (<=>) which is not part of the formalism. As such the user has to define this

relation using two identical meta-predicate definitions with different relation names as follow :

(1) equivalent(p,q) where p(x,y) -> q(x,y)

(2) inclusive_2(p,q) where p(x,y) -> q(x,y)

In addition, the relations between (1) and (2) will have to be expressed by two additional meta-metafacts :

(3) m_inclusive(equivalent, inclusive_2)

(4) m_inclusive_x(equivalent, inclusive_2)

The corresponding meta-rule of (3) and (4) respectively are :

(5) equivalent(p,q) -> inclusive_2(p,q)

(6) equivalent(p,q) -> inclusive_2(q,p)

Preventing Reflexive Metafacts : Restrictive meta-metafacts only prevent rules that have the same premise and contradictory conclusion from being added, but it does prevent reflexive metafacts/domain rules from being added to the knowledge base. Examples of reflexive domain rules are :

(1) p(x) -> not(p(x))

which when applied will clearly lead to a contradiction and

(2) p(x) -> p(x)

which will cause redundancy when applied.

The following are examples of meta-metafacts which prevent reflexive rules like (1) and (2) from being added to the knowledge base :

(3) m_irreflexive(opposite)

(4) m_irrflexive_2(opposite)

The meta-rules corresponding to (3) and (4) respectively are :

(5) opposite(p,q) -> opposite(p,p)

(6) opposite(p,q) -> not(opposite(q,q))

5.7 Summary

This chapter has presented a test example for the sort classifier, the reasoning maintenance system and the meta-knowledge acquisition module. The approach used for implementing the above mentioned modules has also been discussed. The purpose of this chapter has been to discuss each of the three modules and to discuss the strength and limitations of the implementation approach of each module.

Chapter 6
Conclusion and Future work

Over the past few chapters the principles of KLOKE, the system design and the detailed implementation were presented. Evaluation of the system and comparison to other related systems have also been discussed. This chapter concludes this thesis and presents a summary of the major design objectives of KLOKE, the accomplishment of this thesis and possible future enhancements to the KLOKE system. Future work will include designing and implementing a user interface for KLOKE. The final portion of this chapter will discuss possible applications of the KLOKE system and possible further directions of research.

6.1 Summary and Achievements

The objectives of KLOKE have been to :

1. provide an environment for manual knowledge acquisition
2. build a domain model based on the knowledge given by the user
3. detect inconsistencies
4. support terminological knowledge revision
5. support assertional knowledge revision
6. hypothesize about sorts
7. hypothesize about terminological relations
8. hypothesize about properties of facts (inference rules)
9. infer facts from the existing set of facts and rules

Objectives 4, 6 and 7 have been satisfied by the sort classifier which was successfully implemented. The technique employed by the sort classifier for evolving a sort taxonomy is both incremental and reversible. The resulting sort taxonomy that is constructed is independent of the sequence of the set of facts entered by the user.

Objectives 3, 5 and 9 have been met by the reasoning maintenance system which was successfully realized. The reasoning maintenance system provides a mechanism to infer

new beliefs from the existing set of beliefs in the knowledge base. It also has a sub-module to detect contradicting facts and rules and a knowledge revision module to revise beliefs.

The module for acquiring meta-knowledge was successfully completed. It hypothesizes about inference relations and domain rules, this will partially satisfy Objective 8. There are two ways that KLOKE can discover domain rules, one is through the meta-knowledge acquisition module and the other is through the rule discovery module which is partially developed.

Objective 2 has been met by the collective functionalities of the sort classifier, the reasoning maintenance system and the meta-knowledge acquisition module.

Objective 1 is to build a user interface for KLOKE, which is part of the future development. The design of KLOKE's user interface is presented in the next section.

6.2 The User Interface of KLOKE

From the previous chapters, it is clear that the predicate representations of both the sort taxonomy and the data dependency network are highly illegible especially for large scale knowledge bases that are common in the real world. As such one can see that the idea of providing a facility for browsing and navigating the knowledge base (just as in the case of KREME) is both useful and necessary. Three main windows will be provided to allow the user to view the sort taxonomy and the data dependency network pictorially and to allow the user to search for a particular predicate name.

