Practical Techniques for Organizing and Measuring Knowledge

by

Timothy Christian Lethbridge, BSc(CS), MSc(CS)

Thesis

Presented to the School of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree
Doctor of Philosophy (Computer Science)

University of Ottawa*
Ottawa, Ontario, K1N 6N5
Canada

* The PhD program in Computer Science is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

Dedicated

to the memory of

Grandpa Fred Fitzgerald Lydon

Abstract

This research is concerned with the problem of making knowledge acquisition and representation practical for a wide variety of people. The key question investigated is the following: What features are needed in what this research defines as a *knowledge management system*, so that people who are not computer specialists can use it for tasks that involve manipulating complex ideas?

In this research the needs of such users were evaluated, and several prototype systems were created, culminating in the creation of a system called CODE4. A key conclusion is as follows: Users need a tool that involves the synthesis of several techniques for organizing the knowledge.

The research has led to ideas for two types of features: abstract knowledge representation features and user interface features. Some of the proposed abstract knowledge representation features that are novel or improved include: 1) a uniform treatment of units of knowledge which are called concepts; 2) the separation of the names of concepts from their unique identities so that concepts can have multiple names and a given name can be used for several concepts; 3) the classification of conceptual relations using property and statement hierarchies, and 4) facilities for smoothly integrating informal knowledge with knowledge that is more precisely represented. The above ideas are implemented in CODE4-KR, the abstract knowledge representation used in the CODE4 knowledge management system.

Some of the user interface features that are found in CODE4, and are proposed for general use, are: 1) unlimited chained browsers; 2) knowledge maps, representing patterns of knowledge to be explored or processed; 3) fully interchangeable mediating representations operating on knowledge maps; 4) a uniform set of knowledge editing operations in the mediating representations, and 5) masks for highlighting and controlling the visibility of concepts.

CODE4 has had a significant amount of serious use by users in such diverse fields as terminology, organizational modelling and software engineering. This has served as the basis for validating the above ideas about knowledge organization. To this end, several techniques have been used including: 1) a detailed questionnaire about CODE4 administered to many users, and 2) a static analysis of the knowledge bases using several newly-developed metrics. The metrics include seven largely independent measures of knowledge base complexity, as well as compound measures that allow the user to ascertain the completeness, well-formedness and information-richness of a knowledge base.

Acknowledgements

The mental energy of numerous people has influenced the development of this thesis. Doug Skuce, my supervisor, first aroused my interest in artificial intelligence and shaped much of my early thinking. Many of the ideas herein build upon Doug’s previous research. Doug takes great care to provide his students with excellent facilities and as much of his personal time as they need.

Yves Beauvillé provided many early insights and was an invaluable resource as the main programmer of CODE2. Ken Iisaka and Danny Epstein were sources of great inspiration as both friends and colleagues. CODE4 would have been a far inferior product were it not for Danny, Ken and the other members of the programming team: Dave Corbett, Judy Kavanagh and Eric Arseneau.

Peter Clark and Denys Duchier provided in-depth comments about the thesis, and have been a continual source of stimulating conversations. Discussions with Rob Holte showed his vision and great depth of understanding. Numerous people at conferences and workshops helped me broaden my thinking. Bruce Porter, Brian Gaines and Mildred Shaw inspired me greatly.

The users of the CODE systems provided much valuable feedback: Ingrid Meyer’s frequent requests taught me what is really important. Other members of her Cogniterm team including Lynne Bowker, Karen Eck and Bruce McHaffie provided valuable comments and usage data, and above all have been wonderful friends. Many people at Bell-Northern Research helped in the development of ideas, particularly Milan Javor and Roy McLean. The enthusiasm shown by Roy and by Jeff Bradshaw at Boeing boosted my confidence tremendously. Also, I cannot give enough thanks to others who spent many hours creating knowledge bases and answering questionnaires about CODE4: These include Walid Saba, Nagi Ghali, Boyan Broderick, Chris Drummond and Chuck Wang.

Financial support for my research was provided by NSERC through a postgraduate scholarship, by BNR and by Doug Skuce. Doug’s budget was funded by NSERC, BNR, Cognos, URIF, SSHRC and by sales of the CODE systems.

The support of my family and friends have allowed me to reach this stage. My father and my grandfather (who died just as I was completing the thesis) interested me in science and computers at a young age and have always been sources of inspiration and encouragement. My dear departed stepmother taught me the value of self-discipline and hard work. Numerous compatriots on skiing, hiking and canoeing expeditions helped me relax and keep a perspective on other things in life. Alice Lang, Melanie Lukach and other friends at the Graduate Students Association showed me that there is a light at the end of the tunnel.

Finally, my wife Teresa came into my life at the right moment and has been a source of continual encouragement. She participated in this work in numerous ways: By being present to console me, by putting up with a crazy work schedule, by acting as a sounding board for ideas, by helping me structure my reasoning so that somebody from another field can understand it, and by proofreading.

Contents

List of figures

Chapter 1

Introduction

1.1 Overview of the research

This research examines how to make knowledge acquisition and representation technology available and useful to a wide variety of people. The ultimate goal is to make such people more productive with the technology; the proximate goal is to identify key principles that should be applied in the design of knowledge representation and acquisition systems. The primary problems to be overcome are several intrinsic decision-making difficulties people encounter while trying to organize knowledge, as well as various ease-of-use problems encountered with today's technology.

The ideas described in this thesis have been largely implemented in a software system called CODE4 (Conceptually Oriented Design Environment, version 4). The main research procedure has been to study the use of successive versions of CODE, and then to modify or redesign the system to better solve the users' problems.

The biggest insights gained while doing the research has been the following: 1) That developing effective techniques for organizing *sets* of concepts improves the practicality of knowledge acquisition, and 2) That to create a useful system it is necessary to synergistically combine these techniques. Thus the thrust of the research has been to exhaustively categorise concepts and their distinguishing criteria, and then to develop and integrate ways of manipulating, presenting and measuring sets of concepts in these various categories.

This focus on organizing sets of concepts has led to the development, in a parallel, interdependent manner, of both knowledge representation (KR) and user interface (UI) techniques that are at the same time expressive, powerful and easy to use. This research is presented in chapters 3 and 4. A complementary line of investigation, presented in chapter 5, is the development of ways of *measuring* sets of concepts (knowledge base metrics).

CODE4 has been used seriously by a wide variety of people in commercial and educational settings. Observations of this use have led to the belief that knowledge management software embodying ideas such as those in CODE4 will one day become commonplace – once people recognize its value. The development path foreseen is similar to that which led to the widespread use of spreadsheet programs, although the latter are expected to always have far more users[1].

The next section introduces key terms used in the thesis. The subsequent three sections briefly motivate aspects of the work by describing applications of knowledge management and problems users encounter. Section 1.5 gives a preview of some of the ideas developed during this work; and section 1.6 describes the research methodology.

1.2 Definitions of important terms

This section briefly defines important terms that are used repeatedly in the thesis. By understanding the way these terms are used, the reader will gain a clearer understanding of the nature of the research.

A much more exhaustive list defining over 100 terms can be found in the glossary. Detailed descriptions of most of the ideas in the glossary can also be found in the main body of the thesis; the index can be used to locate specific discussions. Also available is a CODE4 knowledge base that describes the terms found in the glossary.

1.2.1 Knowledge acquisition and knowledge management

For the purpose of this thesis, *knowledge acquisition* (KA) is considered the process of gathering knowledge and entering it into a computer so that the computer, and humans using it, can put that knowledge to use. Traditionally, the term knowledge acquisition has referred to gathering *expertise*, primarily in the form of rules, from experts in order to create an *expert system* (Gaines 1987; Neale 1989). Today, the term has evolved to mean more than that. This thesis, for example, is concerned very little with rules and even less with expert systems (although the acquired knowledge may be used to *drive* such systems). The kind of knowledge acquired in this research is typically general knowledge about things in the world, or detailed knowledge about the design of some device, program, organization etc.

An understanding of the term ‘knowledge acquisition’ requires a basic understanding of what ‘knowledge’ is. Unfortunately there are many debates about the definition of the latter, and it is desired to avoid prolonged arguments. In general, knowledge is considered to be any form of information that one might be able to manipulate in one’s brain, but this thesis is primarily concerned with those aspects involving the *categorization, definition and characterization of things and their relationships*. This thesis is deliberately unconcerned

[1] It is possible that ideas from knowledge management might someday be merged with those of spreadsheets.

with repetitive data, procedural knowledge (as might be embedded in a computer program) or knowledge of spatial structures (as might be embedded in a CAD/CAM database)[2].

The word ‘acquisition’ implies an inputting process, whereas the kinds of processes performed in this research go far beyond that: Here, concerns include manipulation, storage, presentation and many other processes performed on the knowledge. Therefore to clarify the nature of this research, the broader term ‘knowledge management’ (Riedesel 1990) is chosen – and the CODE4 software system developed in this research is called a *knowledge management system* (KMS).

The term ‘knowledge management’ as used in this thesis can thus be defined as: The process of acquiring, representing, storing and manipulating the categorizations, characterizations and definitions of both things and their relationships.

Other possible terms that could have been chosen for CODE4 instead of ‘knowledge management system’ include: ‘knowledge representation system’ (except this tends to connote the static storage of knowledge) and ‘knowledge based system’ (except that this has come to refer primarily to expert systems – performance software that *uses* knowledge once it is acquired).

1.2.2 Concepts

In this thesis, all units of knowledge are uniformly called *concepts*. One place where concepts exist is in the brain of an intelligent being; there they are the units that thought processes manipulate. Sometimes when one discovers something, one creates a new mental concept to represent it; sometimes a concept comes into being as a conclusion of a thought. Other times one is explicitly taught a concept. Under any of these circumstances an intelligent entity such as a human tries to relate the new concept to others – to characterize it, differentiate it and link it into networks. The principle that all units of thought should be called concepts is supported by an ISO definition (ISO 1990) which reads, “any unit of thought generally expressed by a term, a letter symbol or by any other symbol”.

Concepts in a knowledge management system such as CODE4 are highly analogous to concepts in a mind[3], with the exception that a computer system has external agents (users) to perform much of the commonsense reasoning required to do such things as integrating a new concept. Hence a useful metaphor used in this research is to model a knowledge man-

[2] Procedural and spatial knowledge and data are important; so important in fact that effective abstractions and tools have been developed to represent and manipulate them. There is a lack, however, of practical tools for the kinds of knowledge on which this thesis focuses.

[3] This does intend to imply that CODE4 is organized like a brain (or vice-versa!). The analogy is merely that both contain identifiable units of knowledge.

agement system as if it were indeed an intelligent entity – this is helpful in the process of deciding what the system's concepts really are and how they should be organized.

In general a concept represents one or more *things*, real or imaginary, concrete or abstract. There can be more than one concept of the same thing; indeed each intelligent entity or running knowledge management system has its own unique concept of any given thing. There can also be *consensus* concepts of a thing: In such cases one imagines a group or a society as the collective intelligent consciousness that has the concept.

One of the goals of knowledge acquisition or knowledge management is to manipulate the state of a knowledge management system so that its concepts match those of a user, or of the consensus of a set of users. This process is very similar to the process that an intelligent entity undergoes when it learns something[4].

It was stated above that all units of knowledge are concepts according to the terminology of this thesis. The following clarifies further: Any component of the representation of knowledge that can be related to other components is called a concept. Thus the following are among the representational entities that are called (or at least represented using) concepts: types, classes, instances, terms, rules, statements, propositions, properties, predicates, relations etc. The glossary contains over twenty terms used for specific kinds of concepts found to be useful in this thesis, plus several additional terms for roles played by concepts. Chapter 3 clarifies exactly how knowledge is represented using concepts.

1.2.3 Knowledge bases and ontologies

A knowledge base is a collection of interrelated concepts. Intelligent entities in general may be considered to have a knowledge base that evolves over time. A computer system has the additional abilities to load, save and share knowledge bases. Such a system may have more than one independent knowledge base active at once.

The word 'ontology' is often confused with 'knowledge base'. In this work, an ontology is considered a specialized kind of knowledge base that consists of descriptions or definitions of types of things. This corresponds roughly with Gruber's definition which is: "a specification of a conceptualization: the objects and relations that exist for an agent" (Gruber 1993). An ontology is generally considered *not* to include descriptions of *specific* things, i.e. instances.

It might also be argued whether a knowledge base of types should be said *to be* an ontology or *to represent* an ontology. The latter implies that ontologies are abstract or philosoph-

[4] Of course, *machine learning* is one of the techniques that can be used to assist knowledge management. This thesis focuses on situations where *humans* do most of the inferencing, rather than machines. The goal is to facilitate human reasoning processes.

ical in nature and that knowledge bases are concrete. In this thesis, the former usage is adopted: Both knowledge bases and ontologies can be imagined to be either abstract (i.e. existing in the brain or society) or concrete.

Most of the knowledge bases created during this research are ontologies as described above. However, this thesis discusses knowledge bases in general; this is partly because instance knowledge is sometimes manipulated by CODE4 users, and partly because there is disagreement about the definition of ‘ontology’: Some only consider a knowledge base to be an ontology if it is very abstract or ‘high level’ in nature, containing truths applicable in a wide variety of everyday experience.

1.2.4 Knowledge organizing techniques

In this thesis *knowledge organizing techniques* are distinct and somewhat independent aspects of the process of representing sets of concepts. Two examples, with which most computer scientists are familiar, are: 1) categorizing concepts when building inheritance hierarchies and 2) associating attributes (properties, slots etc) with concepts. Each of these involves the making of a distinct set of decisions and the performing of a distinct set of actions. *Knowledge organizations* are individual instances of structures within a knowledge base that are formed when a user is working within a particular organizing technique.

One of the objectives of this research is to analyse knowledge organizing techniques (some of which are new ideas) in order to learn ways of making knowledge management more practical. Details of organizing techniques at the abstract knowledge representation level are covered in chapter 3; details of organizing techniques for the user interface are found in chapter 4.

1.2.5 Formality and informality

The contrast between formality and informality, and the need to provide facilities to manipulate both, are themes found in several parts of this thesis. However usage of these terms is controversial.

The word ‘formal’ has a wide variety of meanings in general English. The Concise Oxford Dictionary (Seventh Edition) gives the following relevant ones:

1a:	of the outward form…or external qualities [of a thing];
1b:	of the essence of a thing, … not material;
1c:	concerned with form, not the matter, of reasoning;
3.	observant of forms or rules, precise, regular.

Logicians and theoretical computer scientists use ‘formal’ in sense 1c when they talk about formal proofs or languages. A language would be considered formal if it has a well-defined syntax with a semantics that gives that syntax mathematical meaning in terms of conclusions that may be drawn and manipulations that may be performed.

Philosophers, on the other hand, tend to use sense 1b; but in general conversational use senses 1a and 3 are more often meant (e.g. a formal-looking dress; a formal ceremony).

In this thesis the meaning is a synthesis involving all the above senses, but primarily sense 3. ‘Formal’ is used to describe the situation where concepts are linked together within well-defined patterns (knowledge organizations). Such patterns have a syntax and rules of manipulation, and contrast with arbitrary text strings or pictures which cannot form links that the system can manipulate.The word ‘informal’ is used for the latter.

This usage can be justified in several ways: Firstly, structures that are formal (as described in the last paragraph) are certainly observant of rules (sense 3) and arbitrary strings are governed by no such rules. Secondly, just like in a formal logic or language, the manipulations possible with formal knowledge organizations do not depend on what the concepts represent (sense 1c). Finally, the only way to interpret a string or picture (which is informal) is to look beyond its outward form (opposite of sense 1a) and to understand what it materially represents (opposite of sense 1b).

Objections to the use of ‘formal’ in this thesis might arise because a mathematical or logical semantics is not *imposed* by knowledge organizations. It is not guaranteed that a CODE4 knowledge base can be converted to, say, first order logic. This is because the user is free to ignore any suggested semantics (see section 3.2 for more details).

Due to the possible controversy around the word ‘formal’, one might seek alternate words. A search of thesauri leads to several possible candidates including ‘stylized’, ‘orderly’ and ‘structured’. The last is the best; however, as with all alternatives it doesn’t capture the intended meaning as precisely as ‘formal’. For example, ‘structured’ doesn’t carry the connotation that the thing (knowledge structure) referred to can be manipulated according to rules that are independent of the thing’s referent.

In addition to the arguments provided above, other thinkers support the continued use of the words ‘formal’ and ‘informal’ as in this thesis. For example, an interdisciplinary workshop was held on the topic in 1991 (Lethbridge 1991), and Shipman’s research looks at how excessive formality can be considered harmful because it imposes too many constraints on the representation of ideas (Shipman 1993). Shipman also discusses how development of a knowledge base should proceed smoothly from informal to formal (Shipman 1992).

This thesis shares Shipman’s ideas: Chapter 3 shows how informal elements are integrated into CODE4’s knowledge representation. Chapter 4 shows how the user interface allows the uniform manipulation of both informal and formal elements; and chapter 5 shows that there is a quantifiable, measurable continuum between informality and formality in a knowledge base. However, allowing such a continuum does pose some problems: For example it poses challenges when translating knowledge to other representations that are uncontroversially formal, such as KIF (Lethbridge and Skuce 1992a).

To conclude: As used in this thesis, the word ‘formal’ and its antonym ‘informal’ should not be interpreted in the very strict senses used by some logicians, but in senses closer to ordinary English: A more formal pattern of knowledge is more ‘connected’ and obeys and is manipulable by identifiable rules; whereas a more informal structure has fewer constraints and can only be interpreted or manipulated by an external intelligence.

1.2.6 Mediating representations

A *mediating representation* (Johnson 1989) presents part of a knowledge base to an end user. It can do this in many ways including using formatted text, tables, fill-in-the-blank forms, or graphs with nodes and links. A *mediating representation schema* is the description of the syntax and semantics of a particular class of mediating representations. In general, a mediating representation is designed to highlight certain aspects of the knowledge, or to allow the user to perform a certain task with the knowledge.

This thesis carefully distinguishes mediating representation schemas from both abstract and physical knowledge representation schemas. Physical schemas describe how knowledge is stored in computers; while abstract schemas describe common sets of representational constructs that can be manifested in both physical and mediating representations.

These distinctions are made to preserve flexibility when designing physical and mediating representations. The objective is to prevent the syntax or format described in one schema from constraining the others. For example, if knowledge is physically stored as Lisp expressions, there is no reason why Lisp expressions must be presented to the end-user through a mediating representation (although this is often a problem today). The primary concerns when designing a mediating or physical representation schema should be: 1) ensuring that the principles of the abstract schema are adhered to by using semantics-preserving changes of representation; and 2) Ensuring that the new schema serves its intended task effectively.

Bradshaw and Boose have studied mediating representations in depth (Bradshaw and Boose 1992). Of particular interest in this thesis are *modelling* mediating representations (Bradshaw, Ford et al. 1993; Ford, Bradshaw et al. 1993). These allow the user to actively construct elements of a knowledge base, as opposed to being guided in the process.

1.3 The Task: Practical knowledge management

This section describes the kinds of users for whom CODE4 is designed, and the tasks that CODE4 is intended to help them perform better. Discussions of existing tools for performing these tasks and how CODE4 can help, are deferred to later sections.

1.3.1 Uses of a knowledge management systems

This subsection explores the nature of the tasks in which a knowledge management system of the type described in this thesis might be used. In general, such a system might be used where there are:

- Ideas that are novel, not easily understood, difficult to categorize, difficult to relate to each other, difficult to distinguish or difficult to define precisely.
- Complex interrelationships among ideas or large interconnected networks of ideas.
- Multiple ways of looking at ideas and multiple opinions about ideas.
- Problems expressing ideas completely, clearly and unambiguously in a linear, tabular or diagrammatic format.

The common themes in the above four points are that they describe situations where 1) natural language and special purpose or abstract representations prove inadequate, and 2) there are many details to control. In today's world natural language is used if other representations are inadequate; but this is merely because it is *possible* to represent virtually anything in natural language, not because natural language is the *best* representational format.

The following are some of the specific tasks that might be more productively performed with the assistance of a knowledge management system. Actual examples of usage of CODE4 are described in chapter 6. Applying knowledge management technology to all these tasks is an idea that is only just starting to be recognized. Unfortunately, most knowledge representation technology so far developed has not been practical enough for widespread use.

1. **Developing educational, reference or documentary material**: A student might use a resulting knowledge base to learn either in a programmed manner (navigating the knowledge in a suggested order) or in an exploratory manner. A professional in a field might refer to a knowledge base to look up detailed information about some subject. An engineer or technician might consult the knowledge base to learn the subtleties of some device or process with which he or she is having problems.

   Specialists who perform knowledge management with this task in mind might be teachers, technical authors or system documenters. They also might be terminologists or lexicographers attempting to create a reference source for authors or translators.

2. **Reverse engineering a document, system or other artifact**: In this task the knowledge engineer tries to understand something that is complex, confusing, ambiguously expressed and/or contains errors and inconsistencies. He or she analyses the artifact component-by-component and gradually builds a knowledge base that explains the 'big picture'. During the process, he or she might simultaneously come to understand how the artifact was constructed, uncover fundamental weaknesses in the original artifact and develop ideas for a replacement.

This task might be performed by maintenance engineers or anybody who has a document where knowledge is ‘hidden’. A historian or philosopher might be attempting to shed light on the contents of ancient texts; a forensic accountant or an organizational modeller might be trying to learn the control flow, information flow and money flow in an organization.

3. **Specifying or designing some artifact**: Here, the resultant knowledge is used as the basis for building or implementing the artifact, and later as a reference source about the artifact.

   This task might be performed by software or hardware engineers, or by end-users who are attempting to provide preliminary requirements.

4. **Developing the knowledge base for an expert system or other software**: In this task, the resultant knowledge is used by a reasoning system to plan, diagnose, predict etc.

5. **Synthesizing, generalizing or stimulating new ideas**: Here the task is to help the user clarify his or her thoughts, perhaps as an aid to personal decision making or to help in the writing of a document.

6. **Creating reusable ontologies**: This is a higher-order task – the user develops general purpose knowledge bases that can be used by others as foundation material for any of the above tasks.

1.3.2 Non-computer-specialists as users

A major objective of this research is to make knowledge management technology accessible to a wide range of people. In particular, it is important that people not need a computer science education. At the very basic level, a user should be able to perform productive work if he or she has an average amount of ‘organizational ability’. However, if a user *does* have specialized training in logic, mathematics, linguistics or related fields, he or she should be able to take advantage of additional optional layers of knowledge management features, including those that involve more formality. These optional features should be invisible to (i.e. they should not hinder) the less specialized user.

Currently most people who need to organize ideas do it in unconstrained natural language operated on using a word processor, perhaps with the assistance of a database, spreadsheet or diagram drawing program. These examples of *personal productivity software* allow specialists in a variety of disciplines to perform tasks they could not previously have done; however they have important limitations: Natural language documents tend to contain ambiguity and inconsistency which can be hard to detect; spreadsheets and databases are best for naturally tabular information, and knowledge expressed in diagrams tends to be hard to interrelate with other knowledge.

The dawn of hypertext (Conklin 1987) has dealt with some of these problems, however there is a need for well-thought-out principles to guide the knowledge representer. The latter is the strength of knowledge representation languages and systems developed in the artificial intelligence community.

Observations have shown that it is possible to develop software that embodies well-understood knowledge representation principles, and at the same time has the level of usability of spreadsheets and word processors. This research seeks to understand better how to meld the technologies – providing practical knowledge representation for people without computer backgrounds. A long range objective is to encourage people to better perform the tasks mentioned in the last section by using knowledge management software.

1.4 The Problems: Factors that make knowledge management difficult

This section outlines some problems users encounter when managing knowledge. The material in this section has been developed by observing and interviewing users and analysing the tasks they perform.

Some problems appear to be *intrinsic* to knowledge management, whether it is being performed using a word processor, a formal-language based tool or pencil-and-paper. If a tool proposes to provide practical knowledge management capabilities, it must help the user deal with these problems. The problems that are apparently intrinsic are listed in subsection 1.4.1.

At the same time as solving intrinsic problems, a tool should not introduce its own problems. Subsection 1.4.2 lists problems that are more 'accidental' in nature, arising from flaws or limitations in the tool being used. The categorization of problems into accidental and intrinsic deliberately parallels Brooks' famous 'No silver bullet' paper (Brooks 1987) where he uses the same division to categorize problems in software engineering.

Arising out of each problem is a corresponding high level *requirement*. In the following subsections, each problem implicitly specifies the requirement to provide a *solution*. The problems and requirements form the basis of the analysis of existing knowledge management technology (chapter 2) and the features of CODE4 (chapters 3 and 4). A summary of how the problems are addressed is found in section 7.2.4.

The list of problems and requirements is not intended to be exhaustive, but is intended to focus on the primary concerns of the kinds of users discussed in the last section.

1.4.1 Intrinsic problems

The following problems, expressed as activities which are difficult in some circumstances, appear inherent to knowledge management. They are encountered when performing most of the tasks discussed in the last section. Tools can help in the solution of the problems (making the activities easier to do), but if they are not well designed, tools can also make the problems worse.

Problem I-1: Categorizing

a) Deciding on the *criteria* for categorization, classification or grouping.
b) *Ordering* categories, criteria and elements. (Deciding which categories are more important and which should be listed first.)
c) Deciding on membership in categories. (*Applying* classification criteria.)
d) Categorizing *characteristics* of things.
e) Deciding on the level in a classification hierarchy at which to make a distinction. (Deciding where to place a fact.)
f) Understanding the nature of an *existing* classification.

Problem I-2: Naming things (choosing terms, labels or symbols)

a) Finding or inventing meaningful names for things.
b) Choosing from among synonyms or near synonyms.
c) Distinguishing among concepts that have similar terms. (Understanding their differences.)
d) Understanding and adhering to explicit or implicit naming conventions.

Problem I-3: Making distinctions

a) Differentiating two conceptsb) Expressing second order ideas (knowledge about knowledge).
c) Expressing knowledge about words or terms for things (linguistic knowledge).
d) Expressing knowledge involving mathematics, logic or higher-order relations.

Problem I-4: Understanding the effects of actions or declarations

a) Understanding what conclusions can be made due the presence of a particular fact.
b) Understanding the consequences of performing a certain action.
c) Understanding the effects of declaring a particular new fact.

Problem I-5: Extracting knowledge

a) Finding specific facts.
b) Examining large amounts of knowledge at once.
c) Understanding or visualizing the overall structure of the knowledge.
d) Choosing the perspective from which to view knowledge.

Problem I-6: Handling conceptual errors such as inconsistency, ambiguity, incorrectness and incompleteness

a) Recognizing the possibility of such errors.
b) Detecting the presence of such errors.
c) Finding such errors when they exist.
d) Living with such errors if the resolution is unknown.
e) Correcting such errors.

1.4.2 Accidental problems

The following problems arise primarily by the nature of certain tools; following Brooks' terminology the term 'accidental' is used. The problems are not intrinsic to knowledge management, however their origin may be due to a number of causes:

- They may not be considered important to the designer or the particular class of user.
- They may result from tradeoffs made during design.
- They may result from the fact that many knowledge management tools are research prototypes.

Regardless of the origin , however, all the problems tend to reduce the general practicality of a tool.

Problem A-1: The tool may be only usable for a special purpose.

a) The domain or task may be limited (e.g. to configuration tasks).
b) The type of knowledge representable may be limited (e.g. to rules).

Special-purpose tools are often justifiable on simplicity grounds; also the designer may not care about any other application. However there is a general requirement for technology that is highly adaptable so that when an idea arises, a knowledge base can be immediately built.

Problem A-2: A high degree of expertise may be needed to use the tool.

a) A computer science, logic or mathematics background may be needed.
b) There may be no way to express certain facts that lie outside the specific syntax.
c) A large amount of training may be needed.
d) No simple but useful subset may be available.
e) Support staff may be needed.

A high need for expertise is typically due to poor design. In general the problem can be solved by paying attention to user interface design principles, and making a distinction between basic and advanced capabilities. Another cause of the problem is the insistence on rigid syntax so that the representation or inference mechanisms have certain useful properties such as logical completeness.

Problem A-3: The tool may only be able to handle small knowledge bases.

a) The tool may have performance problems or reach a physical capacity limit.
b) There may be no way to avoid working on the whole knowledge base at once.

This is often the case of research tools. There is some justification for building a tool that is specialized for small knowledge bases: plenty of successful tools exist to manipulate spreadsheets, small databases and other relatively small collections of knowledge. The general requirement, however, is that knowledge management technology be reasonably 'scalable' so users do not hit artificial size barriers.

Problem A-4: The tool may only be usable by an individual; not a group

a) There may be no way of partitioning knowledge bases.
b) There may be no mechanism for concurrent access.

1.5 A preview of techniques

This section briefly outlines the rationale for the main techniques presented in this research, providing a preview of the material found in chapters 3, 4 and 5 respectively. The evaluation of all features can be found in chapter 6.

1.5.1 Knowledge representation techniques

Some knowledge management systems have a variety of types of knowledge unit (e.g. rules, constraints, facts etc). As a result there must be different mechanisms to annotate the units, different ways to categorize them, different ways to inspect them etc. As a first step in creating a simple, practical representation a way must be found to treat all classes of units as uniformly as possible, while still maintaining important distinctions. Chapter 3 presents CODE4-KR, one of whose main tenets is that all units of knowledge (including such things as terms and statements) should be treated uniformly as *concepts*.

Central to knowledge management is the idea of *categorizing*; i.e. arranging concepts in a generalization or inheritance hierarchy. Equally important is the idea of *characterizing* concepts, i.e. specifying properties that define or describe the concepts. However little attention has been paid to the categorization of those properties *themselves*. Chapter 3 discusses the idea of a global property hierarchy in a knowledge base, as well as a number of closely related organizing techniques.

Many knowledge management technologies (e.g. artificial intelligence based systems) require the use of a very formal syntax when representing knowledge. There is usually the possibility of adding comments to the formal constructs, but the comments are typically second-class entities. Other technologies (e.g. hypertext) are the opposite: They have no formal syntax and thus are limited in the kind of inferencing they can do. There is a need for a hybrid technology that allows informal elements to be treated, where possible, just like formal elements. CODE4-KR is an attempt to move in this direction.

1.5.2 User interface techniques

Typical user interfaces of artificial intelligence-based tools are designed as necessary add-ons so that the underlying representation can be manipulated. Hypertext on the other hand is generally designed with usability and information conveyance as its priority. Again, a hybrid is needed: a system designed to convey information that comes from a well-organized representation. In chapter 4, the idea of 'high-bandwidth' browsing is presented, wherein users can rapidly browse through large amounts of knowledge using effective user interface tools called mediating representations, knowledge maps and masks.

1.5.3 Knowledge measuring techniques

The importance of measuring a product is recognized throughout science and engineering. The more physical the product, however, the easier it is to quantify. Software engineering deals with products that are particularly intangible, and therefore research into measuring techniques has been slow and even slower to be accepted by practitioners. Research into measuring the products of knowledge engineering has been even slower, or nonexistent.

Chapter 5 presents some metrics that can be used to assess knowledge bases. The primary reason for developing these is so that knowledge base developers can monitor and improve their work. Being able to measure various aspects of complexity can help users in such tasks as measuring their productivity, estimating the information content in a knowledge base, and finding out whether and where there are 'holes' (areas of apparent incompleteness) in a knowledge base.

1.6 Research history and methodology

This section outlines the history and methodology of the research. Only the *procedures used* are discussed here; insights and conclusions derived from the research are discussed in later chapters.

1.6.1 Research chronology

The research can be divided into three main phases:

• Development and experimentation with early versions of CODE.
• Development of CODE4.
• Evaluation of CODE4.

The next three subsections outline the procedures used in each phase.

Phase 1: CODE2 development (1989-1990)

CODE2 is a knowledge acquisition and representation tool, developed by the AI Laboratory at the University of Ottawa under the direction of Doug Skuce. CODE2 was written in the Smalltalk object-oriented programming language which allowed it to be given significant graphical capabilities. Some of the ideas in CODE2, including its manipulation of fragments of constrained natural language date back to Skuce's early research (Skuce 1977).

The first version of CODE2 was created prior to the commencement of this research. Early descriptions can be found in (Skuce, Wang et al. 1989) and (Skuce 1991). A more comprehensive description is in (Skuce 1993a).

During the current research, CODE2 was used for a year in a real industrial project (Skuce 1992a). It was used to build a large knowledge base about a real-time software engineering tool now called ObjecTime™ (Selic and McGee 1991) being developed at Bell-Northern Research (BNR). In addition to helping the BNR team design their product and clarify its conceptual foundations, the BNR project stimulated the addition of many new features to CODE2.

CODE2 was found to be particularly useful to a documentation group at BNR. This application is discussed in (Lethbridge and Skuce 1992b).

CODE2 was used for several other projects, in particular early phases of the Cogniterm project (Meyer, Skuce et al. 1992) where it was used in support of the work of terminologists. Several graduate students involved with Cogniterm used CODE2 in their research: Bowker (Bowker 1992) investigated how the ‘dimensions’ feature (section 3.8) can help the classification process. Miller (Miller 1992) compared how translations prepared using a CODE2 knowledge base as a translator’s assistant compared with translations prepared using conventional term banks. Within the limited scale of his experiments, CODE2 use did indeed help.

Another project, in which CODE2 was concluded to be useful, involved representing the ideas of well-known philosophers (Longeart, Boss et al. 1993).

CODE3 was developed as a replacement for CODE2, after numerous software modifications rendered the latter unmaintainable. CODE3 was written in Prolog in order to take advantage of that language’s declarative syntax and backward-chaining deduction. Prolog’s lack of user interface flexibility caused CODE3 implementation to be abandoned after only six months. Nevertheless, this short period saw the refinement of many new ideas that were carried forward into CODE4.

The lessons learned from experiences with CODE2 and CODE3 are discussed in section 2.1.

Phase 2: CODE4 development (1991-1992)

Until the commencement of CODE4 development, the research consisted of developing hypotheses about what features would be most useful in pursuing the goal of practical knowledge management. When CODE4 development started, these hypotheses were crystallized in the design of its knowledge representation (Chapter 3) and user interface (Chapter 4).

CODE4 development began in January, 1991; and Smalltalk was once again chosen as the implementation language. Several experienced software engineers worked on the system under the direction of the author; and it was given a layered architecture with a high degree of flexibility and reusability. Chapters 3 and 4 expand on the design of CODE4 in detail (although the principles in these chapters are intended to be more generally applicable).

Although the major principles embedded in CODE4's design have not changed, development continues to this day. Recently added features have met the specific needs of various users (e.g. the addition of various printout formats, inference mechanisms etc.)

Phase 3: CODE4 evaluation (1993-1994)

The final stage of this research is the evaluation of CODE4 (discussed in chapter 6) which was performed in order to provide scientific validity to the research. In most related research, the evaluation, if done at all, focuses on assessing certain mathematical properties of a knowledge representation. The practical-knowledge-management goals of this work dictated a different approach however, one for which there are few precedents: The top level evaluation process involved determining how effectively substantial numbers of serious users worked with CODE4. Evaluation tools included a user questionnaire and the metrics[5] discussed in chapter 5.

1.6.2 Summary of the research methodology

The methodology for this research can be summarized as the following four-step process:

- Study systems to understand prevalent problems (discussed in the next chapter).
- Form hypotheses about features to aid practical knowledge management (the knowledge organization features discussed in chapters 3 and 4, and the metrics discussed in chapter 5).
- Develop software (CODE4) to test these hypotheses.
- Evaluate the results (chapter 6)

Chapter 7 discusses the main contributions of this thesis and proposes directions for future research.

[5] The metrics are intended to represent a distinct contribution of this research. They are both evaluated in their own right and assist in the evaluation of other features.

Chapter 2

Other Technologies for Knowledge Management

This chapter discusses several technologies that can be used to manage knowledge of the type discussed in the last chapter.

Three questions are asked about each technology: 1) What is it and how can it be used for knowledge management? 2) What features are related to those in this research? 3) What problems does it have that reduce its practicality?

Section 2.1 discusses systems from the field of artificial intelligence. These include a) the various versions of the Conceptually Oriented Description/Design Environment (CODE) that preceded CODE4; b) knowledge representation systems such as Cyc, KM and LOOM, and c) so-called knowledge acquisition tools. The chapter continues with a discussion of personal productivity software such as spreadsheets and outline processors. Section 2.3 discusses managing knowledge using hypertext and section 2.4 discusses the use of object-oriented CASE tools.

2.1 Technology from the field of artificial intelligence

This section discusses how various tools from the field of artificial intelligence (AI) can be applied to knowledge management.

2.1.1 Descriptions of the tools

This subsection provides overviews of CODE2, CODE3, Cyc, KM, several KL-ONE derivatives and several so-called knowledge acquisition tools. Discussion of advantages and disadvantages of the tools is deferred to sections 2.1.2 and 2.1.3.

CODE2 and CODE3

These systems were developed in the early stages of this research and have since been superseded. For a chronology of their development and a discussion of applications, see section 1.6.1. A major objective of the CODE project from the very beginning was to make knowledge management practical.

CODE2 was a frame based knowledge management system with a simple knowledge representation and a multi-featured graphical user interface. Its most important distinguishing features were as follows:

1. The information (properties[6]) describing concepts was organized in a fixed three-level hierarchy. The first level distinguished among: 1) properties of the 'view' (i.e. the user interface displaying the concept); 2) 'system' properties (e.g. those describing inheritance relationships), and 3) 'main' properties. The second level allowed the user to group properties using various standard or user-defined categories. The third level contained the properties themselves.
2. A number of 'flags' could be used to control the intended meaning of properties and the effect of inheritance. E.g. a property could be declared 'private', meaning that it did not inherit.
3. All data (values) associated with properties were character strings. The user was encouraged to use a special restricted-English language called ClearTalk for values, but could in fact enter whatever he or she desired. The process of drawing networks of interrelated concepts (other than the basic inheritance hierarchy) involved pattern matching to find which character strings contained references to other concepts. There was also a mechanism that allowed users to find and replace occurrences of particular character strings.
4. The user interface involved four mediating representations: 1) Graphs of various relationships which could be freely manipulated by the user; 2) Lists of concepts, indented to show superconcept-subconcept relationships; 3) 'CD-Views' containing all the information about a concept, and 4) Tables showing various properties (shown on one axis) of various concepts (shown on the other axis).
5. There was a 'mask' facility that allowed the user to restrict the display of concepts based on a fixed set of criteria (such as concept name, properties possessed etc).

The main ideas in points 4 and 5 have been carried forward to this research. The ideas in the other points have been largely superseded by improvements.

CODE3's main contribution was to arrange all knowledge as sets of facets. Several facets composed what was called a property occurrence[7] (or 'pocc'), and several such poccs together formed a concept. All of the facets of a knowledge base were stored physically as Prolog clauses which were then operated on by a Prolog-based knowledge engine.

[6] The word *property* in CODE2 was used differently than in CODE4. In the prior system the idea of property and statement was conflated. The word *concept* also had a more restricted meaning – similar to CODE4's main subjects.

[7] Property occurrences correspond to the statements in CODE4.

Cyc

The name *Cyc* (Lenat and Guha 1990) refers to: a) the large knowledge representation system being developed in a multi-year project at MCC; b) the knowledge base being built using the system, and c) the overall project. Cyc, the project, is the largest knowledge representation effort to date and has a variety of corporate investors who hope that Cyc will become a resource for such activities as natural language understanding and the management of large-scale corporate knowledge bases.

The Cyc system has a variety of important features:

- CycL is Cyc's frame-based knowledge representation language. In common with many other such languages, CycL uses two separate syntaxes: one to represent the *structure* of a knowledge base in terms of units (Cyc's term for concept) and slots, and the other to represent rules and constraints.
- CycL is heavily dependent on the top-level ontology that comes with Cyc: This top-level ontology describes such abstract things as **represented thing**, **stuff**, **collection**, **individual object** and **internal machine thing**.
- CycL contains a wide variety of features designed to deal with difficult problems that appear when representing knowledge about everyday things. For example CycL has mechanisms to handle what it calls *subabstractions*, which are context-dependent aspects of a thing (particularly different 'snap-shots' of a thing at points in time).
- Cyc contains a large number of inference mechanisms. To answer queries, Cyc uses a chain of increasingly complex mechanisms that are called successively until one succeeds. Also present are a truth maintenance system, and mechanisms for analogical reasoning (to suggest new rules or facts) and conflict resolution.

Cyc has a variety of user interface capabilities. The primary focus of these is editing the details of concepts (i.e. the values of slots and associated rules). Multiple users can operate on a Cyc knowledge base using these mechanisms from remote displays. Conflicts that result are detected asynchronously and result in the initiation of dialogues between users, mediated by the system.

- The standard Unit Editor (UE) has a large window tiled with several subwindows each displaying a unit's slots. The user modifies the slots by entering declarative statements in CycL's restricted syntax. Despite the greater sophistication of the other interface modalities described below, the emacs-based UE appears still to be the major way that knowledge is entered into Cyc (Porter 1994).
- The Museum Unit Editor allows the user to traverse the network of units. The relationship of one unit to another is shown using nested boxes: The inner box represents the related unit.

- Another interface mechanism called HITS, developed by MCC's human interface lab (Terveen 1990; Terveen and Wroblewski 1992), provides a *collaborative* interface between the user and Cyc. HITS actively assists the user in making knowledge editing decisions and is based on an edit-by-analogy paradigm – using prior editing tasks as models.

The Cyc project has the ambitious goal to capture a sufficiently large percentage of 'common-sense' knowledge such that the system will begin to be able to 'read' external sources of knowledge and thus augment its knowledge base automatically.

KM

KM is a knowledge representation system developed by Bruce Porter and his team at the University of Texas in Austin. The KM project is usually called the 'Botany' project because KM has been primarily used to develop a single large knowledge base, focussing on plant physiology. An early overview of the project can be found in (Porter, Lester et al. 1988). Acker's PhD thesis (Acker 1992) discusses KM's knowledge representation while Eilerts' MSc thesis (Eilerts 1994) describes its user interface.

KM has its foundations in Cyc. The project in fact started by using Cyc but abandoned it for a number of reasons including Cyc's great complexity.

Like Cyc, KM has a frame-based knowledge representation composed of units which inherit slots. Of particular importance are the unit-slot-value *triples*; these represent the 'facts' in the knowledge base and are the attachment points for such things as 'annotations' and 'embedded units'. The latter represent things that are existentially dependent on the particular unit and slot. Like in CycL, details of the values in triples are specified in a Lisp-based declarative notation.

KM's user interface has two main mediating representations: 1) a unit editor much like Cyc's although somewhat simpler, and 2) a limited graph drawing capability[8]. Unlike Cyc, knowledge is largely intended to be entered by a single individual over a long period of time.

An important inference feature of KM is its 'view retriever'. This mechanism generates coherent subnetworks of concepts using pattern matching: The system searches for a predefined pattern of relationships between concepts.

[8] The developers of Cyc abandoned graph drawing after networks became too complex, however they apparently did not invest much time in finding ways of limiting the graphs so as to reduce their complexity.

Classic and LOOM: KL-One derivatives

One of the most important classes of knowledge representation tools are the descendants of Brachman's KL-ONE system (Brachman and Schmolze 1985). These tools, collectively called description logic languages, range from the AT&T's elegant but restrictive Classic (Brachman, McGuiness et al. 1991) to ISI's more expressive but complex LOOM (ISX 1991; MacGregor 1991b). MacGregor (MacGregor 1991a) describes the evolution of this whole family of languages.

Description logic languages share a number of features:

- The primary units of knowledge representation are called *concepts*[9], of which there are two fundamental classes: primitive[10] and defined[11]. Primitive concepts are those which are asserted to exist in order to represent some set of individual things, any conditions attached to them may be necessary but are not sufficient. Defined concepts, on the other hand, have explicit definitions expressed in terms of necessary and sufficient conditions that refer ultimately to the primitive concepts.
- They have a terminological language, describing concepts (the T-Box) and an assertion language (A-Box) describing constraints or facts in the world.
- In addition to inheritance, which is a capability of all frame-based systems, description logic languages include *classification* as a major inference mechanism. By examining definitions (i.e. the properties possessed by a concept), the classification mechanism is able to place concepts at the correct point in the inheritance hierarchy.

Classic remains true to the spirit of the original KL-ONE. It differs primarily in the details of expressions it uses to represent knowledge – the expressions have been carefully restricted to enable its nine major inference mechanisms to operate efficiently.

Whereas Classic remains a knowledge representation system (with a *modelling* language), LOOM has evolved into a *programming* system by adding a 'behaviour language' with such features as actions, methods and production rules. LOOM can be used to directly build expert systems. While the addition of these facilities has made LOOM less restrictive than Classic in terms of what can be represented, it still remains a largely formal system.

9 This meaning of 'concept' has a narrower meaning than in this thesis, where concepts also include the 'individuals' of description logic languages. Concepts in the latter correspond to *types* in this thesis.

10 The primitive concepts of this thesis refer to something quite different: concepts on which knowledge engine computations depend and must therefore have 'handles' to. The primitive concepts of KL-ONE derivatives correspond roughly to concepts in this thesis that are not dependent concepts.

11 The defined concepts of KL-ONE derivatives correspond roughly to dependent concepts in this thesis, although the latter term has a somewhat wider meaning. See the glossary for more details.

Knowledge acquisition tools

Various *knowledge acquisition* tools have attributes that make them candidates for the kind of knowledge management envisaged in this research. The following are some examples:

Tools in the KSSn family (Shaw and Gaines 1991) can be used to draw graphs composed of nodes and links. Mappings can be defined so that these graphs can manipulate knowledge in an underlying knowledge representation such as LOOM.

KARL (Fensel 1993) has a sophisticated graphical environment for editing knowledge. It is one of many tools used in the KADS project (Weilinga, Schreiber et al. 1992) and its underlying representation is a formal logic-based language.

KEATS (Motta, Eisenstadt et al. 1988; Motta, Rajan et al. 1991) is a classic knowledge acquisition system intended for building expert system applications. KEATS was an influence on Shelley (Anjewierden and Weilemaker 1992), designed specifically for the KADS methodology. Both systems support acquiring conceptual structures and executable knowledge from transcripts; both feature extensive graphic support, and neither are intended to be delivery platforms. Both use a straight-forward frame representation.

Other relevant artificial intelligence technology

The technologies discussed above are *systems*, whereas conceptual graphs (Sowa 1984) is an abstract knowledge representation formalism not tied to any particular implementation. However its relatively simple but powerful notations for expressing knowledge are worthy of note in the search for knowledge management technology. Research into conceptual graphs has recognized the need for graphical representations that people can easily use.

KIF (Genesereth 1992) was developed in an attempt to create a *lingua franca* for knowledge exchange. Gruber has extended KIF to create a language and system called Ontolingua (Gruber 1993) that is more suited for the representation of ontologies. While both involve textual languages, they are worth noting because there is a movement to use them as a common denominator for knowledge exchange; any knowledge management tool might eventually need to be able to translate to and from them[12].

2.1.2 Features of the AI-based tools found most useful in this research

The frame-based representation

The most important AI principle used in this research is the frame-based knowledge representation paradigm. Particular frame-based features used in this research include:

[12] Preliminary steps have been taken to do this as a side-branch of this research: Technology to support this is discussed in section 3.13.

- The reasonably wide definition of the concept or unit in Cyc and KM (e.g. the fact that slots and instances are concepts). However, in this research the definition has been made even wider.
- The hierarchies of slots in Cyc and KM. Again, these ideas have been expanded in this research.
- The triples and embedded units in KM.

Certain graphical representations

Conceptual graphs and user-interface oriented tools like KSSn and KARL have graphical capabilities of use in this research. These languages or tools recognize the need to abstract away from highly mathematics-based representations of knowledge, although such representations may be present ‘underneath’.

2.1.3 Limitations of the tools for practical knowledge management

Complexity of the languages and ontologies

Many AI tools have limited applicability to the kind of knowledge management described in chapter 1 primarily because they are only *intended* to be used by computer specialists.

With a few exceptions (CODE2 and some knowledge acquisition tools such as KSSn), the AI-based tools described above are designed first and foremost with inferential or representational objectives in mind. Examples of such objectives include the ability to perform classification or other inferences, the ability to represent particular complexities that arise in some domain or the ability for the representation to conform to certain formal rules of mathematical logic. User interfaces for these systems are generally developed as a necessity and with expert users in mind.

KL-ONE derivatives, for example, emphasize automatic classification by analysing how definitions and descriptions subsume each other. This has two effects contrary to the goal of practical knowledge management: 1) It means that the tools are severely limited in their expressivity in order for subsumption to be computable, and 2) It constrains them to using a somewhat sophisticated language with many functions and operators to learn. While classification may be useful for certain automated applications, it is probably of no benefit to the user performing knowledge management because such a user must know the properties with which to specify the definitions and descriptions.

Users of Cyc, as another example, must master its ontology and representational language. Both are large, complex, somewhat confusing and require the user to accept many questionable representational decisions (Skuce 1993b).

User interface weaknesses

Where graphical interfaces of AI-based tools are well-developed, they tend to have one or more weaknesses of the following kinds that restrict their general purpose use:

- Loose coupling to the underlying representation (e.g.knowledge acquisition tools that can be used to input only certain types of knowledge but not to convey all of it)
- Much usage of prompts (i.e. modal dialogs)
- A graphical notation that is so heavily tied to the representation that user cannot sketch knowledge, but must understand the representation in detail.
- Restricted abilities to navigate or browse large knowledge bases and extract subsets of knowledge. This latter weakness is the most serious.

Difficulties dealing with names

AI-based tools usually ignore the issue of naming. Naming a concept is left up to the user who must choose a *unique* name that is distinct from other names in the knowledge base and must generally stick with it once chosen (because references are made to the concept using that name). Names are usually tightly connected to the implementation of the system; they are unique and sufficient identifiers for concepts. This has several consequences:

- Choosing concept names becomes a critical activity for the user. In order to prevent confusion, the user: 1) must be consistent with existing naming conventions; 2) must avoid names already used, and 3) must anticipate the addition of other concepts with similar names.
- The user often cannot choose names that mirror those in natural language. Where a natural language name has several meanings, the user of most AI-based tools is forced to invent a new name. Where several natural language names are synonyms for the same thing, the user must choose among them.

The designers of Cyc acknowledge the problem and state that natural language names for all units can be specified in the values of a particular slot. They state that eventually, all manipulation of the knowledge via a natural language front-end would use such slots and the ‘internal’ names could be dropped. However, at the current time knowledge editing procedures continue to use the internal names.

CODE2’s facilities to change all occurrences of a particular string reduced the problem of poor name choices becoming entrenched; however, CODE2 had no way to deal with synonyms and homonyms, which occur frequently.

2.2 Personal productivity software

In sharp contrast with most AI-based tools, software such as spreadsheets and outline processors is expressly designed to be used by non-computer-experts. In this thesis, tools of this kind are called *personal productivity software*. Such software can be used for conceptual knowledge management, even though it was originally designed for applications of a somewhat different nature (i.e. managing numeric data or document structure). For example, one of the early industrial users of CODE2 was the Aluminum Company of America (ALCOA). They had previously used spreadsheets to represent conceptual information in the field of metallurgy, but switched to CODE2 when they saw its benefits.

2.2.1 Description of the tools

Since readers are undoubtedly familiar with tools in this class, only their key features will be noted.

Spreadsheets

Spreadsheets are graphical environments with two-dimensional arrays of cells that contain either data or formulas. When a cell is updated anywhere in the spreadsheet, all cells that depend (directly or indirectly) on that cell are updated.

Outline processors

Outline processors permit users to rapidly arrange the hierarchy of topics in a document. They include facilities to promote and demote topics, to move topics from place to place and to hide and expand sets of topics based on level of depth in the hierarchy. One of the most widely known outline processors is that which comes with Microsoft Word™; others, such as More™, are sold as stand-alone programs.

2.2.2 Features of the tools that were found most useful in this research

The following are useful knowledge-management features found in personal productivity software but typically absent from AI-based tools. The features have been largely adopted in this research.

Large amounts of information displayed in a convenient manner

Personal productivity tools have techniques to maximize the amount of information that can be displayed or made accessible to the user. Displaying much information at once leverages the user's ability to process information[13], since the human visual system is very good at

[13] As opposed to relying on the computer to do the processing as is the case in most AI-based tools.

scanning for interesting facts or noticing patterns. To prevent confusion however, very simple structures are used: tables for spreadsheets and indented lists for outline processors. These structures can convey much knowledge, but are simple enough to prevent users from getting lost when navigating large volumes of data.

