CODE4: A Unified System for Managing Conceptual Knowledge

Doug Skuce
Timothy C. Lethbridge

Department of Computer Science
University of Ottawa
Ottawa, Canada, K1N 6N5
{doug, tcl}@csi.uottawa.ca

Abstract

CODE4 is a general-purpose *knowledge management system,* intended to assist with the common knowledge processing needs of anyone who desires to analyse, store, or retrieve conceptual knowledge in applications as varied as the specification, design and user documentation of computer systems; the construction of term banks, or the development of ontologies for natural language understanding.

This paper provides an overview of CODE4 as follows: We first describe the general philosophy and rationale of CODE4 and relate it to other systems. Next, we discuss the knowledge representation, specifically designed to meet the needs of flexible, interactive knowledge management. The highly-developed user interface, which we believe to be critical for this type of system, is explained in some detail. We finally describe how CODE4 is being used in a number of applications.

1. Introduction

1.1 Rationale for CODE4

This paper describes the main aspects of an interactive, multi-functional, *knowledge management system (KMS)*, CODE4 (Conceptually Oriented Design/Description Environment, version 4). It is the current version of a series begun in 1987 that was motivated by experiences in the mid-eighties in which we used state-of-the art expert system tools to capture knowledge about software. An earlier version of the system, which had a similar philosophy but was much simpler, was described in (Skuce 1993a); that paper makes good background reading.

These experiences lead to our belief that there is a need for knowledge-based systems which primarily act as *amplifiers of human intelligence* rather than systems designed to run autonomously, and for systems that can assist average computer users to store and communicate knowledge more effectively that they do at present. Hence we seek to develop tools that can help people originate, organize, or define concepts or understand and communicate ideas *at the knowledge level* more easily and accurately than most current tools permit.

A KMS, at least as we shall use the term, is an integrated, *multi-functional* tool in that it can support what we believe to be the main knowledge management or processing activities:

- capturing
- organizing, structuring
- clarifying, understanding
- debugging, editing
- finding
- displaying, formatting
- disseminating, transferring, sharing, etc.

A KMS may be either designed for some particular domain, such as software engineering, or be generic. CODE4 is generic, but it is designed so that special features can be added for particular applications.

Few 'knowledge workers' are yet using any kind of knowledge engineering tool to manage their knowledge: They primarily use familiar tools such as word processors (since most knowledge is expressed only in natural language), spreadsheets, drawing programs and database systems. While such tools can be used for a wide variety of applications, the representations and abstraction mechanisms they provide (text, tables, images, relations etc.) are often insufficient for detailed knowledge representation and do no inferencing or semantic checking. Perhaps worse, the tools are usually not interoperable.

Those who do use knowledge engineering tools often use them mainly for a specific aspect of knowledge management, such as acquisition or rule execution (Boose, Bradshaw et al. 1990), or to represent a very particular type of knowledge, such as if-then rules, problem-solving methods, instances with attributes or repertory grids e.g., (Shaw and Gaines 1991). For specific domains and representations, particular kinds of tools have evolved which handle some of the problems well and ignore others. CASE tools are an example –they tend to be excellent for representing ideas that can be expressed with the diagrams they support, but of minimal use if you want to describe something that is unusual according to the tool's particular "ontology" - its methodology or representation. Large 'integrated' CASE tools provide many facilities, but still require you to follow a particular methodology. Often a collection of uncoordinated tools may be used to capture knowledge.

Hence our research addresses the following main problems with systems for knowledge management:

a) Systems are too narrow, focussing on one type of application, one type of user, one type of knowledge representation, one type of knowledge operation, etc.

b) Systems are too hard to use: they lack flexibility, require too much specialized knowledge and have a long learning curve.

c) Systems require conversion to other formats to obtain all functions; this may be impossible

d) Systems are not widely known or available for people not associated with AI research labs.

CODE4 has a very flexible knowledge representation and includes limited support for certain natural language-related problems. Earlier versions of CODE4 have been described in (Skuce 1989); (Skuce, Wang et al. 1989); (Skuce 1993c); (Skuce and Lethbridge, 1994). CODE4 is programmed in Smalltalk-80 and hence runs without modification on all major platforms.