Figure 6.1 shows a window for displaying the sort taxonomy. Horizontal and vertical scroll bars are provided so that the user can navigate through the sort taxonomy for large knowledge bases. To obtain information about a sort, the user can simply click on the bubble which contains the sort name and the extension set of the sort will be displayed through the *sort extension* window. The *argument position* window shows the argument place of each of the predicates which have arguments that are mapped into the sort that is selected by the user.

Figure 6.2 shows the *lexicon* window which the user can use to scan the knowledge base for a particular predicate name or find a predicate name using the find facility. The system will display through the *arguments* window the list of argument (by their sort names) of a predicate that the user selects from the *lexicon* window. If the user selects a particular argument sort from the *argument* window, the system will display the extension set of the selected argument sort through the *extension set* window. The user can also enquire about the number of facts that are represented by the selected predicate by clicking the *fact count button* in the *lexicon* window.

The *data dependency network* window in figure 6.3 shows the dependency of a particular fact on the set of facts and rules in the knowledge base. For large scale knowledge bases, the data dependency network will be extremely large and it is impossible to display the entire network. Instead a focus facility is provided, such that the system will only display a portion of the data dependency network that is in the immediate neighbourhood of a particular node (fact or rule).

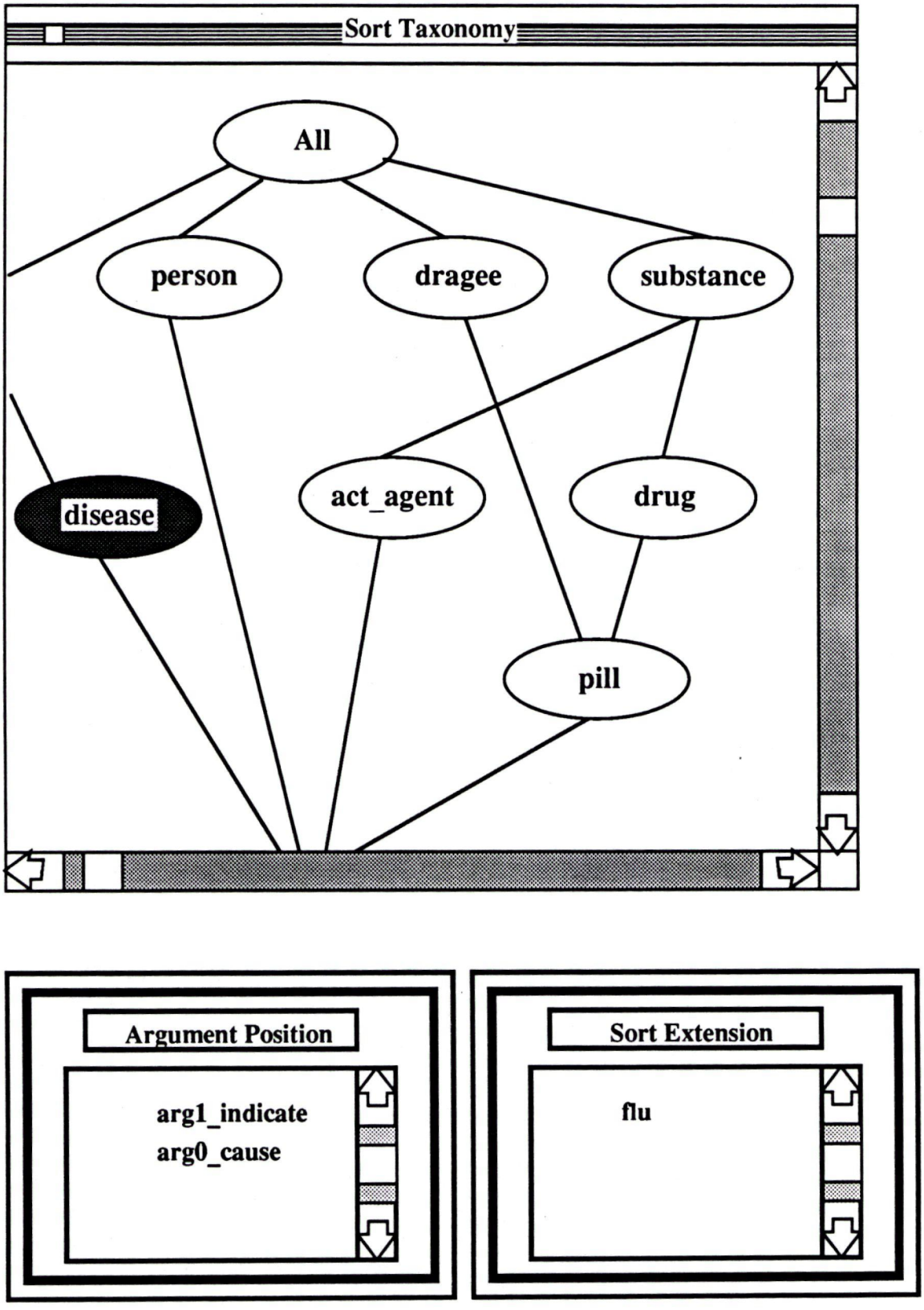

Fig 6.1 Sort Taxonomy Window

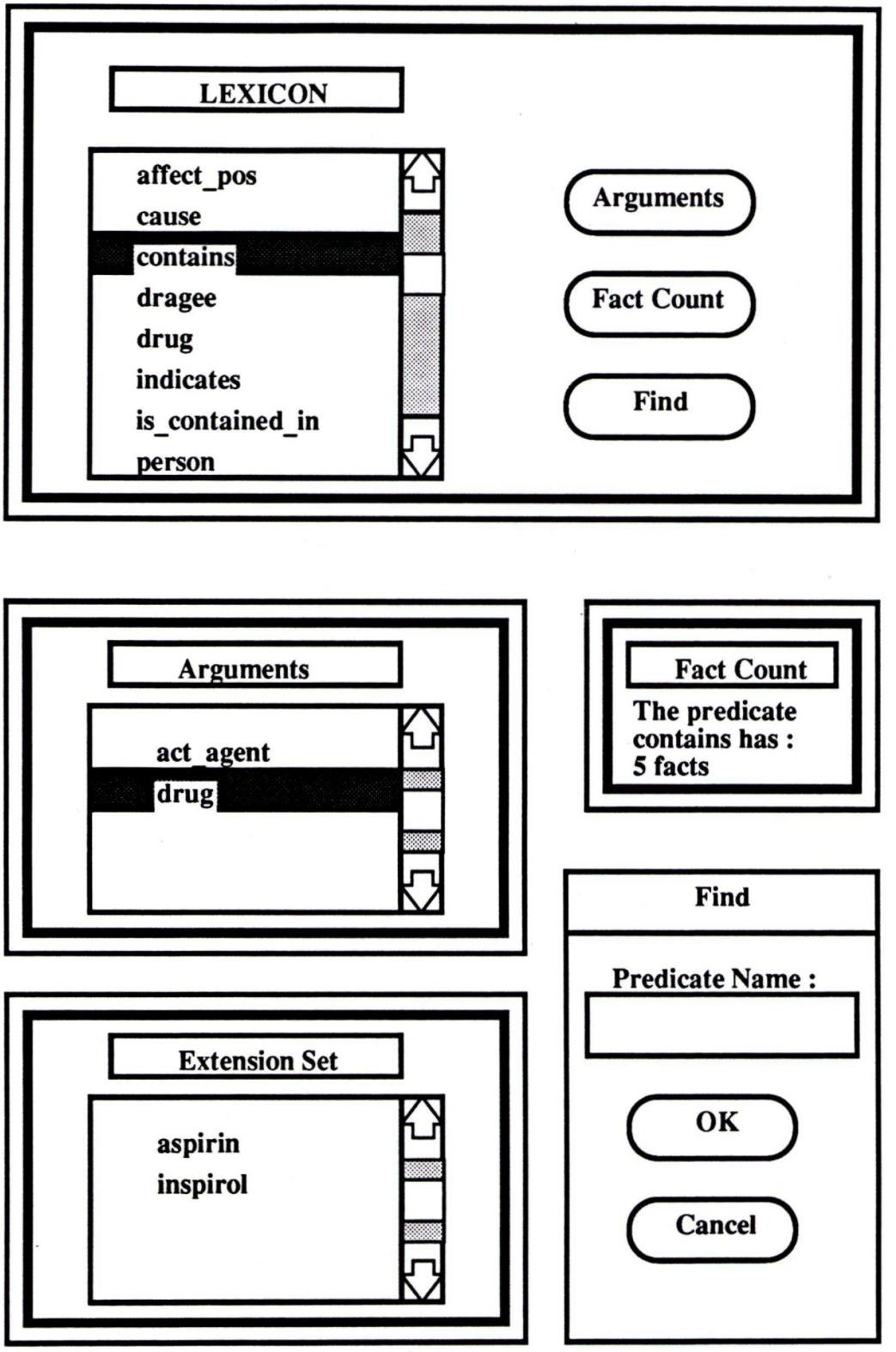

Fig 6.2 Lexicon Windows

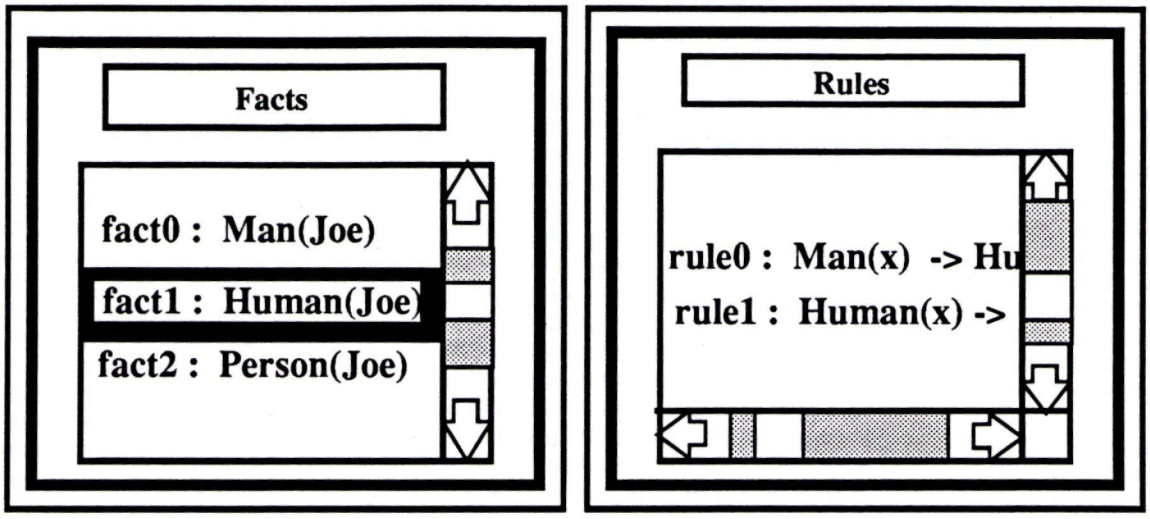

Fig 6.3 Data Dependency Network Windows

6.3 Discussion : Applying the KLOKE System

Systems like LEAP and DISCIPLE, which are applied in a problem-solving environment, employ a learning by doing paradigm (Kodratoff 1988b). The learning mainly consists of improving the performance of problem solving while the systems are actually being used in a real-life operating situation. The idea is that it uses explanation of its behaviour to improve its subsequent performance by modifying some of the rules in its knowledge base. As such the system will require initial information in order to begin the learning process.

Although DISCIPLE does not require a strong domain theory, it still depends heavily on the quality of the domain theory. The representation of entities and their properties in the domain theory will affect the explanation that is being drawn from it. Thus, even though the domain theory may be weak or incomplete, it must still contain a well structured hierarchy of relations describing the properties of the objects in the domain.