The non-modal graphical user interface

To allow users to rapidly manipulate data and headings, most personal productivity software has been developed using graphical user interface techniques. Users can directly select and edit what they see on the screen. Where possible, such software avoids entering 'modes' where the user's actions are limited.

Easy extensibility

The structure of information in both spreadsheets and outline processors can be readily extended by the user. Simple commands are available to add rows or columns, and the user merely types new text to add to existing rows or to extend the document beyond its current bounds.

The ability to work on multiple files at a time

Most personal productivity applications permit users to load several 'documents' at once. This capability facilitates cutting and pasting between documents. It also can help the user with several concurrent tasks by reducing the time lost in repeatedly quitting and reloading.

Automatic updating of multiple windows

Information can be displayed in many windows. Updates to the information that are made in one window are automatically reflected in all others. The form of the representation can also vary widely from window to window: Updating a spreadsheet might result in the automatic update of a graph. Updating an outline might automatically update the full text of the document displayed in another window.

The ability to analyse 'what-if scenarios'

In spreadsheets, the user can make a selection or type a value into one cell and observe how changes ripple through the interface.

Absolute and relative references

In spreadsheets, cell formulas can refer to particular cells or to the cells that happen to be at a certain relative offset from the cell containing the relative formula. This facility greatly facilitates specifying complex patterns of interconnection among cells.

Easy facilities for extracting information

Most spreadsheets include the ability to specify a set of criteria in order to extract a subset of the data. They also have facilities so that all cells fulfilling certain criteria can be ‘selected’. Outline processors have facilities to control which subhierarchies are displayed and to what depth. All personal productivity software additionally has simple text search mechanisms.

2.2.3 Limitations of the tools for practical knowledge management

For the purposes of this research, the advantageous features listed in the last subsection are considered to be generally *necessary* for practical knowledge management, but they are not *sufficient*. The following are very severe limitations:

Limitations on the ability to represent concepts

While all the facts about a concept can be listed in a row or column of a spreadsheet, or in a subhierarchy in an outline, there is no way to treat concepts represented thus as discrete units that can be independently manipulated. In other words, personal productivity software lacks the key advantage of the frame representation of many AI-based tools.

Lack of inference capabilities

Arbitrary formulae can be used to perform computations in spreadsheets, but every required inference must be explicitly programmed by the user. In particular, inheritance is lacking. Outline processors typically have no inference capabilities.

2.3 Hypertext systems

Hypertext (or hypermedia) is another technology with potential application to knowledge management. It encompasses a wide variety of tools and techniques, but its defining feature is a network of nodes, where each node contains a useful chunk of information formatted for use by end-users. Another important feature is facilities to navigate around the nodes (Nielsen 1990). Hypertext systems have been found useful for knowledge management (Barman 1992). Also, hypertext-like facilities can be found in a wide variety of general software. A major example of the increasing importance of hypertext is the World Wide Web (Berners-Lee et al. 1994) – a standard that can be used to turn virtually any on-line source of information into part of an Internet-wide hypertext ‘document’. A structure for formally describing hypertext systems can be found in (Halasz and Schwartz 1994).

2.3.1 Features of the tools that were found most useful in this research

The ability to navigate through complex networks

Hypertext systems are designed so users can explore by clicking on a wide variety of ‘anchors’ that take them to nodes where they can find more information. An important belief is that in knowledge management users should always be able to find out more or say more about whatever is displayed.

Simple end-user oriented displays of informal information

A key to the design of good hypertext is that each node should be a discrete unit of information that is specifically designed and formatted for end-user. A practical knowledge management tool needs a similar capability.

2.3.2 Limitations of the tools for practical knowledge management

The biggest limitations of hypertext *per se*, are the same as the limitations of personal productivity software: The lack of a frame-based conceptual structure and the lack of inference facilities, particularly inheritance. However, hypertext should be considered more a set of ideas that can be applied to software in general rather than a specific set of tools.

2.4 An object-oriented software engineering tool

Object-oriented analysis has strong similarities to the acquisition or modelling part of knowledge management. Both involve identifying and modelling things in a domain and the relationships among those things. One of the most popular object-oriented analysis methodologies is Rumbaugh’s OMT (Rumbaugh, Blaha et al. 1991). OMT emphasizes carefully distinguishing a) the analysis of the domain and problem, from b) the design of the system. OMTool (GE 1993) is a CASE tool that supports the OMT methodology and could be applied to the knowledge management task discussed in this thesis.

2.4.1 Features of OMTool found most useful in this research

Graphical layout and editing of knowledge

In common with certain AI-based tools discussed above, OMTool allows a user to design a network of concepts by placing boxes in a window and linking them together. As is the norm in object-oriented software engineering, concepts are called *classes*[14]. The most

[14] At implementation time there is a justification for saying that concepts are qualitatively different from classes; classes can then be considered as actual software components containing code and other information. At analysis time, however, what the software engineer specifies as a class is really a conceptual entity that *might* be implemented as a class.

fundamental relationship in which OMTool users may place classes is the generalization relationship – in other words to specify inheritance.

OMTool's interface is reasonably non-modal. The user can freely rearrange the elements of the class diagram, constrained only by the need to prevent classes from overlapping each other. Users can type over any text in order to change it and are not forced to name classes.

Associations, attributes and operations

OMTool carefully distinguishes between *associations* and *attributes*. The former represent relations between classes while the latter represent so-called intrinsic properties of a class, properties whose associated data are not instances of another class. In this research, the terms formal and informal are used when describing this distinction. In addition to associations and attributes, OMTool allows users to specify *operations* that represent actions which can be performed on instances. Associations, attributes and operations are all intended to inherit from superclasses to subclasses with no exceptions.

Annotations.

Annotations in OMTool are descriptive information about a class, association etc. They do not inherit.

Classes and instances

OMTool makes a distinction between classes[15] and instances by diagraming each differently. However, the process of attaching attributes, associations and annotations is the same for both. The attributes of instances must be 'grounded' in the sense that they must contain actual character strings or numbers, not data types. The instantiations of associations are called *links* and must also be grounded by being made to point to other instances.

Disjointness

OMTool by default considers all the subclasses of a given superclass to be disjoint, i.e. that they share no common instance. In order to permit multiple inheritance, however, the least common superclass of two classes that share a subclass must be declared 'disjoint'.

Discriminators

According to Rumbaugh, a discriminator "is an attribute whose value differentiates between subclasses". In other words it is possessed by several subclasses, but has a distinct value in each.

[15] Classes correspond to *types* in this research.

2.4.2 Limitations of OMTool for practical knowledge management

Limitations of the representation

OMTool's underlying representation has many features envisaged in a practical knowledge management system, i.e. it has powerful structuring facilities, but a simple easy-to-use syntax. However there are some significant gaps:

- Like most tools discussed in section 2.1, OMTool requires the names of classes to be kept distinct. Also, there is no way to specify information about names.
- There is a lack of flexibility and extensibility. Users must manipulate a fixed set of diagrammatic entities that each have their own syntax and semantics; i.e. users must live within the bounds of the system's *ontological commitment*. For example, because the entities are not treated uniformly: 1) attributes can not have 'second order' attributes attached to them, and 2) it is not possible to categorize associations.
- Although inheritance is part of the conceptualization in the OMT methodology, its results are not actually *displayed* by OMTool. The user is expected to know that in the eventual implementation, an attribute or association possessed by a superclass will be present in subclasses. To ascertain the set of features possessed by a class, the user must manually search up the generalization hierarchy.

Limitations of the user interface

Despite having many useful representation facilities, OMTool's user interface severely constrains its use for general-purpose knowledge management. The root cause of this appears to be the OMT methodology which does not require the ability to deal with highly complex patterns of knowledge or to perform any inference to discover latent information. OMT stresses several things: 1) Users should only include those things (classes, associations, attributes etc.) which bear upon the problem at hand; 2) users should not initially record redundant information (i.e. information that can be derived from other information), and 3) all fundamental relationships (e.g. part-of) should be explicitly diagramed (i.e. not derived through an inference mechanism). In other words, one should record information if and only if it is strictly necessary.

The above limitation severely restricts OMT's ability to be used for knowledge management tasks where it is important to record a wide variety of facts about concepts (whatever comes to mind), and to build very deep inheritance hierarchies. OMTool could not be used for such tasks: It becomes difficult to use when inheritance hierarchies exceed a depth of about 4 and when there are more than about 10 attributes or 5 associations per class. It is especially difficult to use when a large number of rather disparate classes have complicated sets of associations with each other.

Other major problems with OMTool's user interface are

- Browsing facilities are weak.
- There is no way to automatically draw diagrams or extract portions of diagrams.
- Only one set of class diagrams can be loaded at once.

2.5 Summary

The four technologies described above all have much to offer, but each also has limitations. Clearly, what is needed is an integrated approach where key features are borrowed from each technology in order to minimize the number of remaining problems.

Figure 2.1 presents an evaluation of the technologies based on how they deal with the knowledge management problems discussed in section 1.4. For each problem, the technologies are ranked on a scale of 1 to 5 where 1 means the problem is prevalent, 5 means the problem is largely solved and 3 is neutral (the problem is neither prevalent or solved). The following are a few observations about figure 2.1:

- Some problems have different partial solutions in different technologies. For example both AI-based tools and object-oriented CASE tools can be used to easily make different kinds of useful distinctions (as discussed earlier). Similarly, personal productivity software and hypertext have different but useful methods of enabling users to extract knowledge.
- Different tools within the technologies vary widely. The evaluations given are intended to be averages.
- For some problems none of the technologies pose solutions. For example naming is poorly addressed in general. This research attempts to find a new solution to this problem.
- AI-based tools in general have fewer prevalent problems, but personal productivity software generally poses more useful solutions.
- Hypertext is only rated where it contributes solutions because it is seen as a technology for enhancing software user interfaces in general.

	AI-based tools	Personal productivity software	Hypertext	Object oriented CASE
I-1: Categorizing	4	3		3
I-2: Naming	2	2		3
I-3: Making distinctions	3	1		3
I-4: Understanding effects	3	4		1
I-5: Extracting knowledge	3	4	4	2
I-6: Handling errors	4	2		1
A-1: Application restriction	3	4	4	1
A-2: Expertise restriction	2	5	4	3
A-3: Size restriction	3	2		1
A-4: Individual-use restriction	3	1		1

Figure 2.1: How various technologies address knowledge management problems. The scale used is: 1=the problem is significant with most such tools; 3=some tools provide some facilities to solve the problem; 5=the technology contributes significantly to problem solution. The problems are discussed in section 1.4.1. Hypertext has blank fields because it is largely an enabling technology that adds capabilities to tools. Note that problems I-2, I-3, A-3 and A-4 are not really solved by any of the four technologies.

Chapter 3

Knowledge Representation for Practical Knowledge Management

This chapter describes CODE4-KR, a frame-based knowledge representation designed to improve the practicality of knowledge management. The first section summarizes the main features of CODE4-KR. Subsequent sections discuss these features in more depth. It is important to note that most of the details discussed in this chapter *do not have to be understood by the average user*.

3.1 Introduction

As with several other knowledge representations, the basic unit of knowledge in CODE4-KR is called the *concept*. Section 3.2 describes the various classes of concept contained in a CODE4 knowledge base; later sections in the chapter describe in detail the kinds of operations that can be performed with concepts, as well as their intended semantics.

Section 3.3 describes how, in the design of CODE4-KR, paramount importance is placed on practicality and expressiveness, sacrificing rigid semantics and built-in inference capabilities. In the subsequent sections, particular features of CODE4-KR are described first from a practical point of view and then from the point of view of intended semantics.

Section 3.11 shows how the facilities provided by CODE4-KR give rise to a set of richly interrelated organizing techniques for knowledge. Then, section 3.12, compares CODE4-KR with several other knowledge representations. Finally, section 3.13, discusses how most of the ideas in this chapter form an *abstract schema*, and how this schema is made concrete in various ways.

3.2 An overview of concepts

It is critically important to distinguish a concept from the thing the concept represents (Regoczei and Hirst 1989). A concept is a piece of knowledge inside a CODE4 knowledge base[16], whereas most things are *not* inside CODE4. While this may seem an obvious dis-

[16] Or inside a brain or collective consciousness or perhaps some other KR system.

tinction to many, it has been frequently found that this distinction is not clearly made: the two ideas are conflated. Although for simple knowledge representation tasks this conflation causes few problems; unless one is conscious of the distinction, it may be difficult to understand the significance of some of CODE4-KR's features. When this thesis uses phrases like: 'the concept of **car'** or 'the **car** concept', it is *always* referring to things inside minds or knowledge bases, and bold face is used for their labels. On the other hand, whenever bold face is not used (e.g. when talking about cars or a car), the reader can be sure that the thesis is *not* referring to a concept but rather the represented thing. Figure 3.1 shows the relationship between concepts and the things they represent; it includes some special classes of concept that are discussed in the next few paragraphs.

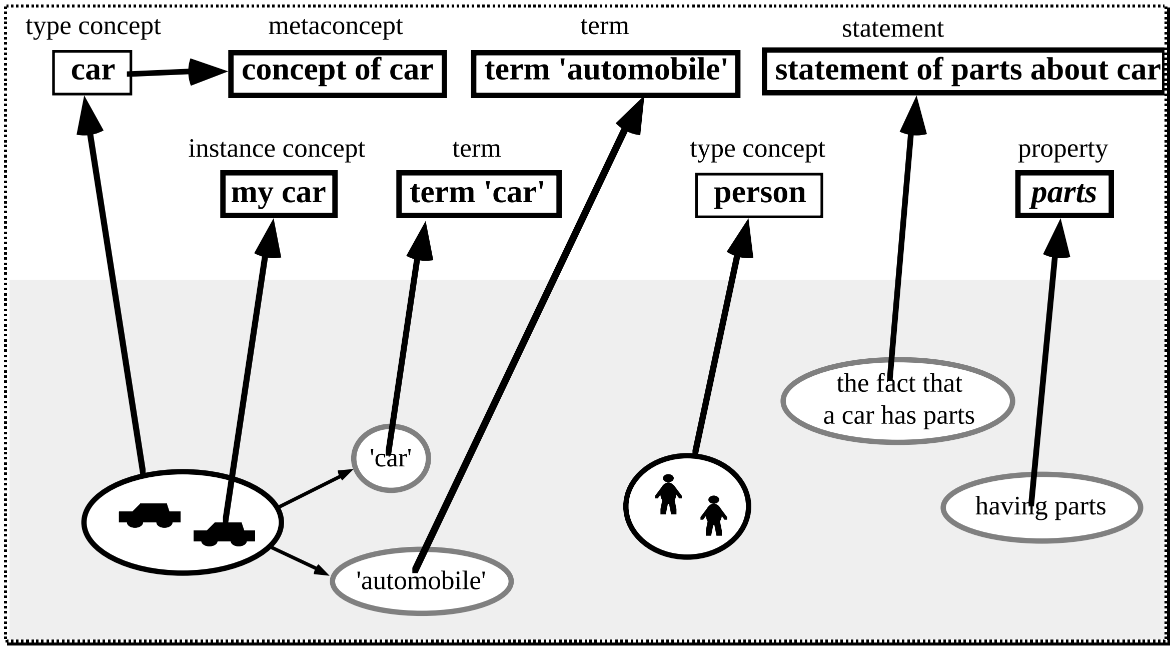


Figure 3.1: Major classes of concept and what they represent. At the top are concepts; in the grey area are things which are not concepts. Arrows link things in general to other things that represent them; bold arrows link things to their concepts. Of the things in the grey area (none of which are concepts), those with dotted outlines are abstract in nature and others are concrete.

3.2.1 Classes of concept

There are several different classes[17] of concept in CODE4-KR. This subsection introduces types, instances, properties, statements, facets, terms, metaconcepts, primitive concepts,

[17] To prevent confusion, the word 'class' is used here as a near-synonym for 'type'. This is done because the discussion is at the *meta-level*, i.e. it is conceptualizing concepts. Confusing sentences like 'types are a type of concept' are replaced by sentences like 'types are a class of concept' – the word *class* is thus used only at the meta-level.

user concepts and main subjects. Subsequent sections as well as the glossary provide more information.

A concept represents a thing: either the collective idea of a set of similar things (represented by a *type* concept, for example the concept of **car** in general) or a particular thing (represented by an *instance* concept, for example the concept of **my car**). See section 3.4 for more about these two classes of concept. The thing represented may be abstract or concrete, real or imagined; it may be an action, a state or in fact anything one can think about[18]. Of course, it follows that concepts are things too since one can think about them (but not all things are concepts as figure 3.1 shows).

Each knowledge base has a most general type concept (the top of the inheritance hierarchy), of which all other concepts are *subconcepts*. By convention this concept is labelled ‘thing’ although the label can be changed by the user. Figure 3.2 shows a simple inheritance hierarchy, containing the concept of **thing** and several other type and instance concepts. In order to keep knowledge management simple for the end user, the only ontological commitment that users are *required* to understand is that there is a single top concept in each knowledge base. Users can give this any name they wish.

The above ideas are similar to those in most other frame-based knowledge representations (the terms ‘unit’ or ‘frame’ are sometimes used for ‘concept’). CODE4-KR departs from the norm, however, in the generality and uniformity with which it treats concepts: Most knowledge representations define *slots* as distinct entities that are inherited by concepts. In CODE4-KR, *properties* perform this role, but properties are just another kind of concept (as in Cyc and KM). In this thesis, the labels of properties are shown in both bold face (because properties are concepts) and in italics. Example properties are ***size***, ***parts*** and ***parents***. Properties are discussed in section 3.5.

In a similar manner, most knowledge representations have some notion of the *association* between a property and a particular slot in a frame. For example in KM the concept-slot-value *triple* fulfils this role. In CODE4-KR such things are called *statements*, because the notion has been deliberately made close to the linguistic notion of a statement. A CODE4-KR statement must have a *subject* (some concept) and a *predicate* (some property); it usually has more. Most statements have a *value* that designates what the subject is related to by way of the predicate (corresponding to the direct object in linguistic terms). Statements are discussed in detail in section 3.6, but for now the important idea is that statements, too, are concepts. Statements may recursively have statements about themselves, called *facets.*

[18] To many people, the word ‘thing’ connotes an *object*, however English speakers use the word some*thing* to refer to anything: e.g. actions: ‘We must do something’ or properties ‘Something bothered me about that man’.

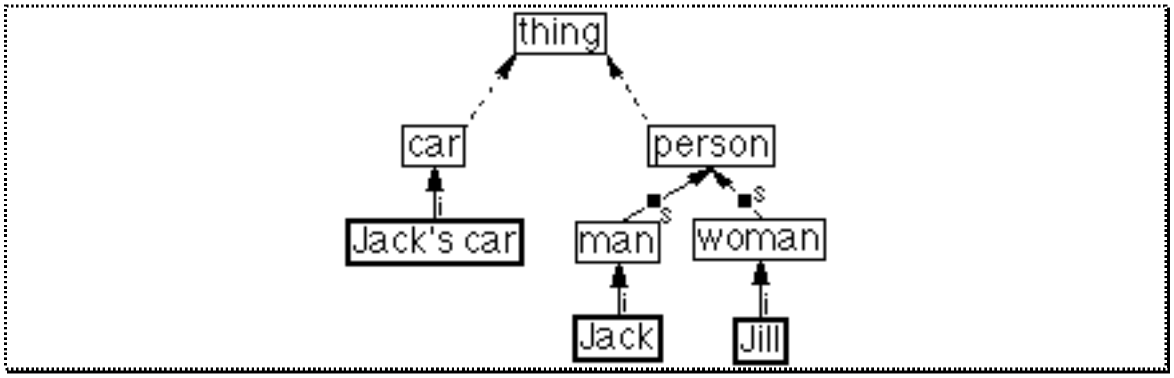


Figure 3.2: A simple inheritance hierarchy. Each box represents a concept. The arrows represent the 'is-a' relation between concepts. Bold boxes (around ***Jack's car****,* ***Jack*** *and* ***Jill****) represent instance concepts while non-bold boxes represent type concepts. An 'i' designates instance links; an 's' designates subtype links. As discussed in section 3.7, the labels in the boxes come from term concepts associated by the user with each concept. For example* ***Jack's car*** *(not intended to be an intensional reference) has a term 'Jack's car'; it may also have other (undisplayed) terms such as 'Car with serial number 12875'. This figure, in which dotted lines indicate that details have been suppressed, was drawn by CODE4.*

By virtue of being full-fledged concepts, it follows that all properties and statements: a) participate in the inheritance hierarchy, b) inherit properties, c) can be the subjects of statements and d) can be referred-to in the values of statements. They behave as concepts do in general and are special only in the sense that they have *additional* semantics associated with them. Figure 3.3 lists some of the key features that all concepts have in common.

CODE4-KR currently implements two other special classes of concepts: *terms* (section 3.7) and *metaconcepts* (section 3.8). Terms represent linguistic entities (words or symbols) that people use to refer to concepts. CODE4-KR is unique in treating terms as full-fledged concepts and allowing a concept to have zero, one, or more than one term. Metaconcepts represent concepts themselves; statements with a metaconcept as subject encode metaknowledge. For example the concept **car** might inherit properties ***maximum speed*** and ***weight***, whereas the metaconcept **concept of car** might inherit the properties ***English description***, ***subconcepts***, ***terms***, ***date entered into this knowledge base*** etc. CODE4-KR is unusual in providing explicit metaconcepts to which are attached those properties that in other knowledge based systems have to be tagged as 'non-inheriting'.

When a knowledge base is created, some concepts are automatically created. For example the top concept commonly named 'thing' is always present and cannot be deleted. This is an example of a *primitive concept*. There are a number of other primitive concepts in a CODE4 knowledge base and these are discussed in section 3.9.

	Type concept	Instance concept	Property	Statement	Term	Meta-concept
Specialized kind of	Concept	Concept	Instance concept	Instance concept	Instance concept	Instance concept
Represents	Collective; set	Particular thing	Relation	Fact about something	Word; symbol	Concept
Example	**person**	**Jack**	***parts***	**Jack has a mother**	**term 'jack'**	**concept of Jill**
Participates in the inheritance hierarchy	•	•	•	•	•	•
Inherits properties	•	•	•	•	•	•
Can be the subject of statements	•	•	•	•	•	•
Can be referred to in the values of statements	•	•	•	•	•	•
Can have associated terms	•	•	•	•	•	•
Almost always has an associated term	•		•			
Can have an associated metaconcept	•	•	•	•	•	•
Can be the predicate of statements			•			
Inherited by concepts			•			
Has a subject				•		
Has a predicate				•		
Can have a value				•		
Has at least one concept that it *means*					•	
Has a concept that it represents						•

Figure 3.3: Important properties of concepts. Note that these are properties of the various types of concepts themselves, not properties of the things represented by the concepts. Many more details can be found later in this chapter. This kind of information can be extracted from CODE4 (see section 4.2.4), although it has been reformatted for presentation here.

If the primitive concepts, statements, metaconcepts and terms were to be ignored, what would be left would be those types and instances that have been explicitly added by the knowledge base developers. These are called *user concepts*[19]. Those user concepts that are the subjects of statements are called *main subjects* (the set of main subjects does not include those user concepts about which nothing has been said).

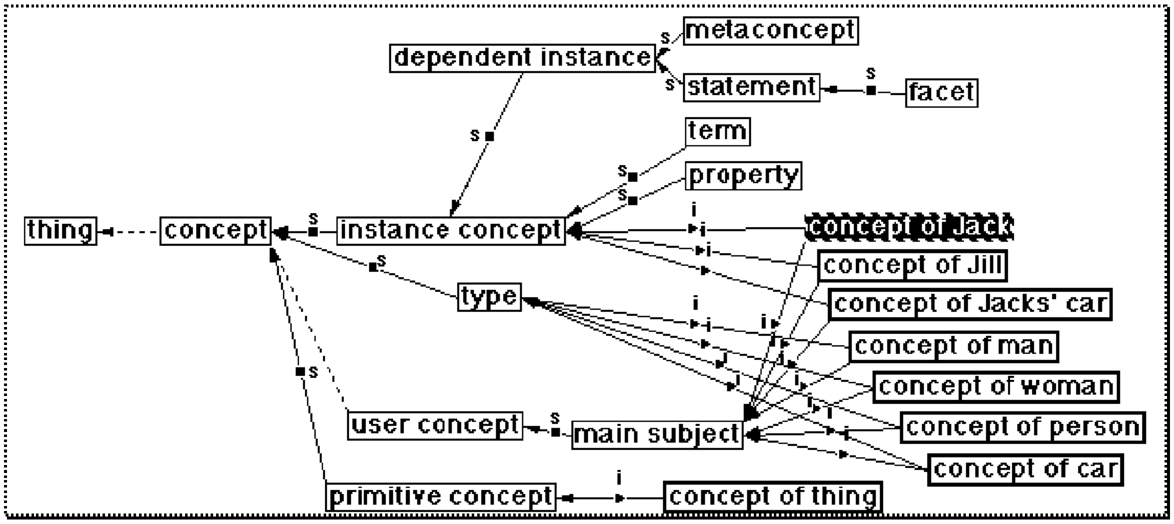


Figure 3.4: An inheritance hierarchy of the most important classes of concept. Note that the diagram is categorizing the kinds of things that one might find in a knowledge base (i.e. concepts themselves) and not things in the world in general. All the instance concepts shown (with bold borders and 'i' links to their superconcepts) are in fact metaconcepts because they are 'concepts representing particular concepts'. They are meta-concepts for the concepts shown in figure 3.2. This subtlety may require deep thinking for many people to appreciate, however it does not need to be understood to make effective use of CODE4. This figure was drawn by CODE4.

All discussion of concepts in this thesis refers to concepts in general; however CODE4 users are often referring only to user types. In fact, in CODE2 the use of the term 'concept' was restricted to refer to what are now called user types and user instances. However, the terminology was changed when the importance of treating all knowledge units uniformly was recognized.

Figure 3.4 summarizes this section by showing an inheritance hierarchy of classes of concept. Figure 3.5 gives a fuller categorization of the classes of concept; details are found in the glossary and will be explained later in the chapter.

[19] Although statements, terms and metaconcepts are created during knowledge base development, these are implicitly added rather than explicitly. For example, a term is added when a concept is named. A metaconcept is implicitly added when any other concept is added etc. These subtleties will become clearer later in the chapter.

3.2.2 Knowledge management problems addressed by the way CODE4-KR organizes concepts

The last subsection introduced a wide variety of things that are called *concepts* in CODE4-KR. The following are some of the partial solutions to knowledge management problems provided by this way of organizing knowledge (the problems are those listed in section 1.4). Note that *particular* classes of concepts handle additional problems, but discussion of these is left until later in the chapter.

- **Problem I-3**: Making distinctions. While most users will not need to work at the meta-level, the various classes of concepts allow sophisticated users to make key distinctions when necessary. Of particular importance are the distinctions between, terms, metaconcepts, statements and other concepts.
- **Problem I-6**: Handling errors: Distinguishing metaconcepts and terms from other concepts helps prevent serious representational errors that can occur in other knowledge representations.

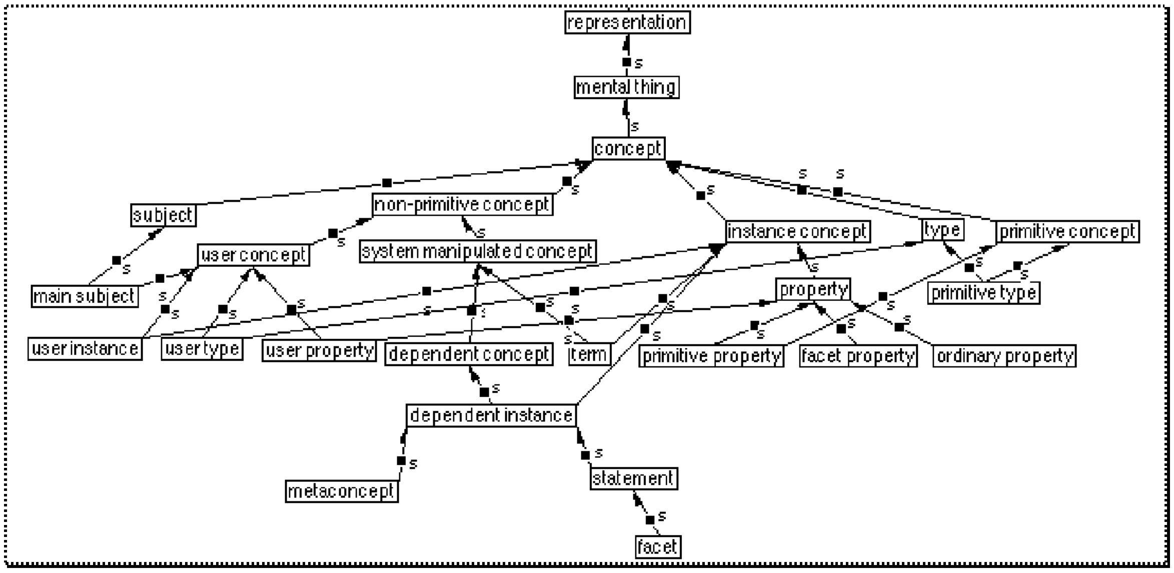


Figure 3.5: More of the classes of concept in CODE4-KR. This figure was drawn by CODE4 from the same knowledge base as figure 3.4.

- **Problem A-2**: Expertise restriction: Calling all units of knowledge *concepts* sets the foundation for a greatly simplified user interface. As users gain expertise they come to know that anything (i.e. any concept) they find in the representation has certain attributes such as a superconcept, a metaconcept, and statements about which it is the subject. They also know that they can always perform certain operations, such as examining the properties of any concept.

The last point appears to be the most important benefit.

3.3 Practicality vs. semantics and expressiveness vs. inference

In the design of CODE4-KR, several tradeoffs have been made. The focus has been on providing practical ways of organizing knowledge and ensuring a high degree of expressivity. To some extent, these emphases mean sacrifices in the ability to provide automatic inference capabilities.

In the discussions of aspects of CODE4-KR that follow, ‘practical effects’ are distinguished from ‘intended semantics’: Each is discussed in a separate subsection of each section.

Practical effects sections

Since CODE4 is designed to be a practical tool for managing knowledge, the primary concern is maximizing expressiveness. In order to solve the problems discussed in section 1.4, users must be able to represent what they want – they must be able to organize knowledge in useful ways and to do this rapidly and intuitively.

When describing CODE4-KR then, what are of paramount importance are the practical effects of an action with regard to: a) the structure of the knowledge base, and b) what constraints are imposed on subsequent actions.

Intended semantics sections

Of secondary importance are the *intended semantics* of operations or structures. These guide the user in interpreting knowledge, but the user may disregard them if he or she so chooses. The user is free to maintain his or her *own* model of what the various structures in CODE4 mean.

To help specify intended semantics, users may make their own annotations (such as special facets attached to statements[20], or statements attached to metaconcepts[21]). Users may also add their own syntactic layers[22] to CODE4.

If the user strictly follows the intended semantics (which suggest a mapping from CODE4-KR to predicate calculus and set theory) then the benefits include: 1) the ability to have knowledge accurately translated to other representations, and 2) a confidence in the consistency and soundness of knowledge.

[20] Discussed in section 3.6.

[21] Discussed in section 3.8.

[22] Using informal values, as discussed in section 3.6.

On the other hand almost all the users who participated in this research[23] cared little about logic or set theory; they just wanted to be able to arrange ideas with the help of useful abstraction mechanisms. The fact that operations in CODE4-KR are designed to have predictable effects on the structure of the knowledge base is sufficient to give users confidence in their *own* interpretations of what they see.

Practical effects and intended semantics of inference mechanisms

As with structures and operations, inference mechanisms in CODE4 (such as inheritance and delegation) that are built into CODE4 are described primarily in terms of their practical effect. The amount of inferencing available in CODE4 is limited due to the fact that many inference mechanisms would have to rely on users adhering to an intended semantics if the results were to have any meaning (i.e. "garbage in, garbage out"). Users can add their own inference layers on top of CODE4-KR if they wish; this involves programming.

Summary

To summarize the above: The consequences of actions performed in CODE4-KR are explicitly defined in terms of how structures change. An intended interpretation of those structures is provided, but the user is not forced to stick to this interpretation.

Each of the following sections discusses an aspect of CODE4-KR. Each discussion is divided into: a) "practical effects", which describes the structures that exist and what happens when certain actions are taken; b) "intended semantics", which indicates what the structures are normally intended to represent; c) "observations on use", which discusses certain pragmatic issues that have arisen out of experiences with CODE4, and d) "problems addressed", which discusses how the problems listed in section 1.4 are addressed by the features presented.

3.4 Types vs. instances

3.4.1 Practical effects

One of the most fundamental partitionings of concepts is into type concepts and instance concepts. A concept is a type concept or an instance concept, but never both at the same time[24]. Type concepts are often just called types, and instance concepts are sometimes just

[23] It might be argued that the *only* users willing to use CODE4 were those that could do their work without mathematical rigour. However, many users would not have been *able* to use CODE4 if it required mathematical rigour, but they very much appreciated the organizing mechanisms provided by CODE4. CODE4 thus fills a significant niche that is largely empty.

[24] In this, CODE4 differs from many other knowledge representation systems, e.g. Cyc.

called instances. When the word ‘instance’ is used alone, it is always referring to instance *concepts*, although CODE4 users sometimes confuse this latter abbreviation with the things the instance concepts *represent*.

Flexibility about whether concepts are instances or types

CODE4 is unusual among knowledge based systems in that the user’s choice about whether a concept is a type or an instance has little effect on the system’s behaviour. Furthermore, the user is provided with simple commands to switch concepts between being types and instances, and vice-versa.

From a purely practical point of view, the main effect of declaring a concept to be an instance is that it cannot have subconcepts – i.e. nothing can then inherit from it. All types can have subconcepts, whereas instances are only found at the bottom of the inheritance hierarchy.

By default, a new user-created concept is created as a type concept unless all its siblings are instance concepts (in which case it is assumed that the user is most likely to be wanting to add another instance concept). A user can turn any non-primitive instance into a type (primitive instances are discussed in section 3.9). On the other hand, a user can only turn a type into an instance if it has no subconcepts.

Disjointness

A useful operation available in CODE4, but typically used only by experts, is to declare two or more concepts *disjoint* from each other. The practical effect of this is to declare that no two of them can have a common subconcept (i.e. no concept can be a direct or indirect subconcept of any two or more). CODE4 prevents any attempt at making a common subconcept of two disjoint concepts. The disjointness property is symmetric: if one concept is disjoint from another, then that other is disjoint from the first. Also, if the concept of **A** is disjoint from the concept of **B**, then the concept of **A**’s subconcepts are automatically disjoint from the concept of **B**’s subconcepts. Note that the disjointness property is attached to a concept’s metaconcept – i.e. it relates concepts, not the things represented by those concepts.

All instance concepts are considered to be inherently disjoint from each other. Any two concepts in a superconcept-subconcept relation to each other or that share a common subconcept (directly or indirectly) are inherently nondisjoint. By default, any two type concepts are potentially nondisjoint unless declared disjoint (directly or indirectly[25]).

[25] Concept **A** would be indirectly declared disjoint from concept **B** if, for example one of concept **A**’s superconcepts were declared disjoint from concept **B**.

The inverse of the operation to make two concepts disjoint is the operation to make the concepts potentially non-disjoint. This can only be applied to type concepts that are not in a superconcept-subconcept relation and that have no common subconcept.

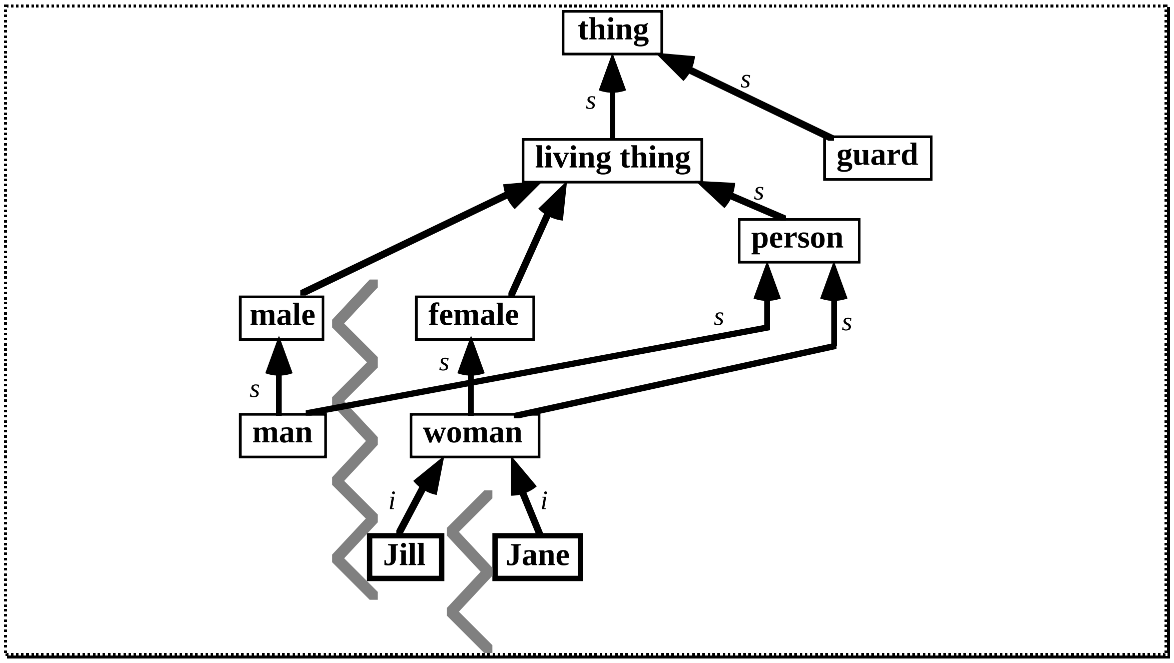


Figure 3.6: The effect of declaring concepts to be instances or disjoint: The jagged lines separate groups of concepts that are disjoint from each other. See figure 3.7 for a semantic interpretation of disjointness, and to see how this diagram partitions things into 18 possible types.

Figure 3.6 illustrates some of the effects of declaring concepts to be instances, or to be disjoint. The concepts for **male** and **female** have been declared disjoint from each other, and the concepts for **Jill** and **Jane** have been declared to be instances. This being the case, the following are true statements:

- **woman** and **female** are nondisjoint, because they are in a superconcept-subconcept relationship.
- **person** and **female** are nondisjoint, because they have a common subconcept.
- **man** and **woman** are disjoint because they have disjoint superconcepts. They cannot have a subconcept in common.
- **Jill** and **Jane** are disjoint because they are both instances. Neither can have a subconcept.
- **person** and **guard** are potentially nondisjoint, because they are types with no superconcept-subconcept relation to each other, and they have not been declared disjoint.

An important point is that the subconcepts (types and/or instances) of a given concept are explicitly ordered. However, it is up to the user to interpret the ordering (e.g. the user could intend it to mean that subconcepts listed first are more important than subsequent ones).

3.4.2 Intended semantics of types and instances

The extension of concepts

The intended semantics of an instance concept is that it corresponds to a *particular* thing, whereas a type concept corresponds to a *set* of things called its *extension* (although see section 3.6 for more clarification about representing sets). The above idea is standard in frame-based knowledge representations except that in some systems, such as KL-ONE derivatives, the word 'concept' refers to types only[26]. An instance concept is considered to represent one of the elements of the extensions of its superconcepts.

As is common in many KR systems, the extension of the primitive top concept is intended to be the universal set, i.e. the set of everything. As one moves down the type hierarchy to successive subconcepts, the extensions of types are intended to represent successive subsets. There is no common element of extension among disjoint types.

If a type **T** has n subtypes, it is assumed that there are some elements of the extension of **T** that are in fact not elements of any of the n subtypes[27]. For example, in figure 3.6 it is assumed that there may be living things that are not persons (perhaps dogs), and persons that are neither men nor women (perhaps children). When concepts are declared disjoint it is *not* intended that this indicate a complete partitioning, so there may be living things that are neither male nor female (and similarly persons that are neither male nor female).

Figure 3.7 illustrates schematically the extensions of the concepts shown in figure 3.6. As usual, the *thing* concept has the universal set as its extension. The world is broken up into 18 bottom-level implicit types (this number grows exponentially with the size of the knowledge base).

A: Female living things that are not persons (e.g. female animals)
B: The same as **A**, but they happen to be guards too (e.g. female guard dogs)
C: Female persons that are not women, and are guards (i.e. girl guards)
D: The same as **C**, but not guards (i.e. just ordinary girls)
E and **F**: Women guards, and ordinary women

[26] This is at odds with the normal English meaning of 'concept': e.g. every intelligent Canadian has in their brain a concept of Canada, which is not a type or set.

[27] A useful extension would be to add an ability to declare that extensions of a set of subconcepts completely cover the superconcepts, i.e. that a categorization is *exhaustive*.

G, **H**, **I**, **J**, **K** and **L**: Same as **F**, **E**, **D**, **C**, **B** and **A**, but male.
M, **N**, **O** and **P**: Same as **I**, **J**, **K** and **L** but neither male not female (neuter?)
Q: Non-living things
R: Non-living things that happen to be guards (e.g. electronic devices with such a function)

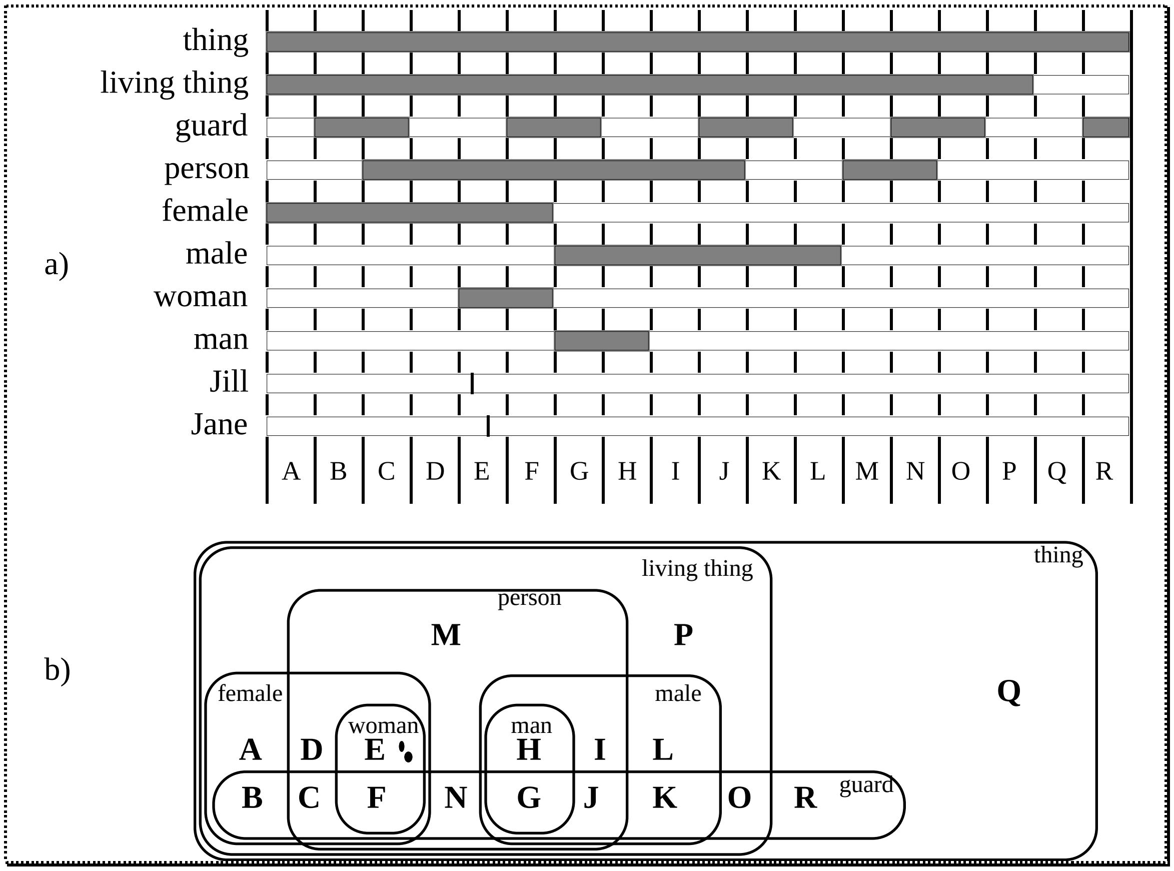


Figure 3.7: The extensions of concepts. In part a), each of the concepts shown in figure 3.6 is listed on the left. The horizontal bars represent the extensions of the concepts. Where the bars overlap vertically, there is an intersection of the extensions. Pairs of concepts where there is no mutual overlap are disjoint. Part b) is a Venn diagram showing the same information.

Quantificational patterns

When the subject of a statement contains a type concept, and the value is also a type concept, the intended semantics (in the absence of modifying facets) is[28]:

$\forall$ s $\exists$ v : <subject> s, <value> v • <predicate>(s,v)

[28] Using a typed logic similar to Z (Spivey 1989), where <type> x means that x is of type <type>.

If the subject is an instance and the value is a type, then the intended semantics is

$\exists$ v : <value> v • <predicate>(<subject>,v)

An example of the type-type case is: “for all **car** there exists a **person** in relation ***owned by***; i.e. every **car** is ***owned by*** some **person**”. An example of the instance-type case is: “**car 28** is ***owned by*** some **person**”.

When the value of a statement contains an instance concept, it is often said to be *grounded*; i.e. it is known which particular thing is related to the subject. If the subject is a type the intended semantics is:

$\forall$ s : <subject> s • <predicate>(s,<value>)

If both subject and predicate are instances, the intended semantics is:

<predicate>(<subject>,<value>)

The above four quantificational patterns are described in the context of KM by Acker (Acker 1992). They are respectively encoded as AE, IE, AI and II; where the two letters indicate whether the subject or predicate are universally (A for all), existentially (E), or not (I for instance) quantified[29]. A further comparison of CODE4-KR and KM can be found in section 3.12.

It might be argued that all of an instance’s statements ought to be grounded (i.e. all the values should be instances). Indeed, this might be a goal for a completely finished knowledge base. When few or none of an instance’s statements are grounded the interpretation can either be: 1) that the particular instances which should be in values are not known, or 2) that they change from time to time. Types, on the other hand can have some grounded statements, but should not generally have all grounded statements, otherwise the type would have an extension of one (the one thing related to the set of things characterized by the values of its statements) and be of little use.

The above intended semantics can be altered by: a) informal values, b) absent values, c) values that have more than one concept, and d) particular facets (such as modality). Discussion of such effects is deferred to section 3.6.

[29] The quantificational patterns listed are sufficient for almost all knowledge representation. The only exception is when the value should be universally quantified. Currently this must be expressed using an annotation in a facet.

3.4.3 Observations on the use of types and instances

Ontologies of types vs. databases of instances

Users of CODE4 and its predecessors have made little use of user instance concepts[30]. Instead they have built extensive type hierarchies, occasionally illustrating or giving examples using instance concepts. In the knowledge acquisition community, the word ‘ontology’ has come to refer to a knowledge base that is primarily a type hierarchy. A common procedure, especially when using a knowledge based system to design something or to model a particular real-world situation, is to first build an ontology, describing the relations and entities, and then to use a separate tool to gather or assert ‘data’ (i.e. instances of the types) (Eriksson et al, 1994). CODE4 users have mostly been building ontologies, although there is considerable disagreement about the usage of this term (Skuce and Monarch 1990; Skuce 1993d).

A knowledge base that is largely full of instances in fact could be called a *database*. There is little to distinguish such an entity from an object-oriented database. A key feature of a knowledge base that distinguishes it from a database is the focus on the relations between, and the manipulation of *types*. In database parlance, when one manipulates the type hierarchy, one is manipulating the *schema* for instances.

Types and instances as siblings in the inheritance hierarchy

A recommendation to users is that the subconcepts of a particular concept be either all types or all instances. At one point this was a hard constraint, but it was relaxed to improve practicality. The rationale for the recommendation is best illustrated by example:

Assume a knowledge base has the type **adult**, and this has subtypes **man** and **woman**. Now assume type **adult** also has the instance **Leslie**. There are several possible interpretations:

a) Leslie is neither man nor woman (in which case the user should add a new subtype of **adult** and make **Leslie** an instance of this).
b) Leslie is both man and woman (a nonsensical, but logically possible interpretation).
c) Leslie is a man (and the user just has not said this).
d) Leslie is a woman.
e) The user does not know what Leslie is.
f) The user does not *care* what Leslie is.

[30] The discussion here refers to *user concepts*. Properties, statements etc. are also instances, but this is not standard among KRs in general.

In CODE4, the above ambiguity disappears if **Leslie** is made a subconcept of either **man** or **woman** (or both). Judging from the participants in this research, knowledge base builders usually know enough and can correctly classify an instance at an appropriate leaf and thus prevent the above ambiguity from arising. Classification is discussed in further detail (e.g. explaining why it is not built in to CODE4) in sections 2.1.3.

Instances of instances

Many KR systems, e.g. Cyc, allow for there to be instances of instances. The rationale is as follows:

Assume a knowledge base has the concept **book**, as in figure 3.8a. An instance of this might be **Gulliver's Travels**. And an instance of that might be **My copy of Gulliver's Travels**. Therefore, according to faulty reasoning, instances can have instances.

Many users participating in this research have failed to see the problem with this logic. The problem is as follows: When one talks about books, is one talking about a) works of literature in the abstract, or b) sheets of paper bound together? The concept **Gulliver's Travels** ought to be an instance of the type **work of literature** (as in figure 3.8b), whereas the concept **My copy of Gulliver's Travels** ought to be an instance of **copy of book**. Of course, the type **copy of book** would have a property, e.g. ***copy of***, indicating the work of literature of which it is a copy.

The above user error is an example of what in this research is called *representational discontinuity*[31]. In general, representational discontinuity refers to situations where a thing is conflated with a representation or concept for that thing. In the above example, a book copy is being conflated with the work in general. Some of the rules[32] imposed in CODE4 are designed to combat representational discontinuity by raising user awareness. Of course, users might still commit representational discontinuity, (e.g. by leaving **Gulliver's Travels** as a type), but once they are told that they cannot have instances of instances, most users will recognize that **Gulliver's Travels** and **My copy of Gulliver's Travels** ought both to be instances of different things.

[31] Another error very commonly made by users is confusing the ***subconcept*** relation with the ***parts*** relation. Since the normal knowledge-base building methodology involves specifying the inheritance hierarchy before adding any relations, users often specify things to be subconcepts that should be parts. At present CODE4 provides no solution to this problem, but see section 7.3.1 for possible future work.

[32] i.e. insisting that instances cannot have instances, or that there be only one inheritance rule.

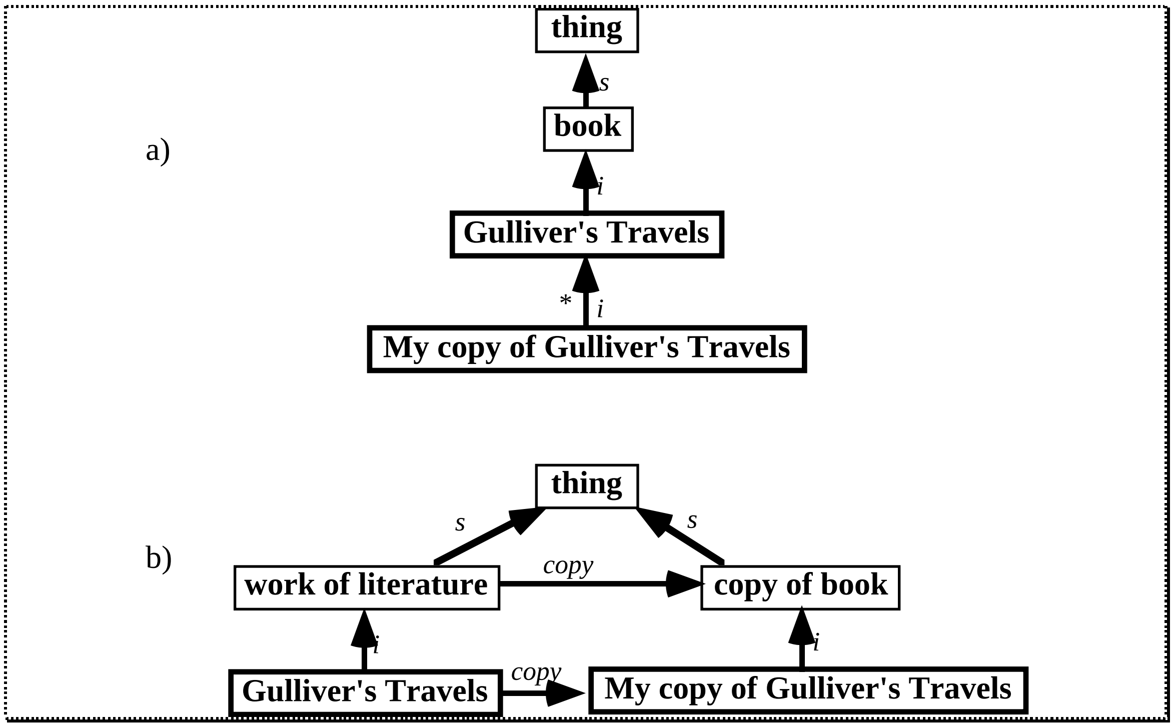


Figure 3.8: Representational discontinuity – the case against instances of instances. Part a) is a representation pattern frequently seen in other KRs (the asterisk indicates an error). CODE4-KR prevents users from making instances of instances in order to encourage correct representation as in part b).