1.2 Design Assumptions

We now summarize the main assumptions we have made in designing CODE4 and its knowledge representation:

- The user need not be a computer specialist.
- A well-designed user interface is essential; the inexperienced user should be able to do knowledge management rapidly and naturally and should have a choice of modes. The user should be able to work on multiple tasks in parallel and should be able to jump between tasks at any time.
- A typical desktop-scale machine should suffice, without the need for AI support staff.
- There should be optional support for language-related problems, mainly relating to terminology and restricting syntax if desired.
- Most users will want to represent largely informal knowledge and will rarely need or benefit from formal syntax and semantics, but these should be available if needed.
- The system should not make inferences automatically if there is a penalty in performance or expressiveness. Any such inferencing should be largely understandable to non-computer people (i.e., it should not need much knowledge of logic or other formalisms). The experienced user should be able to define new kinds of inferencing. Simple computation should be available, as in spreadsheets.
- Flexibility is essential. The user should be able to learn a few simple principles and combine them in natural ways to use the full power of the system. Special cases should be minimized and extensions should build naturally on the general principles.

1.3 Related Representations and Systems

Our work has been influenced by several classes of tool which have some of the knowledge management features we desire, although the term ‘KMS’ has not been used for them.

The primary purpose of traditional *knowledge acquisition* systems (Boose 1988), (Gaines 1987), has been to build expert systems. Indeed, the term ‘knowledge acquisition’ has developed such a strong expert-system connotation that we tend not to use it when describing our work. As the applications described in section 4 attest, CODE4 is more generic in nature. It is unlikely that the knowledge acquisition systems cited in the next two paragraphs would have been applicable to such diverse applications.

KEATS (Motta, Eisenstadt et al. 1988), (Motta, Rajan et al. 1991) is a classic knowledge acquisition system intended for building expert system applications. KEATS’s strength appears to be that it integrates useful classes of tools: 1) tools for informal segmenting and searching of transcripts of expert interviews in a “hypertext” mode; and 2) tools for drawing semantic networks of the relations between frames in a knowledge base (kb). 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 relatively straight-forward frame representation.

SB-ONE (Kobsa 1991) combines both expert system and natural language capabilities. Its knowledge representation is based on KL-ONE, and it is designed to accept natural language queries. Although SB-ONE embodies many useful ideas, its KL-ONE underpinnings put it at

odds with our philosophy that inferencing and formality should not constrain expressive representation: KL-ONE offers *classification* as its main inferencing mechanism, but for our purposes we have found classification to be of no advantage: no user has ever expressed difficulty in performing classification, and in fact rarely really does it However the encumbent restrictions on expressiveness would be a severe disadvantage to the point of rendering the system unusable for the applications we support.

Another class of system is exemplified by Cyc (Lenat and Guha 1990). Cyc can be viewed as a knowledge *resource* for other software more than for unskilled humans to use directly. Like CODE4, it is intended to be generic, although it is not intended to be used as a shell, i.e., where the user develops a knowledge base from scratch. (Most CODE4 kbs have been thus developed, since they have virtually no overlap of subject matter.) Cyc is a very large and complex system (in a class by itself in terms of scale), and assumes that knowledge enterers are highly trained and, in particular, understand its complex ontology (Skuce 1993d).

The Botany Knowledge Base Project, with its underlying system KM (Porter, Lester et al. 1988), overlaps in spirit with the CODE4 project. Like CODE4, KM has been designed as a generic system, but to date has focussed on capturing the kind of knowledge found in university biology textbooks and is therefore intended to be used mainly as a knowledge *source* for students. KM features specific mechanisms for inferencing on certain types of questions common in biology such as causality in processes. It also features user-customized explanations in English. Currently, there are plans to apply KM to descriptions of software systems, an application for which we have used CODE4.

Although not a direct influence on the early development of CODE4, we should point out the significance of KIF (Genesereth 1992) and its derivative Ontolingua (Gruber 1993). KIF (Knowledge Interchange Format) is a highly expressive knowledge representation language with a formal semantics, and is intended to act as a common denominator to allow translation between various representations. Ontolingua is an extension to KIF with abstractions for the descriptions of ontologies (by which its developers mean knowledge bases built around concept hierarchies). Both KIF and Ontolingua are complex and require significant expertise to use, e.g. they are based on logic; both were designed as stand-alone textual languages, not tools, and have abstractions to allow concise expression of aspects of knowledge found in a wide variety of other representations. CODE4 on the other hand was designed as a tool, not a language, with a strong emphasis on its user interface (described in section 3) and this has resulted in an emphasis on abstractions that maximize *usability*. We have built a simple translator that can load a subset of KIF into CODE, and have designed our own siimple "KIF-like" syntax for transfer of kbs to other systems. We intend to redo this and conform to KIF..