Even though learning apprentice systems such as LEAP and DISCIPLE require that the initial background knowledge be present in the knowledge base before learning can actually take place, they do not provide a mechanism for acquiring this initial knowledge. However, learning apprentice systems such as DISCIPLE and LEAP have the advantage over cooperative balanced modelling systems in that they acquire knowledge through a non-explicit learning mode. The approach used by KLOKE in building an initial domain model and can thus be applied to construct the knowledge needed for learning apprentice systems. This has led to the idea of a *cooperative problem solving* systems (Kodratoff 1990), (Tecuci 1991). A cooperative problem solving process consists of two stages.

During the first stage, the system and user will together build an initial model which may or may not be correct or complete. In the second stage the system and user solve problems together and during this learning-by-doing stage, the system will try to discover new rules. The sloppy domain that is constructed from a cooperative balanced modelling system can be refined during the learning-by-doing stages. This will serve as a base idea for designing a system called BINAR. BINAR, shown in figure 6.4 is a possible augmentation of the KLOKE system by incorporating to it a learning apprentice system.

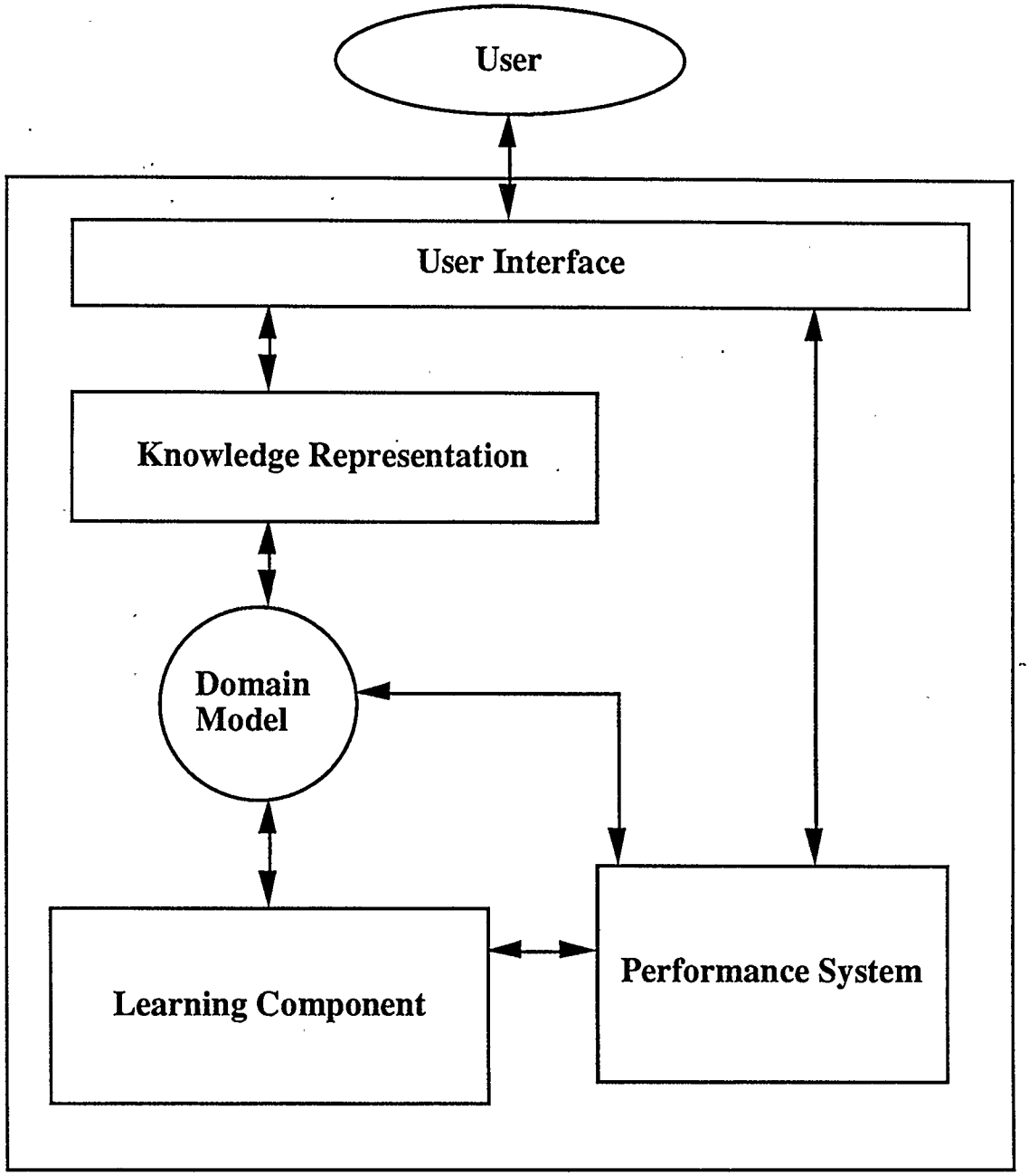

Fig 6.4 The System Architecture of BINAR

6.4 Summary

This thesis has described the principles, design, implementation, testing, and possible augmentation of the KLOKE system. Most of the design principles of KLOKE are based on the BLIP system. The issues of knowledge base maintenance and revision have been addressed using a KL-ONE based hybrid knowledge representation system. KLOKE can be used by any general user to construct and to revise a knowledge base. Some of the design criteria of KLOKE have been to emphasize incrementality and reversibility. As such KLOKE can accept incorrect and incomplete knowledge as its input. The discovery component will attempt to hypothesize missing knowledge while the reasoning maintenance system will detect inconsistencies. Revision of terminological and assertional knowledge is supported by the sort classifier and the reasoning maintenance system respectively. Further work on KLOKE will be taken up in the BINAR system described above.

References