Summary

The idea of types and instances appears very simple for most users to grasp[33], and they do not complain about the basic constraints (e.g. that an instance concept cannot have subconcepts). If, however, a user chooses to take advantage of automatic disjointness maintenance, then a much heavier set of constraints is imposed. But since it is the user's choice to impose those constraints, this appears not to violate the practicality of CODE4-KR.

One more thing should be said about the instances vs. types: Metaconcepts, properties, statements and terms are all instance concepts. This is because the things represented by these special concepts are individual in nature, not sets in nature. For example a term concept represents an individual word, compound word or symbol and a metaconcept represents an individual concept. Questions have been raised with regard to the instancehood of properties; discussion of this is deferred to section 3.5.

[33] The concepts are *understandable*, but this does not prevent important errors from being made.

3.4.4 Knowledge management problems addressed by types and instances

The following problems are partially addressed by CODE4's scheme for types and instances:

- **Problem I-1: Categorizing**. The ability to order subconcepts helps users arrange their thoughts while categorizing.
- **Problem I-3: Making distinctions**. The capability of declaring concepts disjoint helps here.
- **Problem I-6: Handling errors**: The rules that help catch representational discontinuity help with this problem.
- **Problem A-2: Expertise-restriction**. This is facilitated by the ease with which types and instances can be interchanged, and the simple intended semantics.

3.5 Properties

3.5.1 Practical effects

Properties are instance concepts. They are all grouped as subconcepts of a special primitive type concept by default called 'property' (although the user can change that name). Normally the user does not care about the position of properties in the inheritance hierarchy, instead he or she is only interested in how they are used to build descriptions of concepts. The following discusses the latter issue.

Introduction, possession and most general subjects

Every concept has a set of properties associated with it. A property is *introduced* at a certain concept, ***P***; and it is inherited by all the subconcepts of ***P***. The concept at which it is introduced is called the property's *most general subject*. The actual descriptive information associated with the occurrence of a property at a concept is found in a statement (see section 3.6). If ***P*** is a property whose most general subject is **M**, then ***P*** is only inherited by the subconcepts (direct or indirect) of **M**, The only statements that can use ***P*** as their predicate are **M** and its subconcepts.

In CODE4-KR when a concept, **C**, is said to *have* a property ***P***, it means that **C** is either the most general subject for ***P*** or inherits ***P***. From a practical point of view, when a user specifies that ***C*** has ***P***, the user is merely saying the following: That it makes sense to use ***P*** to speak about ***C***, and to use ***C*** as the subject of a sentence whose predicate is ***P***.

Figure 3.9 illustrates the above points. In part a), the property ***job*** is introduced at the concept **person**. **Person** is then the most general subject of ***job***, and ***job*** is inherited by all subconcepts of **person**, such as **woman**. **Person** and all its subconcepts *have* the property ***job***; however other concepts, such as **dog**, do not. In the knowledge base as shown, one can not talk about a dog's job.

If this is decided to be an error, ***job*** has to be moved to a higher most general subject. Since the common intersection of **dog** and **person** is inappropriate (not all living things have jobs), a new concept called **employable thing**[34] is introduced. This is shown in figure 3.9 part b).

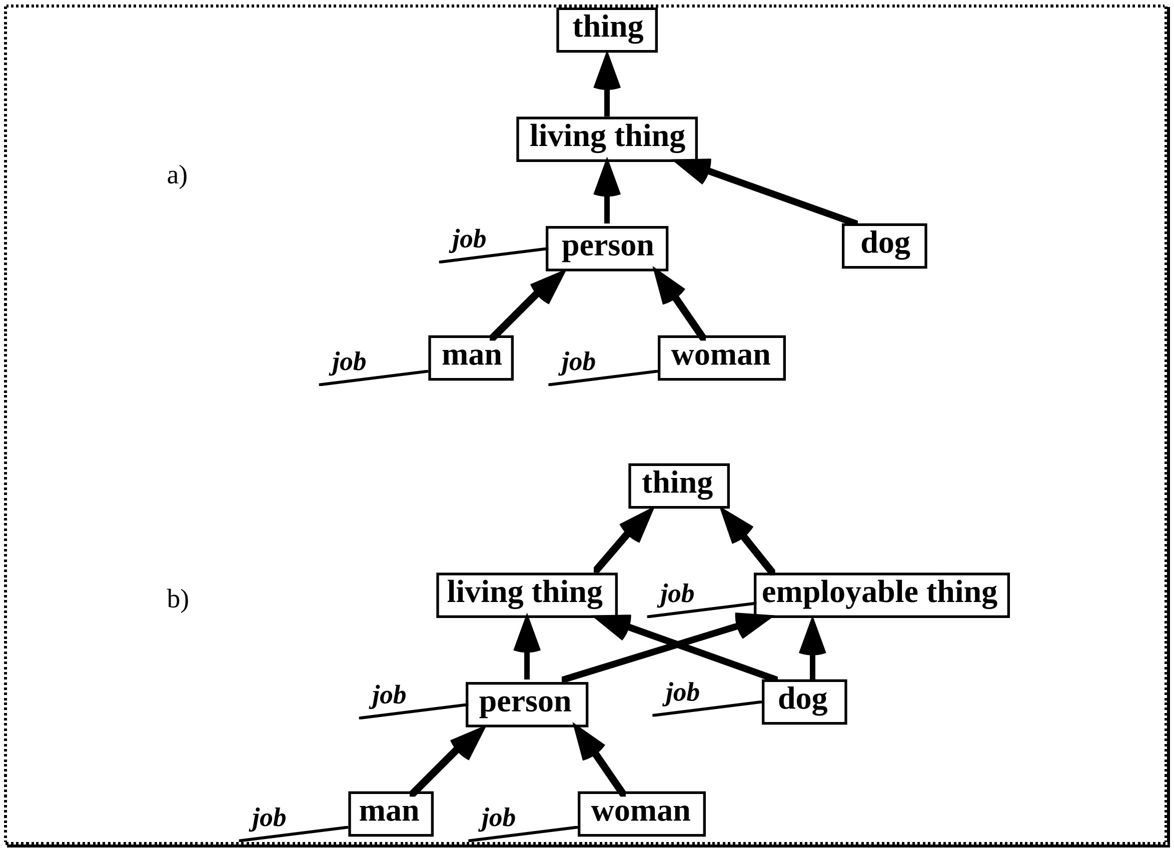


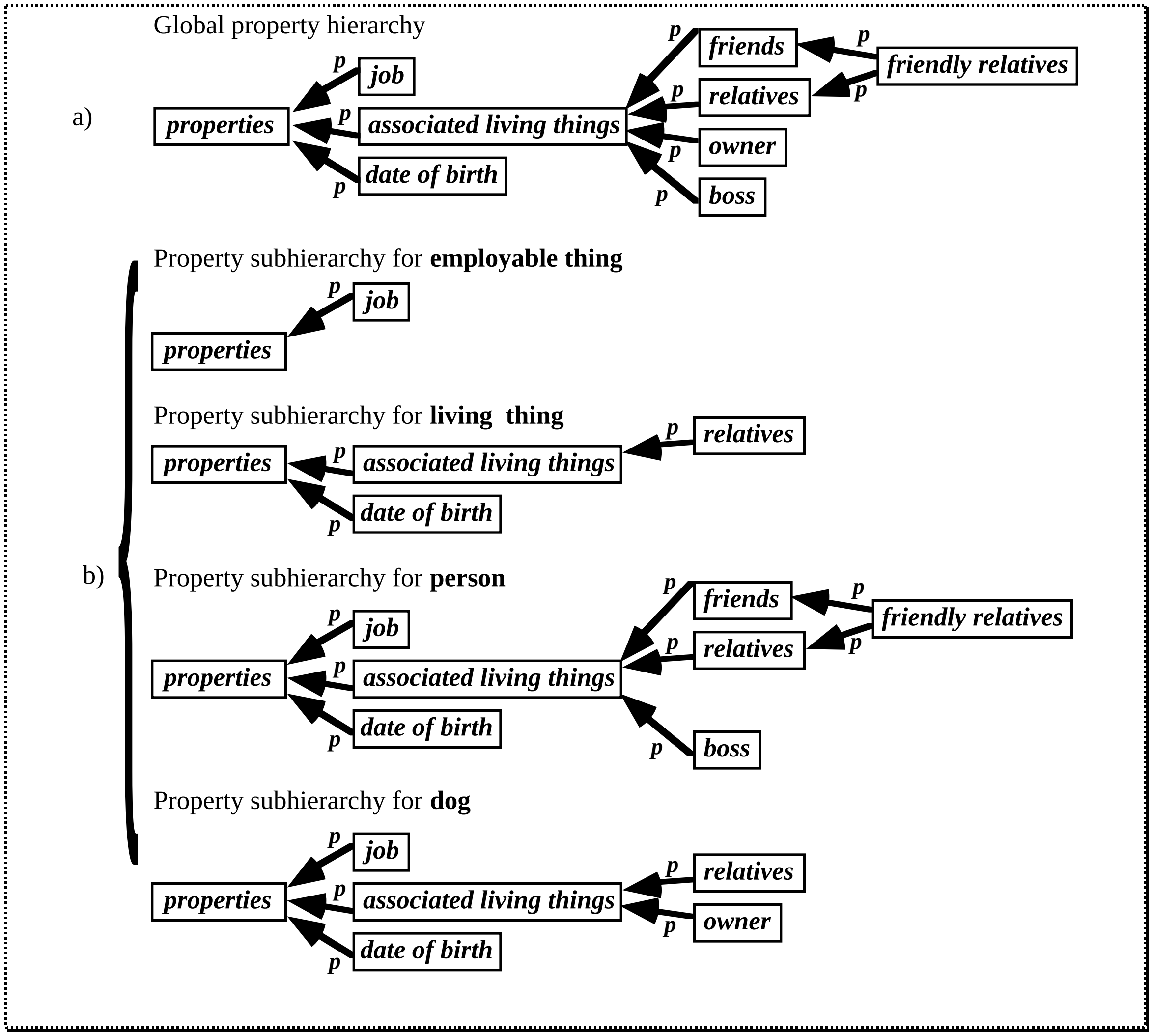
Figure 3.9: Basic ideas about CODE4-KR properties: In part a), **job** *is introduced at its most general subject* ***person****, and inherits to* ***man*** *and* ***woman****. It makes sense to talk (make statements) about jobs of persons in general, men and women. In part b),* **job** *has been moved to* ***employable thing****, so that it can inherit to a wider subtree of the inheritance hierarchy. A thin line with a label represents the possession of a property by a concept.*

Subproperties and the property hierarchy

The set of all properties in a knowledge base is organized into a single global *property hierarchy* based on the ***subproperty*** relation; i.e. each property can have one or more

[34] An **employable thing** would be defined to be a thing that *can* have a job. The user would have to use modality, discussed in section 3.6.2 to specify this.

subproperties. In every knowledge base, this property hierarchy has one single top property which is typically labelled ‘properties’ (for lack of a better name). All other properties are subproperties (directly or indirectly) of the top property. The property hierarchy is a partial order and a property can have more than one superproperty. The major emphasis on the property hierarchy is a distinguishing feature of CODE4-KR[35].



Global property hierarchy
a)
properties
job
associated living things
date of birth
friends
relatives
owner
boss
friendly relatives
p
b)
Property subhierarchy for employable thing
Property subhierarchy for living thing
Property subhierarchy for person
Property subhierarchy for dog

Figure 3.10: The principle of the property hierarchy. Part a) shows an example global property hierarchy (listing all the properties in the knowledge base). Part b) shows how four different concepts (from figure 3.9) have particular property subhierarchies. The p labels indicate links to superproperties. The particular properties inherited by a subject depends only on the most general subjects of the properties. The form of each subhierarchy depends only on the form of global hierarchy.

[35] Cyc and KM are among the systems that organize slots into hierarchies, however CODE4 adds additional features such as: 1) the derivation of the statement hierarchy; 2) introduction involving strict sub-hierarchies; 3) the single most-general-subject, and 4) the importance of the property hierarchy in the user interface (chapter 4).

The set of properties possessed by a concept, **C**, consists of a *subhierarchy*, HC, of the global property hierarchy, rooted at the top property, but omitting some lower level branches. Concept **D**, a subconcept of **C,** has the the union of the property subhierarchies of its superconcepts (in this case just **C**), with additional branches *introduced* at **D**. The properties introduced at **D** must be subproperties of properties found in HC.

Figure 3.10 illustrates the basic idea of the property hierarchy. Part a) shows a global property hierarchy for the concepts found in figure 3.9b). No concept has all of these properties. In figure 3.10b, the concepts **employable thing** and **living thing**, both have small subhierarchies of the global property hierarchy. Their common subconcepts, **person** and **dog** have the union of the small subhierarchies, plus some newly introduced properties. **Person** is the most general subject for properties ***boss***, ***friends*** and ***friendly relatives*** (which has two superproperties). **Dog** is the most general subject for the property ***owner***.

Note that the knowledge base illustrated in figure 3.10 currently excludes the possibility that dogs could have friends. Should there be a desire to change this situation, one solution would be to introduce a concept **sociable living thing** as the most general subject of ***friends***[36].

The property hierarchy is intended to be a categorization of properties. When statements are discussed, it will be seen that the property hierarchy also has some other interesting characteristics.

3.5.2 Intended semantics

Properties can be viewed as logical predicates. This will be discussed further in section 3.6 during the discussion of statements. Another, equivalent, view of a property is as a relation whose domain is the extension of its most general subject. Subproperties can be considered subrelations.

Intensions

The potential set of properties possessed by a concept is called its *intension*[37]. The potential set of properties whose most general subject is a concept is called its *local intension*. Each concept has a local intension that is disjoint from that of all other concepts. The intension of a concept is the union of its local intension and the intensions of all its superconcepts.

---

[36] If this approach is taken often, the tendancy is to create knowledge bases with too many concepts whose sole purpose is to be a most general subject. This can be combatted in the above example by adding **friends** as a property of **living thing**, but giving it the modality of ‘typically not’. This modality would be overridden in the case of those living things that really are sociable.

[37] “Intension” is often used thus in AI; see (Lacey 1976, p98) for a philosophical discussion of the term.

The expression '*potential* set of properties' was used above to emphasise the fact that adding a property does not intrinsically alter the nature of the underlying thing being represented which theoretically might have an unlimited set of properties[38]. When a property is added to a concept, it is merely being stated that now one of the elements of the intension is known.

The property hierarchy: a hierarchy of instances

In systems like Cyc and KM, the property (i.e. slot) hierarchy uses the same hierarchic relation as the inheritance hierarchy. In such systems, subproperties are subconcepts of their superproperties, and there is no distinction between the subconcept and subproperty relations[39].

Upon careful examination, the above has been rejected. Instead, in CODE4-KR properties have a single element of extension (i.e. they are instance concepts) and the subproperty relation is distinct from the subconcept relation. This has been done for the following reasons:

- Many properties of a property do not inherit to subproperties (e.g. ***most general subject***, ***statements with this property as predicate*** etc. do not inherit).
- Although the names of subproperties often seem to be in the 'isa' relation to the names of their superproperties, this is because their respective *values* are in such a relationship. A relation must not be confused with the things it relates.

An intimately related issue considered when designing CODE4 was the following: Whether a statement (discussed in the next section) should be an instance of a property. In such a situation, a property, **P**, would be a type whose instances are the statements of which **P** is the predicate. In this scenario, a property would be considered to represent a set of subject-value tuples (i.e. statements) in the same manner that a type has a set of things in its extension.

This was rejected for three reasons:

- It was decided that properties should be instances, as discussed above.

[38] Properties are mental (or computational) and descriptive entities. In CODE4 they are not considered to be intrinsic to the real-world entity being described. For example, each user is likely to characterise cars using a different set of properties; furthermore, it is trivially easy to 'invent' new properties to describe some latent aspect of a thing, or some obscure relation that one has just noticed. While some definitional properties might be a necessary part of a *concept* (in a mind or knowledge base), this does not limit a user's ability to endlessly invent other properties for a thing.

[39] Of course, the hierarchies contain different concepts, but they appear to be in the same relation to each other.

- The relation between statement and property clearly is not ‘is-a’; a statement *has* a property as predicate, but in no sense *is* a property.
- Statements can be formed using any property in the property hierarchy as predicate (from the top property down to the lowest level subproperty). If statements were instances of properties, then they would naturally only be attached to bottom-level properties.

3.5.3 Observations on use

One of the biggest problems noted in practice with properties is that users tend to introduce two or more properties of the same name at unrelated concepts, with the mental model that they are actually dealing with the same property (or without such a mental model, but when the different properties do in fact have the same meaning and should logically be a single property).[40]

Using figure 3.9a as an example, a user might add a ***job*** property to the concept **person** and another ***job*** property to the concept **dog**. In reality the user ought to have introduced a single property ***job*** higher in the hierarchy at some concept, say **employable thing**, that subsumes both **person** and **dog**. This in fact is done in figure 3.9b.

A possible solution would be to allow a property to have more than one most general subject. This would mean its domain would be the extensions of several concepts, and those concepts would have overlapping intensions. However, this would really be a shortcut for declaring that there is a common superconcept that the user desires not to name. For now, it has been decided that the least complex solution is to keep the status quo in the knowledge representation and to rely on the user interface to highlight potential problems for the user.

In CODE2, where properties were identified by their name, if a user introduced two properties which happened to have the same name at different most general subjects, then the system interpreted them as the same property. This is the way most knowledge representation systems function, however it has been found to lead to a problem that is the *inverse* of the problem described above: A user accidentally gives two distinct meanings to the same property.

The idea of a ‘property manager’ to help users better deal with the above problems is presented in chapter 7.

3.5.4 Knowledge management problems addressed by properties

The following lists some of the way CODE4’s treatment of properties assist in the solution of the problems listed in section 1.4:

[40] This problem arises out of the fact that in CODE4 the name of a concept does not identify it uniquely.

- **Problem I-1: Categorizing**. Property hierarchies provide a ‘second dimension’ with which to categorize knowledge.
- **Problem I-5: Extracting**. When studying a network of related things, the user can narrow (or widen) the scope of information shown by specifying that the network should be generated using successive subproperties (or superproperties).

3.6 Statements and facets

3.6.1 Practical effects

Just like properties, statements are instance concepts; they are instances of the primitive type **statement**. As with properties the user rarely thinks of them in this way: It is their role of associating a particular concept (their subject) and a particular property (their predicate) that is of interest.

The circumstances under which statements exist

When something is to be said involving a property of a particular concept it is said using a statement. A statement is in fact uniquely defined by its subject and predicate pair – there can only be one statement per pair. Also, every pair, composed of a concept and one of its possessed properties, is the basis of a statement. The subject and predicate of a statement are said to be its *defining concepts*.

A statement is a *dependent* concept. As such, it conceptually comes into existence when both of its defining concepts come into existence, and conceptually ceases to exist when either one of its defining concepts ceases to exist. Other types of dependent concept are discussed in sections 3.8. Figure 3.5 relates statements to other classes of concept.

A statement is also a *system-manipulated concept*. A system-manipulated concept is one that is physically created and destroyed as a byproduct of the user’s manipulation of other concepts, but is not manipulated *directly* by the user.

A statement that conceptually exists, because a particular subject possesses a particular predicate, need not physically be created in a knowledge base. If it does not physically exist, it is considered *virtual*. A virtual statement is made real when knowledge is added involving it; i.e. under the following circumstances:

- It is made the most general subject of a property. (A property is attached to it.)
- Its *value* is specified.
- The value of one of its facets (of which it is the subject; discussed below) is specified.
- It is specified as one of the value-items of a value (also discussed below).

A statement is physically discarded and made virtual again when all of these conditions cease to be true.

When the user interface requests to display a statement (all such requests are made by giving just the subject and predicate), a placeholder called a *temporary empty* statement is created if the statement is virtual. This occurs for example when the user wants to specify the value of a virtual statement. The user chooses a subject and predicate and asks to see the (empty) value of the corresponding statement. As soon as the user specifies a value, a real statement is created as a replacement for the temporary empty one.

The statement hierarchy

As discussed in section 3.5, a concept **C** inherits a portion of the global property hierarchy, HC. The statements whose subject is **C** and whose predicates are in HC form **C**'s *statement hierarchy*. Statement **A** is a *substatement* of statement **B** if they both have the same subject, and if the predicate of **A** is a subproperty of the predicate of **B**. In other words, there is a one-to-one mapping between that HC and the statement hierarchy of **C**.

Figure 3.11 shows a statement hierarchy. This happens to be the statement hierarchy for a subject which is also a statement. Thus the statements shown are *facets*. These are discussed next.

Facets

Statements whose subjects are statements are called *facets*. It is in these that a large portion of the important information in a CODE4 knowledge base is stored. As with other concepts, statements possess a set of properties that are called *facet properties*. The most general subject for these is usually the primitive type, **statement**.

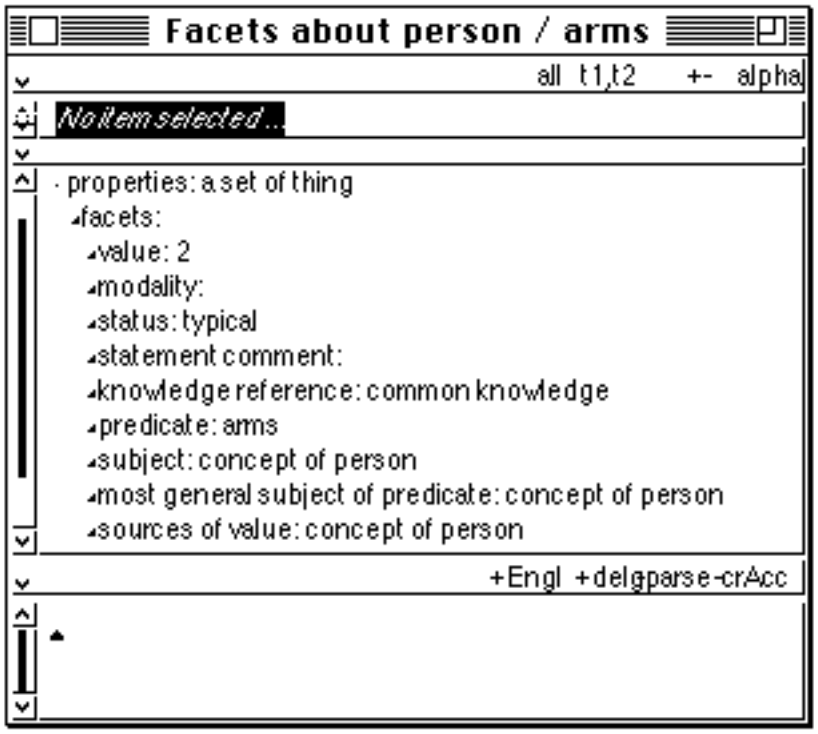
Users are free to add new facet properties in order to cause statements to have new facets. Such new facet properties must be inherited by the statements in question from the *statement's* superconcepts. Possible superconcepts (most general subjects of facet properties) can be:

- The primitive type, **statement**, itself. All statements in the knowledge base would then possess the facet.
- A particular statement (no other statement would then have the facet).
- A special subconcept of the primitive type, **statement**. The special subconcept would be made the superconcept of a specific set of statements which the user wants to have the facet property in question.

Figure 3.11 shows several primitive facet properties found in every CODE4 knowledge base. The following are the main types of facet:

- The value facet (discussed below). This is the most important.

- Facets that can be used to modify the semantics of the statement (e.g. ***modality***, discussed later).
- Facets that are computed automatically and that cannot be changed; for example ***subject***, ***predicate*** and ***most general subject of predicate***. Another example is ***sources of value***; this specifies the subject(s) from which the value of this statement is inherited.
- Facets that are documentary in nature (e.g. ***statement comment***).



```
· properties: a set of thing
  facets:
    value: 2
    modality:
    status: typical
    statement comment:
    knowledge reference: common knowledge
    predicate: arms
    subject: concept of person
    most general subject of predicate: concept of person
    sources of value: concept of person
```

Figure 3.11: Facet properties. Shown are statements whose subject also a statement, i.e. the statement whose subject is **person** *and whose predicate is* ***arms***. *This figure is a CODE4 browser, discussed in chapter 4.*

The value facet and the facet hierarchy

The value facet is treated in a primitive way for the following reason: If one had to look in the value facet to get the value of a statement, an infinite regress would occur. This is because the value facet, as a statement, would also have its value stored in a value facet. One would have to look in the value of the value of the value ad infinitum.

To prevent this problem, a primitive operation is provided to obtain the value when given a subject and predicate. Obtaining the value of any *other* facet would be done in the same manner as looking up the value of a non-facet statement; it requires knowing the facet property (predicate) and statement (subject) in question.

Since facets are statements like any other, their values can be used in the same way as the values of non-facet statements. Furthermore, one can attach *subfacets*[41] to facets (even, interestingly enough, to the value facet) forming an unlimited depth *facet hierarchy*. The lower level facets are usually empty of any new knowledge and only exist implicitly. The fundamental principal here is that 'one can say something about anything in CODE4'. This level of uniformity and flexibility of facets and values is believed to be unique among knowledge representation systems. Some of the benefits of CODE4-KR's treatment of statements and facets will become clearer in the section on intended semantics, below.

Formal values, informal values and value items

Values can be formal or informal (see chapter 1 and the glossary for a discussion of these terms). An informal value contains a character string; i.e. uninterpreted text. A formal value contains an ordered list of *value items*. The value items refer to other concepts in the knowledge base. Section 3.6.2 explains this further.

When a statement has multiple value items, several *value-dependent substatements* are also considered to be present. For example, if the subject were **person** and the predicate were ***parents***, then the value might contain the two concepts **man** and **woman**. Such a statement might be read "a person has parents which are a man and a woman". This statement can be broken down into substatements as follows: Two subproperties of ***parents*** are created, and the statements of these *each have exactly one* of the value items **man** and **woman**. Value-dependent substatements are ordinary statements, except that their presence is determined purely by the presence of multiple value items. CODE4 implements value-dependent substatements, but currently there is no way to attach facets to them[42].

The basic inheritance rule

The inheritance rule for statements in CODE4-KR is as follows: If statement **X**, with predicate ***PX*** and subject **SX**, has value V, then the subconcepts of **SX** have statements with predicate ***PX*** and value V, unless the user explicitly replaces value V with something else or unless multiple inheritance occurs.

In the case of multiple inheritance where inherited values differ, they are combined as follows:

- Any value that is a superconcept of another is dropped (i.e. the most specific value is used in the subconcept).
- Of the remaining concept values, the union of the value items is used (i.e. the subconcept is assumed to be related to possibly multiple things).

[41] i.e. facets whose subjects are facets.

[42] They form the arcs in outline and graphical mediating representations discussed in chapter 4.

• Character string values are combined with the expression ‘one of (value 1, value 2, …)’.

The above uniform rule does not necessarily suit every application and can sometimes give results that users don’t want. However, if the user does not like the result, a replacement value of the user’s choice can be substituted. The benefit of a uniform rule is that the user is not forced to make decisions at the time values are entered in superconcepts.

There are several additional details concerning inheritance; these are deferred to section 3.10 where the *delegation* mechanism is also introduced.

3.6.2 Intended semantics

In the relation interpretation of properties (section 3.5.2), the value of a statement specifies part of the range of the relation formed by its predicate, where the subject specifies a corresponding part of the domain. For the statement involving a predicate’s most general subject, the value corresponds to the whole range.

• If the value is a single concept, then the range is the extension of that concept.

• If the value is several concepts, then the range is a set of tuples: All possible tuples, where one element is taken from the range of each value item.

Multiple value items, including the effect of inheritance and the property hierarchy, are discussed further later.

If the value of a statement is unspecified or if it is a character string, the only semantic interpretation one can place on the statement is the same as if the value were **thing**, the top concept.

An important way of visualizing (or possibly implementing) a knowledge base, is as a *structured set of statements*. In this scheme, each concept is described by a subset of the statements: those for which it is the subject.

The modality facet

One of the primitive facet properties is ***modality***. Aside from values, modality facets are the most common facet specified by users. Intended values for this are ‘necessary’, ‘typical’ and ‘optional’ (although a zero-to-one range might also be used).

If the subject of a modality facet is a non-facet statement (i.e. the top of a facet hierarchy; a statement whose subject is not a statement), then it is specifying the likelihood that a member of the subject’s extension actually participates in the relation. The intended semantics (where subject and value items are types) would be as follows:

∀ s ∃ v : <subject> s, <value> v • prob(<predicate>(s,v)) = <modality>

e.g. if the modality were ‘typical’, and the statement were ‘birds fly’, then a ‘typical’ modality would be interpreted as ‘birds typically fly’. If the modality is missing, the default is assumed to be ‘typical’ – explicit user action is required to make the strong ‘necessary’ assertion.

Modality can also be added lower in the facet hierarchy. If a modality facet is attached to a value facet, then the intended semantics would be as follows (where the top of the facet hierarchy and the value items are types):

$$\forall\ s\ \mathrm{prob}(\exists\ v : \langle\mathrm{subject}\rangle\ s, \langle\mathrm{value}\rangle\ v \bullet \langle\mathrm{predicate}\rangle(s,v)) = \langle\mathrm{modality}\rangle$$

i.e. “for all <subject>, there <modality>ly exists <value> in relation <predicate>“. In other words the modality facet is declaring how common it is that the value is indeed as stated.

If the modality is ‘necessary’, the value is always as stated; e.g. if it is stated to be necessary that the parents of a person are a man and a woman, then one can infer there is no other alternative value. A declaration that the modality is ‘typical’ says that the stated value is the norm, but there may be exceptions.

Value and modality are the only facet properties to have an intended semantics. Users may add others, for example to emulate KM’s likelihood, necessity, cue-validity and uniqueness facets.

Subordinate value facets

A statement can have only one value (although several value items). There are situations however, when it is desired to say *several* things involving a particular subject and predicate.

For example, one might want to say that while the caregivers of a child are typically their parents, another option is that the caregivers are foster parents. This can be expressed by adding a *subordinate value facet* (as a sibling of the regular value facet, i.e. with the same subject and predicate) into which **foster parents** would be placed, and to which would be attached an ‘optional’ modality.

Value and modality consistency maintenance

An inference mechanism proposed as an extension to CODE4 is automatic value and modality consistency maintenance. Such a mechanism (discussed further in chapter 7) would ensure the following:

- **Value consistency**: As one moves down the inheritance hierarchy, the values of a particular property that have the modality ‘necessary’ should represent successive subtypes, i.e. never become more general.

- **Modality consistency**: The values of a particular property that have the modality 'necessary', should remain 'necessary' as one moves down the inheritance hierarchy. i.e. a necessary value should not be overridable.

Mechanisms like this are common in KR systems, but most such systems assume a 'necessary' modality for all statements. It would be important in CODE4 not to force users to maintain such consistency.

Multiple value items as sets

The intended interpretation of multiple value items is that the range of the relation is a set, containing one member from the extension of each value item. For example, if the value of the statement about a person's parents were '(man, woman)' this is saying that a person has a set of two parents, one of which is a man and the other of which is a woman.

A useful extension to CODE4-KR would be to allow for a fuller range of set-construction capabilities in values, and to consider these sets to be concepts themselves. Set-construction capabilities similar to those provided in the Z language (Spivey 1989) would be appropriate. Currently, users are only able to give complex set specifications using informal character strings.

It is intended that for a given statement, **S**, its substatements should have value items that are subsets of the value items of **S**. For example, consider the properties of **person**: A statement at a high level in the statement hierarchy might have as its predicate ***relatives*** and as its value, **set of persons**. Substatements might have predicates ***ancestors***, ***siblings*** and ***descendants***. Each of these would have values that specified subsets of the relatives. In a similar manner, a subset of ***ancestors*** would be ***parents***, which further divides into ***father*** and ***mother***.

3.6.3 Observations on use

The only facet used by about 80 percent of the users who participated in this research was the value facet. Much of the time the users only used informal character strings[43]. The values of statements involving properties high in the property hierarchy are rarely specified. For the above reasons there is little demand for automatic value consistency maintenance.

As a layer on top of the value specification mechanism, certain aspects of a language called ClearTalk (Skuce 1988; Skuce 1992b) were implemented in CODE4. ClearTalk was designed to allow for the easy expression of a wide variety of ideas in values, and to do this unambiguously. ClearTalk is based on linguistic principles and has a grammar that

[43] Statistics about feature usage are given in chapter 6.

reads sufficiently like English to be usable by most people. Certain elements of ClearTalk have proved useful for users to express ideas that have meaning to them.

In the latest versions of CODE4, actual processing of ClearTalk has been reduced to the resolution of references to one or more value-items as described above. Users are still encouraged to use ClearTalk when they specify values using informal character strings.

3.6.4 Knowledge management problems addressed by statements and facets

CODE4-KR's statements contribute to the solution of several problems listed in section 1.4:

- **Problem I-2: Categorizing**. Statement hierarchies, derived from property hierarchies help in this process.
- **Problem I-3: Making distinctions**. The fact that one can attach facets (a whole hierarchy in fact) to any statement helps in the process of making distinctions. Informal values allow specialized syntaxes to be used if the user cannot represent certain knowledge with CODE4's built-in facilities.
- **Problem I-4: Understanding effects**. The values of computed facets such as sources of value help with this. The uniform combination rule helps ensure that users know what values to expect when multiple inheritance occurs.
- **Problem A-2: Expertise restriction**. Informal values reduce the need for the user to learn a particular syntax when representing knowledge

3.7 Terms

3.7.1 Practical effects

CODE4-KR treats the words or symbols used to name concepts as full-fledged concepts themselves. Such concepts are called *terms*.

Naming concepts

An instance of the primitive type **term** is created automatically whenever a user enters a name[44] for a main subject or property, where no term with that name previously existed.

When a concept is first created, it is given a system-created label (e.g. 'instance 12 of car' or 'specialized vehicle'[45]). This label:

[44] A *term* is a kind of concept; a *name* is a string used to identify a concept (associated with a term); a *label* is what is displayed on the screen to identify a concept.

- Helps distinguish the concept from other concepts.
- Avoids forcing the user to immediately think of a name.
- Means the interface can remain non-modal (see chapter 4) because there is no need to prompt the user for something with which to identify the concept.

Such a system-labelled concept does not have any term assigned to it. Furthermore, its label will change dynamically if the context from which the label is derived changes (e.g. if the superconcept changes). Shortly after most main subjects and properties are created, users almost always assign a term to the concept by replacing the generated label with a name of their own choosing.

Properties of terms

Terms have properties such as ***part-of-speech***, ***plural*** or ***French equivalent***. Values for some of these can be automatically obtained from an on-line dictionary to which CODE4 may be connected.

The only *necessary* property of a term is its character string; and there is a one-to-one mapping between terms and character strings. There are few restrictions on the internal syntax of a term's string except that if it starts with an octothorpe, then it is assumed to be an encoded binary bitmap – the term then represents a graphical icon. Encoded bitmaps are never returned when an alphabetic name for a concept is requested. They are used in the graphical mediating representation (see chapter 4).

The assignment of terms to concepts

There may be several terms (synonyms) for a concept, and a single term may refer to several concepts (i.e. a term can have several senses or meanings). For example: **term 'person'** and **term 'human'** may be synonyms for the same concept. Likewise, **term 'bus'** may be a homonym for all of: 1) an activity done by a restaurant employee, 2) a computer part and 3) a vehicle.

In the case of homonyms, users have to distinguish concepts based on their properties (including their relationships to other concepts, graphical layouts, informal values etc). Inside CODE4 however, two identical concepts (e.g. siblings with no terms and no properties) are always considered distinct.

Interestingly, CODE4-KR's uniform architecture allows terms to be assigned to *any* concept, including terms themselves (although there is little value in doing so!)

---

[45] For user instances and types respectively. Properties, statements and terms have their own rules in this regard. When there are multiple parents, the generated labels combine their names.

To find the complete list of terms of a concept, one needs to look in the metaconcept property ***terms***. Also one can find out what concepts a term represents by looking at the term property ***meanings***. The triad of concepts, terms and metaconcepts thus implements a meaning triangle (Regoczei and Hirst 1989).

3.7.2 Intended semantics

There is very little additional semantics attached to terms beyond that of ordinary instance concepts. Their single point of extension refers to the *relationship* between their character string and their meaning.

The ***terms*** property is attached to metaconcepts because it is not the underlying thing that has a name – names are associated with the *concept* of the thing. Different knowledge bases may use different terms for a given thing.

3.7.3 Observations on use

Amongst knowledge management systems, only the Active Glossary system (Klinker, Marques et al. 1993) appears to treat terms as seriously as CODE4-KR does. Experience (with earlier CODE systems) has taught that the flexibility of CODE4-KR's terms is essential for practical knowledge management. The constraint imposed by a one-to-one mapping between words and concepts was too great.

3.7.4 Knowledge management problems addressed by terms

Terms address the following problems:

- **Problem I-2: Naming**. Terms relieve users from being forced to create artificial names just to satisfy internal system needs. This benefit derives from several sources: 1) If a user can immediately think of no term, there is no requirement to make one up; and 2) If a term has already been used to name another concept, there is no reason why it cannot be used again. Terms also prevent users from having to choose between terms – they can give a concept as many as they want.
- **Problem I-3: Making distinctions**: Term properties can be used to express such important linguistic details as language, part of speech, special usages etc. Without terms this knowledge would typically be omitted or attached (inappropriately) to the underlying concept.
- **Problem I-5: Extracting**. Because concepts can be given terms that exactly match those in natural language, it is easier to find such concepts.
- **Problem I-6: Handling errors**. When a user has made a mistake entering a term, it can be readily changed without effecting any formally-expressed knowledge. In some other systems, poorly chosen terms become entrenched.

- **Problem A-2: Expertise restriction**. The ease of naming and extracting provided by terms helps make the system more usable.

3.8 Metaconcepts

3.8.1 Practical effects

When representing knowledge, it is frequently necessary to describe properties not of the thing a concept represents, but of the concept *itself*. For example, a non-metaconcept such as **car** would have properties that describe actual cars. e.g. specifying that they have wheels, a chassis, an engine and a body. However, the metaconcept **concept of car** would have properties such as ***inventor*** or ***comment*** that do not apply to particular instances of car but rather apply to the overall idea (i.e. concept) of cars.

To explain further; metaconcepts allow for distinctions of the following kind to be made: While it is appropriate to say a particular car has wheels, it is inappropriate to say that the concept of car (something in a knowledge base) has wheels. Conversely, while it is appropriate to say that the concept of car was invented in a given year, it is inappropriate to say that an actual car has a particular year of invention – One would not say, “My car was invented in 1896,” one would rather say, “The car [the idea of the car] was invented in 1896.”

The circumstances under which metaconcepts exist

Each concept has exactly one metaconcept associated with it, and vice-versa. A concept represents a thing in the world and its metaconcept represents that concept. A concept is called the *defining concept* of its metaconcept.

As with a statement, a metaconcept is a *dependent* concept. It conceptually comes into existence when its defining concept is created and ceases to exist when its defining concept ceases to exist.

Also as with a statement, a metaconcept that conceptually exists need not be *physically* created in a knowledge base; i.e. it may be *virtual* unless the user desires to add knowledge involving it (making it the most general subject of a property; specifying the value of a statement of which it is the subject, or using it in the value of statement[46]). Likewise, a metaconcept can be physically discarded and made virtual again when all such knowledge

[46] The astute reader might notice the following: Since all concepts except the top concept have superconcepts, and since ***superconcept*** is a metaconcept property, then one might expect most metaconcepts to be physically created to store superconcept information. Although the system acts as if this is true, in fact properties such as ***superconcept*** are primitively managed by optimized mechanisms (see section 3.9) so physical metaconcepts are only created when values of non-primitive properties are to be stored.

is deleted. For mere display purposes, a temporary empty[47] metaconcept can be created; this can be discarded whenever the display is no longer needed. This manipulation of the existence of metaconcepts is done by the system, never directly by the user, hence metaconcepts are a kind of *system-manipulated concept.*

Metaconcepts as a way to support non-inheriting properties

Metaconcepts are instances of the primitive type **metaconcept**. This is the most general subject for a number of properties shared by all metaconcepts, such as:

- ***enterer***, the person who entered the knowledge about the concept;
- ***entry date***, the date the concept was entered;
- properties relating the concept to others (***superconcepts***, ***subconcepts***, ***kinds***, ***dimensions***, ***disjoint concepts***);
- information about how the concept can be referred-to or visualized (***terms***, ***graph layout position***);
- information about which properties have the defining concept as their most general subject (***source properties***);
- descriptive information (***comment***).

These are all primitive metaconcept properties, many of which are described further in various parts of this thesis (including below). Of course, the user is free to add other metaconcept properties.

The important thing to note about all such properties is that they inherit to each individual metaconcept, but *not* to subconcepts of a metaconcept's defining concept. Also, a value of a statement (having one of these properties as predicate and a particular metaconcept as subject) would not be inherited by any other concept. This naturally derives from the fact that metaconcepts are instance concepts; like other instance concepts they can have no subconcepts, and hence nothing can inherit from them.

So by having metaconcepts, CODE4-KR provides an elegant and intuitive mechanism for non-inheriting properties. Whereas most knowledge representations require that non-inheriting slots be tagged as such, CODE4-KR has a uniform rule that all properties inherit, and uses metaconcepts, separate from their defining concepts, to provide non-inheriting properties.

Sometimes it is desired to have values involving metaconcept properties *appear* to inherit. For example, assume user 'Joe' creates a whole hierarchy of concepts. Then assume he

[47] A temporary empty concept is merely a placeholder for a virtual concept.

wants to record his name as the enterer of each concept. Since the property ***enterer*** is a metaconcept property, and hence does not inherit, he might individually fill in values for the metaconcept of every concept he entered. In other cases, it is desired to have groups of unrelated metaconcept statements have similar values. These requirements can be handled in two ways:

- Using delegation (section 3.10). This allows the value of a statement to be computed from another value.
- Using specialized subconcepts of the **metaconcept** primitive type. These can be made the superconcepts of a certain subset of metaconcepts.

The implicit metaconcept hierarchy

Metaconcepts are full-fledged concepts and can be treated just like terms, statements, properties and main subjects. In fact, although rarely used, CODE4's uniformity permits a metaconcept to have its own higher-order metaconcept (e.g. to hold properties describing who updated the first-level metaconcept). If a user asked to look at a metaconcept of a metaconcept of a metaconcept (to any depth), the system would create the appropriate temporary empty metaconcept for display. The user can continue to create successively higher order metaconcepts, although only the first level is usually of any use.

It thus can be said that, as with statements, a knowledge base implicitly contains an infinite number of metaconcepts, most of which serve no conceptual purpose.

Important metaconcept properties

It has been mentioned before that all primitive properties that show the interrelation between concepts (as opposed to between things in general) are attached to metaconcepts. Three interesting such properties are ***dimensions***, and its cousin ***inherited dimensions*** as well as ***kinds***.

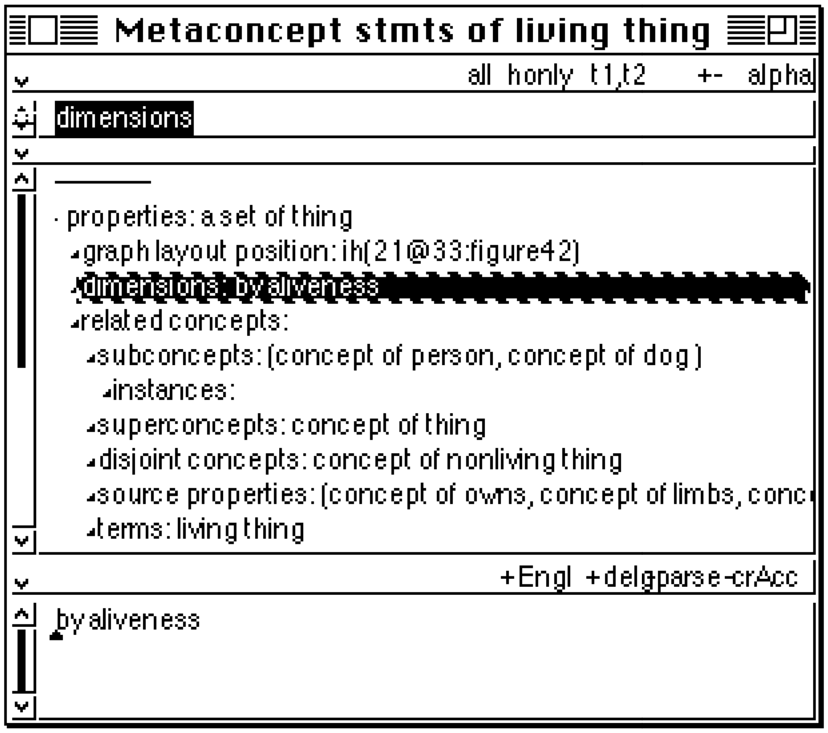
A *dimension* is a partitioning of a concept into subconcepts based on one point of view or criterion, for example, the sex of persons. The concept **person** might have four subconcepts: **female person**, **male person**, **adult** and **child**. The first two subconcepts constitute the dimension 'by sex', while the second two constitute the dimension 'by age'. The dimension property of the metaconcept for **female person**, would contain 'by sex'. CODE4 labels subconcept arcs using dimensions. The regular subconcept relation can be considered a 'nameless' dimension.

The following are normally also true when a concept's subconcepts are divided into several dimensions:

- The concepts *inside* each dimension are usually declared disjoint from each other.
- Concepts with multiple parents are usually created. Each of these concepts inherits from a a different one of the dimensions (e.g. **adult female person**).

• Properties are usually created that have the same name as each of the dimensions.

Since **dimension** is a metaconcept property, it does not inherit. This is valid because usually at lower levels in the inheritance hierarchy the dimension of some remote superconcept is no longer relevant. For example, the fact that **adult** was a subconcept ‘by age’ of **person** is of little relevance to the concept **Class 3 school bus driver** (all of whom happen to be adults). There are cases, however, when one might be interested in inherited dimensions, and it is for this reason that the additional metaconcept property ***inherited dimensions*** was created. The concept **girl**, for example, might inherit from both **child** and **female person**. Its inherited dimensions would be ‘by sex’ and ‘by age’. Note that ***inherited dimensions*** does not use the ordinary inheritance mechanism because it is a metaconcept property.



```
· properties: a set of thing
  graph layout position: ih(21@33:figure42)
  dimensions: by aliveness
  related concepts:
    subconcepts: (concept of person, concept of dog)
      instances:
    superconcepts: concept of thing
    disjoint concepts: concept of nonliving thing
    source properties: (concept of owns, concept of limbs, conc
    terms: living thing
```

```
by aliveness
```

Figure 3.12: Metaconcept properties. Shown are statements whose subject is the metaconcept **concept of living thing** *and whose predicates are the metaconcept properties possessed by that metaconcept. All of the properties shown are primitive*

Another primitive metaconcept property related to dimensions is ***kinds***. Statements whose predicate is the ***kinds*** property contain text that describes which subconcepts are in which dimension. For example, the value of ***kinds*** for **concept of person** may be listed as: ‘by age: (concept of child, concept of adult); by sex: (concept of female person, concept of male person)’. The reason why ‘concept of…’ precedes each concept name is that what are being listed are metaconcepts.

Figure 3.13 illustrates some of the above ideas:

• **thing** is divided into two dimensions, ‘by aliveness’ and ‘by truth’.

• The value of **concept of thing**’s ***kinds*** property would be listed as: “by aliveness: (concept of nonliving thing, concept of living thing); by truth: (concept of nonfictional thing, concept of fictional thing)”.

• **concept of living thing** would have ‘by aliveness’ as the value of its ***dimensions*** property. This is indicated in figure 3.12, and by the label on the arc in figure 3.13.

• **concept of unicorn** would have ‘by truth, by aliveness’ as the value of its ***inherited dimensions*** property.

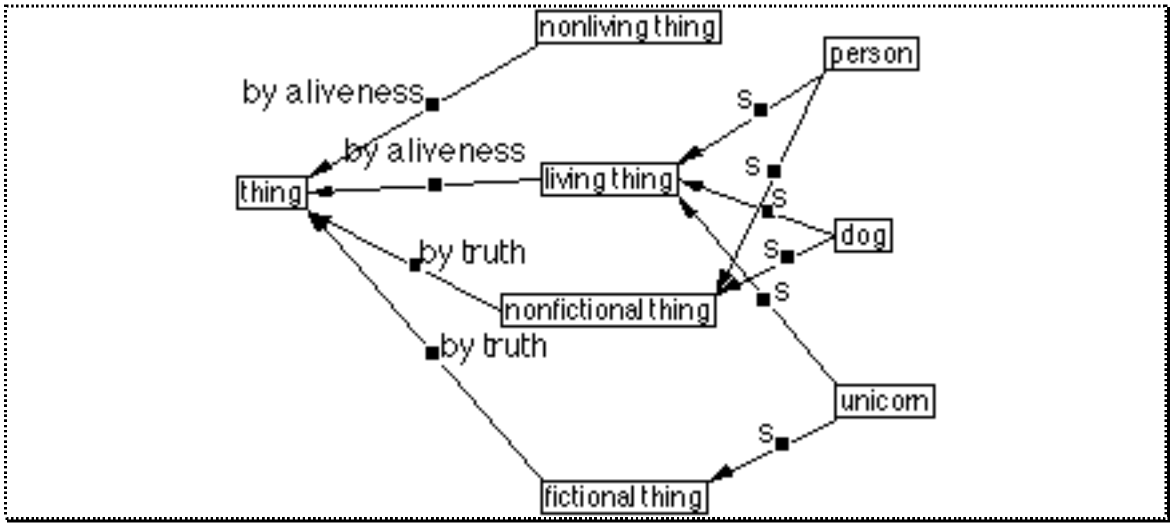


Figure 3.13: An example showing dimensions. Thing is divided into two dimensions, each containing two concepts. Each dimension specifies a criterion by which thing is divided into subconcepts. This figure was drawn by CODE4.

3.8.2 Intended semantics

From the perspective of semantics, metaconcepts are instance concepts just like any other. Their single point of extension is another CODE4-KR concept, whereas the extensions of non-metaconcepts are things outside the currently running CODE4 system.

In the current implementation of CODE4-KR, the ***dimensions***, ***inherited dimensions*** and ***kinds*** metaconcept properties do not have an *intended* semantics. The values of statements involving these are merely character strings.

3.8.3 Observations on use

Most users understand the idea of metaconcepts, although for most users usage of them is limited to:

• Looking at listings of metaconcept properties that are automatically maintained.

• Editing the ***comment*** (i.e. English description).

Dimensions have been found to be an important mechanism for organizing knowledge. A similar idea with this name exists in Classic. Library scientists, however, call a taxonomy breakdown where there are several independent criteria *facetted classification* (Vickery 1960; Hunter 1988; Dahlberg 1993) (there is no relation to CODE4's facets). The principle is called *discriminators* in OMT (Rumbaugh, Blaha et al. 1991).

3.8.4 Knowledge management problems addressed by metaconcepts

Metaconcepts address the following problems:

• **Problem I-3: Making distinctions**. Metaconcepts allow concepts to be distinguished from the thing they represent.

• **Problem I-4: Understanding effects**. Due to the presence of metaconcepts, inheritance is simplified. This helps the user understand what will occur when a particular value is specified.

3.9 Primitive concepts

3.9.1 Practical effects

When a knowledge base is first created, a set of primitive concepts are installed, and terms for them are also installed.

There are two classes of primitive concepts, primitive types and primitive properties. Primitive types are primarily just placeholders in the inheritance hierarchy, whereas primitive properties have several reasons for existence including: 1) giving rise to statements that display computable values and 2) optimization of storage.