2. Knowledge Representation

The knowledge representation used by CODE4 (CODE4-KR) has its roots in ideas borrowed from frame-based inheritance systems, conceptual graphs, object-orientation and description logic systems (also called term subsumption systems). CODE4-KR favours expressiveness over the ability to perform complex automatic inferencing, and is simple enough to be understandable by non-AI specialists. It can be used both informally to clarify preliminary or developing ideas, and more formally if it is desired to make precise definitions or to allow the system to make inferences. Emphasis has been placed on permitting the user a wide choice in the degree of formality, making it “retrofittable”, i.e., one can capture knowledge informally at first and then progressively formalize parts of it if needed.

2.1 Basic Concepts, Terminology, and Ontological Assumptions

As with several other knowledge representations, the basic unit of knowledge in CODE4-KR is called the *concept,* but we use the term somewhat differently from most. In CODE4-KR, a concept is any knowledge structure that participates in the underlying inheritance mechanism. This includes the parts from which a typical frame-like structure is created.

It is critically important to distinguish a concept from the 'thing' the concept represents. A concept is a piece of knowledge inside a CODE4 knowledge base[1], whereas clearly most things are *not* inside CODE4. While this may seem an obvious distinction to many, we have 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, without making the distinction it is difficult to understand the significance of some of CODE4-KR's features.

Hence a concept represents a 'thing': either the collective idea of a collection of similar things (a *type* concept[2], e.g., 'car') or a particular thing (an *instance* concept, e.g., 'my car'). The thing may be abstract or concrete, real or imagined; it may be an action, a state or in fact anything one can think about[3]. Each knowledge base has a most general type concept (the top of the inheritance hierarchy), of which all other concepts are subconcepts. By convention we label this concept 'thing' although this name can be changed by the user.

The above ideas are similar to those in most other frame-based KR's (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 KR's define 'slots' as distinct structures that are inherited by concepts. In CODE4-KR, *properties* perform this role, but properties are just another kind of concept, i.e. they are units of knowledge in CODE4. Thus things have properties, and hence we say (loosely) that their concepts in CODE have properties.

In a similar manner, most KR's 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 the fact that a concept has a property is encoded by a *statement.* This CODE notion has been made close to the linguistic notion of a statement to may it easier for users to grasp: a concept is the *subject*, and its property is encoded by the *predicate* of the statement. Thus a CODE4-KR statement has, at least, a subject and a predicate, and usually more. Statements are discussed in detail below, but for now the important idea is that *statements, too, are treated as concepts that link properties to concepts.*

By virtue of being full-fledged concepts, statements participate in the inheritance hierarchy, inherit properties, can have statements about themselves and can be referred to in other statements. They are special only in the sense that they have *additional* semantics associated with them, but otherwise behave as normal concepts do in general. The same is true for predicates: they are concepts whose function is to participate in statements. CODE4 uses two other primitive concept types called 'terms' and 'metaconcepts' which are described in sections 2.7 and 2.8.

---

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

[2] The term 'concept' is often used in the KR literature to mean only 'type', but this is at odds with its normal meaning: e.g. every Canadian has a concept of Canada, which is not a type.

[3] To many people, the word 'thing' connotes a physical 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 house'.

Any use of these four primitive concept types is represented as an *instance* of the *type, e.g.* a particular statement is an instance of the statement type. Figure 1 shows the four primitive types under the general type 'CODE concept', but their instances are not displayed. When displaying an inheritance hierarchy, most users suppress the display of the primitive types and instances in order to concentrate on their own concepts, which we call *user concepts*. Those user concepts that are the subjects of statements we call *main subjects*.

When we talk about concepts in this paper, we are referring to concepts in general; however we find that CODE4 users often are referring to just 'user concepts'.