Abrett & Burstein (1987)
Glenn Abrett, Mark Burstein. The KREME knowledge editing environment. Int. J. Man-Machine Studies (1987) 27, pages 103-126.

Bareiss (1989)
Ray Bareiss. Exemplar-Based Knowledge Acquisition : A Unified Approach to Concept Representation, Classification, and learning, Academic Press Inc, 1989.

Bareiss & Porter (1990)
E. Ray Bareiss, Bruce W. Porter, Craig C. Wier, An Exemplar-Based Knowledge Acquisition, in Michalski, R and Yves Kodratoff (eds) Machine Learning : An artficial intelligence approach, vol 3, 1990, pages 112 - 139.

Brachman (1979)
Ronald J. Brachman.On the epistemological status of semantic networks. In N.V. Findler (eds) Associative Networks: Representation and used of knowledge by computers., New York academic.

Brachman & Levesque (1983)
Ronald J. Brachman and H. J. Levesque. Krypton : A functional appraoch to knowledge representation. IEEE computer a special issue on knowledge representation, 16(10), pages 67-73,October, 1983.

Brachman (1985)
Ronald J. Brachman. An Overview of the KL-ONE Knowledge Representation System. In Cognitive Science 9, 1985, pages 171-216.

Campbell (1990)
J.A. Campbell, Three novelties of AI: theories, programs, and rational reconstructions, In Derek Partridge and Yorick Wilks (eds) The foundations of artificial intelligence, Cambridge University Press, 1990, pages 237 -245.

Charniak E. (1980)
Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. Artificial Intelligence Programming. Erlbaum, Hillsidale, N.J.

Compton & Jansen (1990)
Compton P and Jansen R. A Philosophical basis for Knowledge Acquisition. In Knowledge Acquisition 2 (3), pages 241-258.

Doyle (1979)
Jon Doyle, A truth Maintenance System. Artificial Intelligence, 12 (3) : 1979, pages 231-272.

Emde and Morik (1987)
Werner Emde and Katharina Morik. Consultation-Independent in BLIP, in A Hutchinson (eds.): Machine and Human Learning Horwood Pub, pages 93-103.

Emde (1987a)
Werner Emde,"Non-Cumulative learning in METAXA.3"; In Proceeding of 10th IJCAI, Milano, Italy, pages 208-210.

Emde (1987b)
Werner Emde. An Inference Engine for Representing Multiple Theories. In K. Morik (eds) Lecture Notes in Artificial Intelligence, No 347, pages 149-175.

Hayes-Roth (1983)
Hayes-Roth F. Using Proff and Refutations to Learn From Experience. In Michalski, R and Mitchell T. (eds) Machine Learning : An artificial intelligence approach, vol 1, 1983, pages 221-240.