In no case is the name of a primitive concept fixed, although they all have default names. Also, with only a few exceptions they may be moved around in the inheritance hierarchy or property hierarchy. Obvious exceptions are that the top concept, **thing**, which must remain at the top of the inheritance hierarchy; and the top property, ***properties***, which must remain at the top of the property hierarchy[48]. Also the most general subject of primitive properties must not be changed such that the properties no longer inherit to their original set of concepts.

[48] These two 'tops' do not impose an ontological committment on CODE4 users. The concepts are merely collecting places; users are not forced to do anything with them.

Primitive types

Primitive types behave just like ordinary types except in the following ways:

- They are created when any knowledge base is first created.
- The user cannot delete them.

The most important primitive type is the top concept, usually labelled ‘thing’. A top concept must exist for all other concepts to be subconcepts of. The four other primitive types are, respectively, the common superconcept of all:

- metaconcepts,
- statements,
- terms, and
- properties.

It is rare for users to directly access the hierarchies below these primitive types (users use primitive operations instead), however the four primitives must exist so that CODE4 knows where to put the special concepts when it creates them. These concepts are named according to the class of concept of which they are a superconcept. For example, the superconcept of all statements is **statement**[49].

Primitive properties

Primitive properties are distinguished along the following three dimensions:

1) by which special class of concept they are inherited (metaconcepts, statements, terms or properties);
2) by their purpose (to optimize storage or for the display of computed values), and
3) by the degree to which users can edit the resulting statement values (no editing allowed; editing constrained by rules, or unlimited editing allowed).

Metaconcepts inherit the most primitive properties (discussed in section 3.8). Statements come next (i.e. inheriting the primitive facet properties discussed in section 3.6) followed by terms and properties. Details of the primitive properties of each class of concept can be found in previous sections.

Computed primitive properties

Most primitive properties are intended to give rise to statements that display computed values; these are called *computed properties*. Examples are: ***subconcepts***, ***subject***,

[49] In older knowledge bases, the terms ‘statement within self’ or ‘statement in this KB’ are used, but users found this confusing so it was changed.

superproperties, and ***meanings***. In an alternate design, some computed properties could be ordinary properties[50] whose values would use the standard property-lookup mechanism. However not all could be ordinary without a paradox being created.

For example, the most important computed metaconcept property is ***superconcepts***. But this property inherits from the common superconcept of all metaconcepts, and thus to look it up CODE4 would need to know about superconcepts in advance; the actual values of computed properties are therefore obtained by implementation-dependent internal mechanisms.

The values of most computed properties contain lists of concepts as their value items; there are a few exceptions however, which can contain strings:

• The ***value*** property itself.

• The ***string*** property of terms.

• The ***dimensions*** property of metaconcepts.

Optimized primitive properties

A few properties whose values contain documentary information are only primitive because they are used very frequently, hence it is desired to increase performance and reduce memory requirements.

An example of this is the metaconcept property ***comment*** (also often renamed 'English description'). This is optimized to reduce the necessity of turning virtual metaconcepts into real ones: In general, CODE4 does not have to explicitly create metaconcepts – information about conceptual relations is maintained using primitive mechanisms associated with the underlying concept (if this optimization were not done, then the whole infinity of implicit metaconcepts would need to be created!). As soon as a user edits a metaconcept, however, by adding a property or filling in the value for some non-primitive property, the metaconcept must be explicitly created. Since ***comment*** is the most heavily used non-computed metaconcept property, it is primitively implemented to save space and computations.

Another example of an optimized property is ***knowledge reference***: This is a facet property and the optimization prevents the creation of real facets from virtual ones. In addition to providing performance optimizations, the presence of non-computed primitive properties helps remind the user about useful statements that she or she should make.

---

[50] i.e. non-primitive properties.

Editable and non-editable primitive properties

The values of most primitive properties are editable: i.e. the user can change them. In the case of a value that expresses a conceptual relation such as ***subconcepts***, editing has the effect of changing the relation (in this case changing the inheritance hierarchy). Each editable primitive property has special editing rules for its values to prevent a violation of structural constraints (e.g. no loops in are allowed in the inheritance hierarchy).

In general, changes to computed primitive property values can be effected in multiple ways. For example, the following are the ways that a user can edit the dimension property of the concept of **child**:

- By directly editing the value of *dimension* in concept of child.
- By editing the label of the superconcept link to person.
- By editing the value of the *kinds* property of concept of person.

Regardless of the method used, all three manifestations of the relationship are changed synchronously.

A few primitive properties have values that cannot be edited; for example, the predicate and subject of a statement. This is because a statement is *defined* by its predicate and subject, as discussed in section 3.6.

3.9.2 Intended semantics

There is no intent to force the user to accept ontological commitment by the default positioning of primitive concepts. However, the default conceptual structure is intended to show the important distinction between concepts and non-concepts.

3.9.3 Observations on use

Users rarely directly edit the values of primitive properties that express conceptual relations; they normally use primitive operations to do the manipulation, only observing the results in computed values.

3.9.4 Knowledge management problems addressed by primitive concepts

Most artificial intelligence-based knowledge management technologies have the idea of primitive concepts, although they are not typically called by that name. As in other systems, CODE4 primitive concepts are a *supporting* mechanism for properties, statements, terms and metaconcepts. The reader should thus refer to the previous four main sections (3.5 to 3.8) for details about the problems they address.

3.10 Further details of inference mechanisms

This section provides additional details about the inference mechanisms provided in CODE4. As explained earlier, since users of CODE4 are largely concerned with its structuring techniques, and since they are free to violate many constraints that would enforce CODE4's intended semantics, only several simple inference mechanisms are provided.

Additional inference capabilities can be added on top of those in the knowledge engine:

- CODE4's user interface provides a 'mask' capability (section 4.3) that constrains what is displayed based on potentially complex criteria.
- CODE4's knowledge map capability (section 4.2) extracts useful patterns of knowledge from a knowledge base.
- Various CODE4 researchers have added facilities such as: a) the ability to embed Smalltalk expressions in values so the values can be computed using arbitrary formulas, and b) a simple expert system that extracts the knowledge it needs from CODE4 (see section 3.13).

3.10.1 Delegation

Delegation is a mechanism whereby a statement's value is determined by referring to the values of other statements; hence it can be used to specify certain kinds of constraints. Delegation can be considered a generalized form of inheritance, but whereas inheritance always occurs unless overridden, the knowledge enterer must explicitly request delegation.

The following is an example of delegation in action: If a person owns something, then he or she is almost always the owner of that thing's parts. It would be nice, therefore, to be able to specify the value of the owner property and have the owner properties of the parts of that entity take on the same owner automatically. As with some other KR systems, e.g. Cyc, it can be said that the 'owner' property *transfers through* the ***parts*** property. To achieve this result in CODE4, one would, at the most general subject of ***part-of*** (e.g. **entity**) specify that the value of ***owner*** should be the same as the ***owner*** of whatever the entity is ***part-of***. A special syntax is provided for this which is documented in the CODE4 reference manual (Lethbridge and Eck 1994).

Delegation can be considered similar to the formula mechanism provided in spreadsheets. Indeed, a reference can be absolute or relative, and most references are relative. In the above example, the delegation directive is inherited by all subconcepts of **entity**, but the owner of all **entity**'s subconcepts is not the same as the owner of **entity**. Instead the owner is the determined relative to the current subject – in other words in the concept **car-engine**, the owner would be determined by looking up the owner of whatever a car-engine is part of (presumably **car**).

3.10.2 Inheritance

In the context of CODE4 there are two senses of the verb ‘to inherit’:

- **Property inheritance**: This is described in section 3.5. A concept always possesses those properties possessed by its superconcepts.
- **Value inheritance**: This is derived from the first sense and is described in section 3.6. Value inheritance is a special case of facet inheritance.

In section 3.6, it was explained how the value facet inherits; in fact the value facet is the only facet that *always* inherits, and it does so using a primitive mechanism. The values of other facets *never* inherit unless explicitly directed to do so using a delegation directive (see above); this directive is specified at a facet property’s most general subject (usually **statement**). Each facet inherits independently, i.e. if the value is overridden in a subconcept, other inheriting facets would still inherit unchanged.

3.11 Knowledge organizing techniques at the representation level

This chapter so far has presented details of CODE4-KR, including details of its organizing techniques. This section summarizes and categorizes the organizing techniques. The evaluation of how much the techniques contribute to practicality is left to chapter 6.

An organizing technique is a way of thinking about a knowledge base; a way of mentally modelling how concepts interrelate to each other. The techniques can be considered ‘dimensions’ of a knowledge base, since they are to some degree orthogonal to each other (one need not be concerned about other dimensions when working within one). As a user is arranging concepts, he or she typically thinks in terms of several organizing techniques; but at any point in time, decisions are being made within just one. For example, the user might spend some time organizing the inheritance hierarchy (one organizing technique) and then may spend more time thinking about synonyms and parts of speech (linguistic knowledge – another organizing technique).

The following list presents the organizing techniques in a sequence in which it would be reasonable to learn them. However, to learn a subsequent primary organizing technique (A to G), it is not necessary to know the special features of a given technique. Each technique is characterized in terms of the main knowledge representation decisions users must make.

The decision about which features should be considered primary organizing techniques, and which should be considered special features is not definitive. Other categorizations are possible.

A summary of the organizing techniques is found in figure 3.14, and a comparison of various knowledge representation systems is presented in figure 3.15.

A Main subjects in the inheritance hierarchy: At the very basic level the user merely lists and names concepts.

- What main subjects should be present?
- What should the order of the main subjects be?
- What should a concept be named?

A1 Categorizing: At this stage a true inheritance hierarchy is being built.

- What should be the children and parent of a main subject?
- How should main subjects be grouped?

A2 Multiple categorizing

- What set of parents should a main subject have?

A3 Types and instances

- Should a main subject be a type or an instance?

A4 Dimensions

- How should subconcept links be labelled?
- Based on what criteria should a concept be divided into subconcepts?

A5 Disjointness

- Which concepts should be disjoint from others?

B Properties: The second fundamental organizing technique involves the user 'saying things' about concepts.

- What properties should a concept 'possess'?
- What should be the order of properties?
- What should a property be named?
- What should the value of a property be at a subject?

B1 Inheritance: Here the user learns the simple inheritance rule of CODE4.

- What properties should a concept and all its subconcepts have?

B2 Property categorizing

- What should be the children and parent of a property?
- How should properties be grouped?

B3 Property multiple categorizing

- What set of parents should a property have?
- How are values combined under multiple inheritance?

B4 Global and local property hierarchy

- How should the global property hierarchy be organized?
- What subhierarchy should each concept possess?

C The uniformity of concepts: Properties and main subjects are subsumed by the idea of concepts. This organizing technique is very fundamental to subsequent ones.

- What properties can properties and other special concepts have?

D Statements as concepts:

- What properties (facets) should a statement have?
- What should the modality of a statement be?

D1 Statement hierarchies

- How should statement hierarchies be derived from the property hierarchy?

D2 Facet hierarchies

- How should a hierarchy of facets (facets of values and other facets) be arranged?

D3 Subordinate values in other facets

- What other values should a statement have?

E Formal and informal values

- Should a value be formal or informal?
- To what concepts should a formal value point?

E1 Arbitrary relation networks

- How should arbitrary networks of concepts be arranged?

E2 Delegation networks

- From what other values should a statement obtain its value and using what method of calculation?

E3 Value specialization and modality

- How necessary is a statement or value?
- How should values of statements be specialized down the inheritance hierarchy?

E4 Value subsetting down the statement hierarchy

- How should values of substatements be related to values of superstatements?

E5 Value-dependent subproperties

- What special statement should be described by virtue of a value item?

F **Metaconcepts**:

- What properties are of the concept rather than the underlying thing?
- What properties should not inherit?
- How should the hierarchy of metaconcepts be arranged?

G Terms and linguistic capabilities

- What names should a concept have?
- What graphic symbols should a concept have?
- What properties should a term have?
- What meanings should a term have?

Organizing technique	Prerequisite techniques	Section discussed	Expertise	Novelty in CODE4
A. Main subjects & inheritance hierarchy		3.2	Beg -	-
A1. Categorizing	A	3.2	Beg	-
A2. Multiple categorizing	A1	3.4	Int	-
A3. Types and instances	A1	3.4	Int	Low
A4. Dimensions	A1	3.8	Int	High
A5. Disjointness	A2	3.4	Exp	Med
B. Properties & property hierarchy	A	3.5	Beg -	-
B1. Inheritance	A1, B	3.5	Beg	-
B2. Property categorizing	B	3.5	Beg	High
B3. Property multiple categorizing	B2	3.5	Int	High
B4. Global and local property hierarchies	B2	3.5	Int	High
C. The uniformity of concepts	A	3.2	Int	High
D. Statements as concepts	B, C	3.6	Int	High
D1. Statement hierarchies	D,B4	3.6	Int	High
D2. Facet hierarchy	D	3.6	Exp	High
D3. Subordinate values in other facets	D2		Exp +	High
E. Formal and informal values	B	3.6	Int	Med
F. Metaconcepts	C	3.8	Int	High
G. Terms and linguistic capabilities	C	3.7	Exp	High

Figure 3.14: Summary of knowledge organizing techniques. The prerequisites column indicates the techniques that must be learned in advance, and which must be present in a system before a given technique can be added. The expertise column indicates the techniques that a given level of user typically understands (beginner, intermediate, expert). The novelty column indicates how rare it is to find the feature in other knowledge management systems such as those discussed in chapter 2.

3.12 Comparison of CODE4-KR with other knowledge management technologies

Figure 3.15 provides a comparison between CODE4-KR and some of the knowledge management technologies discussed in chapter 2[51]. All of these technologies have certain basic features like multiple inheritance hierarchies and the ability to attach properties to concepts. The following are some of the main ways these tools differ from CODE4:

[51] Technologies such as hypertext and personal productivity software, whose strength is in their user interface, are not listed. None of the techniques apply to them.

- Some of the tools treat types and instances significantly differently from CODE4. In KL-One derivatives, instances (individuals) are not represented using the same language as types. These tools also focus on automatic classification (discussed in section 2.1.3). In KM and Cyc, a concept can be both a type and an instance at the same time.
- Several of the tools support dimensions as in CODE4, but this feature does not appear in Cyc and KM.
- Tools like Cyc and KM allow the user to organize slots in a hierarchy, but such a hierarchy is treated very much like the inheritance hierarchy. Also, the property hierarchy is not really used to help organize the knowledge in frames. In CODE4, properties are instance concepts, and the property hierarchy is fundamentally important in the organizing of knowledge.
- CODE4 gives the word ‘concept’ a wider definition than other tools. This introduces more uniformity and flexibility into the way knowledge is organized.
- No tool, aside from CODE4, treats statements as concepts; i.e. as units of knowledge that can have properties etc.
- None of the technologies allows for the flexible integration of formal and informal values. Although tools such as Cyc allow certain slots to be filled with character strings, these are considered ‘special’ slots.
- None of the technologies supports what CODE4 calls metaconcepts. Cyc and KM allow higher order knowledge to be represented, but both mix properties that in CODE4 would be divided between a concept and its metaconcept.
- None of the technologies support terms or the management of linguistic knowledge in as flexible a manner as CODE4. The flexibility of Cyc and KM allows separate concepts to be created that could represent terms; but inflexible internal identifiers would still have to be used when actually representing facts.

Organizing technique	CODE2	KM & Cyc	KL-ONE derivatives	OMTool
A. Main subjects & inheritance hierarchy	•	•	•	•
A1. Categorizing	•	•	•	•
A2. Multiple categorizing	•	•	•	•
A3. Types and instances	•	~	~	•
A4. Dimensions	~		•	•
A5. Disjointness		•	•	•
B. Properties & property hierarchy	•	•	•	•
B1. Inheritance	•	•	•	
B2. Property categorizing	~	~		
B3. Property multiple categorizing		~		
B4. Global and local property hierarchies				
C. The uniformity of concepts		~		
D. Statements as concepts		~		
D1. Statement hierarchies				
D2. Facet hierarchy		~		
D3. Subordinate values in other facets		~		
E. Formal and informal values	~	~		~
F. Distinct metaconcepts		~		
G. Terms and linguistic capabilities		~		

Figure 3.15: Comparison of organizing techniques in various representations. A bullet indicates that the technique is a feature of the tool and works in a manner similar to CODE4. A tilde indicates a capability significantly different from CODE4. A blank indicates the technique is not applicable to the tool.

3.13 CODE4-KR as an abstract schema

The descriptions of the knowledge representation covered so far in this chapter have been very abstract in nature: Little commitment has been made to appearance, implementation, syntax, or medium or mode of transmission. This section discusses how one can consider CODE4 to have several different knowledge representation schemata, all of which are manifestations of the *abstract schema* described so far.

3.13.1 Alternate schemata

Figure 3.16 shows various schemata whereby the CODE4-KR abstract schema can actually be manifested. Figure 3.17 shows how they are embodied in the CODE4 architecture.

The schemata can be divided along several dimensions:

- The top and bottom of the figure distinguish static from dynamic schemata. Dynamic schemata describe syntaxes for commands that allow a knowledge base (a representation using one of the static schemata) to be modified or updated.
- The left and the right of the figure distinguish physical from mediating representations. Physical representations either use the CKB syntax discussed in the next section or an object oriented representation in memory. Mediating representations display knowledge on the screen and are discussed in chapter 4.

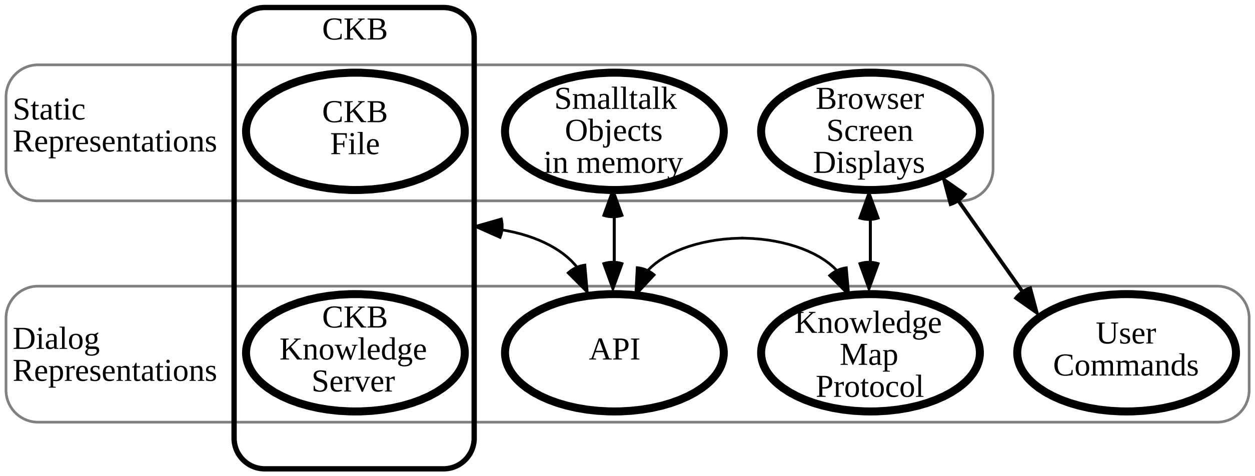


Figure 3.16: Seven manifestations of CODE4-KR. Each ellipse represents a distinct syntax into which the abstract schema of CODE4-KR is mapped. All manifestations have a common semantics. Arrows show syntax translation paths.

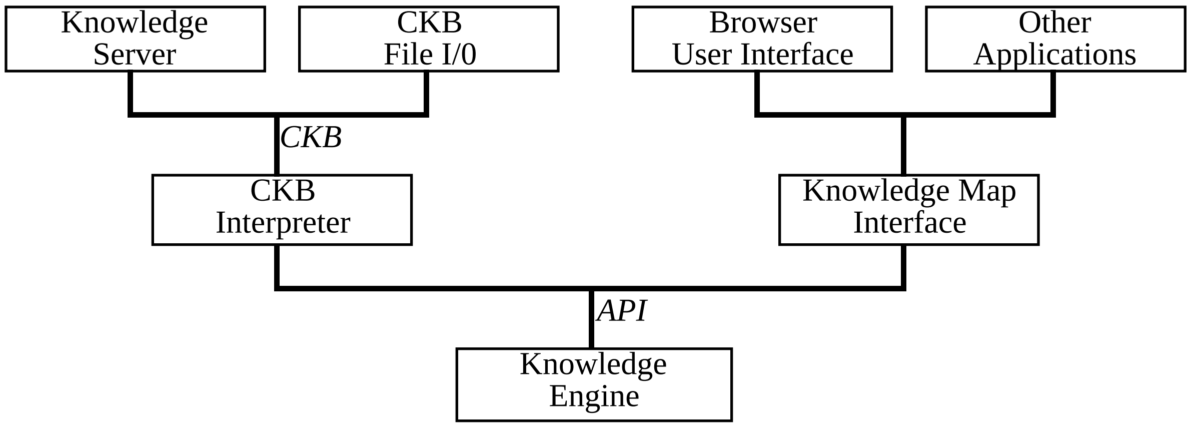


Figure 3.17: The top-level architecture of CODE4. At the top are various applications which interface to the knowledge engine.

3.13.2 The CODE4 API and its manifestation as the CKB syntax

All operations on physical representations of CODE4-KR are performed using a limited set of well-defined commands. The commands are partitioned into two major sets:

- Those that update the knowledge base are called *modifiers*.
- Those that query the knowledge base are termed *navigators* (because applications using navigators use the results of one query to construct the next query, and thus navigate around the knowledge base).

An important subset of modifiers are the *constructors*: commands that add concepts and links between concepts, but do not delete or merely change knowledge. The *basic constructors* form a minimal set necessary and sufficient to construct any CODE4 knowledge base.

The physical representation and the API

In-memory knowledge (i.e. as operated on by CODE4's 'knowledge engine') is stored as a network of Smalltalk objects[52]. The modifiers and navigators are implemented as a set of Smalltalk messages sent to these objects. These messages collectively form CODE4's application program interface (API), and are the only means by which in-memory knowledge can be queried or updated.

When an application dialogues with the API, it passes knowledge backwards and forwards. The API is therefore a language; in fact it is a distinct manifestation of CODE4-KR.

The knowledge map layer

CODE4's knowledge map layer provides another language. This provides high-level abstractions for the user interface and is described in section 4.1.

CKB Format

Another major application using the API is the CKB (CODE4 Knowledge Base) language interpreter. Expressions in CKB are ASCII representations of modifiers and navigators, and form yet another manifestation of CODE4-KR. Appendix D gives an example of CKB syntax.

---

[52] In fact some other physical representation could be substituted leaving all the other manifestations unchanged – it would only be necessary to ensure the new representation used the same API.

Uses of CKB format

CKB is used for two major purposes:

- **For persistent storage**: When knowledge is saved to disk, CODE4 generates the minimal set of basic constructor commands needed to regenerate that knowledge. When the knowledge is loaded from disk, the CKB interpreter translates the commands directly into API messages, and the knowledge base is thus reconstructed. From the perspective of the CODE4 knowledge engine, it makes no difference whether a memory-resident knowledge base was built using the knowledge map layer, the CKB interpreter or some other application.

- **For inter-process communications**: A running CODE4 system can be used as a *knowledge server*. CKB commands are sent using two-way communication between CODE4 and other software, possibly running at distinct geographical locations. This mechanism has been used for two purposes: 1) knowledge-based application programs (e.g. expert systems), and 2) knowledge translators.

  Peter Clark of the University of Texas (not yet published) has built a Prolog-based expert system called 'Electronic Trader' (ET) that attempts to make money by chosing sequences of currency, option and bond transactions. One version of this system connects to a remote CODE4 knowledge base to obtain the knowledge it needs to make inferences.

  To date, two translators have been built: One converts a subset of Ontolingua into CKB and pipes it into CODE4. The other does the same with KM.

Design of the CKB syntax

The CKB language is designed to be compact, but human readable. Human-readability (i.e. using a format composed of printable ASCII characters rather than binary) is believed to be important because it simplifies the job of writing and debugging two kinds of programs (corresponding to the above major uses of CKB format):

- Those that read or write CKB files.
- Remote clients that connect via telnet.

It is *not* an objective to provide sufficient human readability to allow for publication. Nor is it an objective to allow users to study the knowledge's content by reading this particular textual form[53]. This is because knowledge is typically composed of intertwined networks of concepts, and a tool like CODE4's user interface is needed to effectively study it.

[53] CODE4 does provide a number of formats whereby aspects of the knowledge can be printed for publication or study.

CKB's readability is limited by two important objectives: 1) compactness, and 2) extreme ease of machine-parsing. A goal is to minimize bandwidth and maximize the speed of file operations. As a rough rule of thumb, each concept uses about 30 bytes of disk space and each main subject (including the terms, statements, properties and metaconcepts it introduces) uses about 350 bytes of space.

Order dependence

A major difference between CKB and other textual knowledge representations like KIF, Ontolingua and KM is *order dependence*.

As was discussed earlier, CODE4-KR does not rely on external names to identify a concept. A consequence of this symbol independence is that when loading a knowledge base, a concept must be created before it can be referred to in subsequent commands. This imposes a partial ordering on a CKB file which is lacking in a KM or Ontolingua file. Although initially this may appear to be a disadvantage of CKB, in practice it is not. CKB files are never written by humans and there is no need for them to be; the CODE4 mechanism that writes CKB files can easily compute the partial ordering using a near-linear algorithm.

Order dependence does impose a constraint on two-way communication with the server: A client must obtain a reference to a concept (e.g. by name matching) before it can perform an update or query using it. This constraint appears to help maintain integrity: In a typical textual knowledge language, it is possible to assert facts that contain references to nonexistent concepts.

Chapter 4

User Interface Techniques for Practical Knowledge Management

This chapter continues the discussion of CODE4 by describing the main techniques by which the *presentation* of knowledge is organized.

Knowledge maps, described in section 4.1, provide an abstract mechanism for arranging knowledge that is intermediate between the concept level (described in chapter 3) and the mediating representation level (section 4.2). Several different mediating representations can be used to display the information in a knowledge map; and these can be organized into *browser hierarchies*. *Masks* (section 4.3) are used to highlight knowledge and control which details are displayed.

Figure 4.1 is an example CODE4 session showing various windows.

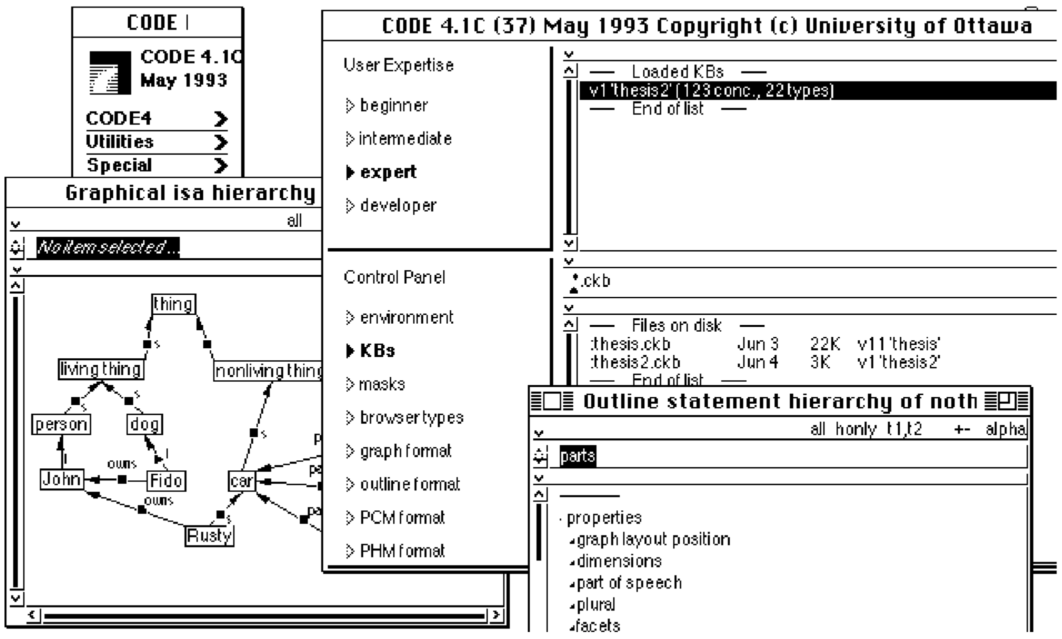


Figure 4.1: An example screen showing the CODE4 user interface. Clockwise from top left: The Smalltalk 'Launcher' which is used to start various applications. The control panel (section 4.4.1). An outline mediating representation (section 4.2.2) showing a statement hierarchy. A graphical mediating (section 4.2.3) representation showing a complex knowledge map.

4.1 Knowledge maps

A *knowledge map* is a software abstraction that allows for the manipulation of a network of related concepts. The knowledge map defines this network in terms of:

a) One or more starting concepts.
b) One or more relations that recursively relate the starting concepts to other concepts. These are discussed in more detail in section 4.1.1.
c) A limit on the traversal depth.

The *nodes* in the network are the starting concepts plus those concepts that can be reached by following the relations to the specified depth. The set of *arcs* in the network are the set of statements that link nodes.

The word 'map' is used instead of 'directed graph', which may be preferred by some mathematicians, for two reasons:

- The word 'graph' could cause confusion with the graphs drawn by the user interface. All mediating representations, not just graphs, use knowledge maps.
- The cartographical analogy is useful: The user can define the map he or she wants to look at, and can then navigate around that map. The user can use masks (associated with knowledge maps) to highlight part of the map. The user can change the kind of information displayed by changing the knowledge map's relations[54].

4.1.1 Details of how relations are specified

Each of a knowledge base's relations is specified by a concept, and each relation defines a set of arcs that will appear. A relation can be specified using any of the following three types of concepts:

1. **A property**. In this case the subject of an arc is the node at the origin of the arc. The predicate of the arc is the given property, or one of its subproperties. An arc appears with a given node as its origin if: a) there is a formal statement whose subject is that node and whose predicate is the specified property or one of its subproperties; and b) the knowledge map's traversal depth limit has not been reached.
2. **A property possessed by metaconcepts**[55]. An arc is defined in the same way as the above, except that its subject is now the *metaconcept* of the node at its origin. This

[54] This is analogous to changing whether a cartographical map shows major roads, all roads, railroads, topographical contours, or any combination of these.

[55] If a property is not inherited by a concept, but is inherited by its metaconcept, then the arcs are defined in this way.

is useful for displaying relations between *concepts*, rather than relations between the things the concepts represent.

3. **A statement**. This directly specifies a particular arc, which appears if its subject appears as a node (and if knowledge map traversal depth has not been reached). Specifying relations using individual statements is useful for the creation of very specific networks. Unless specific statements are used to define relations, the network will have as arcs *all* the statements whose predicate is a given property.

Any number of any combination of the above can be used to define a knowledge map's relations. For example, a complex knowledge map might show all of the following:

- An inheritance hierarchy (specified using the ***subconcepts*** metaconcept property).
- Links among concepts that represent things in a part-whole relationship (specified using the ***parts*** property)
- A few additional arcs of interest, e.g. showing particular people who own certain things (specified with a few statements whose predicate is ***owner***).

Figure 4.2 shows a graph that illustrates the above knowledge map.

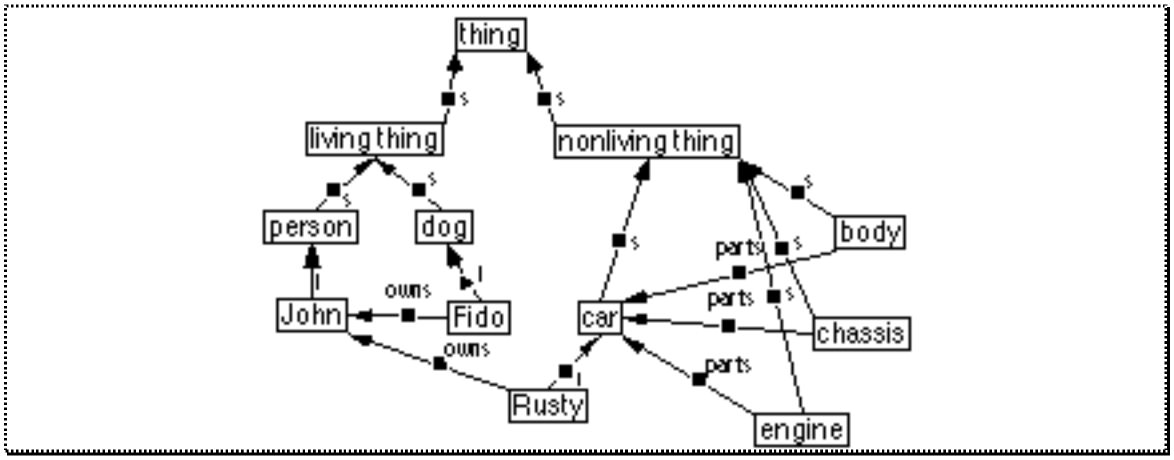


Figure 4.2: A graph showing a knowledge map with several relations. The starting concept is **thing**. *The relations are* ***subconcepts*** *and* ***part****, plus several* ***owns*** *statements.*

When several relations are used in the definition of a knowledge map, the first property listed is known as the *primary relation*. The importance of this will become apparent later.

It is important to note that ordinary users do not need to know any of the details in this section to work with knowledge bases. Such users have a few simple commands available that create the most useful types of knowledge map. More details on types of knowledge map and commands are found in the following two sections and in section 4.2.1.

4.1.2 Classes of knowledge map

The following are the main classes of knowledge map. All are defined in the manner described above, but each has a specific use and a simple command whereby the user can create it. There are separate commands, for example, to open an inheritance hierarchy or a property hierarchy. Also, the implementations of each class are optimized.

Inheritance hierarchies

Knowledge maps showing inheritance hierarchies have the ***subconcepts*** primitive property as their relation:

- The most commonly used knowledge maps are those which display the *entire* inheritance hierarchy. They have a single starting concept, the top concept: **thing**. Because the entire inheritance hierarchy can become very big, masks (section 4.3) are used to help constrain what is visible.
- A knowledge map that displays a *subtree* of the inheritance hierarchy would have a different starting concept, but the same subconcept relation. Several distinct subtrees are shown if there are several starting concepts.

Property hierarchies

These generally have the top property as their starting concept and use the ***subproperties*** primitive property as their relation. It is also possible to display subtrees of a knowledge base's property hierarchy, but this is rarely done.

Statement hierarchies

As their starting concept, these generally have a statement whose predicate is the top property. They use the ***substatements*** primitive property as their relation.

Arbitrary relations graphs

These are the most general knowledge maps, and also the most sophisticated since users must explicitly choose the starting concepts and relations. The following show how to construct particularly useful types of knowledge map:

- To display a finite state machine, create a knowledge map with several starting concepts that are instances of **state** and use the ***outgoing transition*** relation.
- To display a finite state machine that also includes superstates and substates (Harel 1987); do the same as the above, but *also* include the ***substate*** property as a relation.
- To show *all* the possible ways a set of starting concepts are related to each other, create a knowledge map that uses the top property as its relation, and has a traversal depth of one.

- To show all the possible human relations among a particular set of people, do the following: Make **person** possess a property ***related persons***, which has subproperties for various types of human relations (***father***, ***mother***, ***friends***, ***spouse*** etc). Then create a knowledge map whose starting concepts are a set of instance of **person** and whose relation is ***related persons***.

4.1.3 Commands available on the knowledge map interface

As figure 3.17 shows, all communication between the knowledge engine and the mediating representations that actually display knowledge is mediated by the *knowledge map interface*. The interface of the knowledge engine has a set of low-level functions to access and manipulate specific concepts. The interface of the knowledge map layer, on the other hand, is more abstract.

The same knowledge map command can be used for any relation. Thus the user does not have to remember separate commands to add subconcepts, subproperties, parts etc. There is just one command that performs the appropriate action depending on the relation. For example if an inheritance hierarchy is displayed (i.e. the primary relation is ***subconcepts***), then adding a child of a selected node creates a subconcept. On the other hand, if a statement hierarchy is displayed, then a new substatement (and its corresponding subproperty) is created. This benefit is called *polymorphism* in the object-oriented world.

Another way by which knowledge maps provide abstraction is that they allow operations on *sets* of nodes and/or arcs, or even on entire subhierarchies. To do this, knowledge maps automatically maintain a set of *selected* concepts, and one *master selected* concept. These selections can contain either nodes or arcs. More details about selections can be found in section 4.2.1.

The following are the main classes of commands available on the knowledge map interface. Details are not specified since they can be found in the CODE4 user manual (Lethbridge and Eck 1994).

- Adding children or siblings of one or more nodes.
- Deleting one or more nodes or arcs.
- Renaming a node.
- Changing the parent of one or more nodes (reparenting).
- Adding a new parent of one or more nodes.
- Changing the order of sibling nodes.
- Creating a new knowledge map whose starting concepts or relations are one or more nodes or arcs.

4.1.4 Knowledge management problems addressed by knowledge maps

The following are ways in which knowledge maps are designed to help with knowledge management problems (as listed in section 1.4):

- **Problem I-5: Extracting knowledge**. Knowledge maps provide useful ways to define the kind of knowledge that should be extracted. They can also be hooked together so that concepts selected in one knowledge map can change the definition (starting concepts and/or relations) of another knowledge map. This provides the basis for a very flexible browsing mechanism, described in the next section. Knowledge maps are also designed to work with masks, described in section 4.5.
- **Problem A-1: Special purpose restriction**. Knowledge maps are a very general purpose mechanism.
- **Problem A-2: Expertise restriction**. Despite the power of knowledge maps when used by experts, the beginner need see none of that. For the beginner, knowledge maps mean learning fewer commands (due to their polymorphic nature). Knowledge maps also form the foundation on which user-friendly mediating representations can be built. Because knowledge maps provide the same interface to each mediating representation, the latter resemble each other more closely and hence are easier to learn.

4.2 Mediating representations and browsing

In CODE4, a *browser* is a mediating representation that operates on the set of concepts defined by a knowledge map. CODE4 has three types of browsers:

- The *outline* mediating representation displays knowledge in a manner similar to an outline processor, using indentations to show relationships.
- The *graphical* mediating representation displays knowledge as a two-dimensional network of nodes and links.
- The *matrix* mediating representation displays knowledge like a spreadsheet, with cells organized into rows and columns.

There is also a fourth class of mediating representation called the ‘user language’ representation. A member of this class is not called a browser because its purpose is purely to allow the value of a *single statement* to be edited. It uses a degenerate knowledge map with the single statement as its starting concept, and a null relation. It is called a mediating representation in its own right, distinct from the others, for the following reasons:

- To maintain a one-to-one mapping between knowledge maps and mediating representations.
- Because it has a distinct set of commands, e.g. to perform special formatting.
- Because it is a distinct unit in the browser hierarchies discussed in the next section.

4.2.1 Common features of CODE4's mediating representations

The following are descriptions of general features shared by two or more of CODE4's mediating representations. Some features are inapplicable to the user language mediating representation and others are inapplicable to the matrix representation, but they all apply to the outline and graphical representations.

Uniform ways of making selections

Since browsers display *sets* of concepts, CODE4 provides various ways of manipulating these sets. Of most importance are ways to *select* such sets so that commands may be performed on them. Any combination of nodes and arcs may be selected in the following ways:

- By clicking with the mouse on a concept (or sweeping the mouse over a set of concepts), possibly with combinations of special keys (shift, control) also pressed, the user can: a) select or deselect concepts, b) add to or subtract from an existing selection, c) add or subtract whole hierarchies.
- Any command that creates a new concept always selects it so subsequent commands can operate on it. This facilitates performing sequences of commands rapidly.
- The user may use the *highlighting mask*, described in section 4.3, to select concepts that fulfill arbitrary criteria.

Details of the procedures to select concepts can be found in the CODE4 user manual.

Any selected concept appears in reverse video. Figure 4.3 shows three selected concepts (in two different subwindows): **person**, **dog** and ***owns***. Among every selected set of concepts, the most recent one selected is always considered the *master selection*. The master selection is always surrounded by a 'barbershop' pattern, and is used whenever some operation (such as renaming) can only logically operate on a single concept. In figure 4.3, **person** and **owns** are master selections.

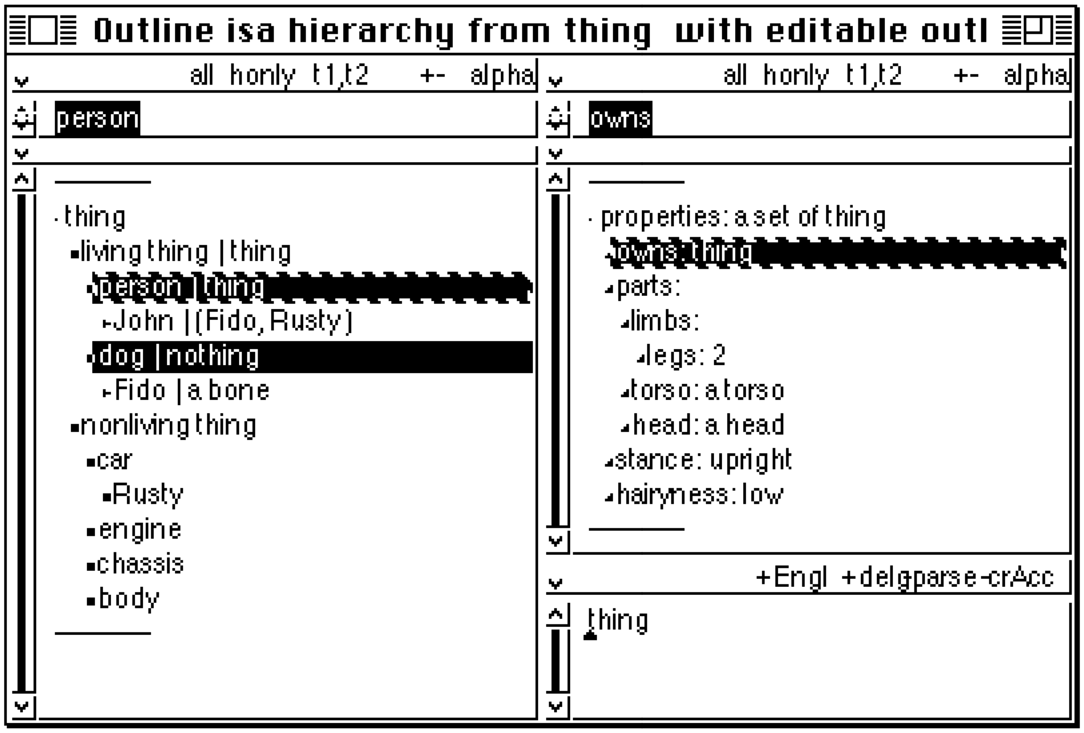
When a set of concepts is selected, that selection is passed to the underlying knowledge map. This may result in the changing of any driven knowledge maps as described in the next subsection.

Organization of mediating representations into hierarchies

It was mentioned in section 4.1 that knowledge maps can be linked together so that the selection in one knowledge map controls the definition of another knowledge map. A knowledge map whose selection affects another is called a *driving* knowledge map (and the corresponding mediating representation is called a driving mediating representation or driving browser). Whole hierarchies of knowledge maps, and thus mediating representations, can be constructed 'on the fly' using this mechanism. These are called *browser hierarchies*, and are considerably more powerful than the kinds of browsers made popular by the Smalltalk environment (Goldberg 1984).

Figure 4.3 shows three mediating representations arranged in a hierarchy.

- At the left is an inheritance hierarchy. The concepts selected in this determine what is shown in the other two subwindows.
- At the upper right is a statement hierarchy. The subject of all statements shown is the master selection in the inheritance hierarchy[56], but a statement is only shown if its predicate is possessed by *all* of the concepts selected in the inheritance hierarchy. The statement selected in the statement hierarchy is passed to the bottom-right subwindow.
- The bottom right subwindow contains a user language mediating representation that displays the value of the statement selected in the statement hierarchy.



```
Outline isa hierarchy from thing  with editable outl

all  honly  t1,t2   +-  alpha        all  honly  t1,t2   +-  alpha
person                               owns

· thing                              · properties: a set of thing
  ·living thing | thing                ·owns: thing
    ·person | thing                    ·parts:
      ·John | (Fido, Rusty)              ·limbs:
    ·dog | nothing                         ·legs: 2
      ·Fido | a bone                     ·torso: a torso
  ·nonliving thing                       ·head: a head
    ·car                               ·stance: upright
      ·Rusty                           ·hairyness: low
    ·engine
    ·chassis                         +Engl  +delgparse-crAcc
    ·body                            thing
```

Figure 4.3: A simple compound browser. On the left is an outline mediating representation showing an inheritance hierarchy. On the right (upper part) is another outline mediating representation showing a statement hierarchy. At the bottom right is a user language mediating representation. Near the top of both outline subwindows is a pane where the user can edit the text of the master selected concept (i.e. **person** *or* **owns***). At the top of all three mediating representations is a list of useful commands (e.g. 'all', 'honly').*

56 If no concept is selected in the inheritance hierarchy, then the top-right subwindow becomes a property hierarchy.

Any selection made in a driving subwindow can thus completely change the knowledge displayed in all driven subwindows. This permits rapid navigation of a knowledge base. The depth and breadth of browser hierarchies is unlimited, although hierarchies created by users rarely exceed four levels deep and twelve mediating representations in total.

Organization of mediating representations into compound browsers

A window containing one or more browsers is called a *browser window*. When such a window contains more than one mediating representation, it is called a *compound browser*. The mediating representations within compound browsers are always in driver-driven relationships. To make it easy to build and use browser hierarchies, CODE4 provides a set of predefined templates for browser windows, most of which describe compound browsers. Figure 4.3 illustrates the most popular of these.

Within a browser window, the subwindow at the top right contains the *primary browser* (its knowledge map is hence called the primary knowledge map).The other browsers in the window (if any) are driven, directly or indirectly, by the primary browser.

To open a new browser window, the user chooses from a menu of possibilities. There are two places from where the user can obtain such a menu:

- **From the control panel** (see section 4.4). Unless the user explicitly requests otherwise, the primary knowledge map of a browser window opened from the control panel uses default starting concepts (generally the top concept or the top property). The user may optionally, however, pick any starting concept from a menu.
- **From any other browser**. In this case, the starting concepts of the primary knowledge map are the selected concepts in the originating browser. The primary browser, however, may or may not be driven by the originating browser. The newly opened browser window is described as *dynamic* if its primary browser is driven (i.e. changes when selections in the driving browser change); otherwise it is termed *static*. A command is available to make any dynamic window static – i.e. so its primary browser is no longer driven. A dynamic browser window is automatically closed when its driving window is closed.

In addition to creating a separate window, it is also possible to add new driven subwindows inside an existing browser window. Furthermore, one can delete or rearrange subwindows.

Displaying the current state of the knowledge base

Unless otherwise specified, mediating representations keep themselves current with respect to the state of the knowledge base. In other words, whenever a knowledge base is updated, all mediating representations must regenerate their contents by querying their knowledge maps. Of course, various optimizations can reduce the amount of computation involved.

Commands are available to switch off this automatic updating to prevent repeated redrawing of complex displays. This can be useful, for example, when the user wants to rapidly enter knowledge in one window while ignoring others. When a mediating representation is out of date with respect to the knowledge base, its background colour changes to grey.

Uniform handling of commands

Mediating representations handle some user commands themselves (e.g. changing their appearance) but pass most to their knowledge map (see section 4.1.3).

Like some personal productivity software, CODE4 has a flexible way of allowing users to tailor how they issue commands. They can be issued in any of the following ways:

- By using particular combinations of keystrokes (definable by the user).
- By selecting from menus.
- By clicking the mouse on icons that appear at the tops of each mediating representation.

The bindings from particular keys, menu items etc. to particular commands can be readily tailored.

Additionally, two commands can be issued by directly typing text. These are: a) the command to rename the master selection (to change the value in the user language mediating representation), and b) the 'fast goto' command discussed in section 4.3. Aside from these textual commands, all commands in mediating representations have three sources of arguments:

1. The set of selected concepts in the knowledge map from which the command is issued.
2. The single master selected concept in the knowledge map.
3. A concept that has been *buffered.* There is a specific command to place in a buffer the currently master selected concept of any knowledge map. This remembers a particular concept that is then globally available to subsequent commands performed on the knowledge base. Buffering acts like the 'copy' function available in most graphical user interface environments, and indeed is an extension of this function (so that the user can 'paste' concepts, or textual representations of concepts, into various parts of the interface). The word 'buffer' is used, however, because 'copy' gives the user the wrong impression about what the buffer command does.
4. A numeric argument that the user can optionally type before issuing the command. If the user does not type a number, then the number 1 is assumed.

There are six basic types of commands; these use the different argument sources as follows:

a) **Commands that require one argument which can be a set of concepts, but which cannot be repeated** (e.g. deleting any number of nodes). These use argument source 1.
b) **Commands that require a set argument and which can be repeatedly performed on the same argument** (e.g. repeatedly adding child nodes). These accept argument sources 1 and 4.
c) **Commands that require one argument which cannot be a set (**e.g. renaming a concept) use argument source 2.
d) **Commands that require two distinct arguments** (e.g. moving concepts to a new parent) use argument sources 1 and 3.
e) **Commands that just take a numeric argument** (e.g. setting traversal depth). use argument source 4.
f) **Commands that take no arguments** (e.g. closing a window).

This scheme for supplying arguments to commands is relatively straightforward since each kind of operation naturally falls into one of the six categories. The only difficulty arises with commands of type d) where users must know which argument has to be buffered and which has to be selected. The labels in menus are designed to make this clear.

The above command structure helps with a major objective: Making the interface largely *non-modal*. Since the interface does not need to prompt users for arguments, they are free at almost all times to choose their next actions – this puts them in control of the interaction.

4.2.2 The outline mediating representation

The outline mediating representation displays one node per line. Indentation indicates the relationship between nodes. Child nodes are found at an indentation level one greater than their first parent, in whatever relationship is being displayed.

The most common use for outline mediating representations is a pair of them displaying, a) the inheritance hierarchy (complete or partial), and b) the statement hierarchy of a particular subject selected from that inheritance hierarchy. A browser with such a pair is shown in figure 4.3. However, it is also reasonable to use an outline mediating representation to display any knowledge map that describes a hierarchy or near-hierarchy[57] (e.g. a meronomy or part-of hierarchy). The outline mediating representation is less useful when cycles can occur (e.g. in a finite state machine). Figures 3.11 and 3.12 also show this mediating representation.

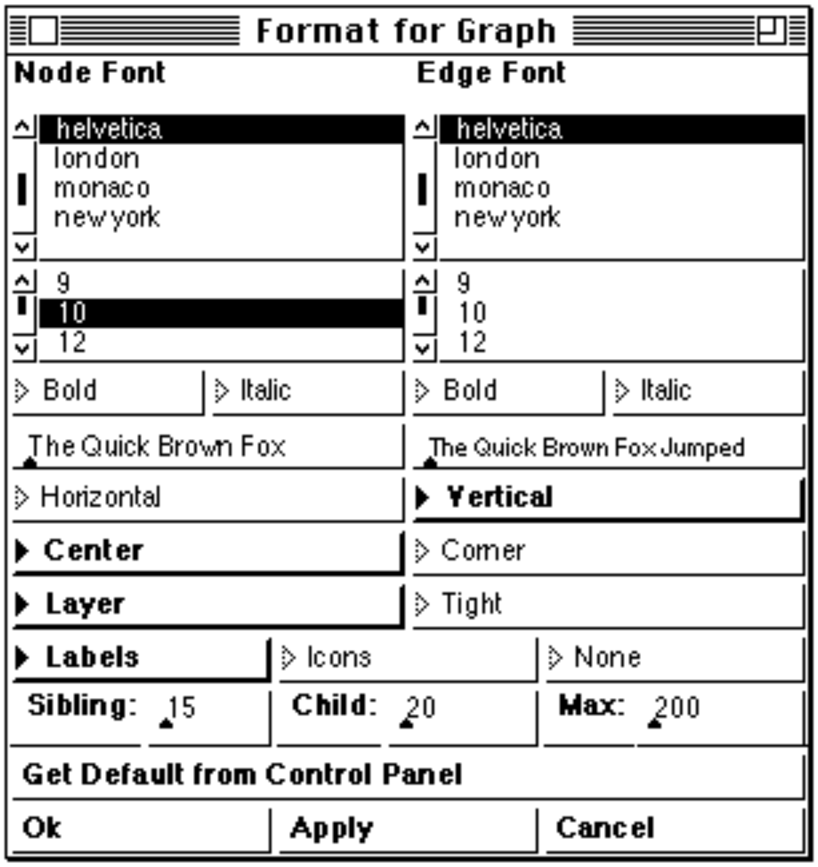
The outline mediating representations has the general features discussed above, plus the following special features:

[57] A near-hierarchy is a hierarchy with the possibility of a node being an immediate child of itself.