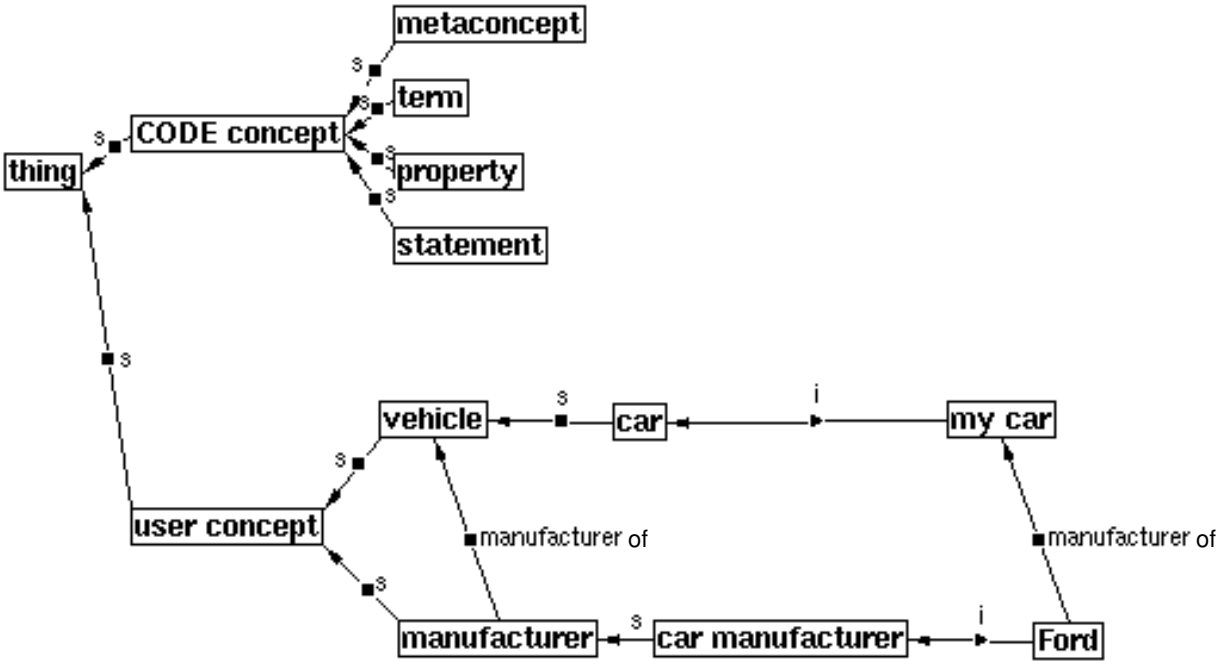

Figure 1: An inheritance hierarchy as displayed by CODE4's user interface, showing a sub-hierarchy of CODE4 primitive concepts (top) and a sub-hierarchy of user-entered concepts (bottom). Users typically employ user interface capabilities to hide the primitive concepts.

A concept has some properties (some or all of which may be inherited from superconcepts) and is characterized by statements involving those properties. User concepts have properties defined by users, whereas special concepts have certain primitive properties. In Figure 1, a 'manufacturer' is stated to be a manufacturer of vehicles, and this 'manufacturer-of' property is refined for 'Ford', an instance of manufacturer of vehicles. The 's' link means "superconcept", from which properties inherit. The figure shows explicitly that

- My car is a vehicle; car manufacturer is a user concept.
- Properties and statements are things..
- 'manufacturer of' is a property of manufacturers.

Note how the display has been designed to make it easy to see these facts, and to see if there are mistakes: here we may note that manufacturers in general manufacture something more general than vehicles, perhaps 'manufactured thing', and 'car manufacturer' should specifically manufacture cars. Only humans can spot such anomalies: to do it automatically would require enormous numbers of formal rules which in turn could easily have just as many mistakes which again could only be detected by humans.

It is also the case, though not directly apparent in this figure, that

"Ford is the manufacturer of my car" is a statement, i.e., an instance of the type 'statement'.

One way to look at a concept for a thing X is as a holder for a set of statements about X, i.e., CODE4's "knowledge of X". Thus the *meaning* of the concept is determined, and possibly defined, by these statements.

Figure 2 shows a statement's relationships with other concepts. In addition to being instances of the primitive type 'statement' and having a subject and predicate, most statements also have a *value.* A value usually corresponds to the direct object of a sentence. Values are discussed in depth in sections 2.3 to 2.6. A value frequently refers to another concept, in which case it can be said that the statement's subject is related to the the concept pointed to, by way of the predicate. Looked at in relational database terms, a property can be considered a relation and the statements involving that property can be considered tuples of that relation.