Kietz (1988)
Kietz, J.-U, "Incremental and Revisible Acquisition of Sort Taxonomies"; in : Proceedings of the European Knowledge Acquisition WorkShop (EKAW 88), GMD-Studien, Bonn, June,1988.

Kodratoff and Tecuci (1987a)
Yves Kodratoff, Gheorghe Tecuci. Rule Learning in DISCIPLE, in A Hutchinson (eds.): Machine and Human Learning Horwood Pub.

Kodratoff & Tecuci (1987b)
Yves Kodratoff, Gheorghe Tecuci., DISCIPLE 1: Interactive Apprentice system in weak theory fields. In Proceedings of the International Joint Conferecnce on Artificial Intelligence, pages 271 - 273, 1987.

Kodratoff & Tecuci (1987c)
Yves Kodratoff, Gheorghe Tecuci. Techniques of Design and DISCIPLE Learning Apprentice. In the International Jornal of Expert Systems, 1987, pages 39 -66.

Kodratoff (1987d)
Kodratoff, Ives: "Is AI a subfield of Computer Science - or is AI the Science of Explanations". In Bratko; Lavrac: Progress in Machine Learning (EWSL 87, Bled, Yugoslawia), Sigma Press, Wilmslow, England, 1987.

Kodratoff & Tecuci (1988a)
Yves Kodratoff, Gheorghe Tecuci. The Central Role of explanation in DISCIPLE. In K. Morik (eds) Lecture Notes in Artificial Intelligence, No 347.

Kodratoff (1988b)
Yves Kodratoff, Introduction to Machine Learning, 1988, pages 59 - 86.

Kodratoff & Tecuci (1990)
Yves Kodratoff, Gheorghe Tecuci. Apprentice Learning in Imperfect Domain Theories, In Michalski, R and Yves Kodratoff (eds) Machine Learning : An artficial intelligence approach, vol 3, 1990, pages 515 - 551.

McAllester (1982). David McAllester. Reasoning Utility Package User's Manual. AI Memo 667, AI Laboratory, Massachusett Institute of Technology, Cambridge, Mass.

Michalski & Winston (1986)
R.S. Michalski. and P. Winston. Variable Precision Logic, Artificial Intelligence 29 North Holland, Amsterdam , pages 121-146.

Michalski (1987)
R.S. Michalski. Concept Learning. In Stuart C, Shapiro(eds) Encyclopedia of Artificial Intelligence Vol 1. John Wiley and Son Pub, pages 185 - 193.

Michalski (1991)
R.S. Michalski.Toward a unified Theory of Learning: An Outline of Basic ideas. Invited papers for the first world conference on the fundamentals of Artificail Intelligence, Paris, July 1-5, 1991.

Mitchell (1985)
T.M. Mitchell, S. Mahadevan, and L.I. Steinberg. LEAP : A learning apprentice for VLSI design. In Proceedings of the International Joint Conferecnce on Artificial Intelligence, 1985, pages 574-580.

Mitchell (1990)
T.M. Mitchell, S. Mahadevan, LEAP : A learning apprentice for VLSI design, In Michalski, R and Yves Kodratoff (eds) Machine Learning : An artficial intelligence approach, vol 3, 1990, pages 271-301.

Morik (1987)
Katharina Morik. Acquiring domain models. Int. J. Man-Machine Studies (1987) 26, pages 93-104.

Morik (1988)
Katharina Morik. Sloppy Modeling. In K. Morik (eds) Lecture Notes in Artificial Intelligence, No 347, pages 107 -134.

Morik (1990)
Katharina Morik. Integrating Manual and Automatic Knowledge Acquisition - BLIP, in Mcgraw, Westphal (eds) Readings in Knowledge Acquisition - Current Practices and Trends, Ellis Horwood Pub, 1990, pages 213-232.

Morik (1991)
Katharina Morik, Underlying assupmtions of knowledge acquisition and machine learning, in Knowledge Acquisition 3, 1991, pages 137-156.