- When a concept appears multiple times (due to multiple inheritance, cycles in a graph etc.), the second and subsequent occurrence is followed by an ellipsis, '…'. To avoid infinite displays, no deeper levels of the hierarchy are redisplayed following such a repeated concept.
- Both nodes and arcs of the knowledge maps are displayed (and can be selected, edited etc). The arcs are the 'bullets' that appear on each line before the node name. Bullets have distinctive shapes depending on their predicate.
- In addition to a concept's name, the values of properties can also be displayed on each line. In figure 4.3, values of the property 'owns' are shown next to every subject that possesses this property. Also, statements are shown as the predicate name followed by the value, followed (optionally) by other selected facets.
- The display can be alphabetically sorted.

4.2.3 The graphical mediating representation

The graphical mediating representation displays nodes in boxes joined by lines which represent the arcs. All of the features of mediating representations in general (section 4.2.1) are applicable, plus the following special features:



Format for Graph

Node Font: [✓ helvetica] london monaco newyork

Node size: 9 [✓ 10] 12

Bold Italic

The Quick Brown Fox

Edge Font: [✓ helvetica] london monaco newyork

Edge size: 9 10 12

Bold Italic

The Quick Brown Fox Jumped

Horizontal [✓ Vertical]

[✓ Center] Corner

[✓ Layer] Tight

[✓ Labels] Icons None

Sibling: 15 Child: 20 Max: 200

Figure 4.4: Format options for the graphical mediating representation.

- Groups of selected nodes can be freely moved around a conceptually infinite ‘canvas’. Users can scroll the display by moving the cursor off the side of the window or by using scrollbars.
- Several graph layout algorithms can be invoked to rearrange the whole graph or just a portion of it. Figure 4.4 shows a non-modal dialog used to control which algorithm will be used as well as various other aspects of a graph’s appearance.
- The current layout of the graph (or of a particular subset of the nodes) can be saved for later recall. Layout information is stored as the values of a particular primitive metaconcept property. This facility helps users by recording their particular visualization of how concepts are arranged.

An example of a graphical mediating representation is in figure 4.2. Other examples are figures 3.2, 3.4, 3.5 and 3.13.

4.2.4 The matrix mediating representation

A matrix mediating representation displays the contents of a special knowledge map that contains several rows and columns of concepts. On one axis is a set of concepts (the subjects[58]) and on the other axis is a set of properties (the predicates). The intersections of rows and columns are ‘cells’ containing statements formed from subject/predicate pairs.

The starting concepts used to generate the display generally come from driving knowledge maps. A number of options are used to compute which subjects and predicates will *actually* appear.

The subjects can be:

- The starting concepts themselves.
- The siblings of the starting concepts.
- The coordinates of the starting concepts (those in the same dimension).
- All concepts above the starting concepts in the inheritance hierarchy.

The predicates can be:

- The starting concepts themselves (or the predicates of the starting concepts if the latter are statements).
- All the predicates possessed by all the subjects (i.e. the union of the subjects’ predicates).
- Only the predicates possessed by all the subjects (i.e. the intersection of the subjects’ predicates).

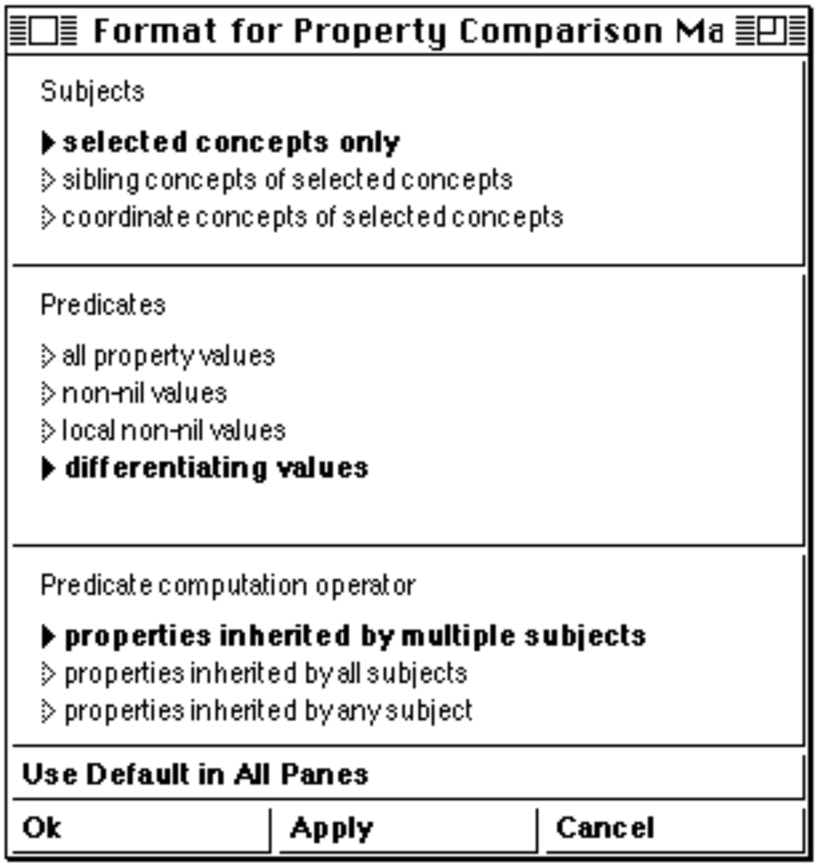
58 These concepts may not yet have anything said about them, so they may not strictly-speaking be called subjects. However, a major purpose of the matrix mediating representation is to allow users to turn them into true subjects by specifying values of statements.

• Only the predicates shared by two or more of the subjects (i.e. the union of pairwise intersections; or the union of predicates less those possessed by only a single subject).

Additionally, the predicates can be restricted to:

• Just those where one of the values is different from the others.
• Just those where at least one of the values is non-nil.
• Just those where at least one of the values is locally specified.

Figure 4.5 shows a non-modal dialog that can be opened to set most of these options.


Format for Property Comparison Ma

Subjects

▶ **selected concepts only**
▷ sibling concepts of selected concepts
▷ coordinate concepts of selected concepts

Predicates

▷ all property values
▷ non-nil values
▷ local non-nil values
▶ **differentiating values**

Predicate computation operator

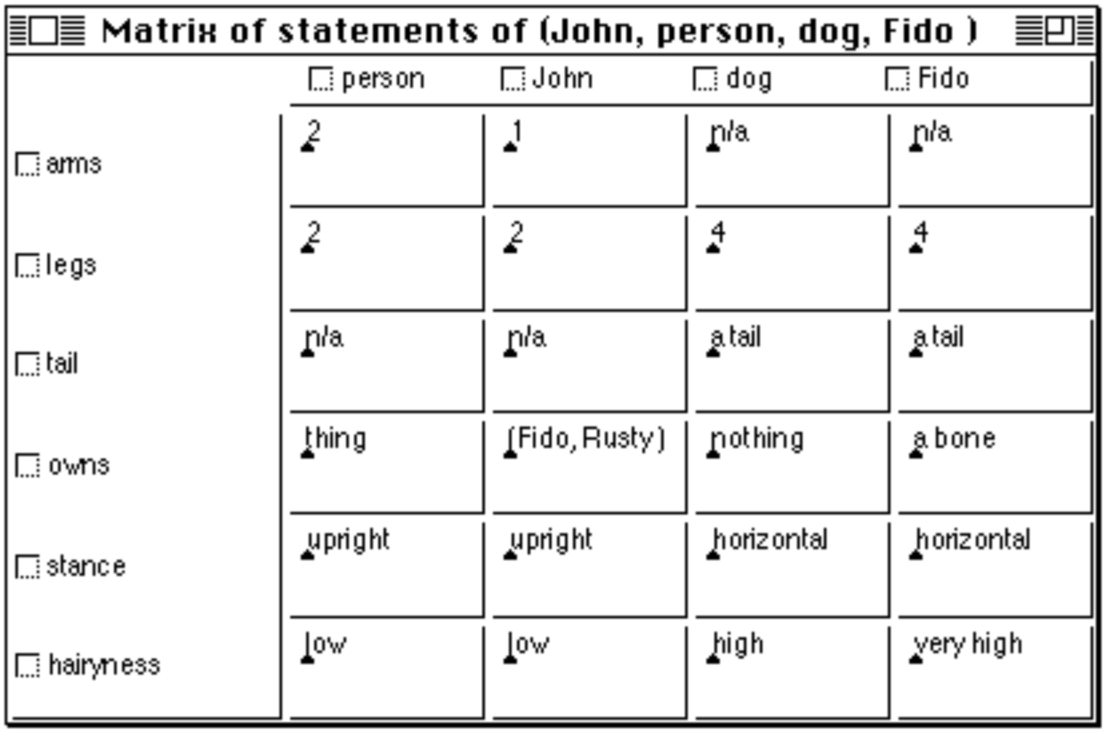
▶ **properties inherited by multiple subjects**
▷ properties inherited by all subjects
▷ properties inherited by any subject

Figure 4.5: Format options for the matrix mediating representation. These options, discussed in the CODE4 user manual, alter how the concepts selected in a driving browser affect what appears in a property comparison matrix.

The matrix mediating representation has many of the features also found in the graphical and outline mediating representations. These include: a) the ability for users to select concepts on which to perform commands; b) the ability to be driven by other browsers; c) the ability to be part of a compound browser; c) the fact that the knowledge displayed is always kept current, and d) the ability to work with the mask (section 4.3).

There are several important functions of the matrix mediating representation:

1) To help a user *understand the differences* among a set of concepts. The user studies the cells that differ. Figure 4.6 illustrates such a situation, where a user is comparing persons and dogs plus particular instances of each. To do this, users request the predicates that differentiate among the subjects.
2) To help a user *make distinctions* between concepts. The user displays all the predicates of the subjects and fills in appropriate cells.
3) To help a user understand how a predicate is refined in successive subconcepts. Figure 4.7 shows such a 'property history' matrix that illustrates how the ***legs*** and ***arms*** properties of **person** are derived. This is particularly useful when trying to understand how values are combined due to multiple inheritance.



Matrix of statements of (John, person, dog, Fido)

	person	John	dog	Fido
arms	2	1	n/a	n/a
legs	2	2	4	4
tail	n/a	n/a	a tail	a tail
owns	thing	{Fido, Rusty}	nothing	a bone
stance	upright	upright	horizontal	horizontal
hairyness	low	low	high	very high

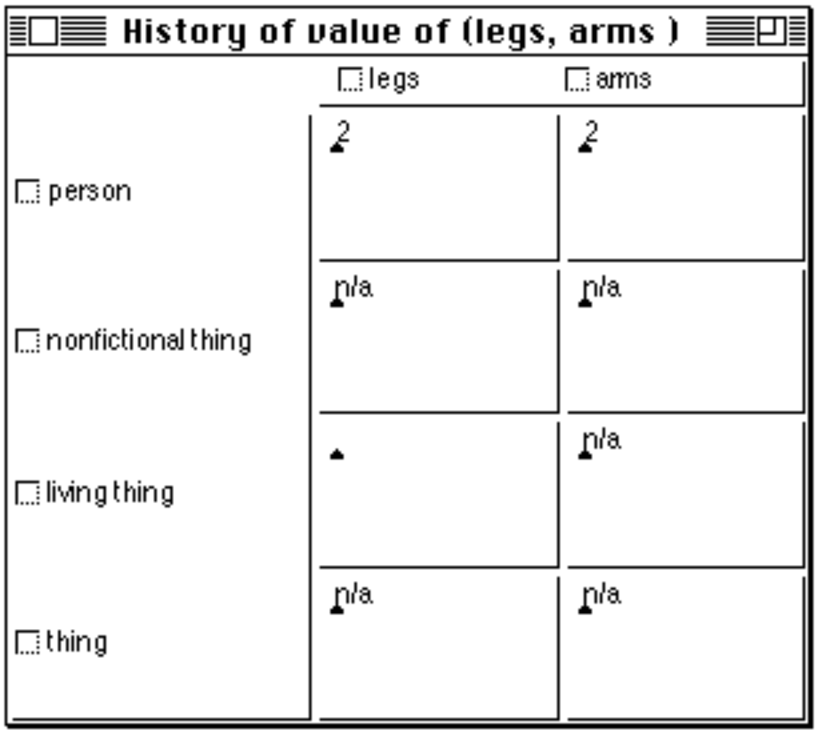
Figure 4.6: A property comparison matrix. This shows the values of all the statements that have particular subjects (columns) and predicates (rows). The subjects and predicates are generally determined by a driving knowledge map in conjunction with the options shown in figure 4.5. 'n/a' means that the subject does not possess the predicate.

4.2.5 Knowledge management problems addressed by mediating representations

CODE4's mediating representations help solve many of the problems listed in section 1.4:

- **Problem I-1: Categorizing**. The mediating representations display categorizations in various different ways so users can understand them better (e.g. by visualizing them in a graph, studying distinctions in a matrix etc.).

- **Problem I-2: Naming**. This is facilitated by the ease of naming concepts, by the ability to rapidly open driven subwindows to manipulate or explore the terms of selected concepts.
- **Problem I-3: Making distinctions**. The matrix mediating representation helps with this by displaying the values of all properties of several concepts.
- **Problem I-4: Understanding effects**. This is aided by the fact that knowledge can be displayed in various ways in several windows, and by rapid browsing of large amounts of knowledge. The matrix mediating representation helps particularly in this regard.



History of value of (legs, arms)

	legs	arms
person	2	2
nonfictional thing	n/a	n/a
living thing		n/a
thing	n/a	n/a

Figure 4.7: A property history matrix. This shows how the values of statements of the properties ***legs*** *and* ***arms*** *become more specialized as their subjects become more specialized. 'n/a' means that the subject does not possess the predicate.*

- **Problem I-5: Extracting**. The chaining of browsers helps with this as does the matrix mediating representation.
- **Problem I-6: Handling errors**. The matrix mediating representation helps users notice inconsistencies and incompletenesses.
- **Problem A-1: Special purpose restriction**. The mechanism is not limited to any domain.
- **Problem A-2: Expertise restriction**. Simple graphs and outline representations are easy to use. The non-modality of commands also help reduce the amount of expertise

required to use them. Although there are numerous facilities available, the beginner can add basic knowledge as soon as she or he learns how to: a) open a basic browser, b) add a child, and c) type a name or statement value.

4.3 Masks

A mask is a logical expression evaluated for each concept in a knowledge map as the concept is being prepared for display. It is composed of a set of conditions (or criteria) related by logical operators, and is either 'true' or 'false' for each concept. Each knowledge map has two masks[59]:

- A **visibility mask** that determines whether the concept will be displayed (true) or hidden (false). The default visibility mask allows *every* concept to be displayed.
- A **highlighting mask**[60] that determines whether a concept will be highlighted (underlined). The default highlighting mask highlights *no* concept.

Whenever a knowledge map needs to be redisplayed (when the knowledge base is edited, when the knowledge map's definition changes or when a mask changes) the masks are reapplied. This is because the set of concepts that fulfil the masks' criteria may have changed.

The set of conditions in the mask is often very simple, e.g. showing or highlighting the concepts whose ***colour*** property has value 'blue'. An expert, however, may create a complex mask combining several conditions into an arbitrary logical expression. Each condition is composed of a *mask predicate* and possibly other arguments. A mask predicate is simply a function that takes a concept as an argument[61] and returns true or false.

4.3.1 Using masks

Masks are used for several related purposes:

- To *focus the display* or reduce 'clutter': For example to show only one or two sub-hierarchies or to show only those concepts that are 'complete' i.e. finished. ('Completeness' could be determined, for example, by examining some metaconcept property).
- To *perform database-like retrieval*: E.g. "show me only concepts having the property 'connected-to' where one of the statement values is 'power supply', and which have been entered since last Friday". The display would be instantly restricted to just these concepts

[59] Knowledge maps for user language mediating representations lack masks.

[60] Called the 'selection criteria' in the system for historical reasons.

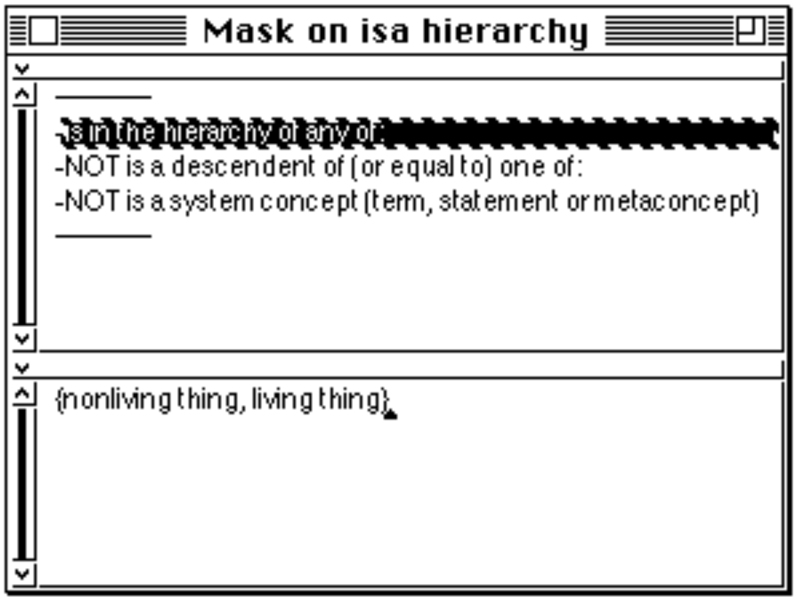
[61] In object oriented terms, a method whose receiver can be any concept.

(in the graphical mediating representation a partial graph may be shown with dotted lines indicating the union of several arcs).

- To *find concepts rapidly*: The highlighting mask may also be used to *select* all concepts that fulfil its criteria. Optionally, the visibility mask can be overridden by the highlighting mask so the user can find concepts that are excluded from view. One might apply such a mask to select some concepts in order to apply some command to them.

There are two primary methods of performing the above functions:

- Users can use special commands that perform specific actions, e.g. to exclude the selected concepts from display. These actions indirectly update one of the masks.
- They can open a mask window, such as the one shown in figure 4.8 in order to edit the mask directly. Once editing is completed they can issue a command to reapply the mask.



———

-is in the hierarchy of any of:
-NOT is a descendent of (or equal to) one of:
-NOT is a system concept (term, statement or metaconcept)

———

{nonliving thing, living thing}

Figure 4.8: An example mask window. This mask has three mask predicates that are logically conjoined. This mask displays or highlights those concepts that are in the hierarchies of either **living thing** or **nonliving thing** (first condition), and that are neither terms, statements nor metaconcepts (third condition). The second predicate currently has no arguments so it has no effect.

Beginners use the first method almost exclusively. A particularly useful command is the ‘fast go to’ facility[62]. If a user types a ‘>‘ symbol[63] followed by a regular expression, the

[62] Not yet available in the matrix mediating representation.

[63] Whenever a ‘>‘ symbol is the first character typed, the ‘fast go to’ command is invoked instead of the rename comand. To actually give a concept a term beginning with ‘>‘ the user would have to type the name directly into the name field at the top of the outline and graphical mediating represenions.

highlighting mask will be used to *select* any concept whose label matches that expression. The mediating representation also scrolls to the first selected concept. If the user types '>>' followed by a regular expression, the same thing happens, except that the visibility mask is temporarily overridden.

4.3.2 Knowledge management problems addressed by masks

The following are some of the problems (from section 1.4) that the mask facilities help solve:

- **Problem I-2: Naming**. This is helped by the ability to find or highlight concepts by any one of multiple names.
- **Problem I-5: Extracting knowledge**. This is the primary use of the mask.
- **Problem I-6: Handling errors**. The mask can help users find errors in their knowledge; for example a user could ask to find all concepts that have mutually exclusive values of particular properties.
- **Problem A-4: Individual-use restriction**. The mask can be used to help a user focus on just those aspects assigned to him or her. It can also highlight parts of the knowledge for group discussion.
- **Problem A-3: Size restriction**. The ability of the mask to restrict what is on display helps deal with large knowledge bases.

4.4 Other interface features

This section summarizes two other important features of CODE4's user interface. This thesis does not focus on these features, but they are mentioned for completeness.

4.4.1 The control panel

As with many software systems, CODE4 has a *control panel window*. This is the master or 'top level' window of the user interface. It is used to initiate all loading and browsing of knowledge bases and to set detailed parameters that control how CODE4's various features will work. Part of a control panel window is visible at the top right of figure 4.1.

One of the most important parameters set in the control panel window is the 'user expertise setting'. At any time when using CODE4, the user can choose one of 'beginner', 'intermediate' or 'expert'. In beginner mode, the whole user interface has fewer menu choices and a simpler appearance than in intermediate or expert mode. The expertise options are always visible at the top left of the control panel window.

Normally, users can rename concepts by typing *anywhere* in the mediating representation.

Also permanently visible, below the expertise settings, are a set of selections that control which *specific* control panel appears in the right of the control panel window. The following are some examples of specific control panels:

- The 'knowledge bases' control panel allows the user to perform such operations on knowledge bases as loading, saving, merging, opening browsers etc. Figure 4.1 shows a 'knowledge bases' control panel.
- Various format control panels allow the user to tailor the look and feel of the different mediating representations, either globally or on a window-by-window basis.
- The help control panel gives access to an online help system[64].

The primary benefit of the control panel is to help solve problem A-2 (expertise restriction).

4.4.2 The feedback panel

In section 4.2 it was mentioned that CODE4's user interface has been made largely non-modal by the use of commands that do not require prompts. However, requesting arguments is not the only reason why software systems display prompts (also known as modal dialogs). Other reasons of the display of such dialogs are:

a) To indicate to the user that something major is to be deleted (or some other drastic action is to take place) and to ask whether the user really wants to go ahead.

b) To warn the user that a requested command has side-effects of which he or she may not be aware, and to ask whether the user still wants to go ahead. An example occurs when deleting a property: The user may not realize that the property will be deleted from the system as a whole and not just from one main subject.

c) To indicate to the user that a command cannot be executed because some condition is not met, and to ask the user for an acknowledgement . An example occurs when attempting to delete a primitive property: The system responds that this cannot be done.

d) To indicate to the user that there are alternative ways of performing a command and to ask the user to chose one. An example occurs when changing the most general subject of a property: In this case the system needs to know whether to *also* move the value of the statement at the old most general subject.

In CODE4 non-modality is maintained in the above circumstances by the use of the *feedback panel*. Instead of presenting a modal dialog, the problem the system wants to point out is placed in the feedback panel which is brought to the front of the display. In the cases where further action is needed (cases a, b and d), the choices are listed in a pane of the feedback panel. The user may complete the command by selecting a choice, but in no

[64] Currently only partially implemented. This can also be invoked for any command by holding down the shift key when requesting that command.

case is obliged to do so. He or she is free to do some other action, optionally returning later to complete the command.

This is important because users frequently need to gather further information before they can decide which of several choices to take. The feedback panel thus helps with the following problems: problems I-4 (understanding effects) and problem I-6 (handling errors). As will be discussed in chapter seven, the feedback panel is only partly developed in the current implementation of CODE4.

Chapter 5

Knowledge Base Measurement

In chapter 1 it was stated that the goal of this research is to provide techniques to make knowledge management practical – so people can productively construct useful knowledge bases. In addition to facilitating construction, however, it is also necessary to be able to analyse what has been created to determine how much work has been done, how good it is and how it compares with other work. Chapters 3 and 4 contained ideas about the representation and presentation of knowledge; this chapter discusses the measurement of that knowledge.

It must be emphasised that this chapter is *not an evaluation* of the thesis research – all evaluation (including evaluation of the metrics themselves) is found in chapter 6.

5.1 Introduction

There are several purposes in attempting to measure knowledge bases:

- To allow knowledge engineers to monitor their work and to provide baselines for its continual improvement.
- To facilitate research into how knowledge bases, users and domains differ in terms of characteristics such as quality, experience and structural characteristics.
- To facilitate research into user interface and knowledge representation features, by providing grounds both for their comparison and for the comparison of knowledge bases built using them.
- To provide means whereby research using different systems can be put on a common footing.

Measuring knowledge bases appears to be a novel idea in knowledge representation; no precedents for the ideas described here have been found in the artificial intelligence literature.

The approach taken in this chapter is as follows: Section 5.2 defines exactly what measurement means and what is being measured. The section also surveys measurement in the related field of software engineering. Section 5.3 discusses the kinds of general measuring tasks people may want to perform; and Section 5.4 then presents some ideas for metrics that can be used in the measuring tasks.

The ideas in this chapter are largely intended to stimulate further research. Only with the analysis of very large amounts of data can metrics, like those proposed here, gain comparable stature to those in software engineering.

5.2 Some important definitions and background

5.2.1 Measurements vs. measures vs. metrics

It is necessary to understand subtle differences between the words ‘measurement’, ‘measure’ and ‘metric’. The word ‘measurement’ has two main senses that are of concern to this work: 1) The general process of measuring, and 2) a specific value (generally numeric) found as the result of a particular activity of measurement (in sense 1). These two senses are almost always distinguishable by context.

In normal English, an important sense of the word ‘measure’, as a noun, is as follows: A scale or unit for taking measurements. Often implicit in a measure is a procedure for making those measurements. Two simple examples: 1) The meter is a measure of length. 2) Words-per-minute is a measure of productivity for a typist, and implies timing a typist over a fixed interval and counting the number of words produced. A measurement (in sense 2) is the result of *applying* a particular measure.

In computer science and in other technical fields the word ‘metric’ has come into use, as a slightly specialized type of measure. A metric is a measure that normally has some specific purpose, often providing a concrete or objective way of gauging some abstract or subjective phenomenon. In general, a metric is chosen or proposed for standard use because it has useful qualities such as understandability. In the field of typing for example, words-per-minute is the normal metric for the abstract phenomenon of productivity; whereas milliseconds-per-word and keys-pressed-per-hour (while conveying much the same information) are measures that would not generally be used as metrics. In this thesis the word ‘metric’ is used as opposed to ‘measure’, in accordance with common practice in computer science.

This chapter discusses aspects of knowledge bases that might be measured and proposes specific metrics that can be used in that process. The major goal in developing the metrics is maximizing their usefulness.

5.2.2 Open-ended vs. closed-ended metrics

A closed-ended metric is one where measurements can only fall within a particular range – and where it is logically impossible for them to fall outside that range. The ratio of some part to its corresponding whole is of this type: Its range can only be from zero to one. An example is the fraction of all concepts in a knowledge base that are type concepts.

An open-ended metric is one where at least one of the ends of its range are not absolutely fixed. Although the probability of a measurement outside a particular subrange might be very small, it is still finite. An example of an open-ended metric is the number of type concepts in a knowledge base.

5.2.3 Important metrics in software engineering

In the field of software engineering the need for metrics has been widely recognized. It is therefore worth trying to learn some lessons from them before setting out to develop metrics for the allied field of knowledge management. Software engineering metrics are used as measures of productivity and as tools in project planning. The following are some of the more widely used:

Lines of code

Lines of code (LOC or KLOC for thousands) is a very basic and intuitively obvious metric; it is widely used but has many flaws. A major use, now frowned upon by many people, is to judge programmer productivity. The main problem is that LOC varies widely with the application, programming language, coding style and type of module being programmed. Also, programmers can ‘pad’ their apparent productivity by writing unnecessarily verbose programs or adding whitespace. A further problem is an inconsistent definition: Does LOC include comments? Does it include blank lines?

Another use to which lines of code has been put is helping project the completion time of a software engineering effort. The typical scenario is as follows: Based on the requirements, a management team makes a subjective estimate of the number of lines of code (using previous experience as a guide). Various heuristics are then used to project how long it will take to develop that number of lines: For example, depending on the environment and the individual, a programmer might be able to program between 5 and 100 lines per day. These methods for projecting project completion have fallen into disrepute due to persistent difficulties in making accurate estimates. As was discussed in the previous paragraph, lines of code is just not well enough correlated with volume of work.

Function points

Function points (Low and Jeffrey 1990) is becoming widely recognized as a better metric than lines of CODE. It measures *functionality*, and is determined by counting specific items in a software requirements specification (unadjusted function points), and then factoring in other subjectively-estimated aspects of a project such as the importance of reuse.

The International Function Points User Group (IFPUG 1994) is now the custodian of the function points methodology. This organization works to refine the counting methods and adjustment formulas and to adapt them to different technologies. A major goal is that the subjective perception of the amount of functionality in any project should correlate well with that project's function point count.

Function points works far better than lines of code for judging productivity since it is independent of programming language and cannot be padded during implementation (although it can certainly be padded during analysis in order to increase the value of a contract).

The function points metric is also much better than lines of code as a basis for estimating project completion time. Project planners follow a defined methodology when calculating function points, whereas lines of code estimates are often little more than guesses. Function points can be calculated early in development, because they are based on an actual document that is produced in the early stages. On the other hand, lines of code cannot be truly calculated until coding is complete: they can only be projected.

The biggest flaws with function points are: 1) They require significant training to calculate, and cannot be automatically calculated from an informal specification. 2) The formula for computing function points considers many aspects of complexity but weights them all equally, even though there may be huge differences in the importance of these factors. Despite these flaws, function points is a very useful metric.

Constructive Cost Modelling (COCOMO)

COCOMO (Boehm 1981) is a methodology for projecting elapsed time to completion (in calendar months) as well as total person-time. It provides a number of formulas that take into account various aspects of the complexity of a project. A major flaw, however, is that COCOMO requires as input an estimate of the number of lines of code. COCOMO is also becoming obsolete because of lack of ongoing research. It is also questionable whether COCOMO formulas can be called metrics at all – unlike lines of code and function points they are only used to project, never to measure an existing entity (source code or elements in a requirements document).

Function points, lines of code and COCOMO can be used together as follows: A function point estimate is first made. This is then converted into an estimate of lines of CODE after decisions about implementation technology are made (using heuristics that state how many lines in a particular language it takes on average to produce a function point). The lines of code estimate is then converted into time estimates using COCOMO.

Lessons learned

As a result of studying the above metrics, the following are some of the lessons that should be considered when designing a metric in another field such as knowledge management:

- One should ensure that the metric correlates well with the subjective phenomena about which it is designed to yield information (unlike lines of code).

- One should try to make the metric automatically calculable from a document (unlike function points and COCOMO).

Luckily there are a great number of possible aspects of a knowledge base that can be counted, and thus there is wide scope for experimenting with potentially-useful metrics.

5.2.4 Measuring knowledge bases vs. measuring knowledge

A question that could be asked about this research is: Should one talk about measuring *knowledge* or measuring *knowledge bases*?

In other domains it is conventional to give equivalent meanings to analogous pairs of phrases (where the objects are in a part-whole or substance-container relationship). For example measuring the quality of the stocks in a portfolio is considered to mean roughly the same as measuring the quality of the portfolio. Similarly, measuring the maintainability of a software system implies measuring the maintainability of the source code.

Knowledge is contained in a knowledge base. So in this thesis the measurement of knowledge is considered to be equivalent to the measurement of knowledge bases.

5.2.5 The kind of knowledge bases to be measured

This study is concerned with the types of knowledge bases discussed earlier in this thesis; i.e. concept-oriented (frame-based or semantic-net based) representations such as CODE4-KR, KM, CycL and KL-ONE. These are now sometimes called object-oriented knowledge representations, although they should not be confused with object-oriented programming languages. However, the design of the metrics in this chapter does not preclude their use with other forms of knowledge base such as those containing rules or problem-solving methods[65].

The reason for the latter assertion is that concept-oriented representations can be made to subsume the others; or the latter can be seen as more specialized abstractions for the purposes of certain tasks. For example, when representing rule-oriented knowledge in CODE4-KR one might encode the rules as instance concepts and have additional concepts for: 1) the things affected by these rules; 2) properties of the rules; 3) statements about them, etc. One can likewise imagine encoding problem solving methods in a concept-oriented way. In order to extend a concept-oriented knowledge base so that it can handle more specialized representations the primary tasks would be to provide new primitive mechanisms and concepts as well as new mediating representations.

---

[65] A major area of research in the knowledge acquisition community involves encoding problem solving methods in knowledge bases.

5.3 General tasks for which knowledge base measurement may be useful

The following subsections list tasks that involve measuring attributes of knowledge bases. Many of the tasks have analogs in conventional software engineering, but others do not. The purpose of this section is to further motivate the development of metrics; suggestions for actual metrics and examples of their use can be found in section 5.4.

The measuring tasks can be divided into three rough categories: 1) assessing the present state of a single knowledge base (for completeness, complexity, information content and balance); 2) predicting knowledge base development time and effort, and 3) comparing knowledge bases (with different creators, domains, knowledge acquisition techniques and knowledge representations). These tasks are certainly not independent – most tasks depend explicitly on others; however, categorizing the measuring tasks provides a useful basis to decide what metrics should be developed.

A clear mapping between tasks and metrics should not be expected: Some metrics might help with several tasks, and some tasks may require several metrics.

5.3.1 Tasks A to D: Assessing the present state of a single knowledge base

Present-state assessment tasks are those where the measurer wants to determine how a particular knowledge base fits on one of several possible scales. These tasks can be subtasks of task A, but can also be performed for other reasons.

Task A – Assessing completeness

Important questions in knowledge engineering are: What does it mean for a concept-oriented knowledge base to be complete? And furthermore, how can one tell when a knowledge base *is* complete? A related question is: Given a particular knowledge base, how close is it to a state of completeness? Unless these questions are answered, it cannot be possible to predict the effort required to get to that state.

In a well-managed standard software engineering project (where requirements are not constantly changing), completeness can be assessed by determining if the software fulfils its specification (e.g. passing appropriate testcases). The degree of completeness for a partially finished project can be estimated based on such factors as the proportion of function points represented by testcases that have been passed. However, there are a number of fundamental differences between standard software engineering and knowledge engineering that make it necessary to find different ways of determining completeness:

- In knowledge base development, there rarely is a specification. Knowledge engineering typically progresses fluidly, perhaps using a prototyping approach or perhaps only with the rough objective of “documenting a domain” or “designing a widget”.
- Whereas standard software can be broken down into well-defined modules with a well-defined interface that can be tested; most knowledge bases have a much finer grain – being composed of concepts or rules that can be referred to by many others in widely different parts of the knowledge base.
- Knowledge can be expressed in a rough form that is *partially* suited to the task at hand, and then gradually refined and formalized so that it becomes more suited to the intended task. Standard software tends either to work or not to work so there is less scope for incremental development at the detailed level.

For a knowledge base developed with a particular performance task in mind (e.g. a set of rules for a diagnosis system) a way of measuring completeness is to apply the performance system to a set of test tasks that have optimum expected outcomes (e.g. correctly diagnosing faults), and to measure how well the system performs. Unfortunately where no specific performance task is envisaged (commonly the case for the kinds of concept-oriented knowledge bases created by users in this research) the above approach is not feasible.

An alternative approach might be as follows: The first step is to recognize there may be no certain state of completeness. Secondly, it should be possible to measure the amount and kind of detail supplied about each main subjects in the knowledge base. Finally it should be possible to ascertain how close this is to the statistical average for knowledge bases that have been *subjectively* judged complete. This method has promise if users sketch out the inheritance hierarchy (i.e. the main subjects) as the first step in developing a knowledge base (or a part thereof).

Task B – Assessing complexity

A number of metrics allow software engineers to determine the complexity of modules or subsystems. There are several reasons for doing this: High complexity may be predictive of greater cost in future stages of development; and it may expose situations where designs can be improved. A measure of complexity may also help in accurately measuring productivity, because a small but complex project may take as much work as a large but simple project.

In software engineering, some measures of complexity include cyclomatic complexity (McCabe 1976), coupling, fan-out and hierarchy depth (descriptions of these can be found in any good software engineering textbook, e.g. (Sommerville 1992)). Adaptations of the latter three may well apply to knowledge bases, but the special nature of knowledge bases may suggest additional useful metrics.

Task C – Assessing information content

It is unusual to pose the following query to conventional software: How much information is in this system? Rather, one asks: How well does it perform one of its limited number of tasks? With many knowledge bases however, a major goal is that they be queriable in novel ways – deducing new facts using various inference methods. The objective is to be able to uncover *latent* knowledge. The word ‘multifunctional’ has been used for such knowledge bases (Acker 1992).

Estimating information content can help determine both the potential usefulness of a knowledge base and the productivity of its development effort. Such metrics should take into account the fact that some knowledge is a lot less useful than other knowledge.

Task D – Assessing balance

The *balance* of a knowledge base is defined as either: 1) the degree to which a group of measurements using related metrics are close to their respective ‘normal’ values, or 2) the degree to which measurements of different parts of a knowledge base using a single metric are close to each other.

Knowledge bases are typically composed of a mixture of very different classes of knowledge; e.g. concepts in a type hierarchy, detailed slots, metaknowledge, commentary knowledge, procedural knowledge, rules, constraints etc. Each project may require a different proportions of these classes; a balance metric of the first type would indicate how normal a knowledge base is. For example, a knowledge base would be considered unbalanced if it contains a large amount of commentary knowledge but very few different properties. This might indicate that the person building the knowledge base has not been trained to use an appropriate methodology that involves identifying properties.

The second type of balance metrics are those that show whether different parts of a knowledge base are equally complete or complex, i.e. whether completeness and complexity are focussed in certain areas or not. A knowledge base would be unbalanced if one major part of the inheritance hierarchy contained much detail while another part did not.

There is likely to be a strong relationship between metrics for completeness and metrics for balance; and some balance metrics may be used in the calculation of a composite metric of completeness. For example if a knowledge base has a low ratio of rules to types, it may be concluded that there is scope to add more rules; likewise if one subhierarchy is far more developed than another, the overall KB may be incomplete. However, there are reasons for measuring balance variables separately (perhaps after a subjective judgement of completion): They can allow one to characterize the nature of knowledge and to classify knowledge bases. For example it may be that for some applications, different proportions of the various classes of concept are normal.

There are not many analogs to the idea of balance in general software engineering, but one example is the measure of the ratio of comment lines to statement lines.

5.3.2 Task E. Predicting knowledge base development time and effort

This is one of the main tasks for which software engineering metrics are developed: People are interested in ascertaining the amount of work involved in a development project so they can create budgets and schedules, or so they can decide whether or not to go ahead with a project as proposed.

The following general framework illustrates the scenario for such metrics; here the term ‘product’ stands for either ‘software system’ or ‘knowledge base’:

a) Measurements have been taken of a number of products, $P_1 \ldots P_{n-1}$.
b) There is an interest in making predictions about product under development, P_n.
c) One of the metrics, M_t, represents time to create a product.
d) Another metric, M_s, can be calculated early in product development and is found to be correlated (linearly or otherwise) with M_t.
e) By analysing $P_1 \ldots P_{n-1}$, a function, f_p, is developed that reasonably accurately predicts M_t, i.e. $f_p(M_s)$ -> M_t.

In the software engineering world, function points is used for M_s and COCOMO has formulas that fulfil the role of f_p, predicting the number of person-months to complete the project, given M_s.

No similar metrics or functions for the production of concept-oriented knowledge bases have been found in the literature. Some people consider the production of such knowledge bases to be a case of software engineering (especially when a KB is being developed for use in an inference oriented system such as an expert system); however, it is intuitively apparent that lines of code, function points or COCOMO formulas have little meaning and hence no predictive value for knowledge bases. Furthermore other measures of standard software engineering projects (e.g. the number of modules, classes etc) are irrelevant in a concept-oriented framework.

For knowledge management, there is a need to come up with new candidates for M_s and new functions for f_p. Furthermore, there is a need to understand the set of assumptions under which M_s and f_p are valid. For example the original function points metric is only effective for data processing software; and COCOMO has become outdated as a predictive technique due to improvements in software engineering methods.

As with standard software engineering, prediction cannot be an exact science: Domains, problems and knowledge engineers differ; and furthermore completeness can only be statistically estimated (as discussed in section 5.3.1). Nevertheless it still seems a worthwhile effort to give knowledge engineers metrics to help judge how much work they might reasonably need to do.

5.3.3 Tasks F to I: Comparison

The last section described some uses of assessing a single knowledge base in isolation, perhaps comparing it with a goal or norm. Another task for metrics is to find *relative* differences between knowledge bases so as to indirectly compare their creators, domains, techniques and representations.

Task F – Comparing users

Most of the metrics derived from tasks listed above can lead to useful differential measures of the skills or productivity of users. By examining knowledge bases they have built, one can also determine which users are familiar with which features.

Task G – Comparing domains

By measuring various attributes about knowledge bases in particular domains, it may be possible to ascertain certain constant factors that characterize a domain, distinguishing it from others.

This kind of knowledge can feed back into the prediction process (task A). For example in the COCOMO method, different predictive formulas are applied to embedded software and to data processing software. Similarly it may be possible to distinguish classes of knowledge base that would lead to the creation of different prediction formulas (or different coefficients in the same formulas) for, say, medical rule based systems, electronics diagnosis systems and educational systems.

Task H – Comparing development techniques

By comparing measures of knowledge bases developed using different knowledge acquisition techniques it might be possible to decide which techniques are better for certain tasks. A subsequent objective might be to incrementally improve the techniques using metrics to evaluate success.

Task I – Comparing representation schemata

In a similar manner to comparing development techniques, it is useful to compare representation schemata. Even though such a process might require converting knowledge bases to some common denominator before measuring, it might still be possible to gain useful knowledge about the effectiveness of the abstractions in each schema.

5.4 Proposals for metrics

This section proposes actual metrics that can be used in the general measurement tasks discussed in the last section. Evaluation of the metrics, including actual measurements of knowledge bases, is deferred to chapter 6. There are three classes of metrics: 1) general open-ended measures of size, 2) measures of various independent aspects of complexity, and 3) compound metrics.

5.4.1 Metrics for raw size

One of the most basic questions that can be asked is: How *big* is a knowledge base; what is its *size*? Knowing this can help predict development time and judge information content. But what do 'big' and 'size' mean? In conventional software engineering, lines of code is a useful concrete metric that is easy to calculate given some source code. Knowledge engineering needs a similar metric. Several options were considered, the following two being the most promising:

The total count of all CODE4 concepts, M_{ALLC}

This very physical size measure is appealing as an analog to lines of code which, despite its problems, is understandable and easy to calculate. Concepts in a knowledge base, however, can be far more diverse in nature than lines of code: They may include main subjects, properties, statements, metaconcepts etc. Some concepts may even be created without typical users realizing it. E.g. a term concept in CODE4 is automatically added when a user types a new name for another concept.

M_{ALLC} might be most useful as a measure of how much memory and loading-time a knowledge base might require, or how much time certain search operations might take.

The number of main subjects, M_{MSUBJ}

M_{MSUBJ} is the count of just the main subjects, i.e. the important concepts that users specify directly. It excludes concepts about which nothing is said (e.g. example instances) and also excludes CODE4's 'special' concepts: properties, statements, metaconcepts and terms.

The benefit of such a metric is that since it ignores the detail that has been 'filled in' around each main subject, it helps in the process of separating the sheer size of a knowledge base from other aspects of its complexity. M_{MSUBJ} should also be intuitive to users because most of them directly look at lists or graphs of main subjects, but only indirectly work with other concepts. Furthermore, since users typically develop knowledge bases by initially drawing a hierarchy of main subjects and then filling in details, M_{MSUBJ} has the potential to provide a baseline for the estimation of completeness and the prediction of development time. All these advantages make it potentially more useful than M_{ALLC}.

Both M_{ALLC} and M_{MSUBJ} have the problem that since concepts differ in importance, what they count are not 'equal' to each other. For example, in many CODE4 knowledge bases people create a core of concepts which are central to their domain and about which they add much detail. Typically though, users also add a significant number (10-30%) of concepts that are outside or peripheral to the domain. In a typical case, a user creating a zoology knowledge base might also sketch out a rough hierarchy of plants. Both M_{ALLC} and M_{MSUBJ} would consider these peripheral concepts to be as important as the main zoological concepts.

5.4.2 Independent and closed-ended metrics for complexity

A number of factors contribute to the complexity of a knowledge base. Seven complexity metrics have been developed that are logically independent of the raw size. They have also been designed to be as independent of *each other* as possible. Of course, just because one metric is designed to be independent of another does not prevent correlations from appearing in practice: It may happen that the normal process of knowledge acquisition results in increasing complexity along the different scales in parallel. Correlation is discussed in section 6.2.1.

Each of these complexity measures falls in the range of 0 to 1 so that they can be easily visualized as percentages, and so they can be easily combined to form compound metrics.

Relative Properties, M_{RPROP}

This is a measure of the number of user properties, relative to the number of main subjects. It is calculated as the square of the ratio of user (non-primitive) properties to the sum of user properties and main subjects:

$$M_{RPROP} = (M_{UPROP} / (M_{UPROP} + M_{MSUBJ}))^2$$

Where M_{UPROP} is the number of user properties in the knowledge base.

If a user adds no properties, then M_{RPROP} is zero; M_{RPROP} approaches one as large numbers of properties are added to a knowledge base. In the average knowledge base, the number of user properties tends to be slightly more than the number of main subjects (0.55, i.e. people seem to add just over one new property for every main subject), hence M_{RPROP} averages 0.3 (which is 0.55 squared).

The metric is squared to reduce a problem that arises due to the deliberate decision not to make it open-ended: For each additional property, the increase in the metric is less. Thus in a 100 main-subject knowledge base, increasing the number of user properties from zero to 100 causes a 0.25 increase, whereas raising it another 100 only causes a 0.19 increase. If the metric were not squared, this problem would be much worse (the first 100 properties would cause a 0.5 increase and the next 100 only a 0.17 increase). In practice these diminishing returns only become severe at numbers of properties well above what have been encountered. Intuitively there appears to be diminishing returns in the subjective sense of complexity as well.

Figure 5.1 shows how M_{RPROP} varies as additional properties are added to a 100 main subject knowledge base.

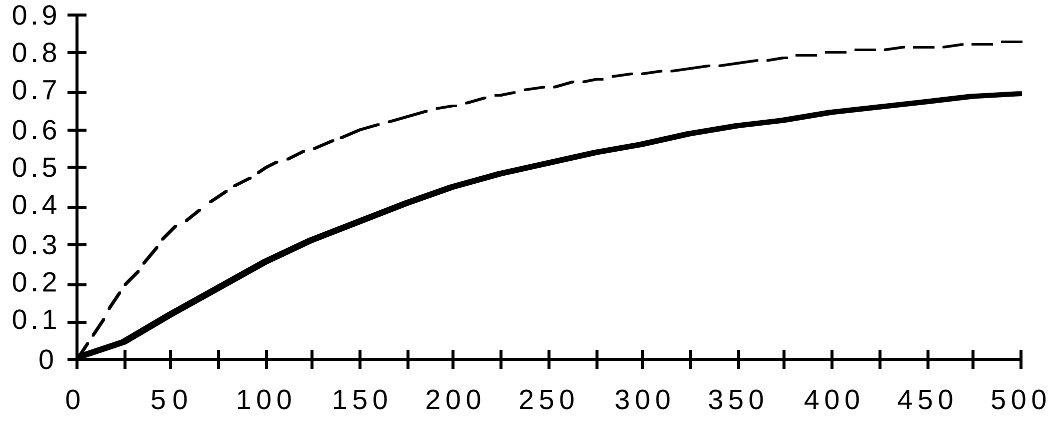


Figure 5.1: Values of M_{RPROP} when there are 100 main subjects. The x axis shows the effect of an increasing number of user properties. The bold plot is that of M_{RPROP} while the dashed plot shows what the metric would look like if it were not squared. The squaring flattens the curve and thus makes the metric more useful.

Detail, M_{DET}

This metric is the fraction of statements about main subjects, that have a locally specified value. Such statements contrast with those that have an inherited value, or no value at all as can be the case if a property is added and no statements are created using it. If no statement values are filled in at all in a knowledge base, M_{DET} is zero. If every possible statement about every main subject is given a specialized value, then M_{DET} equals one. However should M_{DET} have a value too near one it would indicate that inheritance is not being properly taken advantage of, and thus there is no knowledge reuse with its complexity-reducing effect.

To summarise, the formula for M_{DET} is as follows:

$$M_{DET} = M_{MSSLV} / M_{MSS}$$

where M_{MSS} is the number of main subject statements (statements whose subject is a main subject)
and M_{MSSLV} is the number of main subject statements which have local values

Whereas M_{RPROP} measures the *potential* number of specialized statements , M_{DET} indicates the amount of *use* of that potential. The next measure, M_{SFORM}, goes one step further and measures to what degree that use results in the interconnection of concepts.

Statement Formality, M_{SFORM}

This measures the fraction of values (of statements about main subjects) that contain actual links to other concepts in the knowledge base. If M_{SFORM} is zero, then the user has merely placed arbitrary expressions in all values. The higher M_{SFORM} is, the more a knowledge management system would be able to infer additional network structures (i.e. knowledge maps describing such things as the part-of relation) inherent in the knowledge. The idea of formality vs. informality, and why these terms are used, is discussed in section 1.2.5.

The formula for M_{SFORM} is as follows:

$$M_{SFORM} = M_{MSFS} / M_{MSSLV}$$

where M_{MSSLV} is the number of main subject statements with local values (see M_{DET})
and M_{MSFS} is the number of main subject formal statements, i.e. those with local values containing value items that are other concepts

The denominator of M_{SFORM} is composed of only those statements that form the numerator of M_{DET} – those statements with locally specified values. It thus should be independent of M_{DET} : i.e. regardless of whether M_{DET} is near zero or near one, M_{SFORM} can still range from zero to one. The one exception is when there are no statement values, in which case M_{DET} is zero and M_{SFORM} is undefined.

Diversity, M_{DIV}

While high measurements of relative properties, detail and formality may indicate that substantial work has been done to describe each main subject, that detail may be largely in the form of very subtle differences. There may be a large amount of mere data, expressed as statements of the same set of properties about each main subject. For example in a knowledge base describing types of cars, hundreds of classifications of cars may be described but only using a fixed set of properties (engine size, fuel consumption etc.) – such a knowledge base would be subjectively judged to be rather simple despite having a lot of ‘facts’; M_{DIV} attempts to quantify this subjective feeling.

The diversity metric, M_{DIV}, measures the degree to which the introduction of properties is evenly spread among main subjects. If all properties are introduced in one place (e.g. at the top concept) then M_{DIV} is close to zero because the knowledge base is judged to be simpler. Likewise, M_{DIV} is close to zero if properties are introduced mostly on leaf concepts (so there is no inheritance). Maximum M_{DIV} complexity of one is achieved when some properties are introduced at the top of the inheritance hierarchy, some in the middle and some at all of the leaves.

The method of calculating M_{DIV} is described in the following paragraphs; figure 5.2 is used to illustrate the calculations.