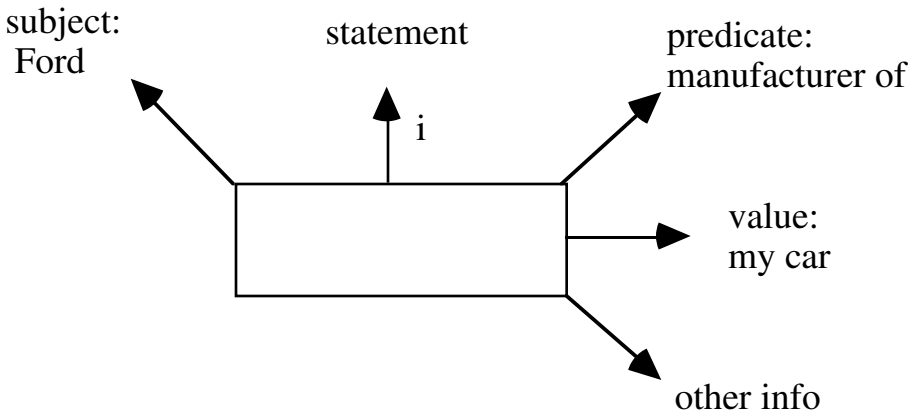

Figure 2: Shows how the statement 'Ford is the manufacturer of my car' is related to other concepts. Its subject is 'Ford' (an instance concept), its predicate is the property 'manufacturer of', its value points to 'my car' (an instance concept), and the statement itself is an instance concept of the primitive type concept 'statement'.

Statements can be displayed on a CODE4 graph[4] as links between the subject and the value concept (in fact all links on graphs represent some statement). The property name labels the link. Superconcept statements appear as links in inheritance hierarchy graphs; other statement instances are displayed as extra links in such graphs only when requested (e.g., for 'manufacturer of' in Figure 1). Statements can also be shown explicitly by displaying a statement hierarchy, as described in section 2.2.

As mentioned earlier, every concept is either an instance or a type. There are two explicit user instance concepts in Figure 1, indicated by the 'i' link. The purpose of a type is to specify statements that must or may be true about its instances. For type concepts, all statements are interpreted as *typically* true of its instances unless a specific modal qualifier is present as a facet (see section 2.3).

2.2 The Inheritance, Property, Statement, Relation and Facet Hierarchies

Statements link CODE4 concepts into a complex network build around several useful abstract structures. In fact, it is essential for users to work with such abstractions, since working with the entire network at once would be impossible. These structures usually take the form of hierarchies, and we refer to them as such, even though some can be general directed graphs. These hierarchies form the basis for knowledge maps, which are discussed in section 3.3.

Below we describe the five kinds of hierarchy: Inheritance hierarchies are common to all frame-based KR's. Property and statement hierarchies are less common, and where present may be given less prominence. Relation 'hierarchies' (semantic nets) are not a new idea either, but CODE4's treatment of them uniformly with other 'hierarchies' has advantages for ease of use. Facet hierarchies, a CODE4-KR innovation, are introduced here and are described in more detail in section 2.3. In addition to these five structures a sixth but abstruse hierarchy can be formed by following metaconcepts of metaconcepts (section 2.8) recursively.

- **The Inheritance Hierarchy**: All concepts participate in the 'inheritance' or 'isa' hierarchy[5], including "empty" ones: the user can locate a new concept beneath an existing one but not give it any statements yet (of course it will inherit statements from its parent). The purpose of this hierarchy is conventional: to permit taxonomic structuring of knowledge and property inheritance, described in section 2.4. CODE4-KR permits a concept to have multiple superconcepts (parents).

- **The Property Hierarchy**: The second hierarchical structure is termed the 'property' or 'predicate' hierarchy. All properties (instances of the primitive type 'property') are arranged in a hierarchy distinct from the inheritance hierarchy. There is a single top property and users typically create several levels of subproperties.

  There are several ways of interpreting or designing a property hierarchy. It may just be seen as a convenient way of grouping properties – higher level properties can be considered property *categories*. A unique feature of CODE4-KR's property hierarchies is that a property can have *multiple* superproperties (it can be in several categories).

---

[4] In fact, as will be seen in section 3, links (statements) are also shown on outline-format and matrix-format displays.

[5] Sometimes also called the 'concept hierarchy', although this is misleading because all the hierarchies involve concepts.

If the user desires more formality, the partial order in the property hierarchy can be interpreted as “implies”. A property P implies its superproperties (implicants), i.e., any statement formed from P will imply all similar statements formed from its implicants. Example: if ‘walks’ is a subproperty of ‘moves’, then any statement that ‘X walks’ implies that ‘X moves’. No inheritance occurs in the property hierarchy because this would mean that “properties of properties” inherit. This is the case since properties have few properties and these normally do not inherit. (However this can be arranged if needed; see section 2.6.)