Moser (1983)
M.G. Moser. An Overview of NIKL, the new implementation of KL-ONE. In research In knowledge representation and natural language understanding, BBN report No. 5421, Bolt, Beranek and Newman Inc., Cambridge, Mass., 1983, pages 7-26.

Nebel (1990)
B.Nebel, Reasoning and Revision in Hybrid Representation Systems, Lecture Notes in Artificial Intelligence Vol 422, Springer-Verlag, 1990.

Porter & Bareiss & Holte (1990)
Bruce W. Porter, E. Ray Bareiss, and Robert C. Holte, "Concept Learning and Heuristic Classification in Weak-Theory Domains", Artificial Intelligence 45(1-2):229-263.Reprinted in "Readings in Machine Learning", J. Shavlik and T.G. Dietterich (editors), Morgan-Kaufmann, pages 710-746.

Salzberg (1985)
Steven L Salzberg, Heuristics for Inductive Learning, In Proceedings of the International Joint Conferecnce on Artificial Intelligence, 1985, pages 603 - 609.

Salzberg (1990)
Steven L Salzberg, Learning with Nested Generalized Exemplars, foreword by William A. Woods, Kluwer Academics Publisher.

Schmidt-Schaub (1989)
Schmidt-Schaub M. Computational aspects of an Order-Sorted logic with Term declarations, Lecture Notes of Artificial Intelligence No 395, pages 149-155.

Shanahan (1988)
Murray Shanahan, Incrementality and Logic Programming, In Barbara Smith and Gerald Kelleher (eds) Reason Maintenance Systems and their applications. Ellis Horwood Publications, 1988, pages 21 -34.

Shanahan (1989). Murray Shanahan and Richard Southwick. Search, Inference and Dependencies in Artificial Intelligence, Ellis Horwood Publication.

Shrager & Langley (1990)
Jeff Shrager and Pat Langley. Computational Models of Scientific Discovery and Theory Formation, Morgan Kaufman.

Simon & Lea (1991)
Herbert A. Simon and Glenn Lea. Problem Solving and Rule Induction: A unifying view. In "Readings in Machine Learning", J. Shavlik and T.G. Dietterich (editors), Morgan-Kaufmann, pages 26-37.

Thagard (1988)
Paul Thagard. Computational Philosophy of Science, MIT Press.

Thieme (1987)
Sabine Thieme. The Acquisition of Model-Knowledge for a Model-Driven Machine Learning Approach. In K. Morik (eds) Lecture Notes in Artificial Intelligence, No 347, pages 177-191.

Tecuci (1991)
Gheorghe Tecuci. A Mulistrategy Learning Approach to Domain Modelling and Knowledge Acquisition. In Yve Kodratoff (Eds) Machine Learning - EWSL -91, Lecture Notes in Artificial Intelligence No 483, 1991.

Vilian (1985)
Marc B. Valian. The restricted language architecture of a hybrid representation system. In Proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, Cal., August, 1985, page 547-551.

Walther (1985)
Christoph Walther. A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution. In Proceeddings of AAAI-84,pages 330-334, revised version in Artificial Intelligence 26, 2, pages 217-224.

Wilkins (1988)
David C. Wilkins. Knowledge Base Refinement Using Apprentice Learning Techniques. In K. Morik (eds) Lecture Notes in Artificial Intelligence, No 347.

Wilkins (1990)
David C. Wilkins. Knowledge Base Refinement as Improving an Inccorect and Incomplete Domain Theory. In Michalski, R and Yves Kodratoff (eds) Machine Learning : An artficial intelligence approach, vol 3, 1990, pages 493 - 513.

Wrobel (1987a)
Stefan Wrobel. Design goals for sloppy modelling systems. Int. J. Man-Machine Studies (1988) 29, pages 461-477.

Wrobel (1987b)
Stefan Wrobel,"Higher order concept in a tractable knowledge representation" In Morik (ed) Proceeding of the GWAI-87,11th German Workshop in Artificial intelligence,Guericke; Springer Verlag, Berlinpages, pages 129-138.

Wrobel (1987c)
Stefan Wrobel. Demand-driven Concept Formation. In K. Morik (eds) Lecture Notes in Artificial Intelligence, No 347, pages 249-319.