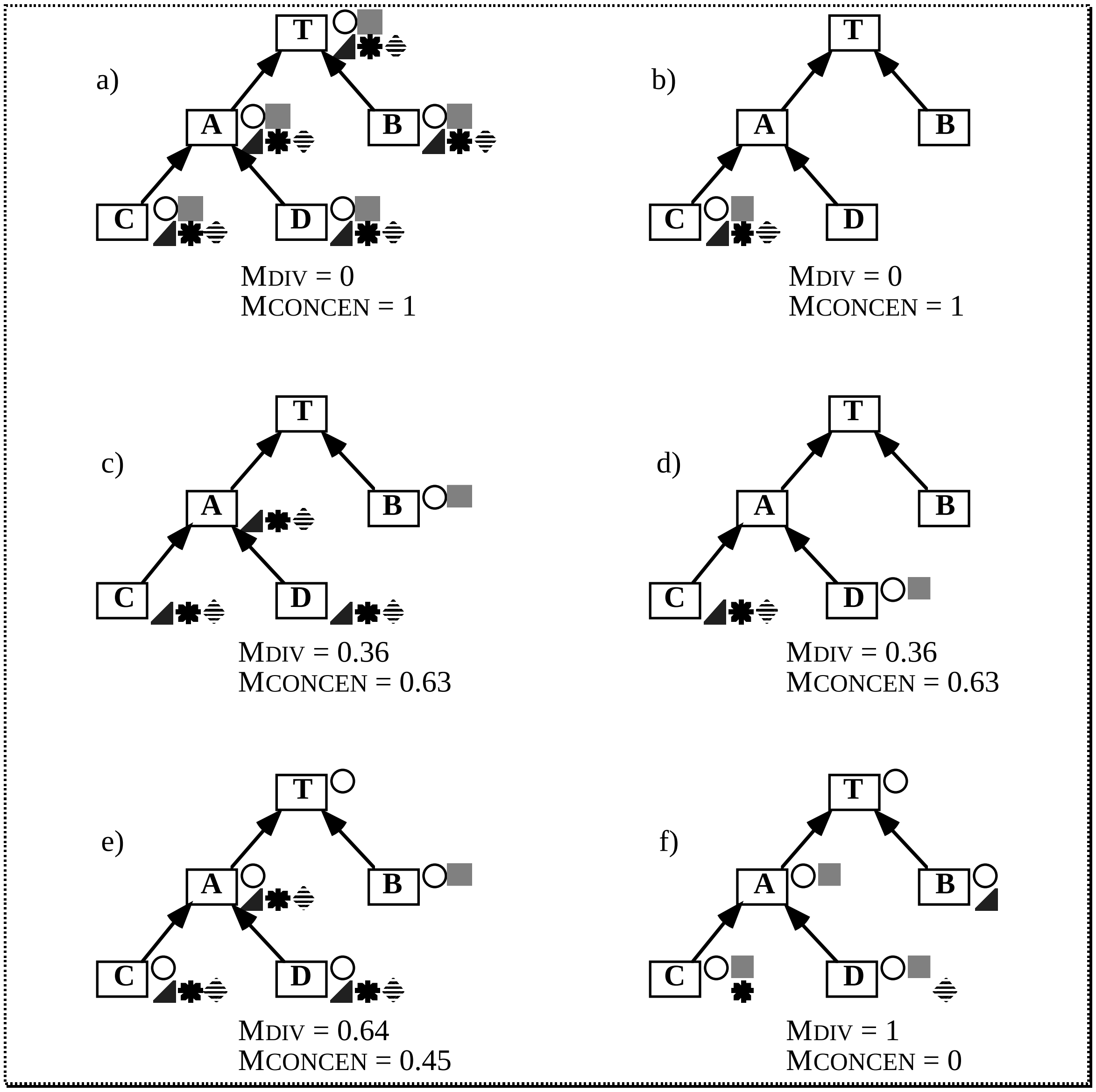


Figure 5.2: Calculation of M_{DIV}. Each part shows an inheritance hierarchy with five main subjects and five properties (shown as five distinct shapes). Parts a and b show pathological situations where properties are all introduced at a single concept (the top and a leaf respectively) and M_{DIV} is thus zero. Part f shows a well balanced case, where each concept introduces exactly the same number of new properties (one in this case) and M_{DIV} is one. Parts c, d and e show intermediate cases.

To calculate M_{DIV}, the first step is to calculate the standard deviation of the number of properties introduced at each main subject. This is zero if the introduction of properties is evenly distributed, and some higher number otherwise. The next step is to normalize this standard deviation into the range zero to one. The following formula accomplishes this:

$$M_{CONCEN} = ((\sigma \text{ PI}) / M_{UPROP}) * \sqrt{M_{MSUBJ}}$$

where M_{UPROP} is the number of user properties in the knowledge base
and PI is the number of properties introduced at a main subject
and σ is the standard deviation operator

A simple way to convert M_{CONCEN} into a diversity metric would be to calculate 1-M_{CONCEN}. However, thus results in measurements being too crowded towards zero. M_{DIV} is actually calculated using the following more subjectively appealing formula:

$$M_{DIV} = (1 - M_{CONCEN}^2)^2$$

Figure 5.3 plots the relationship between M_{CONCEN} and M_{DIV}.

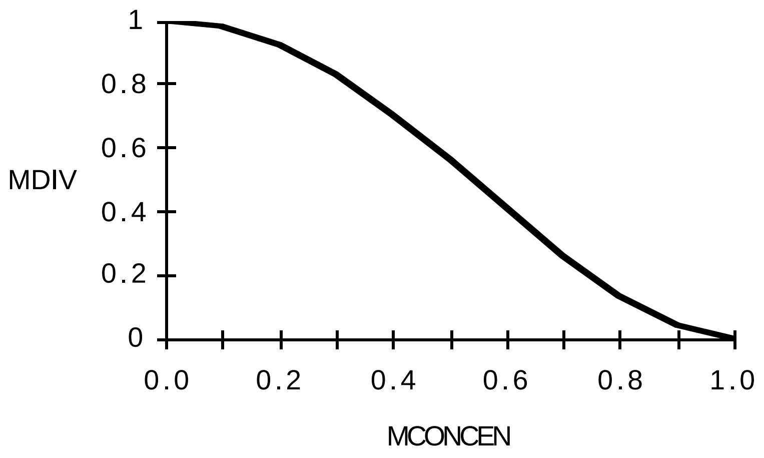


Figure 5.3: The relationship between M_{CONCEN} and M_{DIV} . M_{CONCEN}, shown in the x axis is a normalized version of the standard deviation of the number of properties introduced at each main subject. This is transformed using a sigmoid function, as plotted in the above graph, in order to obtain a metric that has a good subjective 'feel'.

Second Order Knowledge, M_{SOK}

The metrics described so far totally ignore the presence of second order knowledge, i.e. knowledge not about the things in the world represented by concepts, but about the representations of those things. M_{SOK} measures the use of two types of second order knowledge:

• The use of metaconcepts, i.e. the concepts that describe concepts themselves
• The use of terms, i.e. symbols that represent things.

Every main subject in principle has a metaconcept, but most are ignored by users. Of interest are those which users have explicitly made the subjects of statements (i.e. those which users have said something about). Similarly, every main subject generally has a term that is used to name it. However, of interest are those cases where extra terms are specified: This indicates that users have paid attention to linguistic issues.

M_{SOK} is zero if the user has neither worked with metaconcepts nor specified synonyms for any concept. The metric gives a measurement of one if every main subject has both multiple terms and metaconcept statements. M_{SOK} is thus calculated as follows:

$$M_{SOK} = (M_{FMETA} + M_{FTERM}) / 2$$

where M_{FMETA} is the fraction of main subjects whose metaconcept has a user-specified statement
and M_{FTERM} is the fraction of main subjects that have more than one term

Isa complexity, M_{ISA}

This metric combines two factors in order to measure the complexity of the inheritance hierarchy. The first factor is the fraction of types that are leaves of the inheritance hierarchy, ignoring instance concepts which are always leaves (i.e. the fraction of types that have no subtypes). This measure is called M_{FLEAF}:

$$M_{FLEAF} = M_{LEAF} / M_{TYPES}$$

where M_{LEAF} is the number of leaf types
and M_{TYPES} is the total number of types

For any non-trivial knowledge base, M_{FLEAF} is a function of the branching factor, M_{BF}[66], however the latter is not used as a metric because it is open ended (i.e. not in a zero-to-one range). The formula relating branching factor to M_{FLEAF} is as follows:

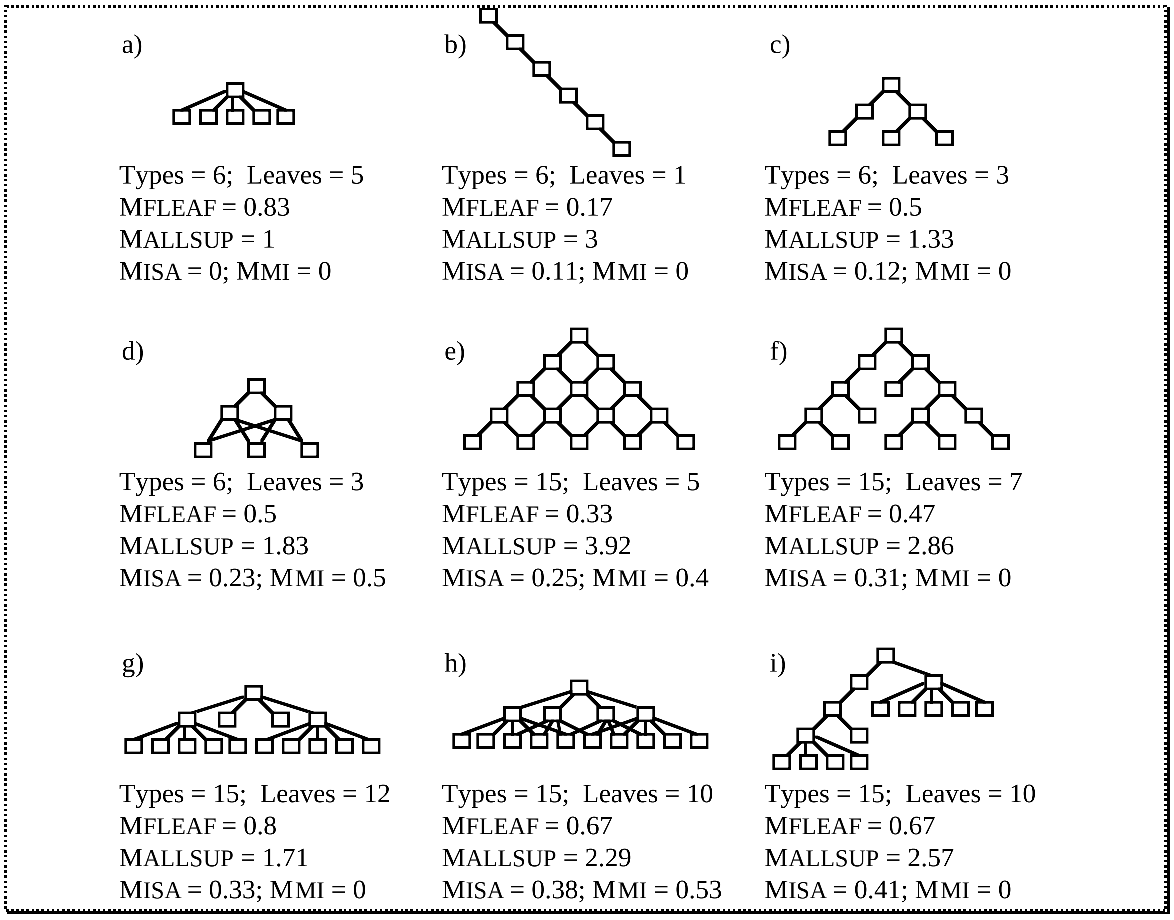
$$\lim_{M_{TYPES} \to \infty} M_{FLEAF} = (M_{BF} - 1) / M_{BF}$$

where M_{TYPES} is the total number of types

Figure 5.4 shows several simple inheritance hierarchies along with their measures of M_{FLEAF}. M_{FLEAF} approaches 0.5 when the inheritance hierarchy is a binary tree (figure 5.4,

66 The branching factor is 2 for a perfect binary tree, 3 for a perfect ternary tree etc.

parts c and f). It approaches zero in the ridiculous case of a unary ‘tree’ with just a single superconcept-subconcept chain (part b) and it approaches one if every concept is an immediate subconcept of the top concept (part a).



a) Types = 6; Leaves = 5
MFLEAF = 0.83
MALLSUP = 1
MISA = 0; MMI = 0

b) Types = 6; Leaves = 1
MFLEAF = 0.17
MALLSUP = 3
MISA = 0.11; MMI = 0

c) Types = 6; Leaves = 3
MFLEAF = 0.5
MALLSUP = 1.33
MISA = 0.12; MMI = 0

d) Types = 6; Leaves = 3
MFLEAF = 0.5
MALLSUP = 1.83
MISA = 0.23; MMI = 0.5

e) Types = 15; Leaves = 5
MFLEAF = 0.33
MALLSUP = 3.92
MISA = 0.25; MMI = 0.4

f) Types = 15; Leaves = 7
MFLEAF = 0.47
MALLSUP = 2.86
MISA = 0.31; MMI = 0

g) Types = 15; Leaves = 12
MFLEAF = 0.8
MALLSUP = 1.71
MISA = 0.33; MMI = 0

h) Types = 15; Leaves = 10
MFLEAF = 0.67
MALLSUP = 2.29
MISA = 0.38; MMI = 0.53

i) Types = 15; Leaves = 10
MFLEAF = 0.67
MALLSUP = 2.57
MISA = 0.41; MMI = 0

Figure 5.4: Complexities of various inheritance hierarchies. Parts a through i show increasing complexity using the M_{ISA} metric. Parts a and b show simplistic cases (M_{ISA} would be small regardless of the size of the knowledge base). Parts c and d show the same structure, with variation only in multiple inheritance. Pairs (e,f) and (g,h) also show similar structures with variation in multiple inheritance, however M_{ISA} can be seen to be somewhat independent of the adding of extra parents (such an action may either increase or decrease M_{ISA}).

On its own, M_{FLEAF} has the undesirable property that for a very shallow hierarchy (e.g. just two or three levels) with a high branching factor it gives a measurement that is unreasonably high, from a subjective standpoint (part a of figure 5.4 illustrates this)

To correct this problem with M_{FLEAF} , an additional factor is used in the calculation of M_{ISA}: the average number of direct and indirect superconcepts per non-root[67] main subject,

[67] i.e. excluding the top concept, which cannot have parents.

M_{ALLSUP}. This second factor is related to hierarchy depth but depends to some extent on the amount of multiple inheritance.

M_{ISA} is thus calculated using the following formula:

$$M_{ISA} = M_{FLEAF} - (M_{FLEAF} / M_{ALLSUP})$$

M_{ISA} approaches zero in either of the following cases: 1) when there is just a single layer of concepts below the top concept (no matter how many concepts); or 2) when the branching factor is one. M_{ISA} approaches the value of M_{FLEAF} (e.g. 0.5 for a binary tree) as the hierarchy becomes deeper (as the average number of direct or indirect parents per main subject increases).

Multiple Inheritance, M_{MI}

This measures the extent to which main subjects have more than one parent, thus introducing complex issues associated with multiple inheritance such as the combination of values when two inherited values conflict. If just single inheritance is present, this metric is zero.

M_{MI} measures the ratio of *extra* parents to main subjects, thus if all main subjects had two parents (impossible in fact because the concepts directly below the top concept cannot have more than one parent) then the metric would be one; it could also be one if a substantial number of concepts have more than *two* parents. Although the metric, as described, could theoretically give a value above one, it is assumed that this could not be the case in any reasonable knowledge base. Thus a ceiling of one is imposed that the metric cannot exceed, even if there are an excessive number of parents.

5.4.3 Compound metrics for complexity

In this subsection, the simple metrics for complexity described in section 5.4.2 are combined into compound metrics in an attempt to give useful overall pictures of a knowledge base. In the previous subsection, the metrics were designed with reasonable confidence that they are generally applicable. Here however, the necessity of *combining* metrics requires finding numerical coefficients that optimize the usefulness of the result. Determining such coefficients requires extensive examination of data. In this research the only data available was that of the user study described in the next chapter. While the amount of data is quite large for testing metrics, it is not nearly large enough to act as a 'training set' for metric optimization. As a result, the main contributions of this subsection are proposals for *procedures for designing metrics*, rather than proposals for actual formulas (the formulas presented are examples of the application of the procedure).

The generic procedure for designing a compound metric is as follows:

1) Choose the metrics that are to be combined by looking for those with certain desired characteristics.

2) Normalize the metrics so that they are not arbitrarily biased.
3) Weight the metrics based on decisions about which are most important, and how much their values should contribute to the resulting metric
4) Design a formula that combines the metrics.

Although both step 2 and step 3 involve finding coefficients that modify the original values of the metrics, they are logically separate steps. After step 2, accidental bias is removed. Such bias may be caused by the fact that the metrics might have quite different expected values or ranges. In step 3, deliberate weighting is added. For this initial research, step 3 is ignored since it would require a very large amount of data to justify an unequal weighting. The most important step in this research is step 4.

Apparent completeness, M_{ACPLT}

This metric combines those metrics that, intuitively, should steadily increase as a project progresses. The idea is to create a metric that ranges between 0 and 1 so that users can obtain an impression of how much work needs to be done on a knowledge base to make it 'complete' (i.e with a reasonably high percentage of detail, formality etc). See section 5.3.1 for a discussion of what it means for a knowledge base to be complete.

The metrics chosen to compose M_{ACPLT} are:

• M_{RPROP}, indicating the extent to which properties have been added
• M_{DET}, indicating the proportion of potential statements with actual values, and
• M_{SFORM} indicating the extent to which the knowledge base has been formalized.

The possibility of adding M_{SOK} was considered, but it was decided not to do this because the amount of second order knowledge might be heavily dependent on the domain, whereas the extent to which statements are filled in and formalized appears much more likely to measure the state of completeness, independent of domain.

To remove accidental bias from the metrics, actual knowledge bases were studied to see how the component metrics range in practice. None ever approached one, and the maximums for well-worked out knowledge bases were all about 0.7. It seems reasonable that none of these metrics would ever reach one for the following reasons:

• M_{RPROP} can only asymptotically approach one as ever larger numbers of properties are added. A measurement as high as 0.7 already indicates that there are over five times as many properties as main subjects. Experience has shown that it is unlikely for a knowledge base to have a much higher proportion of properties than this.

• If M_{DET} were to approach 1, it would mean that hardly any statements are inherited; all would be locally specified (i.e. they all override inherited values). This seems unlikely to happen in most knowledge bases.

- If M_{SFORM} were to approach 1, the user has been able to formalize all statements. This seems unlikely in a normal knowledge base.

It was thus decided that all metrics should be normalized so that when a measure using them has a value of about 0.7, it is considered to indicate that the particular aspect of the knowledge base is reasonably complete. It is purely coincidental that the 0.7 is used for all three. The coefficient used to normalize the metrics is the reciprocal of 0.7, i.e. 1.4. As a result of this, M_{ACPLT} can give measurements greater than one – that would merely indicate that more detail has been provided than in a normal complete knowledge base.

The following points explain how it was decided to combine the metrics:

- M_{RPROP} must dominate the calculation for the following reason: If there are few properties, then high measurements of detail and formality mean little (because there can only exist a few statements with potential for having detail and formality). Thus M_{RPROP} should be a multiplicative value for the whole metric formula – if M_{RPROP} is zero then M_{ACPLT} should be zero.
- For similar reasons, M_{DET} must dominate M_{SFORM}. If there are few statements, then a high measurement of formality means little because there are only a few statements to *be* formal.

The basic formula for combining the three metrics is therefore:

$$M_{ACPLT} = C * M_{RPROP} * (W + C * M_{DET} * (X + Y * C * M_{SFORM}))$$

Where C is the constant 1.4 used to unbias the metrics; i.e. to convert their 0 to 0.7 range to a range of 0 to 1. If this factor were missing , the resulting metric could only yield measurements that would approach 70%.

And where W, X and Y are coefficients used to weight the metrics. As discussed earlier it was decided to weight the metrics equally. These three coefficients should all be 0.33 then. That would mean that if all three input metrics approached one, the result would be 0.33 * 3 = 1.

Simplifying the above, where Z is used in place of W, X and Y, gives:

$$M_{ACPLT} = M_{RPROP} * (C * Z + M_{DET} * (C^2 * Z + C^3 * Z * M_{SFORM}))$$

After application of the constants, the derived formula for M_{ACPLT} is thus as follows:

$$M_{ACPLT} = M_{RPROP} * (0.47 + M_{DET} * (0.65 + 0.91 * M_{SFORM}))$$

An interesting derivative use of M_{ACPLT} would be to apply it to different subhierarchies of a knowledge base in order to determine which areas need work, or alternatively, which areas contain more useful knowledge.

Pure complexity, M_{PCPLX}

The objective of this metric is to combine all the independent complexity measures into a size-independent metric for the 'difficulty' or 'sophistication' of a knowledge base. It was decided to use a mechanism similar to that used to calculate function points (see section 5.2.3). M_{MACPLT} is used as the analog for 'unadjusted function points'. The four metrics not included in the calculation of M_{MACPLT} are then used as 'complexity adjustment factors'. After the application of each to either increase or decrease the unadjusted metric, the resulting metric M_{PCPLX} is an analog of 'adjusted function points'.

To calculate adjusted function points, the first step is to measure fourteen complexity adjustment factors using simple scales and then to sum the measurements (without weighting). The sum can be called the 'compound adjustment factor'. In the second step, a formula involving the compound adjustment factor results in the multiplication of the unadjusted function points by a factor ranging from 0.65 to 1.35. In the following paragraphs, a similar two-step approach is applied to knowledge base metrics.

Step 1: Scaling and summing. The four metrics not included in M_{MACPLT} have theoretical ranges of zero-to-one. However, upon examining data from user studies, it was determined that measurements of three of the four metrics rarely approach the top of that range (M_{SOK} is the exception).

Thus it was decided to apply multiplicative scaling factors to M_{SOK}, M_{ISA} and M_{MI} in order to unbias them (so that each metric contributes fairly to the result). The scaling factors (2.5, 1.6 and 1.2 respectively) were determined by examining the means and ranges of the test data. These scaling factors are the least objective aspect of the calculation of M_{PCPLX} – extensive data analysis would be needed to fine tune them; however since this research is intended to propose a method for developing metrics rather than a definitive formula, these figures are considered adequate. The resulting compound adjustment factor has a range of zero to four (see formula below).

Step 2: Creating the adjusted metric: It was decided to adjust M_{MACPLT} so that the resulting metric is M_{MACPLT} multiplied by between 0.25 and 1.85. This is a wider range than that used for function points, but it was decided the four adjustment factors should have a larger influence than those used for function points. Again, more extensive data analysis would be needed to arrive at a more objective range.

The derived formula for M_{PCPLX} is thus as follows:

$$\begin{aligned} M_{PCPLX} = \; & M_{MACPLT} * (0.25 + 0.4 * \\ & (M_{DIV} + 2.5 * M_{SOK} + 1.6 * M_{ISA} + 1.2 * M_{MI})) \end{aligned}$$

where the second line is the compound adjustment factor

Overall complexity, M_{OCPLX}

The overall complexity measure is simply M_{MSUBJ} multiplied by M_{PCPLX}. In other words the count of the number of main subjects is adjusted using the measure of ‘pure’ complexity of the knowledge base. M_{OCPLX} might be considered to measure ‘fully specified main subjects’. It is intended to serve as a measure of productivity or information content.

5.5 Desirable qualities of the metrics

This section discusses various useful qualities of the metrics presented in the previous section[68]. The following criteria are used to judge the quality of the metrics: 1) Does each metric have use that is independent of th others? 2) Is each metric understandable? And, 3) does each metric have a reasonable mapping onto the subjective phenomenon (including behaviour at extremes)?

5.5.1 How subjectively useful are the metrics?

Each metric must perform some useful task. There should be some reason why a user might want to use the metric independently of the others. The following paragraphs indicate that each metric apparently has a valid use by showing what the user can learn by using it.

- **All concepts: M_{ALLC}**: This gives an idea of the amount of memory and disk space used by the knowledge base. It also gives an idea of the number of discrete facts in the knowledge base.
- **Main subjects: M_{MSUBJ}**: This indicates to the user the number of things being talked about in the knowledge base, and hence is a natural measure of size that is independent of complexity. If the user follows a methodology of sketching out the inheritance hierarchy before filling in details, this metric can help him or her estimate the eventual size of the knowledge base and hence the amount of work that might be required to create it.
- **Relative Properties: M_{RPROP}**: This indicates to the user whether a knowledge base has a reasonable number of properties, relative to the number of main subjects. A low measurement might indicate that more properties should be added.
- **Detail: M_{DET}**: This indicates whether a knowledge base (or portion thereof) has a reasonable number of statements. A low measurement might suggest that additional statements should be added.

[68] This section should not be confused with an evaluation of how the metrics fared in actual use. Some discussion of the latter type is found in section 6.2.

- **Statement Formality: M_{SFORM}**: This indicates whether a knowledge base has a reasonable number of formal links. A low measurement indicates that the user would be unlikely to be able to generate sophisticated graphs of relations.
- **Diversity: M_{DIV}**: This indicates the degree to which knowledge is focussed at the top or bottom of the inheritance hierarchy. A low value might indicate that inheritance is not being properly used and that many concepts are merely placeholders.
- **Second Order Knowledge: M_{SOK}**: This indicates the extent to which knowledge is included about concepts themselves as opposed to the things represented by concepts. A low measurement indicates that there is probably significant knowledge missing.
- **Isa Complexity: M_{ISA}**: This metric indicates the degree to which the inheritance hierarchy has a non-trivial pattern of branching. A low measurement indicates that many concepts do not have siblings and thus relatively few distinctions are being made.
- **Multiple inheritance: M_{MI}**: This indicates the extent of the complexity added due to multiple inheritance. A low measurement indicates that little multiple inheritance is being used.
- **Apparent completeness: M_{ACPLT}**: By combining those metrics that should logically increase as a knowledge base approaches completion, this metric gives the user an idea of how close to completion the knowledge base might be.
- **Pure complexity: M_{PCPLX}**: By combining all the complexity metrics this gives the user an overall value of the ‘difficulty’ of the knowledge base, independent of size.
- **Overall complexity: M_{OCPLX}**: In combining pure complexity with size, this metric is intended to give the user an idea of the information content in a knowledge base.

5.5.2 How intuitive or understandable are the metrics?

To be preferred are simpler metrics, i.e. ones where users can easily understand the reasons for the calculations.

The methods of calculating M_{ALLC} and M_{MSUBJ} are the simplest, being mere counts. M_{DET}, M_{SFORM} and M_{MI} have a slightly lower level of understandability since they are simple ratios of reasonably understandable counts. M_{RPROP} is lower again in understandability since it is squared in order to give it better responsiveness to the underlying phenomenon. The other metrics all have relatively complex formulas.

5.5.3 How good is the mapping between the metric’s function and the subjective phenomenon?

The function described by the metric (the objective phenomenon) must correspond well throughout its range with the subjective phenomenon in the mind of the metric’s interpreter.

If the correspondence is poor, it might be because some objective factor is omitted or incorrectly weighted.

Metric	Meaning approaching 0	Meaning near 0.5	Meaning approaching 1
M_{RPROP}	No properties	2.5 properties per concept	Very many properties per concept
M_{DET}	No values specified	Values specified on half of statements	Value specified wherever possible
M_{SFORM}	No formal values	Half of values are formal	All values are formal
M_{DIV}	Properties introduced on one concept	Properties introduced on half of all concepts	Properties introduced evenly on all concepts
M_{SOK}	No second order knowledge	Metaconcept detail and extra terms on about half of concepts	Both metaconcept detail and extra terms on every concept
M_{ISA}	Each concept has about one parent - very simple	Binary tree	Very bushy tree - very complex
M_{MI}	No multiple inheritance	Half of concepts have an extra parent	Very high degree of multiple inheritance

Figure 5.5: Interpreting metrics – meanings of various values. Rows show metrics defined that are closed-ended. Columns indicate the interpretation of measurements at the extremes of that range and in the middle.

One test of correspondence for metrics with a zero-one range is as follows: The metric should yield a value of 0.5 when the subjective phenomenon is at about the half-way point, i.e. the subjectively 'normal' point. Another test is to ensure that the endpoints correspond with what one would intuitively expect. Figure 5.5 gives interpretations for the endpoints and centre points of the closed-ended metrics and shows that they behave well in these respects. The only possible exception is M_{RPROP} whose middle point indicates that there are 2.5 properties per concept – however this occurs in order to ensure that equal *deltas* of the metric correspond more closely with equal changes in the subjective phenomenon.

5.5.4 Summary of desirable qualities of the metrics

Figure 5.6 summarizes how well the metrics appear suited to the various tasks described in section 5.3.

Figure 5.7 lists each metric and then rates it using the three criteria discussed in the above subsections. The following are some general observations:

- The most useful metrics appear to be M_{OCPLX}, M_{DET} and M_{SFORM}. The overall complexity metric correlates reasonably well with time-to-complete, whereas M_{DET} and M_{SFORM} give clear indications of where work needs to be done in a knowledge base.
- The metrics that have the best combination of understandability and usefulness are M_{MSUBJ} and M_{DET}. These should require the least explanation to users. M_{ALLC} requires an understanding of CODE4's uniform representation of all knowledge units as concepts, whereas M_{ISA} and M_{ACPLX} have less-than-simple formulas.
- Most metrics correspond well with subjective phenomena.
- All the metrics with a zero-to-one range have good meanings for the ends of the ranges

Task	**Suitable Metrics**	**Observations**
A to D. Assessing the present state of a knowledge base		
Completeness	M_{ACPLT}	The compound metric of apparent completeness.
Complexity	M_{PCPLX}, M_{DET}, M_{SFORM}	Pure complexity, and two of its most important components: detail and statement formality
Information content	M_{OCPLX}	The overall complexity of the knowledge base
Balance	M_{DIV}	The diversity of distribution of properties; other balance metrics could be created.
E. Predicting	M_{MSUBJ}	The number of main subjects can typically be determined earlier than other metrics
F to I. Comparing	all	

Figure 5.6: Measuring tasks and how well they can be performed. At the left are the measuring tasks listed in section 5.3. The middle column lists some of the metrics that may be useful in the performance of the task.

5.6 Summary

The primary purpose the research in this thesis is to make knowledge management practical. It is very hard to manage something, however, unless one can quantify it.

This chapter has discussed several metrics that can be applied to frame-based knowledge representations. The metrics can be divided into three classes: 1) raw measures of size; 2) measures of various attributes of complexity, and 3) compound measures intended to help users assess their productivity.

With the help of metrics, users can better do such things as the following:

1. Estimate completeness of a knowledge base, or a component thereof. Using a metric like M_{ACPLT}, a user can decide how much work might need to be done and where.
2. Judge the overall volume of knowledge in a knowledge base, using M_{OCPLX}.
3. Obtain a rough idea of how difficult a knowledge base might be to navigate or modify; M_{PCPLX} can help with this.
4. Compare subjectively 'complete' knowledge bases to see how domains differ, in order to help in the estimation of future knowledge base development tasks.

The main scientific contribution of this chapter has been to point out several kinds of things that one might wish to measure in a knowledge base. It is hoped that the chapter work will stimulate further research.

Criterion:	1. Useful	2. Understandable	3. Mapping (well-behaved at center and extremes)
Raw size metrics			
All concepts: M_{ALLC}	2	5	5
Main subjects: M_{MSUBJ}	4	5	5
Independent complexity metrics			
Relative Properties: M_{RPROP}	3	3	5
Detail: M_{DET}	5	4	5
Statement Formality: M_{SFORM}	5	3	5
Diversity: M_{DIV}	4	2	4
Second order Knowledge: M_{SOK}	4	3	5
Isa Complexity: M_{ISA}	3	2	3
Multiple inheritance: M_{MI}	3	4	5
Compound complexity metrics			
Apparent completeness M_{ACPLT}	4	2	2
Pure complexity: M_{PCPLX}	4	3	4
Overall complexity: M_{OCPLX}	5	4	3

Figure 5.7: Ratings of the metrics based on three criteria. On the y axis (listed at left) is each of the metrics. On the x axis (listed at top) are the evaluation criteria discussed in sections 5.5.1 to 5.5.3. The numerical scores are on a scale of 0 to 5, with 5 being the best. Numbers are subjective judgements on the part of the author.

Chapter 6

Evaluation

This chapter presents an evaluation of the work discussed in the previous three chapters. The first section discusses the basic procedures used. Subsequent sections discuss the evaluation of the metrics (from chapter 5) and of the ideas built into CODE4 (from chapters 3 and 4).

6.1 The basic evaluation procedure

The basic procedure used to evaluate this research was as follows: To have as many people as possible build knowledge bases using CODE4, and then to study their work experiences and the knowledge bases they produced. Details of this procedure are discussed in section 6.1.1. Subsequent sections discuss the questionnaire that was used to gather information, statistical procedures used, and alternative procedures that were considered and rejected. Discussion of actual results is deferred to sections 6.2 and 6.3.

6.1.1 Details of the evaluation procedure

The basic steps were: 1) Solicit people to build knowledge bases; 2) Train them; 3) Enhance the system to accommodate their needs, and 4) Study the results of their knowledge base building efforts. Details of these steps follow:

Step 1: As many people as possible were solicited to build knowledge bases using CODE4.

A total of at least 30 individuals are known to have used CODE4 to build knowledge bases. These fall into three main groups: 1) industrial customers, 2) researchers and 3) graduate students taking courses. In addition, about ten other groups are known to have obtained copies of a demonstration version that cannot save knowledge bases.

The primary industrial customers for CODE4 have been groups headed by Jeff Bradshaw, formerly of Boeing Aircraft Corporation in Seattle, and now of Eurisco in Toulouse, France. Bradshaw's work has primarily involved building CODE4 into other applications. In (Bradshaw, Boose et al. 1992) an application involving design rationale capture is presented. Another application involving organizational modelling is discussed in (Bradshaw, Holm et al. 1992). Bradshaw is in the process of starting a further project involving an application of CODE4 to a medical domain. This will be taking place at the Fred Hutchinson Cancer Research Center in Seattle (Bradshaw, Chapman et al. 1992)

The Cogniterm project, which started with the use of CODE2 (see chapter 2), has continued its very successful work with CODE4 as its knowledge management tool. Some of the published literature includes Eck's MSc thesis (Eck 1993) and papers by Skuce and Meyer (Skuce 1993c; Meyer, Eck et al. 1994). Bowker (Bowker and Lethbridge 1994) is using CODE4 as a major tool for her PhD research.

Several other student research projects and theses have used CODE4 as their major tool; these include (Iisaka 1992), (Ghali 1993) and (Wang 1994). CODE4 has been used as a teaching tool in several courses in artificial intelligence, and there are plans to use it as a resource for programming languages and operating systems courses at the University of Ottawa.

Others who have used or studied CODE4 in significant ways and have expressed enthusiasm for it include David Aha (of the Naval Research Laboratory), Peter Clark (of the University of Texas at Austin) and Denys Duchier (of the University of Ottawa). The latter two have worked on applications that connect to the CODE4 knowledge server (see section 3.13). Another important user is Chris Drummond who has created several knowledge bases and exercised almost all the features available.

Step 2: The users were trained to use the system, or trained themselves.

A 100-page user manual (Lethbridge and Eck 1994) was created to train users. Additionally, users received training: 1) from graduate lectures, 2) by following a step-by-step tutorial created by members of the Cogniterm team, and 3) by personal consultation with members of the development team.

Step 3: CODE4 was regularly enhanced in response to requests from the users.

It was not possible to have CODE4 remain static during the research. Significant new features were added over a three-year period in order to meet the requirements of users. The kinds of features added after CODE4 started to be used seriously include:

- Knowledge representation features such as dimensions.
- New mask predicates to permit specialized queries.
- New primitive properties to permit the display and storage of specialized information such as multiple graph layouts
- New display options for existing mediating representations.
- An entirely new mediating representation: the property comparison matrix.

CODE4 was designed with a modular architecture to allow this kind of modification to be made relatively easily. Also, an automatic problem reporting feature is included in the system: When a user has a problem (or suggestion, request etc.) he or she can invoke the problem reporter. The problem reporter emails text of the problem, along with the current state of the system and various debugging information, to the CODE4 developers.

Step 4: Information was gathered from as many users as possible

A significant percentage of CODE4 users were willing to have their work studied in detail. These users are called the *participants*[69] in the following discussions.

The total number of participants was twelve; all used CODE4 at the University of Ottawa. Eleven were graduate students and one was a professor. Of the graduate students, five were terminologists participating in the Cogniterm project. Student usage was divided between those using CODE4 for course work, and those using it in thesis research.

The participants created 25 knowledge bases. These covered a very wide range of topics, categorized as follows:

- Computer languages and operating systems (eleven knowledge bases).
- Other technical topics (e.g. optical storage, electrical devices, geology, matrices – nine knowledge bases).
- General purpose knowledge (e.g. people, vehicles, fruit, top level ontology – five knowledge bases).

The total amount of work involved in creating these knowledge bases was reported to be about 2000 hours, i.e. an entire person-year.

The work of the participants was studied in two ways:

1) By analysing the participants' responses to a detailed questionnaire and follow-up interview questions. Further details about the questionnaire are found in the next subsection.
2) By measuring the knowledge bases, in particular using the metrics discussed in the previous chapter.

6.1.2 The CODE4 user questionnaire

Twenty copies of a questionnaire were distributed and twelve were returned. Some of the questionnaire is repeated in Appendix A, along with summaries of the answers. Other parts of the questionnaire are presented directly later in this chapter. This thesis omits consideration of responses to questions of the following types:

[69] There were other users in *addition* to the participants. These did not actively particpate (filling out questionnaires, being interviewed etc.) because they were busy, their work was confidential etc.

- Questions that were merely administrative (e.g. requesting names, permission-to-use etc).
- Questions that received too few responses to draw statistically significant conclusions (see next subsection).
- Questions which, as was discovered in follow-up interviews, users had misunderstood or had interpreted in different ways.

The questionnaire contains six parts containing a total of 55 main questions, many of which have sub-questions: There are a total of 259 sub-questions about CODE4 in general and 130 sub-questions about each knowledge base. Most participants took several hours to complete the questionnaire and some were quite enthusiastic about it despite the amount of work involved.

The first two sections of the questionnaire ask general questions in order to ascertain how much users know about CODE4, how they learned it and how much they have used it. The subsequent two sections focus on the user's reactions to specific features of the software. The fifth section deals with users' experiences developing individual knowledge bases. Questions in this section were designed to complement the measurement of the knowledge bases. The final section of the questionnaire covers problems and desires users encountered in the use of CODE4.

6.1.3 Statistical tests

In order to verify that numbers reported are not likely to be chance occurrences, statistical tests were applied to data gathered from the questionnaires and from the measurement of knowledge bases. There were three types of tests:

Test type 1: Confirming that the *mean* reported for a question or measure is significant

The objective here is to confirm that each mean reported from the sample (of users or knowledge bases) is sufficiently close to the population mean (the actual mean that would be obtained if data could be gathered from all users or knowledge bases of the type studied). For this objective, *sufficiently close* means that, 19 times out of 20, the population mean must be within an interval defined as the sample mean plus-or-minus 10% of the range of the sample.

Standard t-tests were used to make the above confirmations. Such tests require that the underlying distributions of data be normally distributed or, failing this, that the sample size be greater than about 30 (thanks to the central limit theorem). In this research both conditions are close to being fulfilled much of the time: The number of knowledge bases studied was 25; also most of the measurements and questions were of the type where values were far more likely to occur in the center of a range than in either tail, but where values do occur in both tails. The following are examples of the kinds of populations from which samples were taken and which would appear close to normally distributed:

- Complexity measurements that range from zero to one but for which a value in the middle of the range is to be expected.
- Size measurements, where very small and very large knowledge bases are far less common than medium-sized ones.
- Users opinions placed on a scale, where intermediate values are to be expected.

If there are cases where the prerequisites for a t-test might not have been fully met, many of the conclusions presented are still likely to be valid because many of the tests indicated statistical significance even in a 99% confidence interval.

If confirmation of significance could not be reached for a given question or measure, the data is not reported in this thesis. This generally occurred when not enough participants responded to a particular question and/or when they disagreed markedly in their responses.

Test type 2: Confirming that the coefficient of linear correlation between two measures (or a measure and a question) is meaningful.

The objective here is similar as that in test type 1: To confirm that the correlation coefficient is sufficiently close to the population correlation coefficient. The criterion is that it should be within plus or minus 0.1, of the sample value, 19 times out of 20. As above, t-tests are used to confirm this.

Note that this test is not attempting to show that measures are independent or dependent.

Test type 3: Verifying that there is a statistically significant difference between two means.

Such tests were made to compare several related measures or questions (e.g. when determining which of two features users preferred). The objective is to determine that the actual difference between means is non-zero, with 95% confidence. Again t-tests were used to do this.

6.1.4 The indirect nature of the evaluation

The evaluation of this research must be understood to be constrained by a number of factors including the following:

- The benefits of most aspects of the research can only be assessed indirectly; i.e. subjective interpretation of users and/or the author is involved.
- A particular tool, CODE4, is used in the evaluation. It therefore may not be possible to extrapolate all conclusions to the general case. For example: CODE4 may constrain users

to choose only certain domains or to represent only certain details (and thus the users may encounter more or fewer of the problems listed in section 1.4).

- Only a relatively small selection of users participated in this study, and they were largely students. It is possible that different results may be obtained from other sets of users. One should thus consider all statistical results as only being applicable to populations similar to the users in the study.

6.1.5 Why not run a ‘controlled’ experiment?

The methodology described above for the evaluation of this thesis has been that of a *natural* experiment (i.e. trying to reach conclusions about an activity by observing as many cases of that activity as possible, but without interfering). An alternative approach would have been to design a *controlled experiment.*

In the field of knowledge engineering, a controlled experiment might involve a procedure like the following: First, create two versions of a knowledge management system that differ according to the presence or absence of some feature. Then have separate sets of users create knowledge bases with each system. Finally, measure the resulting knowledge bases to see if the presence or absence of the feature causes any difference in the resulting work.

Such an experiment involves varying one parameter (in this case a feature) while ensuring that differences in other factors, including ongoing changes to the system, do not influence any conclusions. This was judged impractical for the following reasons:

- It is not reasonable to assign a small number of people to create knowledge bases and expect that the resulting work will be comparable. Knowledge enterers have dramatically different levels of skill (analysis ability and understanding of knowledge representation) as well as background (general knowledge that they can apply to the problem).
- It might be possible to control for skill and background differences by using a large number of knowledge enterers. However it was found to be too difficult to find enough people with sufficient motivation to work for tens of hours on a single assigned problem. People were only willing to be participants if they could work on a topic that interested them.
- It might be possible to overcome biases imposed by a particular domain by having participants create several knowledge bases from a small pool of domains. Again however, finding willing people is extremely difficult (it was even difficult to get enough participants, who had already created knowledge bases, to fill in a questionnaire).
- During the conduct of this study, users would frequently ask for new features. If all participants were *assigned* a topic, it might have been possible to deny their requested modifications for the sake of gathering ‘good data’. However since most users were working on their own research projects, it was necessary to make regular incremental

modifications to the system. Otherwise many users would not have felt comfortable trusting CODE4 to be a reliable tool for their research.

In conclusion, without a very significant amount of money with which to pay participants for long periods of time, it was only feasible to perform an exploratory study of the type described in the previous subsections.

6.2 Evaluation of the metrics

This section presents an evaluation of the knowledge base metrics that were introduced in chapter 5, focussing on their independence from each other.

Figure 6.1 summarizes measurements taken of the knowledge bases prepared by the participants in this research. Details for individual knowledge bases can be found in appendix B.

Metric	**Theoretical Range**	**Mean**	**Minimum**	**Maximum**	**St. Dev.**
Raw size metrics					
All concepts: M_{ALLC}	(40-∞)	842	224	2825	612
Main subjects: M_{MSUBJ}	(0-∞)	91	21	278	61
Independent complexity metrics					
Relative Properties: M_{RPROP}	(0-1)	0.33	0.05	0.70	0.17
Detail: M_{DET}	(0-1)	0.18	0.03	0.69	0.14
Statement Formality: M_{SFORM}	(0-1)	0.16	0.00	0.67	0.19
Diversity: M_{DIV}	(0-1)	0.71	0.00	0.99	0.27
Second Order Knowledge: M_{SOK}	(0-1)	0.06	0.00	0.41	0.11
Isa Complexity: M_{ISA}	(0-1)	0.40	0.19	0.59	0.11
Multiple inheritance: M_{MI}	(0-1)	0.19	0.00	0.87	0.23
Compound complexity metrics					
Apparent completeness: M_{ACPLT}	(0-2)	0.20	0.03	0.39	0.11
Pure complexity: M_{PCPLX}	(0-5.6)	0.19	0.02	0.38	0.10
Overall complexity: M_{OCPLX}	(0-∞)	16	2	56	14

Figure 6.1: Statistics about knowledge bases created by the participants. Each row corresponds to one of the metrics discussed in chapter 5.

The following are some general observations about figure 6.1:

- The knowledge bases differ substantially in size. When measured using M_{ALLC} and M_{MSUBJ} the ratio of largest to smallest is about 13:1. When measured using M_{OCPLX}, however, a more realistic ratio appears, i.e. 28:1.
- The knowledge bases vary widely according to all of the independent complexity metrics, in particular according to M_{DIV} and M_{MI}. Of these, M_{DIV} is probably the most interesting since it seems to be more of an indicator of the individual style of the knowledge base developer than the other metrics.

6.2.1 How independent is each complexity metric from the others?

Metrics should be as independent as possible because it would be redundant to make measurements using two metrics that are dependent on each other. The simple complexity metrics were *designed* to be independent of each other – all involve separate aspects of a knowledge base. However, the only way to really test for independence is to calculate correlation coefficients.

As figure 6.2 indicates, for the most part success has been achieved. The biggest exception is the reasonably strong negative correlation between the isa complexity and the amount of multiple inheritance. This can be accounted for theoretically because if there are more parent concepts to be multiply inherited (including by leaves), then the proportion of leaf types should decrease. Despite this moderate correlation, having a separate measure of multiple inheritance still appears to be useful.

	Multiple Inher.	Isa Complex.	Second Order	Diversity	Statement Formality	Detail
	M_{MI}	M_{ISA}	M_{SOK}	M_{DIV}	M_{SFORM}	M_{DET}
Relative Properties: M_{RPROP}	0.15	-0.23	0.11	-0.13	-0.17	0.17
Detail: M_{DET}	0.12	-0.28	-0.21	-0.06	-0.16	
Statement Formality: M_{SFORM}	-0.19	0.04	-0.02	0.34		
Diversity: M_{DIV}	-0.18	0.07	-0.08			
Second Order Knowledge: M_{SOK}	-0.21	0.14				
Isa complexity: M_{ISA}	-0.58					

Figure 6.2: Coefficients of linear correlation among the seven complexity metrics. The data used in calculating these coefficients was obtained from the knowledge bases prepared by the participants.

6.3 Evaluation of CODE4

This section presents an evaluation of the ideas built into CODE4 as part of this research. The ideas themselves are found in chapters 3 and 4.

The evaluation is divided into four subsections: Section 6.3.1 discusses conclusions that can be drawn from the enthusiastic reception of CODE4 by users. Sections 6.3.2 to 6.3.4 discuss responses to the questionnaires. Section 6.3.5 evaluates the system by looking at measurements.

6.3.1 Evaluation by observing the general use of CODE4

One of the most important attributes of this research has been its use by significant numbers of users. The inference that can be drawn from this is that CODE4 has achieved its general goal of providing practical knowledge management.

The following are two specific questions that must be answered affirmatively in order to be sure that the software has a broad practicality.

Do people *choose* to use the software in their work over a significant period of time and in significant projects?

If this is true, then there must be an advantage to the software. The software must be, in some sense, more practical than other software available.

For CODE4, the answer to this question is yes. In particular, the Cogniterm project and Jeff Bradshaw’s projects demonstrate that users perceive CODE4 to have desirable features.

Is the software used in a variety of domains?

Positive answers to this indicate that the software is general purpose. It is far easier to create special purpose software; however a practical knowledge management tool must be general purpose.

For CODE4, the answer is again yes. Knowledge bases have been developed in such diverse domains as software engineering, geology, organizational modelling, optical storage devices and top-level ontology development.

6.3.2 Features users found useful

Users were asked various questions in order to assess the usefulness of the various features built into CODE4.

What aspects of knowledge bases convey more of their content?

For each knowledge base, users were asked to estimate how much various aspects of the knowledge representation contributed to conveying its actual content (i.e. to expressing the important facts or information in the knowledge base).

The results are shown in figures 6.3 and 6.4. Figure 6.3 lists the 14 knowledge representation aspects about which users were asked. Note that the aspects are not necessarily disjoint; e.g. users were asked about values (of statements) in general as well as particular types of value.

The aspects are listed in decreasing order by mean, however there is no statistically significant difference[70] between aspects whose means are close together. An analysis of where statistically significant differences do exist showed that the aspects can be divided into eight groups (i.e. equivalence classes) labelled A through H. Figure 6.4 can then be used to determine which groups convey significantly more knowledge than others. For example none of the aspects within group A were considered more important than others, whereas all of group A was considered more important than all of the others (groups B through F). Similarly the aspect in group B is only more important than aspects in groups D and F, but not more important than any of the aspects in group C.

Some general observations about figures 6.3 and 6.4:

- As group A shows, names are considered to be of paramount importance. Users appear to attach great significance to the fact that humans will be using the knowledge base and thus concepts must have effective names. In a purely formal system, operated on by a computer, names might be less significant.
- Group A supports the intuitive notion that the inheritance hierarchy, with labelled concepts and attached properties, is the fundamental framework for building knowledge bases. Other aspects of knowledge can only be added once a rough inheritance hierarchy is specified.
- The way the property hierarchy is structured has as much importance as the values of statements. This validates CODE4's emphasis of this feature.
- Informal values are considered significantly more important than formal ones (and more important than the graphs of relations that the latter permit to be drawn). This suggests that developing effective ways of dealing with informal values can be a worthwhile exercise. Substantial and useful knowledge bases can be built that contain largely informal knowledge.
- As would be intuitively expected, values of statements about main subjects were considered more important than values of statements about metaconcepts; although metaconcept values were important to some.
- Graph layouts and dimension labels, features supported strongly in CODE4, were considered moderately important. Among knowledge bases, however, these features varied a lot in importance.

[70] According to pairwise t-tests.

Group	**Aspect**	Mean	Max	Min	Std. Dev	Responses
A	The names given to main subjects	9.3	10	7	1.0	21
	The names given to properties	9.1	10	6	1.6	21
	The structure of the inheritance hierarchy	9.1	10	5	1.5	21
B	The structure of the property hierarchy	8.0	10	0	2.5	21
C	Values of statements about main subjects	7.5	10	0	3.2	21
	Informal values as carefully thought-out natural language expressions or sentences	7.2	10	3	2.8	21
	Informal values that are commentary and descriptive	7.2	10	1	3.2	21
D	The layout of graphs	5.9	10	0	3.6	21
	The dimension labels	5.6	10	0	4.0	21
E	Values of statements about metaconcepts	4.7	10	0	3.9	21
	The structure of other relation graphs	4.5	10	0	3.5	21
	Formal values	4.1	10	0	3.7	21
	The order of subproperties in a property hierarchy	3.6	9	0	3.5	21
F	The order of subconcepts in an inheritance hierarchy	2.8	9	0	3.4	21

Figure 6.3: Aspects of a knowledge bases that convey its content. For each knowledge base, each aspect was ranked by the participants on a scale where 0 means unimportant and 10 means essential. The horizontal lines distinguish groups of aspects that are independently comparable with other groups, as shown in figure 6.4

	F	E	D	C	B
A	√	√	√	√	√
B	√	√	√		
C	√	√			
D	√				

Figure 6.4: Significant differences among groups of knowledge-conveying aspects. The x and y axes indicate groups of content-conveying aspects from figure 6.6. Wherever a check mark appears, users believed that row aspects were significantly more important than column aspects. Significance was determined using t-tests.

How useful were the particular knowledge representation features?

Users were asked how useful they perceived various knowledge representation features to have been in their work. This set of questions differs from the last set in the following ways:

- The previous set of questions ask about specific knowledge bases, whereas these questions ask for overall impressions of users. The sample sizes are thus smaller.
- These questions ask about features in more detail than the previous ones. The result is that fewer users were able to answer any given question.

- These questions ask about usefulness in general, whereas the previous ones focussed on how information content was conveyed. These questions use a scale where -5 indicates that the feature was harmful to their work (i.e. negatively useful), 0 indicates that the feature is of no use, and +5 indicates that the feature is very useful. The previous questions on the other hand, had just required responses on a scale of zero to 10.

Complete statistics for this set of questions is found in appendix A2. The following are some comments:

- Inheritance and multiple inheritance were judged more important than any other feature, and were classed as 'essential'. This corroborates the conclusion (from the questions about information content) that the inheritance hierarchy is of paramount importance.
- Significantly less important, statistically speaking, was the property hierarchy, however it too was judged very close to essential.
- The next most important feature was multiple parents in the property hierarchy, however no conclusion can be drawn about whether it is in fact more important than the twelve features that follow it in the ranking. Despite this, the emphasis CODE4 places on the property hierarchy appears to correspond with users' needs.
- A large number of features were ranked close together and no conclusions can be drawn about the exact order. In this group are formal and informal values (formal values are ranked marginally higher than informal ones, contrary to findings discussed earlier, but not significantly higher).
- Ranked close together at a moderate level of importance were several features associated with naming (synonyms, unrestricted syntax for names, and treating terms as concepts). However the feature of CODE4 that results in automatic default naming was ranked far lower (close to 'unimportant'). The latter might be because users almost always name a concept as soon as they create it, so they consider the default name of little use. However, generating a default name permits the interface to be non-modal (this was ranked highly in a question discussed in the next subsection).
- Although no features were given an overall assessment of 'harmful' (i.e. negatively useful), three features were given such an assessment by individual users. These were automatic combination under multiple inheritance, treating facets as properties and automatic default naming.

How useful were particular user interface features?

In a similar manner to the questions about knowledge representation features, users were asked a series of questions about user interface features. They used the same scale as the above (where -5 means harmful and 5 means essential).