The property hierarchy forms a second “dimension” in a knowledge base, after the ‘isa’ hierarchy, one we find extremely useful for further classifying knowledge. Although property hierarchies (or recursive slot structures) are also found in Cyc, KM, and KL-ONE systems, they are typically not treated with the same importance as they are in CODE4. For example, in KM, slots are in a global hierarchy, but the display of triples in the user interface does not show this.

- **Statement Hierarchies**: Each concept inherits a sub-hierarchy of the property hierarchy: all those which apply to it. The statements formed from this sub-hierarchy form the ‘statement’ hierarchy for the concept. A knowledge base has one property hierarchy but many statement hierarchies (one for each concept). Systems that lack this capability typically have long, flat and unstructured lists of slots associated with each frame. Statement hierarchies greatly assist in indexing or creating appropriate statements, since a new statement (its property or predicate) usually “fits under” some existing one. Often a user may display only statements in only one hierarchy for a period of time, to reduce visual overload.

  An example partial property and hence statement hierarchy for the concept of car might be:

  car:

  properties
  related entities
  parts
  electrical parts
  ignition parts

  The last of these is read as: 'cars (typically, the default) have ignition parts', since if the propery name is a noun, the verb ‘has’ is inserted to make the statement. We also see that having ignition parts implies having electrical parts.

  ‘Isa’ and ‘statement’ hierarchies are treated as equal participants in making inferences from statements. We exploit this duality much further however. The user can think or browse equally well in terms of either the ‘isa’ or the ‘property/statement’ “dimension”. Thus a statement should be thought of in two "dimensions": under the super of the subject and under the super of the predicate (property). When entering a new statement, the user must choose both the super of the subject and the super of the predicate.

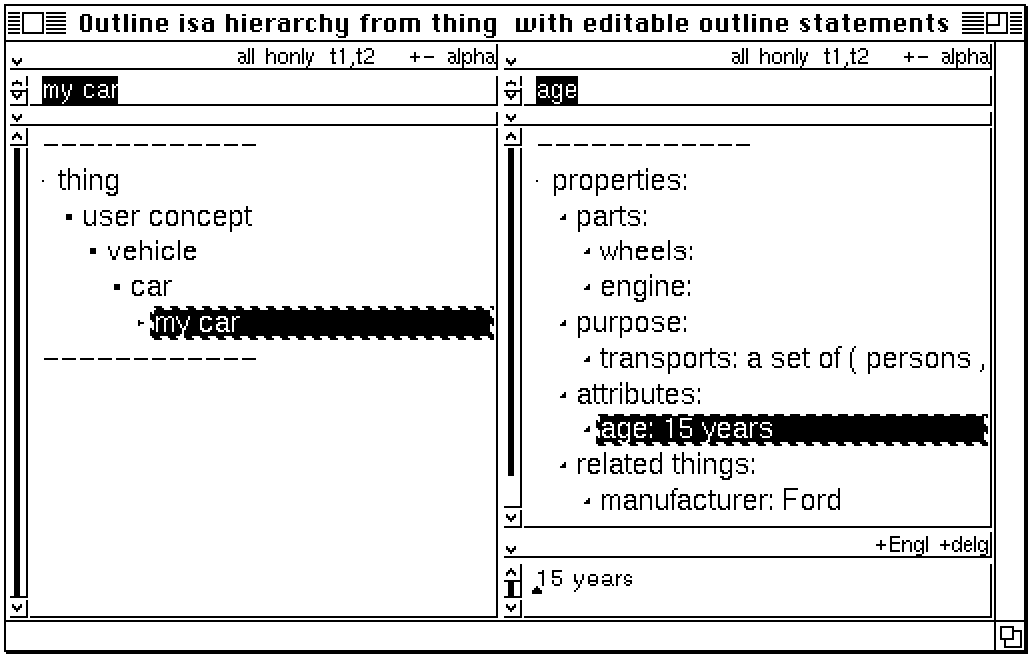

Figure 3: Outline views of an inheritance hierarchy (left) and a statement hierarchy (right) of statements of 'my car'. Components and functionality of the browser are discussed in section 3. The value for the age statement appears in the bottom pane.

Figure 3 shows a typical CODE4 user interface component, an *outline isa hierarchy browser with editable outline statements.* (See Section 3 for more discussion of these components). Here we can see a partial hierarchy of user concepts on the left and the statement hierarchy on the right for the selected concept, 'my car'.