Most of the features about which questions were asked are either novel in CODE4 or are emphasised far more in CODE4 than in other knowledge management technology.

Complete statistics are in part 3 of appendix A; the following are some specific observations:

- Four features were ranked as close to essential and were significantly more important than most of the others. Foremost of these was the outline mediating representation. Close behind this was: a) dynamic updating of windows, b) direct typing to change a name and c) the graphical mediating representation.
- Ranking significantly behind the above four, but still ‘very important’ were the ability to save multiple graph layouts, the control panel and the ability to make multiple selections. These features are not ‘new ideas’, but their inclusion appears to greatly enhance the system.
- A very large number of features were ranked indistinguishably close to each other as ‘important’. Examples are the non-modality of the interface, the matrix mediating representation, relation knowledge maps and features concerned with the mask.
- No features were ranked as ‘unimportant’ or ‘harmful’ but a few features were ranked as only slightly useful. These included the ability to attach a graphical icon to a concept and the ability to have multiple knowledge bases loaded at once. The latter feature was ranked as harmful by one user.

Which mediating representation did users use most when entering knowledge?

For each knowledge base, users were asked to estimate how much of the knowledge was entered using the four major mediating representations. In agreement with the above general questions, users reported using the outline representation far more than any other.

The matrix representation was used relatively little, although it was developed after some users had started their work. Also, it is not (yet) possible to *start* creating a knowledge base using a matrix window; it can only be used as a query mechanism or to fill in missing values.

Figure 6.5 gives the data about mediating representation usage. The statistics for the ‘user language’ mediating representation and the outline representation are likely to be too low and high respectively. This is because many users are not aware that the user language representation is considered distinct from the outline representation (the former, which is used only for entering a single property value, is usually associated with an outline but can be associated with a graph).

Mediating Representation	Mean	Max	Min	Std. Dev
Outline	64.4	100	30	22.8
Graphical	14.8	50	0	18.7
User language	16.3	50	0	16.7
Matrix	4.6	20	0	6.1

Figure 6.5: Amounts of knowledge entered using different mediating representations.

What other information can be learned about formality and informality in CODE4?

One of the major tenets in the design of CODE4 was that users should be able to represent knowledge both formally and informally. The following questions were designed to ascertain whether participants took advantage of this:

Participants were asked to judge the degree of formality of their knowledge bases on a scale where 0 means completely informal and 10 means completely formal. The mean response was 4.8 with a standard deviation of 2.7. Responses ranged from 0 to 9 for the 24 knowledge bases from which responses were received. Interestingly, individual participants gave very different responses for different knowledge bases they created. It thus appears that informality and formality are both necessary, although as was discussed above, users feel the informal aspects might convey more knowledge.

Participants were also asked to indicate how rapidly they would represent an idea when it occurred to them. The principle here is that informal capabilities should allow users to rapidly record their thoughts, if they so wish. The scale used was as follows: 0 means that they would represent the idea immediately and later on worry about whether it was correct; and 10 means that they would never represent the idea until they were absolutely sure they could do it correctly. The mean response was 4.2, with a range of 0 to 8 and a standard deviation of 2.4. Being in the middle of the range, this suggests that users need both facilities to rapidly enter ideas, and also facilities to rapidly help them decide whether they *should* enter an idea.

6.3.3 Tasks performed during knowledge management

Users were asked several questions pertaining to the tasks or activities they perform while doing knowledge management. The objective of asking these questions was to ascertain whether the features provided in CODE4 are likely to be solving the most prevalent problems or not.

How difficult do users find various knowledge management tasks?

For each knowledge base, users were asked how difficult various tasks were to perform. Difficulty was ranked on a scale where zero indicates extremely easy and ten indicates extremely difficult.

The data is shown in figures 6.6 and 6.7. Figure 6.6 orders tasks by mean. Figure 6.7 shows which groups of tasks were significantly more difficult than others. The following are general observations:

- None of the tasks was judged to be extremely difficult; most of the tasks were closer to ‘easy’ than to ‘difficult’. This suggests that CODE4 is helping users.
- The only possible exception was ‘Determining whether a part of the knowledge base is correct, coherent and consistent’. This task was judged significantly more difficult than almost all others. This suggests the need for more facilities to provide feedback.
- Judged to be less difficult than the above, but still moderately difficult, were two ‘understanding’ tasks: understanding the domain, and understanding CODE4’s error messages. The latter again suggests that better feedback facilities are needed.
- Several tasks related to specifying properties are rated moderate (neither easy nor very difficult). These tasks include naming a property, dealing with several properties that have the same name, determining the most general subject for a property, determining whether a property is already present or not, and moving a property to a new most general subject. This suggests that a ‘property assistant’ tool would prove useful.
- Tasks that are largely mechanical in nature were judged reasonably easy by the participants. Such tasks included reparenting and setting up windows.
- Two tasks were judged significantly easier than the others: These were finding a main subject that already exists and understanding the effects of inheritance. This helps show that CODE4’s querying and display capabilities are effective.

Group	**Knowledge management task**	Mean	Max	Min	Std. Dev	Responses
A	Determining whether a part of the knowledge base is correct, coherent and consistent	5.8	10	2	2.4	23
B	Understanding the subject matter being entered into the knowledge base	5.2	10	0	3.2	23
	Understanding what is the problem when the system does not permit a command	5.1	10	0	3.2	23
C	Naming a property	4.3	8	0	2.9	23
	Determining the interrelationships among a particular set of concepts	4.2	9	1	2.5	19
D	Determining whether a part of the knowledge base is complete enough	4.2	8	1	2.9	20
	Dealing with the situation where there are several properties with the same name which should really have been the same property	4.2	9	0	2.9	20
E	Figuring out where to put a concept in the inheritance hierarchy	4.1	8	1	2.2	23
	Figuring out the most general subject for a property	4	9	2	1.8	23
F	Specifying a special relationship (e.g. part-of) between concepts	3.8	9	0	2.9	22
G	Determining whether a property is important enough to add	3.6	8	1	2.0	23
	Determining whether a property exists already or not	3.3	8	1	2.1	22
	Changing a character string (possibly in many places)	3.3	10	0	3.3	23
H	Determining whether a concept is important enough to add	3.2	8	1	1.8	23
H	Filling in a value, in general	3.1	10	0	2.1	23
I	Moving a property to a new most general subject	3.1	9	0	3.5	22
H	Figuring out where to put a property in the property hierarchy	3.1	7	1	1.9	23
I	Reparenting a property	3.1	9	0	3.5	20
I	Reparenting a concept	3.0	9	0	3.2	21
H	Determining on what statement to put a value	3.0	7	0	2.0	21
H	Setting up windows to show the knowledge in which you are interested	2.8	10	0	2.8	23
I	Naming a main subject	2.7	7	0	2.6	23
I	Finding a property that already exists in the knowledge base	2.6	7	1	2.1	21
J	Finding a main subject that already exists in the knowledge base	1.9	7	0	1.7	21
	Understanding how inherited values get their content	1.8	9	0	2.3	23

Figure 6.6: Users' perceptions of the difficulty of tasks. For each knowledge base, each task was ranked by the participants on a scale where 0 means extremely easy and 10 means extremely difficult. The horizontal lines distinguish groups of tasks that are independently comparable with other groups, as shown in figure 6.7. Groups H and I could not be separated by a unique line.

	J	I	H	G	F	E	D	C
A	√	√	√	√	√	√	√	√
B	√	√	√	√				
C	√	√	√					
D	√	√						
E	√	√	√					
F	√	√						
G	√							
H	√							
I	√							

Figure 6.7: Significant differences among groups of knowledge management tasks. The x and y axes indicate groups of knowledge management tasks from figure 6.6. Wherever a check mark appears, users believed that row tasks were significantly more difficult than column tasks. Significance was determined using t-tests.

How difficult were naming tasks?

Users were asked what percentage of main subjects or properties did not have standardized terms; i.e. they had trouble putting a term on a concept, deciding if two terms had the same meaning or even *what* a term meant. The mean response was 29% (range 10% to 90%; standard deviation 22%). This indicates naming is not a straightforward task, and justifies CODE4's attempts to provide better facilities to handle names.

As figure 6.6 shows, users found naming properties to be significantly more difficult than naming main subjects.

Users were also asked what percentage of main subjects and properties had a name that the participant invented, as opposed to one that might be found in an ordinary dictionary or technical glossary. The mean was 17% (range 0% to 70%; standard deviation 18%).

6.3.4 General benefits of CODE4

How much insight into the *domain* did participants obtain?

In almost all instances, the participants report that they obtained substantial insight about the subject matter when using CODE4. On a scale where 0 indicated that constructing the knowledge base resulted in no new insights and 10 indicated that many new insights were obtained, the mean response was 7.1 (range 2 to 10; std. dev. = 2.6).

Was creating the knowledge base a worthwhile exercise?

Users were asked, for each knowledge base, whether creating it was a worthwhile exercise. Answers were put on a scale where -5 meant 'strongly disagree', 0 meant 'no opinion' and 5 meant 'strongly agree'.

The mean response was 4.2 (range 0 to 5; std. dev. 1.5), indicating a reasonably high feeling that using CODE4 was worthwhile. Users however gave significantly different answers for different knowledge bases: As would be expected, those knowledge bases used for serious research were judged more worthwhile.

How do users find CODE4's capabilities compare with those of other representations and tools?

For each knowledge base they developed, each participant was asked to compare CODE4 with other representational technologies with which he or she was familiar.

The question posed was: "How easily could you have represented the same knowledge using the following types of software or representation methods?". The answers were placed on a scale where -5 meant that the knowledge could have been represented much more *easily* using the alternate technology than using CODE4; zero meant that CODE4 equalled the technology in representational ease, and 5 meant that using the alternate technology instead of CODE4 would have resulted in much more *difficulty* (i.e. that CODE4 was much better for the task than the alternative technology would have been be).

In all cases, the mean rating indicates CODE4 is easier for the task than alternate technologies. Figure 6.8 summarizes the results and figure 6.9 shows a plot of the results. In particular it is notable that CODE4 was perceived as easier to use than both informal representation techniques (e.g. natural language) and formal representation techniques (e.g. predicate calculus and conceptual graphs). Hypertext is CODE4's only significant competition: One user ranked hypertext as far easier than CODE4, but on average, users still found hypertext a bit harder to use for knowledge management.

Technology	**Mean**	**Max**	**Min**	**Std. Dev**	**Responses (KBs)**
Spreadsheet	4.2	5	2	1.2	17
Drawing program	3.9	5	2	1.2	17
Relational database	3.0	5	1	1.6	15
Natural language (word processor)	3.0	5	0	1.7	20
Outline processor	2.9	4	0	1.5	7
Hypertext	1.0	5	-5	2.4	11
Predicate calculus	4.6	5	1	1.1	14
Prolog	4.4	5	1	1.2	11
Other AI tools (any the user had used; e.g. expert system shells)	3.9	5	-1	2.0	8
Conceptual graphs	3.3	5	-1	2.0	12

Figure 6.8: Perceived ease of use of representational technologies. larger numbers indicate that the technology is perceived as being more difficult to use for the kind of representational tasks studied. CODE4 (the baseline for comparison) is zero. A negative number would indicate that the technology is perceived as easier than CODE4. There are no negative means; only a few individual negative values.

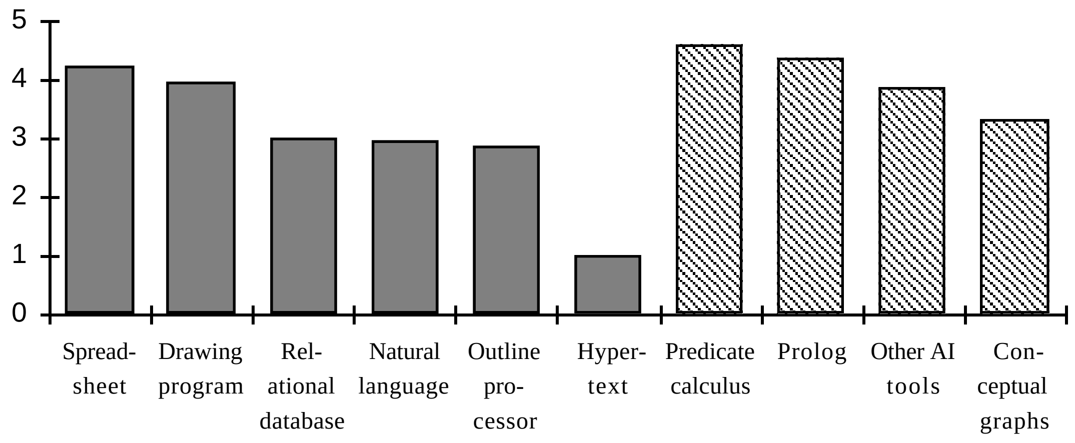


Figure 6.9: Histogram of perceived ease of use (see data in figure 6.8). The technologies on the right are those from the field of artificial intelligence.

6.3.5 Analysing the knowledge bases created by participants

The knowledge bases created by the participants can be analysed by measuring various attributes. Figure 6.10 shows counts of the various classes of concepts in CODE4 knowledge bases. Figure 6.1 shows measures using the various metrics proposed in chapter five.

Class of concept	**Total**	**Mean**	**Minimum**	**Maximum**	**St. Dev.**
Types (includes 4 primitives each)	2311	96	31	288	60
User instances	182	8	0	55	16
Properties (inc. 32 primitives each)	3744	155	52	397	60
Statements	7399	308	34	1614	355
Terms	6362	265	93	739	168
Metaconcepts	209	9	0	80	20
All concepts: M_{ALLC}	20207	842	224	2825	612
Main subjects: M_{MSUBJ}	2177	91	21	278	61
Statements per main subject		*2.45*	*0.67*	*8.24*	*1.61*
Terms per main subject		*1.03*	*1.00*	*1.20*	*0.05*

Figure 6.10: Counts of classes of concepts created by participants

The following are some general observations arising from the measurements:

- The total number of concepts and main subjects confirms that substantial work has been done using CODE4. The size of the larger knowledge bases confirms that some of the work was serious in nature.

- The measured formality of the knowledge bases differed substantially from the estimates of users. Whereas users considered their knowledge bases to be 48% formal, MSFORM indicates that the knowledge bases were only 16% formal on average. The coefficient of linear correlation between measured and estimated formality was 0.52. This indicates that users can recognize formality but they cannot quantify it well.
- The data confirm that users did make use of such features as independent metaconcepts and terms.

6.4 Summary

The features CODE4 provides have been used by a significant number of users and have enabled the production of a significant number of useful knowledge bases. Most users are happy with the system and use it productively. Most of the features appear to be important because the removal of any individual one would significantly impact either individual productivity or the ability of the system to be adapted to specific needs. The evaluation provided in this chapter can guide the development of future knowledge management technology.

Readers should note, however, that the evaluation is constrained in several ways that limit the generality of any conclusions. In particular, although a variety of users was involved, they were mostly students. Also the only tool tested was CODE4, and it cannot be guaranteed that favourable results with this tool will necessarily extend to other tools.

The following summarizes the results of evaluating important features. More discussion of the contribution of features can be found in section 7.2. It is important to note that many features of CODE4 are interdependent. For example, users did not directly judge the uniformity of concepts to be important; they nevertheless found facilities to deal with terms, metaconcepts and properties to be useful. Treating concepts uniformly allows the latter to be designed more easily; such treatment also facilitates browsing and other useful capabilities. Similar dependences exist that justify the inclusion of such features as multiple selections and knowledge maps.

- **Features for dealing with the inheritance hierarchy**: Users judged the inheritance hierarchy to be one of the most generally important features and to convey much of the knowledge in their knowledge bases. While this is not unexpected, it suggests that knowledge management systems must provide effective facilities, like CODE4's outline mediating representation, for rapid manipulation of this hierarchy.
- **The property hierarchy and features for dealing with it**: Users found these both generally important and useful in conveying knowledge.
- **Features for dealing with terms and naming**: Users found names of properties and main subjects to be very important in conveying meaning. They also found naming to

be intrinsically difficult since they had to frequently invent names or choose among several alternatives.

- **Features for handling informality**: Users judged that informality was important in conveying much of the knowledge. The actual percentage of informality in the knowledge bases indicates that these features are important.
- **The non-modality of the interface and dynamic updating**: These features were rated important by users.
- **Mediating representations**: Users rated all three major mediating representations as useful, particularly outlines. They considered the ability to save graph formats to be important.
- **The mask**: Users found the mask to be useful. They also indicated that finding concepts, a task frequently performed using the mask, was one of the easiest tasks to perform.

The following needs were identified during the evaluation. These points are discussed further in the future work section of the next chapter (section 7.3).

- **Tools to help specify properties**: Users found naming properties to be particularly difficult. They also found manipulating them to be difficult and in need of support.
- **Tools to assist with analysis**: Since users reported that one of their most difficult problems was evaluating correctness and coherence of the knowledge base, there is an obvious need for analysis tools. The metrics, which were unavailable to users when they created their knowledge bases, might help in this process.
- **Improved facilities for feedback**: Users found that understanding CODE4's error messages was one of the most difficult tasks.

The following are some points from the evaluation that show that, overall, CODE4 is a valuable tool:

- Users regarded developing their knowledge bases as a worthwhile exercise.
- Users used CODE4 repeatedly and produced significant knowledge bases with it.
- Users judged CODE4 to be easier to use than other tools for knowledge management.

Chapter 7

Contributions and Future Work

7.1 General summary of the research

The goal of this research has been to develop practical techniques for knowledge management; i.e. the process of acquiring, representing storing and manipulating complex patterns of concepts and their interrelationships. To be practical, a knowledge management system ought to be usable in a variety of non-computer tasks (see section 1.3) by people with little computer background; however it should also have sophisticated features for the expert.

As the first stage of the research a set of problems were identified (section 1.4). These were found to exist in a variety of existing technologies that can be used for knowledge management. Some of the problems appear intrinsic to knowledge management while others appear due to inadequate tools.

Many of the ideas developed to assist knowledge management can be described as capabilities or techniques the user can employ to better *organize* knowledge. Some of these organizing techniques fall into the domain of abstract knowledge representation (chapter 3), while others fall into the domain of user interface (chapter 4).

To serve as a testbed for the knowledge organization techniques (and indeed, to help develop those ideas themselves) a large knowledge management system called CODE4 was developed. The discussions in chapters 3 and 4 focus on how CODE4 and its abstract knowledge representation CODE4-KR implement the organizing techniques. In addition, a set of metrics was developed to help analyse knowledge bases.

To assess the research, a study was made of the use of CODE4 by about twenty users over three years. The study concludes that the features built into CODE4 combine to make knowledge management more productive.

7.2 Contributions of this research

This section lists those ideas developed in this research which are believed to contribute to a general understanding of how to create a practical knowledge management system. Some ideas are novel while others represent improvements of existing ideas or else new applications of old ideas.

The most significant contribution is the synthesis of a large number of these ideas in one system.

Most of the contributions can be thought of as techniques for better organizing knowledge, both in the knowledge representation as well as in how the knowledge is presented to the user through the user interface. The degree to which the contributions solve problems is indicated in figure 7.1.

7.2.1 Contributions to knowledge representation

The following are some ways in which CODE4-KR contributes to knowledge representation. In addition to these features, CODE4-KR has several others, such as its treatment of informal values, that are less novel but that combine with these in useful ways.

a) **The uniformity with which concepts are treated**: In other systems fewer things are generally called concepts, whereas in this research an effort has been made to call any unit of representation a concept. The result of the uniform treatment is that a common set of operations can be applied to all such units. This research contributes a useful taxonomy of classes of concept. While other approaches, such as Cyc, have a more uniform treatment of knowledge representation units than is traditional in artificial intelligence, CODE4 apparently takes concept uniformity further than any other tool.

b) **The separation of terms from the concepts they stand for**: Making terms concepts in their own right permits effective representation of linguistic knowledge. It also allows for the easy handling of synonyms, multiple natural languages, homonyms etc. This is believed to be novel in a knowledge management tool.

c) **The separation of metaconcepts from the concepts they represent**: This clarifies an important distinction and reduces representational errors. Although the idea of metaconcepts is not new, certain details are: For example: 1) the assumption of the existence of a metaconcept for every concept, and 2) the attachment of all non-inheriting properties to metaconcepts.

d) **The treatment of statements as concepts**: This apparently novel idea improves the uniformity of the knowledge representation. It means for example that the arcs or indentations shown in a mediating representation, which correspond to statements, can be queried to find their properties.

e) **The property and statement hierarchies**: These are techniques for organizing knowledge that users have found useful. Other systems have hierarchies of slots, but the idea of a global hierarchy and subhierarchies for each subject is believed to be novel.

f) **Accessing knowledge using constructors and navigators**: This approach to a text based language for knowledge exchange, while running counter to the trend towards ‘declarativeness’, helps achieve compactness as well as integration with user interface commands that operate on knowledge.

g) Informal and formal knowledge representation: CODE4 nearly seamlessly allows both formal and informal aspects of knowledge to be managed.

7.2.2 Contributions to user interfaces for knowledge management

Each of the following contributions is relatively small in its own right. However when the contributions are combined, the 'emergent' abilities of the resulting system are very useful to the user. For example a matrix browser can be controlled both by masks and by browsers higher in a chain; which in turn are affected by various selection capabilities, by their own masks and perhaps by the updating of the knowledge base in yet another browser.

h) Rapid, easy browsing of knowledge bases using hierarchies of browsers: Users have found it useful to be able to look at large collection of concepts organized according to some criteria, and then to be able to switch criteria and study the effect of the switch. This 'high-bandwidth browsing' is a feature that other tools for knowledge management appear not to support.

i) Interchangeable mediating representations: Providing similar commands and interaction mechanisms in each mediating representation reduces complexity for the user. In most tools with multiple mediating representations, each presents a different interface to the user.

j) The matrix mediating representation: Matrices to view knowledge are found elsewhere, such as in spreadsheets. New ideas, however, can be found in the details such as the operators available to select which sets of properties or main subjects are shown.

k) Knowledge maps to control user interface displays: By creating an abstraction that defines the kind of knowledge to be displayed in a mediating representation, a significant degree of flexibility is added to the system.

l) The use of masks to control visibility and highlighting: A contribution of this research is the use of masks as a flexible mechanism to allow database-like queries, focussing of the display and highlighting of concepts of interest.

m) The ability to store and recall multiple graph layouts: Users have found that the way they arrange their knowledge is, to them, a kind of latent knowledge. Being able to automatically store this knowledge, in the same format as other knowledge, appears to be a new idea.

n) The ability to display arbitrary relation graphs: While drawing semantic nets is an old idea, this research contributes by showing the benefit of being able to dynamically alter both the set of concepts displayed and the set of relations.

o) The variety of ways of selecting concepts: Sets of concepts to be operated on can be chosen in numerous ways including using masks, using various combinations of keys or using simple regular expressions. Having these mechanisms available greatly facilitates the knowledge management task. All these mechanisms are generally not available in other systems.

p) The variety of editing techniques: Links can be made between concepts by direct manipulation in the various mediating representations, by cutting and pasting or by typing concept names. This flexibility helps in the editing task.

7.2.3 Other general contributions

q) Allowing multiple knowledge bases to be loaded at once: Most knowledge based systems allow a single knowledge base to be loaded, or allow partitioning of a knowledge base into subhierarchies. Allowing several independent knowledge bases to be simultaneously loaded (even several copies of the same knowledge base) allows the user to rapidly switch among various ongoing tasks and to compare and copy portions of knowledge bases.

r) Measuring knowledge bases: Being able to analyse the quality or complexity of a knowledge base is a new idea introduced in this research.

7.2.4 Problems addressed by the research

Section 1.4 listed a series of problems users have while managing knowledge. The following subsections summarize how CODE4 addresses some of these problems. CODE4 contributes most by the solutions it presents to problems I-1 (categorizing), I-2 (naming), I-5 (extracting) and A-2 (the expert user restriction).

Problem I-1: Categorizing

CODE4's three main mediating representations help with this problem. The outline representation allows users to rapidly rearrange concepts in simple indented lists, while the graphical representation allows for a two-dimensional view. The matrix representation helps users to understand existing categorizations and to decide on appropriate distinctions.

The property and statement hierarchies help users with the organization of categorization criteria. The dimensions and disjointness facilities help fine tune categorization decisions.

The fact that commands and operations are similar among mediating representations helps reduce the cognitive overhead associated with performing categorization. Mask facilities assist similarly by helping users identify concepts by content and by helping them narrow the scope of their work.

Problem I-2: Naming things

Three of CODE4's abilities allow users to use natural names for concepts, reducing the need for them to invent artificial ones: a) The handling of synonyms; b) allowing several concepts to have the same name, and c) the ability to accept any character string as a name.

Problem:	I-1 Categ-oriz-ation	I-2 Nam-ing	I-3 Dist-ingu-ishing	I-4 Under-stand-ing	I-5 Ext-racting	I-6 Hand-ling errors	A-1 Special - purpose	A-2 Expert-only	A-3 Large size	A-4 Single user	Overall contrib-ution
KNOWLEDGE REPRESENTATION (section 7.2.1)											
a) Uniform treatment of concepts	*		**	*	*						**8**
b) Separate terms	*	**	**	*				*			**9**
c) Separate metaconcepts	*		**	**				*			**8**
d) Statements as concepts			**								**7**
e) Property and statement hierarchies	**		*	**							**9**
f) Constructors & navig-ators/ Knowledge server		*			*					*	**5**
g) Formal and informal knowledge	*	*	**			*		**			**6**
USER INTERFACE (Section 7.2.2)											
h) Hierarchies of dynamic browsers	*		*	**	**	**		*			**7**
i) Interchangeable mediating representations	**		*	**	**		**	**			**8**
j) Matrix mediating representations	**	*	**	**	**	**		*			**8**
k) Knowledge maps	*			*	**		**		*		**9**
l) Masks	*	*	*	*	**	**	**	*	*		**8**
m) Graph layouts as knowledge	*	*	**					*			**8**
n) Arbitrary relation graphs			*	**	*		**				**7**
o) Multiple ways of selecting concepts	*			*	**		*	*			**5**
p) Variety of editing techniques	*	*	*				*	*			**5**
OTHER CONTRIBUTIONS (section 7.2.3)											
q) Multiple loaded KBs				*					*	*	**8**
r) Metrics						*			*	*	**8**
Overall contribution of the thesis	**8**	**8**	**7**	**6**	**7**	**5**	**8**	**6**	**3**	**1**	

Figure 7.1: How contributions of this thesis solve knowledge management problems. Rows represent the contributions discussed in section 7.2. Columns represent problems presented in section 1.4. Two asterisks indicate a significant contribution to the problem's solution. One asterisk indicates a weaker contribution. The rightmost column places the contributions on a scale of 0 to 10, where 10 is most significant. Factors determining significance are both novelty and impact. The bottom row uses a 0 to 10 scale to indicate the the relative contribution of CODE4 to the problem's solution, where 0 is no contribution and 10 is very strong. All entries are the authors' opinion.

The fact that CODE4 does not use the names of concepts as internal identifiers allows users to freely change a name if they find a better one, and not to worry about getting the name right in the first place. Several other features allow CODE4 users not to depend on accurate naming in the initial stages of knowledge base developing: a) The fact that concepts are automatically given names that are reasonably sensible by looking at the names of their superconcepts; b) the fact that concepts are almost always entered in an outline or graphical structure so that the user can identify them by their visual placement, and c) the fact that concepts can be readily identified by content, using the highlighting mask.

Yet another way in which CODE4 helps in naming is its treatment of terms as full-fledged concepts. This allows users to focus on linguistic issues at a very detailed level.

Problem I-3: Making distinctions

CODE4 allows properties to be attached to any concept, and includes terms, metaconcepts and statements among the things it treats as concepts. This means that the user can easily record fine details in knowledge, and is not forced to commingle properties of what should be different concepts (e.g. to mix regular inheriting properties with noninheriting metaconcept properties). The fact that graph layouts and informal values are stored in the same manner as other knowledge also helps in this regard.

The matrix mediating representation also helps in making distinctions by highlighting cases where inadequate distinctions have been made.

Problem I-4: Understanding effects

Attaching non-inheriting properties to metaconcepts, and having a strict rule for inheritance, simplify the user’s understanding the effects of inheritance. The ability to display large amounts of knowledge in various windows and in various formats helps users visualize the effects of their actions. Due to automatic updating of windows, the effects of changes are always visible.

Problem I-5: Extracting knowledge

The mask, chained browsers and the matrix mediating representation are the main facilities by which CODE4 helps users extract knowledge. CODE4 emphasizes the presentation of knowledge so that the user can easily extract facts from what is displayed; rather than requiring the user to specify a query in some artificial language.

Problem I-6: Handling errors

Mask facilities help users detect incompleteness and some inconsistencies. The matrix mediating representation, and the ability to display multiple windows containing related knowledge, help the user visually scan for inconsistencies. Various metrics discussed in chapter 5 can help users identify excessive complexity or incompleteness in knowledge bases.

Features such as informal values and value combination allow the user to temporarily live with inconsistencies until appropriate resolutions can be found. Thus the user can follow an incremental approach to knowledge base development.

Problem A-1: Special purpose restriction

CODE4 is not restricted to any particular domain; in a wide variety of domains there is the need to represent concepts and their properties and to carefully specify arbitrary relations.

Problem A-2: The expert user restriction

A number of features are very usable by novices. Foremost of these are the simple-to-use mediating representations wherein the user can sketch hierarchies and arbitrary graphs with no need to understand details of the knowledge representation. A second key feature is the ability to enter informal text in any facet value, resulting in the ability of the user to set their own standards and not be tied down to a sophisticated syntax.

Problem A-3: The small knowledge base restriction

CODE4 has been used to create medium-sized serious knowledge bases. The fact that CODE4's user interface allows the user to focus on a particular aspect of a knowledge base helps in this regard.

Problem A-4: The single-user restriction

Although CODE4 remains largely a single-user environment when it comes to editing, the knowledge server capability and the world-wide-web interface allow large numbers of users to simultaneously query a knowledge base.

CODE4's ability to manage multiple knowledge bases at once, and its ability to split and merge knowledge bases have helped in team-oriented development.

7.3 Future Research

This section summarizes ideas for potential future work. This list does not include minor improvements or cosmetic changes that are in the 'to do' list for CODE4. Also omitted are various planned internal design changes (e.g. to improve maintainability of the system).

7.3.1 Future work on user interfaces for knowledge management systems.

- **Improve feedback panel facilities**: The current feedback panel (section 4.4.2) provides rudimentary information when a user executes a command. The following are some necessary enhancements:
  a) A multi-level undo capability. This would require defining an inverse operation for every knowledge base editing operation.
  b) Active suggestions when certain changes are made. For example when a user adds a property where there already exists a property with that name, a non-modal warning should be displayed. This could be expanded into a general 'property manager' capability.
  c) More useful suggestions when a user performs an 'illegal' action. Ideally the user should be given a non-modal list of possible alternative actions that might be taken – now this only occurs for a few commands. Generally, the problem is described to the user in terse wording that can be hard to understand; and the user must infer the corrective action to take before completing the command.

- **Improve facilities for users to manipulate the look of concepts and the layout of mediating representations**: Currently users can specify the layout of a graph and can set a number of 'format' parameters in any mediating representation. The graph layout is the only aspect of the 'look' that is actually stored as knowledge (i.e as the values of a certain metaconcept property). Also, the format parameters apply to an entire window. What users need are: 1) to be able to manipulate the fonts, colours, box shapes etc. of individual *concepts*; and 2) to be able to treat this information as knowledge.

- **Make the outline and graphical mediating representations have more of the features of outline processors**: Abilities such as dragging and dropping of subhierarchies to reparent them would be very useful; some gesture would have to be found to distinguish this from mere visual reordering.

- **Improve graph layout algorithms**: The interactions among multiple graph structures, masking and dynamic updating cause a high level of complexity in the automatic layout algorithms. These algorithms need analysing and improving.

- **Enhance the matrix mediating representation so users can actively add concepts and properties**: Currently only cells can be edited.

- **Allow users to load and save various configurations of windows and masks**: Users need to be able to save and restore various screen layouts, and other session parameters. They also need to be able to maintain a library of masks.

7.3.2 Future work in knowledge representation

- **Make a distinction between words and terms**: Currently there is a one-to-one mapping between character strings used to name concepts and term concepts. Every new string entered as a name results in the creation of a new term (although terms can stand for multiple other concepts). However it becomes difficult to attach such properties as ***part of speech*** to such concepts because a given character string may be used as several different parts of speech. The solution is to add an extra class of concept called ‘word’. There would be a one-to-one mapping between character-strings and words, a one-to-many mapping between words and terms (i.e. senses of the word), and a many-to-one mapping between terms and other concepts.

- **Implement additional classes of dependent concepts as well as inverse properties, sets and value-dependent subproperties**: These features need to be implemented as described in this thesis.
- **Improve the delegation mechanism**: This mechanism could be made to work similarly to the formula mechanism in spreadsheets. As in spreadsheets, references could be relative (to the current subject and/or property) or absolute (to a particular concept).
- **Improve capabilities to deal with the ClearTalk language**: Provide fuller parsing capabilities for a restricted version of English that would be intermediate between formal concept references and informal text.
- **Provide explicit support for rules and constraints**: Currently this knowledge is entered informally. While most users have not found this a serious hindrance, it does preclude automatic checking mechanisms and executable user rules.
- **Implement checking for type consistency over inheritance**: Currently if a necessary property has one value for a particular subject, it is not constrained to have a narrower value for a subconcept of that subject. While users should continue to be able to informally override such constraints, a checking capability should be added so the user can find inconsistencies on demand.
- **Provide a capability to store knowledge about the knowledge base as a whole**: Currently this can only be done by adding a user concept to represent the knowledge base. A primitive mechanism is needed so that knowledge bases can be indexed.
- **Implement capabilities to track updaters of the knowledge base**: Currently there are primitive properties for this purpose, but they are never used.
- **Generate dimensions automatically**. This can be done by finding statements that are ‘definitional’; i.e. necessary and sufficient ones; those that state the reason why the concept exists at all. A facet could be used to declare a statement definitional. For example, if the the statement about age were declared definitional for the concept **child**, then the dimension ‘by age’ could be generated automatically. A distinction between definitional and assertional statements is fundamental in some KR systems like KM, but it has been found that the types of users interested in CODE4-KR rarely care about precise definitions, they are more interested in *describing* as much about a concept as they can.

- **Divide *subconcepts* into value-dependent subproperties**. Rather than having a property called ***kinds*** whose statements are purely textual, a useful extension would be to consider the various kinds as subproperties of the ***subconcepts*** property. A generalization of the value-dependent substatements mechanism would be needed.

7.3.3 Future work on CODE4 in general

- **Controlled experiments to prove the efficacy of features**: CODE4 has too many features to individually test, however given a large enough pool of participants it would be useful to perform an experiment of the following kind on certain key features: Give a control group the system and have them build a significant number of knowledge bases. Have the test group create knowledge bases using a version of the system that lacks a certain feature. Then compare the time spent, quality of the knowledge bases produced etc. Unfortunately this kind of experiment would require a significant amount of money to properly execute. This is because, to obtain comparable results, participants would have to spend tens of hours and build knowledge bases on the same topics as other participants. Because finding volunteers to do this has proved difficult, participants would probably have to be paid.
- **Enhance knowledge-server capabilities**: The objective would be to permit 'groupware' use of CODE4, wherein several users, using separate CODE4 systems, can dynamically interact with the same knowledge base under the control of a master CODE4 server.
- **Enhance the saving and loading mechanism**: This currently operates on a whole knowledge base at a time. The objective would be to make it incrementally save and load at a much finer grain, even down to the statement level. This would lift the cap on knowledge base size, which currently is the amount of available virtual memory. It would also ease multi-user development of knowledge bases.

7.3.4 Future work involving metrics

The metrics presented in this research are preliminary ideas. The following suggests lines of future research:

- **Further develop the complexity metrics**. A number of additional complexity metrics could be devised, and some of the existing ones could be enhanced:
  a) The complexity of the property hierarchy should be measured in a similar manner to M_{ISA}.
  b) The complexity of graphs computable by following formal links should be measured (e.g. part-of hierarchies, dependency graphs etc.)
  c) The use of dimensions and disjointness should be factored into the computation of M_{ISA}.

d) Additional balance metrics could be developed – for example to measure how complexity differs in different parts of the inheritance hierarchy or property hierarchy.

- **Test users' perceptions of the metrics**: The metrics should be evaluated and adjusted so as to better correspond with subjective phenomena. Participants who are CODE4 experts should be presented with a series of knowledge bases, and should be asked to give estimates for various types of complexity, as well as overall complexity and pure complexity. Next, various formulas and weightings should be adjusted to minimize the difference between the metrics and participants' estimates. After this, a further experiment should be run to verify that the adjusted formulae yield useful measurements (the participants could be divided into a 'training set' and 'test set').
- **Adjust M_{OCPLX} so that it has more meaning**: In a similar manner to the above, the formula for overall complexity could be adjusted so that it becomes a better predictor of time to create a knowledge base. The idea would be to make M_{OCPLX} more analogous to function points.

7.4 Conclusions

Knowledge management systems can be made usable by people who are not trained in computer science. This has been demonstrated by the development of CODE4 which has attracted a supportive and on-going set of users. The following paragraphs discuss major conclusions of this research; these are phrased as recommendations to the developers of future systems – proposals for features which should all be synergistically combined. It must be borne in mind that these conclusions are limited to the context of concept-oriented knowledge management using a tool like CODE4. Such a tool would not be appropriate to manage, for example, repetitive data (as in a database), bitmapped images or expert-system rule bases – i.e. large numbers of instance concepts. However, it might be possible to use a knowledge management system to manage the *types* of these instance concepts.

Future system developers should focus on enhancing the user's ability to overcome intrinsic difficulties such as categorizing and naming concepts. They should also recognize that there is a large need for technology that can permit average people to manage concept-oriented knowledge. Cues should be taken from personal productivity software such as word processors and spreadsheets: The majority of the success of such tools comes from a very few fundamental ideas that help users, rather than from sophisticated formal representations. Most of users' knowledge management work revolves around the manipulation of hierarchies of named concepts and the attachment of properties to those concepts.

A focus on *knowledge organizing* techniques has highlighted many areas where knowledge management technology can be made more practical. Such techniques help with various aspects of the handling of *sets* of concepts. In particular developers of knowledge management systems can better organize knowledge both at the abstract representation level and at the user interface level.

At the abstract representation level, major suggestions for future knowledge management systems are: 1) to treat terms, metaconcepts and statements as distinct classes of concepts; and 2) to organize properties and statements into hierarchies. Both of these suggestions can help users to more easily perform categorization and to make needed distinctions. In particular, it is important to ensure that the names of concepts can be readily changed, that a particular name can be used for several concepts and that concepts can have more than one name.

Another major suggestion is to allow for the integration of formal and informal knowledge. Formal knowledge is knowledge that is interconnected by networks of links, and informal knowledge is typically represented as uninterpreted character strings. Whereas much existing knowledge management technology focuses on managing the formal knowledge, users need equally effective ways to organize informal knowledge. Experience has shown that most knowledge actually handled by users is informal and that users consider their informal knowledge to convey a large fraction of the content of the knowledge base.

At the user interface level, the most important idea proposed is to create a 'high-bandwidth' browsing environment using browser hierarchies, interchangeable mediating representations, knowledge maps and masks. The objective of such an environment is to leverage the human brain to make inferences about knowledge, rather than relying on the machine to make those inferences. Browser hierarchies can allow the user to rapidly navigate knowledge. Graphical and outline mediating representations can allow the user to visualize and easily manipulate knowledge. Matrix mediating representations permit the user to compare concepts and to find inconsistencies. Knowledge maps and masks work together so that the user can extract knowledge that fulfils certain criteria.

Two other things are proposed as important user interface features of a knowledge management system: 1) As is standard in personal productivity software, the user interface should be as non-modal as possible; i.e. the number of prompts that have a limited number of possible responses should be minimized so the user can rapidly switch tasks and remain 'in control' of the interaction. 2) The ways the user embellishes graphical displays of knowledge using particular layouts, colours, box shapes and fonts etc. all should be recognized as valid knowledge and should be stored in knowledge bases as is any other knowledge. The principle here is that a knowledge base should be able to represent its user's thoughts, and many users apparently use spatial and similar cues as part of their representation of conceptual structures.

A further proposal is that future knowledge management systems support a number of ways of measuring knowledge. Possibilities include simple metrics for various types of complexity and compound metrics that might help the user better understand the true size of a knowledge base.

Each of the above suggestion can prove useful in its own right, however the most important conclusion of this research is the following: *That it is only by the synthesis of the suggestions into a well-designed system that users can be provided with a truly practical tool for knowledge management*. When the suggestions are integrated, the resulting whole has many useful emergent properties.

Bibliography

Acker, L. (1992) *Access Methods for Large, Multifunctional Knowledge Bases, TR AI92-183,* PhD Thesis, University of Texas at Austin.

Anjewierden, A. and J. Weilemaker (1992). "Shelley - Computer-Aided Knowledge Engineering." *Knowledge Acquisition* 4: 109-125.

Barman, D. (1992). "Capturing Design Rationale with Semi-structured Hypertext". *AAAI92 Workshop on Design Rationale Capture and Use*, San Jose, 15-22.

Berners-Lee, T., R. Cailliau, A. Luotonen, H. Neilsen and A. Secret (1994). "The World-Wide Web". *CACM* 37(8) Aug 1994: 76-82.

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs NJ: Prentice-Hall.

Bowker, L. (1992) *Guidelines for Handling Multidimensionality in a Terminological Knowledge Base,* Master's thesis, University of Ottawa.

Bowker, L. and T. C. Lethbridge (1994). "Terminology and Faceted Classification: Applications Using CODE4". *Third International ISKO Conference*, Copenhagen, 200-207.

Brachman, R., D. McGuiness, P. Patel-Schneider, L. Resnick and A. Borgida (1991). "Living with CLASSIC: When and How to Use a KL-ONE Like Language" in *Principles of Semantic Networks*. J. Sowa ed. San Mateo: Morgan Kaufmann: 401-456.

Brachman, R. and J. Schmolze (1985). "An Overview of the KL ONE Knowledge Representation Language." *Cognitive Science* 9(2): 171-216.

Bradshaw, J., J. Boose, D. Shema, D. Skuce and T. Lethbridge (1992). "Steps Toward Sharable Ontologies for Design Rationale". *AAAI-92 Design Rationale Capture and Use Workshop*, San Jose, CA.

Bradshaw, J., C. Chapman and K. Sullivan (1992). "An Application of DDucks to Bone-Marrow Transplant Patient Support". *1992 AAAI Spring Symposium on Artificial Intelligence in Medicine*, Stanford University, March.

Bradshaw, J., P. Holm, O. Kipersztok and T. Nguyen (1992). "eQuality: A Knowledge Acquisition Tool for Process Management". *FLAIRS 92*, Fort Lauderdale, Florida.

Bradshaw, J. M. and J. H. Boose (1992) *Mediating Representations for Knowledge Acquisition,* Internal Report, Boeing Computer Services, Seattle, Washington.

Bradshaw, J. M., K. M. Ford, J. R. Adams-Webber and J. H. Boose (1993). "Beyond the Repertory Grid: New Approaches to Constructivist Knowledge Acquisition Tool Development" in.*Knowledge Acquisition as Modeling*. K. M. Ford and J. M. Bradshaw ed. New York: John Wiley. 287-333.

Brooks, F. P. (1987). “No Silver Bullet – Essence and Accidents of Software Engineering.” *IEEE Computer*, 20(4), April, 10-19.

Conklin, J. (1987). “Hypertext: An Introduction and Survey.” *IEEE Computer* (September 1987): 17-41.

Dahlberg, I. (1993). "Faceted Classification and Terminology". *TKE'93, Terminology and Knowledge Engineering*, Frankfurt, Indeks Verlag: 225-234.

Eck, K. (1993) *Bringing Aristotle Into the Twentieth Century: Definition-Oriented Concept Analysis in a Terminological Knowledge Base*, Master's thesis, University of Ottawa.

Eilerts, E. (1994) *KnEd: An Interface for a Frame-Based Knowledge Representation System*, Masters Thesis, University of Texas at Austin.

Eriksson, H., Puerta, A., Musen, M. (1994). "Generation of Knowledge Acquisition Tools from Domain Ontologies". *8th Banff Knowledge Acquisition for Knowledge-based Systems Workshop*, Banff, 7.1-7.20.

Fensel, D. (1993) *The Knowledge Acquisition and Representation Language KARL*, PhD thesis, University of Karlsruhe.

Ford, K. M., J. M. Bradshaw, J. R. Adams-Webber and N. M. Agnew (1993). "Knowledge Acquisition as a Constructive Modeling Activity." in.*Knowledge Acquisition as Modeling*. K. M. Ford and J. M. Bradshaw ed. New York: John Wiley. 9-32.

Gaines, B. (1987). “An Overview of Knowledge Acquisition and Transfer.” *International Journal of Man-Machine Studies*. 26: 453-472.

GE (1993) *OMTool User Guide, version 2.0*, , General Electric Advanced Concepts Center.

Genesereth, M., Fikes, R. (1992) *Knowledge Interchange Format Version 3.0 Reference Manual*, , Computer Science Department, Stanford University.

Ghali, N. (1993) *Managing Software Development Knowledge: A Conceptually Oriented Software Engineering Environment*, MSc Thesis, University of Ottawa, Dept of Computer Science.

Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment*. Reading, Mass: Addison-Wesley.

Gruber, T. (1993). “A Translation Approach to Portable Ontology Specifications.” *Knowledge Acquisition* 5: 199-220.

Halasz, F. and M. Schwartz. “The Dexter Hypertext Reference Model”. *CACM* 37(2) Feb 1994: 30-39.

Harel, D. (1987). “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming* 8: 231-274.

Hunter, E. J. (1988). *Classification Made Simple*. Aldershot: Gower.

IFPUG (1994) *Function Point Counting Practices Manual*, , International Function Points User Group, Release 4.0.

Iisaka, K. (1992) *Knowledge Base of Fundamental Concepts of the Smalltalk-80 Language and Programming Environment*, Undergraduate Senior Report, University of Ottawa.

ISO (1990). *ISO 1087: Terminology Vocabulary*. ISO: Geneva.

ISX (1991) *LOOM Users Guide*, ISX Corporation.

Johnson, N. E. (1989). "Mediating Representations in Knowledge Elicitation" in.*Knowledge Elicitation: Principles, Techniques and Applications*. D. Diaper ed. New York: John Wiley.

Klinker, G., D. Marques and J. McDermott (1993). “The Active Glossary: taking integration seriously.” *Knowledge Acquisition* 5: 173-197.

Lacey, A (1976). *A Dictionary of Philosophy*, Routledge & Kegan Paul: London.

Lenat, D. and R. Guha (1990). *Building Large Knowledge Based Systems*. Reading, MA: Addison Wesley.

Lethbridge, T. C. (1991). "A model for informality in knowledge representation and acquisition.". *Workshop on Informal Computing*,, Santa Cruz, Incremental Systems: 175-177.

Lethbridge, T. C. (1993) *CKB Command Documentation*, Unpublished Report, University of Ottawa AI Lab.

Lethbridge, T. C. and K. Eck (1994) *CODE4 Reference Manual*, Internal Report, AI Lab, University of Ottawa.

Lethbridge, T. C. and D. Skuce (1992a). "Informality in Knowledge Exchange". *AAAI-92 Workshop on Knowledge Representation Aspects of Knowledge Acquistion*, San Jose, CA: 93-99.

Lethbridge, T. C. and D. Skuce (1992b). "Beyond Hypertext: Knowledge Management for the Technical Documenter". *SIGDOC 92*, Ottawa, ACM, 313-322.

Longeart, M., G. Boss and D. Skuce (1993). “Frame-based Representation of Philosophical Systems Using a Knowledge Engineering Tool.” *Computers and the Humanities* 27: 27-41.

Low, G. C. and D. R. Jeffrey (1990). “Function Points in the Estimation and Evaluation of the Software Process.” *IEEE Transactions on Software Engineering* 16: 64-71.

MacGregor, R. (1991a). "The Evolving Technology of Classification-based Knowledge Representation Systems" in *Principles of Semantic Nets*. J. Sowa ed. San Mateo, CA: Morgan Kaufmann. 385-400.

MacGregor, R. (1991b). "Using a description classifier to enhance deductive inference". *7th IEEE Conf. on AI Applications*, 41-45.

McCabe, T. J. (1976). “A complexity measure.” *IEEE Trans. Software Engineering* 2(4): 308-320.

Meyer, I., K. Eck and D. Skuce (1994). "Systematic Concept Analysis Within a Knowledge-Based Approach to Terminology", in *Handbook of Terminology Management,* S. E. Wright and G. Budin eds. Amsterdam/Philadelphia: John Benjamin.

Meyer, I., D. Skuce, L. Bowker and K. Eck (1992). "Towards a New Generation of Terminological Resources: An Experiment in Building a Terminological Knowledge Base". *13th International Conference on Computational Linguistics (COLING).*, Nantes, 956-960.

Miller, D. (1992) *Toward Knowledge-Base Systems for Translators,* Master's thesis, University of Ottawa.

Motta, E., M. Eisenstadt, K. Pitman and M. West (1988). “Support for Knowledge Acquisition in the Knowledge Engineer’s Assistant (KEATS).” *Expert Systems* 5(1): 21-50.

Motta, E., T. Rajan, J. Domingue and M. Eisenstadt (1991). “Methodological foundation of KEATS, the knowledge Engineer's Assistant.” *Knowledge Acquisition* 3: 21-47.

Neale, I. M. (1989). “First generation expert systems: a review of knowledge acquisition methodologies.” *The Knowledge Engineering Review* : 105-145.

Nielsen, J. (1990). “The Art of Navigation Through Hypertext.” *CACM* 33(3): 298-309.

Porter, B. (1994) Personal Commmunication.

Porter, B., J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker and T. Jones (1988) *AI Research in the Context of a Multifunctional Knowledge Base: The Botany Knowledge Base Project*, , The University of Texas at Austin.

Regoczei, S. and S. Hirst (1989) *The Meaning Triangle as a tool for the acquisition of abstract conceptual knowledge*, Technical Report, University of Toronto, Computer Science Dept., TR CSRI-211.

Riedesel, J. D. (1990). "Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems". *The 5th Annual AI Systems in Government Conference*, Washington, D.C., 80-87.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991). *Object-oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall.

Selic, B. and J. McGee (1991). "Object-Oriented Design Concepts for Real-Time Distributed Systems.". *Real-Time and Embedded Systems Workshop, ACM Conference on Object-Oriented Programming, Systems, and Languages (OOPLSA 91),* Phoenix.

Shaw, M. and B. Gaines (1991). "Using Knowledge Acquisition Tools to Support Creative Processes". *proc 6th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop,* Banff, 27.1 - 27.19

Shipman, F. M. (1992) *Supporting Knowledge-base Evolution using Multiple Degrees of Formality*, Technical Report, Dept. of CS, Univ. Colorado at Boulder, CU-CS-592-92

Shipman, F. M. (1993) *Formality Considered Harmful: Experiences, Emerging Themes, and Directions*, Technical Report, Dept. of CS, Univ. Colorado at Boulder, CU-CS-648-93.

Skuce, D. (1977) *Toward communicating qualitative knowledge between scientists and machines*, PhD thesis, McGill.

Skuce, D. (1988). "An English-like Syntax and a Smalltalk Tool for Conceptual Graphs.". *Workshop on Conceptual Graphs*, Minneapolis, p3.1.5.1-3.1.5.10.

Skuce, D. (1991). "A Language and System for Making Definitions of Technical Concepts." *Journal of Systems and Software* 14(1): 39-59.

Skuce, D. (1992a). "Managing Software Design Knowledge: A Tool and an Experiment." *Resubmitted with reviewers changes to: IEEE Transactions on Knowledge and Data Engineering* .

Skuce, D. (1992b) *Notes on ClearTalk*, Internal Report, AI Laboratory, University of Ottawa.

Skuce, D. (1993a). "A Multifunctional Knowledge Management System." *Knowledge Acquisition* 5: 305-346.

Skuce, D. (1993b). "A Review of "Building Large Knowledge Based Systems" by D. Lenat and R. Guha." *Artificial Intelligence* 61: 81-94.

Skuce, D. (1993c). "A System for Managing Knowledge and Terminology for Technical Documentation". *Third International Congress on Terminology and Knowledge Engineering*, Cologne, 428-441.

Skuce, D. (1993d). "Your thing is not the same as my thing: reaching agreeement on shared ontologies". *proc. Formal Ontology in Conceptual Analysis and Knowledge Representation*, Padova.

Skuce, D. and I. Monarch (1990). "Ontological Issues in Knowledge Base Design: Some Problems and Suggestions.". *5th Knowledge Acquisition for Knowledge Based Systems Workshop*, Banff.

Skuce, D., S. Wang and Y. Beauvillé (1989). "A Generic Knowledge Acquisition Environment for Conceptual and Ontological Analysis.". *4th Knowledge Acquisition for Knowledge Based Systems Workshop*, Banff, 31.1-31.20.

Sommerville, I. (1992). *Software Engineering, 4th Ed*. Wokingham: Addison-Wesley.

Sowa, J. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison Wesley.

Spivey, J. M. (1989). "An Introduction to Z and Formal Specifications." *Software Engineering Journal* : 40-50.

Terveen, L. (1990) *A Collaborative Interface for Editing Large Knowledge Bases*, Technical Report, MCC, ACT-HI-187-90.

Terveen, L. and D. Wroblewski (1992). “A Tool for Achieving Concensus in Knowledge Representation.” *AAAI-92* : 74-79.

Vickery, B. C. (1960). *Faceted Classification*. London: Aslib.

Wang, C. (1994) *Towards Conceptually-Oriented Software Requirements Analysis and Design*, MSc Thesis, University of Ottawa.

Weilinga, B. J., A. T. Schreiber and J. A. Breuker (1992). “KADS: A modelling approach to knowledge engineering.” *Knowledge Acquisition* 4(1).

Appendix A

Data About the Users

This appendix contains summaries of data concerning users who participated in the CODE4 user study. The methodology for the study is discussed in section 6.1.

The data comes primarily from questionnaires, however many of the questions are not included in this appendix. The following are reasons for omissions:

- The answers to some questions yielded no statistically significant results because:

    a) Some questions were answered by an insufficient number of respondents.

    b) For some questions involving rating several attributes, the attributes were rated too similarly to draw conclusions about any differences.

- It was discovered that some questions were misunderstood.

A.1 General questions about experiences with CODE4.

			Average	Max	Min	Std. Dev	Number of responses
A2	Approximately how many hours in total do you think you have spent using CODE4?		**271**	1000	40	296	11
A4	Estimate what percentage of the system functionality you use regularly when using CODE4 (think in terms of the percentage of menu items that you use)?		**38**	65	20	17	11
B1	What percentage of your current understanding of CODE4 was obtained from:						
	h	Exploring the system on your own	**25**	65	0	22	10
	a	Studying the reference manual	**18**	50	0	17	10
	c	Studying scientific papers	**15**	33	0	13	10
	f	Being taught individually in front of the system	**14**	60	0	17	10
	i	Asking questions to CODE4 developers	**8**	20	0	6	10
	g	Watching others use the system	**7**	40	0	12	10
	j	Asking questions to CODE4 users	**5**	11	0	5	10
	b	Studying a beginner's tutorial	**4**	11	0	5	10
B5	How many days of training do you think it would take an average university graduate with basic computer literacy, but no artificial intelligence knowledge, to become an intermediate user of CODE4?		**7**	20	2	5	11

A.2 Questions about CODE4 knowledge representation features

			Average	Max	Min	Std. Dev	Number of responses *
D1	In this question, please rank your perceptions of various knowledge representation features. How useful have the features been to you in the tasks you have been performing? (-5 = very harmful …0 = of no use … 5 = essential)						
	A	Inheritance, in general	**4.9**	5	4	0.3	10
	B	Multiple inheritance	**4.9**	5	4	0.3	10
	C	The property hierarchy, in general	**4.8**	5	4	0.4	9
	D	Multiple parents in the property hierarchy	**4.1**	5	0	1.7	8
	E	The ability to have formal references to other concepts in any facet value (either by parsing or cut-and-paste)	**3.9**	5	1	1.4	10
	F	Facets, in general	**3.9**	5	2	1.2	9
	G	The ability to have informal text in any facet value	**3.6**	5	-1	2.0	10
	H	The modality facet	**3.6**	5	0	1.7	9
	I	Metaconcepts	**3.5**	5	2	1.2	6
	J	The ability to add new facets	**3.4**	5	1	1.4	9
	K	The ability to have more than one term for a concept (synonyms)	**3.2**	5	0	2.0	10
	L	The ability to use almost any character string in a term (the name of a property or main concept)	**3.2**	5	0	1.5	10
	M	Treating terms, properties and statements as full-fledged concepts (with their own properties etc.)	**3.1**	5	0	1.8	7
	N	Automatic combination of conflicting values in multiple inheritance	**2.8**	5	-1	2.2	9
	O	Delegation	**2.7**	5	0	1.8	6
	P	The set of built-in facets	**2.6**	4	1	1.0	7
	Q	Instance concepts	**1.9**	5	0	2.4	8
	R	Terms as separate concepts from the concept(s) they designate	**1.9**	5	0	2.0	7
	S	The maintenance of information about which concepts are disjoint	**1.5**	5	0	2.0	6
	T	Treating facets as properties	**1.4**	5	-2	2.2	8
	U	The fact that you need not explicitly name a concept (automatic default naming)	**0.2**	5	-3	2.2	9

* The numbers of responses to the questions differ because some people did not actually understand the terms used in some questions. For example, only six people knew what the term ‘metaconcept’ meant. This does not mean, however, that questions containing such

non-understood terms should be counted as ‘of no use’. In fact, almost all users implicitly used metaconcepts when they specified non-inheriting properties – they were just never exposed to the term.

The following indicates which knowledge representation features are significantly more useful than others (calculated using t-tests). The letters on the rows and columns correspond to the letters next to the features listed on the previous page. The lack of a check mark in a cell indicates that no conclusion can be drawn as to whether the row feature is more important than the column feature.

		U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C
Inheritance, in general	A	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
Multiple inheritance	B	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
Property hierarchy	C	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√		
Multiple prop. parents	D	√	√	√	√		√													
Formal references	E	√	√	√	√	√	√													
Facets, in general	F	√	√	√	√	√	√	√												
Informal text in value	G	√	√																	
The modality facet	H	√	√	√																
Metaconcepts	I	√		√			√													
Ability to add new facets	J	√	√	√			√													
Synonyms	K	√																		
Any character string	L	√	√	√																
Full-fledged concepts	M	√																		
Automatic combination	N	√																		
Delegation	O																			
The set of built-in facets	P	√																		

A.3 Questions about CODE4 user interface features

			Average	Max	Min	Std. Dev	Number of responses
C1		In this question, please rank your perceptions of various user interface features. How useful have the features been to you in the tasks you have been performing? (-5 = very harmful …0 = of no use … 5 = essential)					
	A	The outline mediating representation, in general	**4.8**	5	4	0.4	10
	B	Dynamic updating of subwindows	**4.7**	5	4	0.5	11
	C	Direct typing to change a concept name in the outline and graphical mediating representations	**4.6**	5	3	0.8	11
	D	The graphical mediating representation, in general	**4.6**	5	4	0.5	11
	E	The ability to save various graph layouts	**4.5**	5	3	0.8	8
	F	The control panel, in general	**4.5**	5	3	0.9	8
	G	The ability to make multiple selections on the graphical and outline mediating representations	**4.5**	5	3	0.7	11
	H	Relation subwindows (e.g. showing part-of or other arbitrary relations of combinations of relations)	**4.3**	5	2	1.1	9
	I	The matrix mediating representation, in general	**4.3**	5	3	1.0	10
	J	Different mediating representations (outline, graphical, matrix) with commands that work in a similar manner in each	**4.3**	5	2	1.1	10
	K	The variety of ways of making or extending a selection (dragging, marquee, shift key, control key)	**4.2**	5	2	1.1	11
	L	The mask, in general	**4.2**	5	2	1.1	11
	M	The full selection-criteria capability (The ability to highlight concepts matching a pattern using mask-like predicates)	**4.2**	5	2	1.1	11
	N	The fast goto capability (Typing the name of a concept following a '>' sign to highlight it)	**4.2**	5	2	1.1	11
	O	Non-modality of most of the interface (the fact that you are rarely prompted for anything when executing a command)	**4.2**	5	2	1.1	11
	P	Splitting and merging of knowledge bases	**4.2**	5	2	1.1	11
	Q	The display of statistics about knowledge bases	**4.2**	5	2	1.1	11
	R	The ability to easily request or add predicates for the mask	**4.2**	5	2	1.1	11
	S	The selection of predicates available in the mask	**4.2**	5	2	1.1	11

		Average	Max	Min	Std. Dev	Number of responses
T	The ability to open a subwindow from any selected concept or set of concepts	**4.2**	5	2	1.1	11
U	The ability to design your own browser (containing various subwindows of different sizes)	**4.2**	5	2	1.1	11
V	Tailorable hotkeys	**4.2**	5	2	1.1	11
W	The ability to label isa links (dimensions)	**4.2**	5	2	1.1	11
X	Multiple ways of adding links: Typing or pasting into a facet value, using add sibling/add child commands, etc	**4.2**	5	2	1.1	11
Y	The ability to focus a hierarchy around one or several concepts (selecting the button marked 'honly')	**4.1**	5	2	1.4	8
Z	The format options available in the matrix mediating representation	**4.1**	5	2	1.3	9
A'	The ability to reorder subconcepts and subproperties in the outline mediating representation. (the rotate-up and rotate down commands)	**4.0**	5	2	1.2	10
B'	The format options available in the graphical mediating representation	**3.9**	5	0	1.7	9
C'	In general, the 'high-bandwidth browsing' nature of the user interface where a large amount of knowledge is displayed and is editable at once (as opposed to results from individual queries)	**3.9**	5	0	1.7	9
D'	The options available in the control panel (e.g. the format options)	**3.4**	5	0	1.9	10
E'	Unlimited chains of driving and driven subwindows	**3.4**	5	0	1.6	11
F'	The ability to arrange an outline both alphabetically and hierarchically	**2.6**	5	0	2.0	10
G'	The ability to have multiple knowledge bases loaded at once	**2.6**	5	-5	3.5	9
H'	The ability to show placeholders for hidden concepts (+/-)	**2.1**	5	0	2.0	8
I'	The ability to attach a graphical icon to a node in the graphical mediating representation	**1.2**	4	0	1.8	5

The following indicates which of the above features are significantly more useful than others (calculated using t-tests). The lack of a check mark indicates that no conclusion can be drawn as to whether the row feature is more significant than the column feature.

		I’	H’	G’	F’	E’	D’	C’	B’	A’	Z	Y	X	W	V	U
The outline med. rep., in general	A	√	√		√	√	√	√	√	√	√	√	√	√	√	√
Dynamic updating of subwindows	B	√	√		√	√	√	√	√	√	√	√	√	√	√	√
Direct typing to change a name	C	√	√		√	√	√	√	√	√	√		√	√	√	√
Graphical med. rep., in general	D	√	√		√	√	√	√	√	√	√	√	√	√	√	√
Saving graph layouts	E	√	√		√	√				√						
Control panel, in general	F	√	√		√	√										
Multiple selections	G	√	√		√	√	√			√						
Relation subwindows	H	√	√		√	√										
Matrix med. rep., in general	I	√	√		√	√										
Different mediating representations	J	√	√		√	√										
The variety of ways of selecting	K	√	√		√	√										
Mask, in general	L	√	√		√	√										
Selection-criteria capability	M	√	√		√	√										
Fast goto capability	N	√	√		√	√										
Non-modality of the interface	O	√	√		√	√										
Splitting/merging KB’s	P	√	√		√	√										
KB statistics display	Q	√	√		√	√										
Ability to add mask predicates	R	√	√		√	√										
Predicates available in the mask	S	√	√		√	√										
Open subwindow / any concept	T	√	√		√	√										
The ability to design browser	U	√	√		√	√										
Tailorable hotkeys	V	√	√		√	√										
Dimensions	W	√	√		√	√										
Multiple ways of adding links	X	√	√		√	√										
Ability to focus a hierarchy	Y	√	√		√											
Matrix format options	Z	√	√		√											
Ability to reorder subconcepts etc.	A’	√	√		√											
Graphical format options	B’	√														
High-bandwidth browsing	C’	√														
Options in the control panel	D’															
Unlimited chains subwindows	E’	√														

Below is the the continuation of the top four rows of the table; the other cells were blank

		T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D
The outline med. rep., in general	A	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
Dynamic updating of subwindows	B	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	
Direct typing to change a name	C	√	√	√	√	√	√	√	√	√	√		√					
Graphical med. rep., in general	D	√	√	√	√	√	√	√	√	√	√	√	√		√			

Appendix B

Summary of Data Gathered about Knowledge Bases

This appendix contains statistics about each of the knowledge bases whose construction was studied as part of this research. More information can be found in chapter 6.

B.1 General data about the knowledge bases studied

Summary data for this appendix can be found in figure 6.10

KB	Description	*	Est dev. hours **	Est NL Pages ***	Types	User insts.	User props.	Stmts.	Terms	Meta-conc.
	Computer technology									
C1	Computer terms	B8	10	15	45	0	71	112	145	0
C2	Operating systems	B7	8	3	34	0	98	117	134	0
C3	Unix	A8	60	27	156	0	71	100	285	1
C4	Computer languages	B8	10	10	31	11	43	44	108	0
C5	Smalltalk in general	A9			112	51	73	218	272	1
C6	Smalltalk collections	A1	20		61	0	138	76	216	0
C7	Smalltalk sets and dictionaries	A2	200	7	66	15	365	420	541	15
C8	MVC	B8	100	35	95	55	135	492	269	0
C9	ML	A1	20		89	6	61	115	191	0
C10	Pascal	A2	70	5	132	0	207	363	339	0
C11	C Language	A7	100		123	31	141	484	264	0
	Mean for group		*60*	*14*	*86*	*15*	*128*	*231*	*251*	*2*
	Other technical topics									
T1	Optical storage	A3	400		288	0	228	1614	580	80
T2	Lasers	A3	50	25	79	0	135	853	218	51
T3	Electrical devices 1	A4	25	5	55	0	83	117	175	0
T4	Electrical devices 2 ****	A7	15							
T5	Electrical devices 3	A5	15	15	90	3	24	179	156	0
T6	Automatic teller machine	A2	300	20	148	8	324	321	519	2
T7	Telephone	A4			171	2	342	508	739	25
T8	Geology - Bedrock field	A5	70	100	159	0	79	691	297	0
T9	Matrices	A6	50	40	124	0	69	172	247	4
	Mean for group		*116*	*34*	*139*	*2*	*161*	*557*	*366*	*20*

KB	Description	*	Est dev. hours **	Est NL Pages ***	Types	User insts.	User props.	Stmts.	Terms	Meta-conc.
	General purpose knowledge									
G1	Top level ontology	A1	500		61	0	66	98	182	30
G2	Vehicles	A4	25	3	40	0	20	35	94	0
G3	Person 1	A4	15	3	53	0	41	34	129	0
G4	Person 2	B8	10	5	31	0	27	174	93	0
G5	Fruit	B6	20		68	0	100	62	169	0
	Mean for group		*114*	*4*	*51*	*0*	*51*	*81*	*133*	*6*

* The codes in this column indicate which user created the knowledge base. Additional information about the users can be found in appendix A.

** This is the number of hours that the user estimated taking to create the knowledge base.

*** This is the number of pages of natural language text that the user estimated it would require to describe the same knowledge as is in the knowledge base.

**** The user filled out a questionnaire, but did not make this knowledge base available.

B.2 Measurements of the knowledge bases

For each of the knowledge bases listed in appendix B1, this section lists the twelve metrics discussed in chapter 5. Summary data can be found in figure 6.1. T4 is missing because the user filled out a questionnaire but did not make the knowledge base available.

KB	M_{allc}	M_{msubj}	M_{rprop}	M_{det}	M_{sform}	M_{div}	M_{sok}	M_{isa}	M_{mi}	M_{acplt}	M_{pcplx}	M_{ocplx}
Computer technology												
C1	409	35	0.45	0.30	0.08	0.82	0.00	0.30	0.23	0.31	0.27	9.5
C2	419	24	0.65	0.21	0.00	0.00	0.00	0.32	0.67	0.39	0.30	7.3
C3	647	146	0.11	0.03	0.04	0.74	0.01	0.54	0.00	0.05	0.05	7.0
C4	273	32	0.33	0.08	0.00	0.81	0.00	0.19	0.56	0.17	0.17	5.3
C5	761	151	0.11	0.09	0.44	0.79	0.05	0.39	0.01	0.06	0.05	7.9
C6	526	51	0.53	0.29	0.00	0.09	0.00	0.32	0.12	0.35	0.19	9.9
C7	1457	71	0.70	0.13	0.01	0.99	0.11	0.33	0.00	0.39	0.38	26.6
C8	1045	87	0.37	0.23	0.67	0.96	0.01	0.21	0.29	0.28	0.25	22.1
C9	497	85	0.17	0.13	0.00	0.87	0.04	0.44	0.06	0.10	0.09	7.8
C10	1075	120	0.40	0.12	0.07	0.93	0.02	0.48	0.08	0.22	0.22	26.4
C11	1078	134	0.26	0.46	0.20	0.88	0.01	0.42	0.18	0.22	0.22	29.0
Avg	*744*	*85*	*0.37*	*0.19*	*0.14*	*0.72*	*0.02*	*0.36*	*0.20*	*0.23*	*0.20*	*14.4*
Other technical topics												
T1	2825	278	0.20	0.13	0.22	0.83	0.14	0.59	0.01	0.12	0.13	36.5
T2	1371	69	0.44	0.15	0.00	0.23	0.37	0.49	0.01	0.25	0.25	17.6
T3	465	45	0.42	0.13	0.09	0.91	0.01	0.38	0.42	0.24	0.25	11.4
T5	486	80	0.05	0.20	0.00	0.59	0.00	0.41	0.01	0.03	0.02	1.9
T6	1357	143	0.48	0.14	0.20	0.99	0.02	0.52	0.07	0.28	0.29	42.1
T7	1822	162	0.46	0.16	0.45	0.97	0.12	0.54	0.14	0.29	0.35	56.1
T8	1260	149	0.12	0.10	0.17	0.47	0.05	0.54	0.02	0.07	0.06	8.2
T9	650	114	0.14	0.08	0.32	0.79	0.02	0.39	0.19	0.08	0.07	8.3
Avg	*1280*	*130*	*0.29*	*0.14*	*0.18*	*0.72*	*0.09*	*0.48*	*0.11*	*0.17*	*0.18*	*22.8*
General purpose knowledge												
G1	473	49	0.33	0.08	0.15	0.80	0.41	0.29	0.22	0.17	0.22	10.8
G2	224	30	0.16	0.11	0.03	0.60	0.00	0.30	0.87	0.09	0.10	2.9
G3	292	43	0.24	0.04	0.48	0.61	0.03	0.39	0.00	0.12	0.09	4.0
G4	361	21	0.32	0.69	0.00	0.72	0.00	0.28	0.29	0.29	0.25	5.2
G5	434	58	0.40	0.13	0.10	0.71	0.00	0.45	0.00	0.23	0.19	10.8
Avg	*357*	*40*	*0.29*	*0.21*	*0.15*	*0.69*	*0.09*	*0.34*	*0.28*	*0.18*	*0.17*	*6.8*

Appendix C

CODE4 Design Principles

This appendix summarizes some of the main design principles that have guided the development of CODE4. Most of these are derived from the primary goal: to make knowledge acquisition practical for a wide variety of people.

C.1 Assumptions about the user and environment

- The user need not be a computer specialist.
- Few users are interested in representing knowledge with mathematical precision.
- A typical desktop-scale environment should suffice.
- The end user should not require support staff to install or use the system.

C.2 General principles

- Flexibility is essential.
- The user should be able to learn a few simple principles and combine them easily to do more complex things.
- Special cases should be minimized.
- Where a feature is available in one context, it should be available in all other analogous contexts.
- Features should be designed recursively (e.g. concepts of concepts; subwindows of subwindows).
- The system should be optimized for the most popular and powerful facilities.
- The system should be open to the addition of new features by developers and by users.
- The system should be easily divisible so higher level subsystems can be removed with out impacting the core of the system.
- The system should not be biased towards applications in any particular domain.

C.3 User interface principles

- A well-designed user interface is essential.
- The inexperienced user should be able to easily enter knowledge.
- The user should be able to manage knowledge rapidly
- The user should be able to jump between tasks at any time.
- Look and feel should be consistent throughout the system.
- The user should be provided with informative feedback about the results of all operations.
- The interface should be tailorable wherever possible.
- It should be possible to select, edit and open subwindows about any knowledge displayed in the user interface.
- It should be possible to operate on multiple elements of any kind of set at once.
- There should be support for terminology choice and disambiguation.

C.4 Knowledge representation principles

- All units of knowledge should be treated in a uniform manner.
- The user should be able to represent all necessary distinctions.
- Artificial structures such as flags or tags should be avoided where possible.
- There should be a suggested restricted syntax for user entered knowledge.
- The user should be able to violate the suggested syntax in order to enter informal knowledge.
- There should be an intended semantics for all elements of the knowledge representation.
- The system should behave consistently even if the user ignores the intended semantics of the knowledge representation.
- The representation's ontological commitment should be minimal.

C.5 Inferencing principles

- The system should not make inferences automatically if there is a penalty in performance or expressiveness.
- Inference mechanisms should involve very simple rules of operation.
- Any automatic inferencing should be largely understandable to non-computer people.
- The experienced user should be able to define new kinds of inferencing.

Appendix D

CKB Format for a Sample Knowledge Base

The following, in CKB-format, is the knowledge base used to create the figures in chapter 4 and some of the figures in chapter 3. Details of CKB syntax, including a full BNF grammar and descriptions of primitives, can be found in (Lethbridge 1993). However, the following is a general guide:

- The first line (split into two in the following for compactness) identifies the knowledge base.
- Each subsequent line contains a constructor (see section 3.13). The type of constructor is identified by the first character of the line.
- Constructors take arguments that are either
    - Concept references (base 36 numbers with modifiers such as ‘m’ or ‘s’ to indicate dependent concepts).
    - Informal strings.
- The following are the main kinds of constructor:
    - z: Identifies a primitive type, and specifies its relationships to other concepts.
    - y: Identifies a non-primitive type, and specifies its relationships to other concepts.
    - t: Creates a term concept.
    - a: Adds an additional superconcept link
    - q: Identifies a primitive property and specifies its relationships to other concepts.
    - p: Identifies a non-primitive property and specifies its relationships to other concepts.
    - v: Specifies the value of a statement
    - u: Specifies the mapping between a term and a concept.

```
CKB FORMAT CODE 4.1C
  (37),4,thesis2
z1>#thingConcept.
y29>1.
y2B>29.
z5>2B#selfMetaconceptConc
  ept.
y2D>29.
z2>2D#selfTermConcept.
```

```
tJ>2'graph layout
  position'.
t16>2'modality'.
t17>2'dimensions'.
t18>2'superproperties'.
t19>2'all changes'.
t1A>2'metaconcept'.
t1B>2'most general
  subject'.
t1C>2'plural'.
```

```
t1D>2'statement'.
t1E>2'knowledge base'.
t1F>2'properties'.
t1G>2'status'.
t1H>2'last changer'.
t1I>2'value'.
t1J>2'thing'.
t1K>2'related concepts'.
t1L>2'ordinary property'.
t1M>2'disjoint concepts'.
```

```
t1N>2'predicate'.
t1O>2'subject'.
t1P>2'terms'.
t1Q>2'all changers'.
t1R>2'comment'.
t1S>2'source properties'.
t1T>2'meanings'.
t1U>2'part of speech'.
t1V>2'statement comment'.
t1W>2'superconcepts'.
t1X>2'sources of value'.
t1Y>2'instances'.
t1Z>2'last change
  reason'.
t20>2'most general
  subject of predicate'.
t21>2'knowledge
  reference'.
t22>2'subconcepts'.
t23>2'last change'.
t24>2'string'.
t25>2'kinds'.
t26>2'last change time'.
t27>2'term'.
t28>2'subproperties'.
t2A>2'concept'.
t2C>2'metaconcept'.
t2E>2'term'.
t2G>2'property'.
t2I>2'statement'.
t2K>2'facets'.
t2N>2'living thing'.
t2P>2'person'.
t2R>2'dog'.
t2T>2'John'.
t2V>2'Fido'.
t2X>2'owns'.
t30>2'nonliving thing'.
t32>2'car'.
t36>2'body'.
t37>2'chassis'.
t38>2'engine'.
t3B>2'parts'.
t3F>2'Rusty'.
t3I>2'limbs'.
t3K>2'legs'.
t3O>2'arms'.
t3R>2'stance'.
t3W>2'torso'.
t3X>2'head'.
t41>2'tail'.
t47>2'furryness'.
t48>2'hairyness'.
t4E>2'fictional thing'.
t4G>2'nonfictional
  thing'.
t4I>2'unicorn'.
y2F>29.
z3>2F#selfOrdinaryPropert
  yConcept.
y2H>29.
z4>2H#selfStatementConcep
  t.
y2M>1.
y2O>2M.
i2S>2O.
y2Q>2M.
i2U>2Q.
y4H>2M.
y2Z>1.
y31>2Z.
y3E>31.
y33>2Z.
y34>2Z.
y35>2Z.
y39>1.
y4F>1.
a2O:4F.
a2Q:4F.
a4H:39.
q6>3::1#valueProperties.
qI>3:6:5#valueLayout.
qK>3:6:5#valueDimensions.
q14>3:6:2#valuePartOfSpee
  ch.
q15>3:6:2#valuePlural.
p2J>3:6:2H.
qQ>3:2J:4#value.
qR>3:2J:4#valueModality.
qS>3:2J:4#valueStatus.
qT>3:2J:4#valueStatementC
  omment.
qU>3:2J:4#valueKnowledgeR
  eference.
qV>3:2J:4#valuePredicate.
qW>3:2J:4#valueSubject.
qX>3:2J:4#valueMostGenera
  lSubjectOfPredicate.
qY>3:2J:4#valueSourcesOfV
  alue.
q12>3:6:2#valueString.
q13>3:6:2#valueMeanings.
q7>3:6:5#valueRelatedConc
  epts.
q8>3:7:5#valueSubconcepts
  .
q9>3:8:5#valueInstances.
qA>3:8:5#valueKinds.
qB>3:7:5#valueSuperconcep
  ts.
qC>3:7:5#valueDisjointCon
  cepts.
qD>3:7:5#valueSourcePrope
  rties.
qE>3:7:5#valueTerms.
qP>3:6:5#valueComment.
qG>3:6:5#valueAllChangers
  .
qH>3:6:5#valueLastChanger
  .
qL>3:6:5#valueAllChanges.
qM>3:L:5#valueLastChange.
qN>3:M:5#valueLastChangeR
  eason.
qO>3:M:5#valueLastChangeT
  ime.
qF>3:6:5#valueKnowledgeBa
  se.
qZ>3:6:3#valueMostGeneral
  Subject.
q10>3:6:3#valueSuperprope
  rties.
q11>3:6:3#valueSubpropert
  ies.
p2W>3:6:2M.
p3A>3:6:1.
p3H>3:3A:2M.
p3J>3:3H:2M.
p3N>3:3H:2O.
p3U>3:3A:2M.
p3V>3:3A:2M.
p40>3:3A:2Q.
p3Q>3:6:2M.
p44>3:6:2M.
u1:1J.
v1:6{'a set of thing '}.
vm1:I{'ih(77@-
  6:figure42)(-
  58@108:dimensions)'}.
u29:2A.
u2B:2C.
u5:1A.
u2D:2E.
u2:27.
u2F:2G.
u3:1L.
u2H:2I.
u4:1D.
u2M:2N.
v2M:2W{1}.
v2M:3V{'a head'}.
v2M:3U{'a torso'}.
vm2M:I{'ih(21@33:figure42
  )(56@103:dimensions)'}.
vm2M:K{'by aliveness'}.
u2O:2P.
v2O:44{'low'}.
v2O:3J{'2'}.
v2O:3Q{'upright'}.
vs3N/2O:S{'typical'}.
vs3N/2O:U{'common
  knowledge'}.
v2O:3N{'2'}.
vm2O:I{'ih(5@66:figure42)
  (162@56:dimensions)'}.
u2S:2T.
v2S:3N{'1'}.
v2S:2W{2U,3E}.
vm2S:I{'ih(10@99:figure42
  )'}.
u2Q:2R.
v2Q:2W{'nothing'}.
v2Q:44{'high'}.
v2Q:3J{'4'}.
v2Q:3Q{'horizontal'}.
v2Q:40{'a tail'}.
vm2Q:I{'ih(62@66:figure42
  )(166@124:dimensions)'}.
u2U:2V.
v2U:44{'very high'}.
v2U:2W{'a bone'}.
```

```
vm2U:I{'ih(74@100:figure4
  2)'}.
u4H:4I.
vm4H:I{'ih(166@186:dimens
  ions)'}.
u2Z:30.
vm2Z:I{'ih(114@34:figure4
  2)(54@46:dimensions)'}.
vm2Z:K{'by aliveness'}.
u31:32.
v31:3A{35,34,33}.
vm31:I{'ih(123@100:figure
  42)'}.
u3E:3F.
vm3E:I{'ih(97@133:figure4
  2)'}.
u33:38.
vm33:I{'ih(189@142:figure
  42)'}.
u34:37.
vm34:I{'ih(211@106:figure
  42)'}.
u35:36.
vm35:I{'ih(219@68:figure4
  2)'}.
u39:4E.
vm39:I{'ih(56@222:dimensi
  ons)'}.
vm39:K{'by truth'}.
u4F:4G.
vm4F:I{'ih(41@152:dimensi
  ons)'}.
vm4F:K{'by truth'}.
u6:1F.
uI:J.
uK:17.
u14:1U.
u15:1C.
u2J:2K.
uQ:1I.
uR:16.
uS:1G.
uT:1V.
uU:21.
uV:1N.
uW:1O.
uX:20.
uY:1X.
u12:24.
u13:1T.
u7:1K.
u8:22.
u9:1Y.
uA:25.
uB:1W.
uC:1M.
uD:1S.
uE:1P.
uP:1R.
uG:1Q.
uH:1H.
uL:19.
uM:23.
uN:1Z.
uO:26.
uF:1E.
uZ:1B.
u10:18.
u11:28.
u2W:2X.
u3A:3B.
u3H:3I.
u3J:3K.
u3N:3O.
u3U:3W.
u3V:3X.
u40:41.
u3Q:3R.
u44:47.
u44:48.
```

Glossary

The following is a list of all the important terms used in this thesis. For more details about these terms the reader is referred to the following places: 1) The index (so as to find the relevant section in the main body of the thesis); 2) The 'thesis.ckb' CODE4 knowledge base developed to describe the main ideas in the thesis, and 3) Figure 3.5 which shows the inheritance hierarchy of the main categories of concept.

Where wording in this glossary differs from that in the main body of the thesis, both wordings are intended to convey the same meaning. Where a definition is preceded by two terms, separated by a comma, the terms are synonyms. Small capitals within definitions are used to indicate terms defined elsewhere in the glossary.

It is important to note that most users of CODE4 need only know a small fraction of these terms. Terms that should be understood by beginners are double-underlined. Terms that should be understood by intermediate users are single-underlined. Terms that are not underlined at all are only for sophisticated experts.

Unless a term is specified to be a verb (v) or adjective (a), it is a noun.

Terms are also categorized according to the degree to which their use in this thesis is novel, specific or unusual.

- Three asterisks (***) indicate that the term has been invented during this research and has a non-obvious meaning (any other usage in the literature is largely coincidental).
- Two asterisks (**) mean the term has a meaning that differs substantially from conventional usage in artificial intelligence or other fields of computer science. Two asterisks is also used for invented terms whose meaning could probably be closely guessed by an artificial intelligence expert.
- One asterisk (*) means that the term has been used in a slightly specialized way, or that there is significant disagreement in the literature about the meaning of the term.
- No asterisks indicates that the term is not being used in a special way, but is included in the glossary for completeness or to help the reader whose background may not be in artificial intelligence.

Abstract representation schema, abstract schema: (*) An idealized KNOWLEDGE REPRESENTATION SCHEMA describing representation principles, but containing simplifying assumptions or constructs that may not be finitely realizable. An abstract schema neither describes how knowledge is presented nor how it is stored. (Specializations are PHYSICAL REPRESENTATION SCHEMA and MEDIATING REPRESENTATION SCHEMA).

Application program interface, API: A protocol composed of the interfaces to callable procedures, whereby an application program can obtain services (in the form of functions performed or data stored or retrieved) from a low-level layer of software such as a knowledge engine.

Browser: (*) A MEDIATING REPRESENTATION that presents a set of CONCEPTS described by a KNOWLEDGE MAP and allows sets of concepts to be selected and to have operations performed on them.

Browser hierarchy: (**) A set of BROWSERS arranged in a hierarchy, such that the selection of a CONCEPT or set of concepts in an ancestor browser changes the KNOWLEDGE MAP of a descendant browser, and results in the display of a new network of concepts in the descendant browser.

CKB: (***, proper noun) The language used by CODE4 for representing a CODE4-KR KNOWLEDGE BASE (or some of the CONCEPTS in a knowledge base) when operating as a KNOWLEDGE SERVER or working with ASCII files. CKB represents knowledge in the form of MODIFIERS and NAVIGATORS.

CKB-format: (***, a) Describes a file containing an entire CODE4 KNOWLEDGE BASE represented in CKB syntax as a minimal sequence of CONSTRUCTORS sufficient to reconstruct the knowledge base.

CODE4: (***, proper noun): The KNOWLEDGE MANAGEMENT system developed at the University of Ottawa by the author and other collaborators.

CODE4-KR: (***, proper noun) The ABSTRACT KNOWLEDGE REPRESENTATION SCHEMA implemented in CODE4.

Combination: (**) A process that automatically resolves conflicts which arise from the MULTIPLE INHERITANCE of different VALUES by a given STATEMENT, by creating a value consistent with the inherited values. Combination results in a value that is more specific than any of the multiply inherited values. Combination can result in a formal or informal value; if any of the inherited values is informal, then the combined value is informal.

Concept: (*) A representation of a THING or set of things. A discrete unit of knowledge representation to which it is possible to explicitly refer. Something that acts as the locus or possessor of a set of facts about a thing or set of things. (IMMEDIATE SUBCONCEPTS: TYPE, INSTANCE CONCEPT, PRIMITIVE CONCEPT, NON-PRIMITIVE CONCEPT. A more restrictive meaning is often intended in other literature. The words ‘unit’ or ‘frame’ are used for ‘concept’ in other literature).

Constructor: (***) A MODIFIER that is one of a minimal subset of all modifiers that is sufficient to build any KNOWLEDGE BASE.

Defining concept: (***) A CONCEPT on whose existence a DEPENDENT CONCEPT depends. A concept used in the definition of a dependent concept. A dependent concept exists by definition, even if not physically created, if its defining concept(s) exist. (Role term applicable to any concept when discussing a dependent concept).

Delegation: (*) The process whereby the VALUE of a STATEMENT is automatically computed by using a DELEGATION FORMULA. A more generalized mechanism than inheritance, but one which requires individual specification of each case rather than being automatically present.

Delegation formula: (**) An optional component of a VALUE that specifies that the value is to be computed as a function of the values of other STATEMENTS. A delegation formula INHERITS and can refer to values that have further delegation formulae.

Dependent concept: (**) A CONCEPT which is considered to exist by virtue of its relation to certain other concepts (its DEFINING CONCEPTS) and which implicitly comes into existence upon the creation of its defining concepts and is destroyed upon the destruction of its defining concepts (specialization of SYSTEM-MANIPULATED CONCEPT; DISJOINT with TERM).

Dependent instance: (***) A DEPENDENT CONCEPT, one or more of which comes into existence when *any* CONCEPT is created (specialization of DEPENDENT CONCEPT and INSTANCE CONCEPT).

Dependent type: (***) A DEPENDENT concept whose DEFINING CONCEPT is a particular STATEMENT. A dependent type has the value of its defining statement as its superconcept. (specialization of DEPENDENT CONCEPT and TYPE CONCEPT).

Dimension: (***) An identifier that indicates the criterion used to distinguish two or more SUBCONCEPTS from their common SUPERCONCEPT. A dimension is commonly the name of a PROPERTY the VALUES of whose STATEMENTS distinguish among the subconcepts. (See also KNOWLEDGE ORGANIZING TECHNIQUE for a distinct use of this term).

Discontinuity: See REPRESENTATIONAL DISCONTINUITY.

Disjoint: (a, *) Applies to set of CONCEPTS for which a declaration has been made that no two or more of them may have a common SUBCONCEPT.

Extension: The potential set of all the THINGS represented by a CONCEPT.

Facet: (*) A STATEMENT whose SUBJECT is a STATEMENT.

Facet hierarchy: (***) A hierarchy of STATEMENTS where the root of the hierarchy is a STATEMENT whose SUBJECT is not a statement, and where successive descendants (FACETS) each have their immediate ancestor as subject. A facet hierarchy specifies layers of recursive detail about the root statement. (Specialization of KNOWLEDGE ORGANIZATION).

Facet property: (**) A PROPERTY possessed by one or more STATEMENTS.

Focus: (v, **) To apply a VISIBILITY MASK that restricts the display of CONCEPTS in a BROWSER to those that are the descendants (according to the relation specified in the browser's KNOWLEDGE MAP) of a particular set of concepts.

Formal: (a, *) applies to aspects of a KNOWLEDGE BASE where CONCEPTS have explicit relationships to other concepts. Especially applies to VALUES that are composed of VALUE ITEMS, forming explicit relationships between concepts. (Contrasts with INFORMAL).

Hierarchy: see PROPERTY HIERARCHY, STATEMENT HIERARCHY, INHERITANCE HIERARCHY and FACET HIERARCHY.

Highlighting mask: (***) A MASK which results in the highlighting or selection of any CONCEPT for which it evaluates true, and the normal display or deselecting of any concept for which it evaluates false.

Immediate: (a) When applied to SUPERCONCEPT, SUBCONCEPT, SUPERPROPERTY, SUBPROPERTY etc. indicates the next higher or lower in the hierarchy.

Informal: (a, *) applies to aspects of a KNOWLEDGE BASE involving text strings or pictures which cannot be interpreted by a KNOWLEDGE MANAGEMENT system. Especially applies to VALUES composed of text strings. (Contrasts with FORMAL).

Inherit, sense 1 applicable to properties: (v) The possessing of a PROPERTY by a CONCEPT, because the concept is a SUBCONCEPT of the MOST GENERAL SUBJECT of the property.

Inherit, sense 2 applicable to values: (v) The possessing of a VALUE by a STATEMENT, because the statement's PREDICATE is inherited (sense 1) by the statement's SUBJECT, and because the statement does not have a LOCALLY SPECIFIED VALUE. Also applies to FACETS other than the VALUE FACET if those facets are declared to inherit using DELEGATION.

Inheritance: The process or mechanism whereby PROPERTIES (and hence VALUES) are caused to be INHERITED. A specialized and automatic mechanism more general in scope than DELEGATION and which cannot be overridden or altered.

Inheritance hierarchy: A partial order, arranged by the user, of all the CONCEPTS within a KNOWLEDGE BASE, rooted at a single PRIMITIVE TYPE that represents all THINGS. A concept is an ancestor of another if and only if its EXTENSION is a superset of the extension of the other; or equivalently if its INTENSION is a subset of the intension of the other. (Specialization of KNOWLEDGE ORGANIZATION).

Inherited value: (*) A VALUE that is INHERITED (sense 2). (DISJOINT with LOCALLY SPECIFIED VALUE).

Instance concept: (*) A CONCEPT representing an individual THING, i.e. having a single thing as its EXTENSION. A concept which can not have SUBCONCEPTS. (Specialization of CONCEPT; DISJOINT with TYPE).

Intension: The potential set of all the PROPERTIES possessed by a particular CONCEPT. The union of the LOCAL INTENSION of the concept, and the intension of all the concept's SUPERCONCEPTS. The intension is an uncountable set whose existence is posited solely for the purpose of making various formal definitions.

Introduce, sense 1 applicable to values: (v, **) To specify a VALUE of a particular FACET of a STATEMENT that differs from the value that would have been INHERITED, and so that the value can be inherited by the SUBCONCEPTS of the statement's SUBJECT. (*The value is introduced at the statement).*

Introduce, sense 2 applicable to properties: (v, **) To specify a PROPERTY such that a particular CONCEPT is its MOST GENERAL SUBJECT. (*The property is introduced at the concept*).

Knowledge acquisition: The process of gathering knowledge and entering it into a computer so the computer, and humans using it, can put that knowledge to use.

Knowledge base: A collection of interrelated CONCEPTS.

Knowledge management: (**) The processes of creating and maintaining KNOWLEDGE BASES, and making them available for use. The manipulation of those aspects of knowledge involving the categorization, definition and characterization of THINGS and their relationships.

Knowledge map: (**) A software abstraction that specifies a network of interrelated CONCEPTS within a KNOWLEDGE BASE, and provides operations for their manipulation. The network is defined in terms of a set of starting concepts and a relation that is repeatedly applied beginning with the starting concepts.

Knowledge organization: A structure within a knowledge base formed by performing the actions of a particular KNOWLEDGE ORGANIZING TECHNIQUE.

Knowledge organizing technique: (**) A distinct aspect of the process of representing knowledge involving the making of certain types of decisions and the performing of certain types of actions, both of which are distinct from those required by other knowledge organizing techniques.

Knowledge representation schema, **representation schema**: (*) A description of a syntax and semantics for the representation of knowledge.

Knowledge server: A running program containing one or more KNOWLEDGE BASES, to which a remote program can connect via a telecommunications line, and exchange knowledge.

Label: (*) Text or other symbols displayed to identify a CONCEPT in a MEDIATING REPRESENTATION. May be generated or be derived from one of the concept's TERMS.

Local intension: (**) The potential set of all the PROPERTIES whose MOST GENERAL SUBJECT is a particular CONCEPT. The local intension is an uncountable set whose existence is posited solely for the purpose of making various formal definitions. (See also INTENSION).

Locally specified statement, local statement: (***) A STATEMENT having one or more FACETS with LOCALLY SPECIFIED VALUES. (Role term applicable to any statement when discussing its subject).

Locally specified value, local value: (**) A VALUE that is explicitly specified (DISJOINT with INHERITED VALUE).

Main subject: (***) A USER CONCEPT that is the SUBJECT of at least one STATEMENT.

Map: see KNOWLEDGE MAP.

Mask: (**) A logical expression evaluated for each CONCEPT in a KNOWLEDGE MAP as the concept is being prepared for display in a MEDIATING REPRESENTATION. A mask is composed of a set of MASK PREDICATES related by logical operators, and the result of its evaluation is either true or false.

Mask predicate: (***) A software function that takes a CONCEPT as its argument and performs a calculation whose result is true or false.

Mediating representation: A representation of CONCEPTS in a particular KNOWLEDGE BASE that conveys their meaning to a user through a user interface. A particular case of a MEDIATING REPRESENTATION SCHEMA.

Mediating representation schema: (**) A KNOWLEDGE REPRESENTATION SCHEMA that describes the syntax and semantics of a MEDIATING REPRESENTATION. A description of how a portion of an ABSTRACT REPRESENTATION SCHEMA can be mapped onto user interface constructs.

Metaconcept: (**) A CONCEPT representing another concept, which is the metaconcept's DEFINING CONCEPT. A concept possessing facts about another concept. (Specialization of DEPENDENT INSTANCE; DISJOINT with STATEMENT).

Modality facet, modality: (*) A FACET whose VALUE specifies the degree to which the facet's STATEMENT is considered true (e.g. necessarily, typically or optionally). The PREDICATE of every modality facet is a particular PRIMITIVE PROPERTY.

Modifier: (***) A command applied to a KNOWLEDGE BASE that adds or deletes a CONCEPT or VALUE.

Most general subject: (***) A CONCEPT that is the highest concept in the INHERITANCE HIERARCHY to possess a particular PROPERTY. All the subconcepts INHERIT the property. (Role term applicable to any concept when discussing its role with respect to a property).

Multiple inheritance: INHERITANCE of a PROPERTY where the inheriting CONCEPT has more than one SUPERCONCEPT possessing the property. Also applies to the resulting inheritance of VALUES, which results in COMBINATION of values.

Navigator: (***) A command applied to a KNOWLEDGE BASE in order to traverse a network of CONCEPTS.

Non-disjoint: (a, *) Applies to a set of CONCEPTS that share a common SUBCONCEPT and thus have EXTENSIONS with a non-empty intersection.

Non-primitive concept: (**) A CONCEPT that can be deleted from a KNOWLEDGE BASE without preventing any computational processes in a KNOWLEDGE MANAGEMENT system from being able to be run. (Specialization of CONCEPT; DISJOINT with PRIMITIVE CONCEPT).

Ontology: (*) A KNOWLEDGE BASE lacking USER INSTANCES. An ontology describes all those TYPES and PROPERTIES significant to particular agent for its thought or computational processes.

Physical representation: A representation of CONCEPTS in a particular KNOWLEDGE BASE as stored in data structures in the memory of a computer or on mass-storage media. An instantiation of a PHYSICAL REPRESENTATION SCHEMA.

Physical representation schema: (*) A KNOWLEDGE REPRESENTATION SCHEMA that describes the syntax and semantics of a PHYSICAL REPRESENTATION. A description of how an ABSTRACT REPRESENTATION SCHEMA can be mapped into computer data structures.

Potentially non-disjoint: (a, **) Applies to a set of CONCEPTS for which no declaration has been made that any pair is DISJOINT, but for which no pair has a common SUBCONCEPT.

Predicate: (*) A PROPERTY used by a STATEMENT in the description of the statement's SUBJECT. (Role term applicable to any property when discussing its role in a statement).

Primitive concept: (**) A CONCEPT on whose existence in a KNOWLEDGE BASE certain computational processes in a KNOWLEDGE MANAGEMENT system depend. (Specialization of CONCEPT; DISJOINT with NON-PRIMITIVE CONCEPT; IMMEDIATE SUBCONCEPTS: PRIMITIVE TYPE and PRIMITIVE PROPERTY; in some literature 'primitive concept' has a totally different meaning – a concept that is not a defined concept).

Primitive property: (***) A PROPERTY that is also a PRIMITIVE CONCEPT; DISJOINT with PRIMITIVE TYPE and USER PROPERTY.

Primitive type: (***) A TYPE that is also a PRIMITIVE CONCEPT; DISJOINT with PRIMITIVE PROPERTY and USER TYPE.

Property: (*) A CONCEPT representing a relation between concepts (Specialization of INSTANCE CONCEPT; DISJOINT with SYSTEM-MANIPULATED CONCEPT and USER INSTANCE).

Property hierarchy: (**) A partial order, arranged by the user, of all the PROPERTIES within a KNOWLEDGE BASE, rooted at a single PRIMITIVE PROPERTY. A property should be made an ancestor of another if and only if, for any given SUBJECT, STATEMENTS using the two properties as PREDICATE have VALUES where the EXTENSION of the first is a superset of the EXTENSION of the second. (Specialization of KNOWLEDGE ORGANIZATION).

Representation schema: see KNOWLEDGE REPRESENTATION SCHEMA.

Representational discontinuity: (***) An error of knowledge representation whereby a thing is conflated with a representation or type of that thing.

Schema: See KNOWLEDGE REPRESENTATION SCHEMA.

Statement: (***) A CONCEPT representing an assertion of fact about a concept, known as the statement's SUBJECT. The representation of the possession by the subject of a particular PROPERTY called the statement's PREDICATE. The subject and predicate are a statement's DEFINING CONCEPTS. (Specialization of DEPENDENT INSTANCE; DISJOINT with METACONCEPT).

Statement hierarchy: (***) A partial order of all the STATEMENTS whose SUBJECT is a particular CONCEPT. A statement is an ancestor of another if and only if the predicate of the former is a superproperty of the predicate of the latter. (Specialization of KNOWLEDGE ORGANIZATION).

Subconcept: A CONCEPT that is a descendant of another concept in the INHERITANCE HIERARCHY. (Role term applicable to any concept when discussing its ancestors in the inheritance hierarchy).

Subfacet: (***) A FACET whose SUBJECT is another facet. (Role term applicable to any facet when discussing its role in a FACET HIERARCHY).

Subject, sense 1: (*) A CONCEPT that a STATEMENT describes or is about. The SUBJECT and predicate together are the DEFINING CONCEPTS of a statement. (Role term applicable to any concept when discussing its role in a statement).

Subject, sense 2: (*) A CONCEPT that is the subject (in sense 1) of one or more locally specified STATEMENTS. A concept that has something said about it. (Specialization of CONCEPT).

Subproperty: (***) A PROPERTY that is a descendant of another property in the PROPERTY HIERARCHY. (Role term applicable to any property when discussing its ancestors in the property hierarchy).

Substatement: (***) A STATEMENT with the same SUBJECT as another statement, and whose PREDICATE is a SUBPROPERTY of the predicate of the other statement. (Role term applicable to any statement when discussing its ancestors in a STATEMENT HIERARCHY).

Superconcept: A CONCEPT that is an ancestor of another concept in the INHERITANCE HIERARCHY. (Role term applicable to any concept when discussing its descendants in the inheritance hierarchy).

Superfacet: (***) A FACET which is the SUBJECT of another facet. (Role term applicable to any facet when discussing its role in a FACET HIERARCHY).

Superproperty: (***) A PROPERTY that is an ancestor of another property in the PROPERTY HIERARCHY. (Role term applicable to any property when discussing its descendants in the property hierarchy).

Superstatement: (***) A STATEMENT with the same SUBJECT as another statement, and whose PREDICATE is a SUPERPROPERTY of the predicate of the other statement. (Role term applicable to any statement when discussing its descendants in a STATEMENT HIERARCHY).

System-manipulated concept: (**) A CONCEPT created and manipulated by a KNOWLEDGE MANAGEMENT system as a side-effect of the manipulation by a user of certain other concepts. (Specialization of NON-PRIMITIVE CONCEPT; DISJOINT with USER CONCEPT and PROPERTY).

Term: (**) A CONCEPT representing a textual or graphical symbol. (Specialization of SYSTEM-MANIPULATED CONCEPT and INSTANCE CONCEPT; DISJOINT with DEPENDENT CONCEPT).

Thing: Anything real or imagined, concrete or abstract. Actions, states, conditions, entities, facts and relations and concepts are all things. Anything that can be thought of or about is a thing.

Type, type concept: (*) A CONCEPT representing a set of similar THINGS, and which may have SUBCONCEPTS. The EXTENSION of a type is a set of things, which may be uncountable. (Specialization of CONCEPT; DISJOINT with INSTANCE CONCEPT; IMMEDIATE SUBCONCEPTS: USER TYPE and PRIMITIVE TYPE. The word ‘concept’ in some literature has this meaning of ‘type’).

User concept: (**) A CONCEPT whose existence in a KNOWLEDGE BASE is not dependent on the existence of any other concept in that knowledge base, and which is explicitly created in the knowledge base by a user. (Specialization of NON-PRIMITIVE CONCEPT; DISJOINT with SYSTEM-MANIPULATED CONCEPT).

User instance: (***) An INSTANCE CONCEPT that is also a USER CONCEPT. An instance concept explicitly added by a user as a subconcept of one or more USER TYPES.

User type: (***) A TYPE that is also a USER CONCEPT. A type explicitly added by a user in an INHERITANCE HIERARCHY.

Value: (*) A component of a STATEMENT, composed of either a text string or one or more CONCEPTS. A value indicates what the statement’s SUBJECT is related to by way of its PREDICATE. (Role term applicable to any text string or set of concepts when discussing its role in a statement).

Value facet: (***) A FACET whose VALUE is the value of its SUBJECT (a STATEMENT). The sole purpose of a value facet is to allow the attachment of other facets to a value (as statements whose SUBJECT is the value facet). The PREDICATE of every value facet is a particular PRIMITIVE PROPERTY.

Value item: (***) One of the CONCEPTS composing a VALUE.

Value-dependent subproperty: (***) A PROPERTY whose existence depends on a particular VALUE ITEM and on the STATEMENT to which that value item is attached. The value-dependent subproperty's SUPERPROPERTY is the PREDICATE of the value item's statement; the MOST GENERAL SUBJECT is the SUBJECT of the statement where the value item is INTRODUCED; the VALUE is the particular value item. (Specialization of PROPERTY and DEPENDENT CONCEPT).

Value-dependent substatement: (***) A STATEMENT whose SUBJECT is a VALUE-DEPENDENT SUBPROPERTY. (Specialization of STATEMENT).

Visibility mask: (***) A MASK which results in the display of any CONCEPT for which it evaluates true, and the suppression from display of any concept for which it evaluates false.

Index