- **Relation hierarchies**: 'Relation' hierarchies are formed by following chains of concepts via the values of statements of one or more properties. This fourth kind of hierarchy is typified by the 'part of' relation, i.e., 'part-of' is a property that applies recursively: the parts of things are usually themselves things that have parts. Organizing knowledge in 'part of' or other such hierarchies is also a very important requirement of knowledge management systems. In CODE4, any relation can be defined thus, assuming it make sense. The system has features for graphing such hierarchies easily, and defining inheritance behaviour over such relations (similar to "transfers through" in Cyc).

- **Facet hierarchies**: Statements can be made recursively about statements and this results in the formation of 'facet' hierarchies. Facets are described in the next section.

2.3 Facets

Facets are secondary statements representing incremental additions to a statement. There are three main types:

1) Facets corresponding to noun complements following the predicate. e.g., John bought <a car> <from Ottawa Cars> <on Jan 4> would have 3 facets, of which the first is the statement value (the statement has subject: John, predicate: bought). The user is responsible for creating and naming these additional facets, e.g., 'seller' and 'date'.

2) Facets for additional information that would be part of a sentence, such as modal, quantificational, temporal and adverbial modification. These do not correspond to noun phrases.

3) Documentation or background information, such as the source, enterer, or date of the statement.

Figure 4 shows the facets for the statement about age on 'my car' above, i.e., they are all statements attached to the statement 'age' about 'my car' with value '15 years' which could be simply rendered as "my car (necessarily) has an age, which is 15 years"[6]. The value and modality facets participate in this minimal version of the statement. The other facets would be rendered as separate statements about this statement.

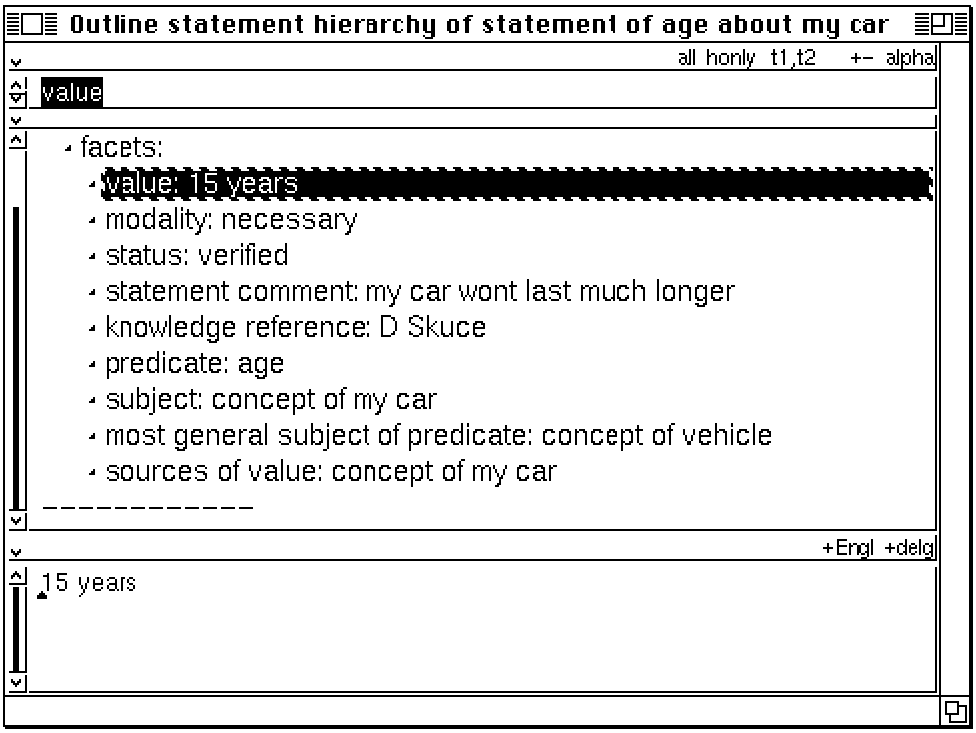

Figure 4: Facets for the statement about 'age' of 'my car'

Properties can be viewed as logical predicates; most are binary or higher arity, i.e., statements (facets) using them require a value[7]. The value facet automatically inherits; others do not unless specified, though we often make modality inherit. Facets considered as predicates are almost always binary; i.e., facets always take a statement as their first argument (the subject) and either

[6] CODE2 had a facility to output statements in such an English-like form. We intend to reintroduce such a capability into CODE4.

[7] One might suspect infinite regress here: A statement has a value facet, which is a statement, which has a value facet etc. In practice no problem arises: When one asks for the value, one gets what is stored, the system does not actually look for a value facet; the value facet is *virtual*.

a reference to other concept(s) or other values as their second argument, termed the 'facet value'[8] (see section 2.5).

Frequently used facets include:

- *comment*, which is purely informal, i.e., for human processing only.
- *status* (whether the statement has been verified by someone)
- *knowledge reference* (where the knowledge came from). CODE4 has extensive facilities for locating statements based on the contents of such facets, e.g., "find all statements referring to 'Ford' that were entered by Bill since Jan 1 and have not been checked" (see section 3.5).
- *modality*: We find it essential to be able to explicitly record modal information, such as whether a statement is a) necessarily true (for all instances of the subject), b) typically true (in most instances; the default), c) optionally true (possible), d) impossible or absurd (in no instances; because does not make sense), or e) false (in no instances, because it is contingently not true). By following these conventions, optional checking of modality consistency in subconcepts can be done.
- *quantification*: Statement subjects are either types or instances. In the former case, the subject is universally quantified and any other facets in the statement are by default existentially dependent on it. No quantification facet is attached to these arguments in this case. Default quantification can however be explicitly overridden using the quantification facet.

As an example of the use of facets, 'full-time students can take 2 to 6 courses' would be rendered:

full-time students
take: courses | modality: o | quant: 2 to 6

This statement has three facets (separated by '|'), the statement value (courses) plus a modality (optional) and a quantification. If an inconsistent change is made lower in the hierarchy, such as changing the quant to 1, this can be detected in the form of a warning.

Thus, in contrast to some KR systems which would not permit entering an inconsistent statement, CODE4 allows it but can issue a warning, which may be recorded as a flag on the statement that further attention is needed. Later during knowledge reviews, one can ask for all statements flagged with inconsistencies. We have taken this approach because we have found that often as knowledge is emerging, users want to state beliefs that may be inconsistent, and deal with them later, possibly in consultation with others. For our purposes, enforcing literal consistency would prevent many users from entering knowledge at all: imagine if you told someone "you cannot speak unless you are always consistent". CODE4 is intended for capturing knowledge of any quality; hopefully rough knowledge would be later refined.

---

[8] Facet values are not to be confused with the value of a statement, which is a facet which holds the statement value, another name for the linguistic direct object. All other facets also have values, but these are not "values of the statement", and do not correspond to a direct object.

2.4 Inheritance

In most frame systems, all facets of a slot inherit together. We have found such "slot-as-a-whole" inheritance undesirable in many cases, particularly in two main situations: a) during single inheritance when only a part of a statement is changed (e.g., a change in a comment that should inherit); b) during multiple inheritance, where facets may inherit from different parents so that the statement is a hybrid.

CODE4 uses a built-in inheritance rule for the value facet, since it always inherits. However the value may require combining expressions, e.g., the following inheritance structure can be created using a structure called 'both of' (a logical 'and' for noun phrases) primitive value combining function:

pets	cats:
eat: expensive food	eat: fish

pet cats
eat: both of (expensive food, fish)

In 'pet cats', the value is a hybrid, i.e., it comes from two parents, and a change at 'pets', say, will be reflected in 'pet cats'. For the other facets, the user may create various other kinds of inheritance behaviour such as inheritance over 'part of' (similar to Cyc's "transfers through") links using block computation (section 2.6). Many frame inheritance systems do not have this flexibility.

2.5 ClearTalk, Formality and Informality

If the user wishes, certain syntactic conventions may be used when entering certain expressions; for example, property values and concept names can be restricted to simple noun, verb, or adjectival phrases. These conventions, termed ClearTalk, have the following benefits:

- They allow a statement value to be interpretable as a reference to other concept(s) and hence to be *interpretable* or *formal* in the sense that it refers to another concept, thus permitting certain inferences to be made. If the user has so requested, all statement value expressions entered are parsed and if concept(s) exist that correspond to the expression, pointers are automatically placed in the value, rather than the expression itself. What the users sees is the current name of that concept, i.e., the pointer is followed to the concept(s) and the name(s) is shown. As an example of a trivial but important inference, if a concept is subsequently renamed, a commonly needed operation, all references to it will still exist correctly and the new name will appear everywhere. Few systems support renaming, a critical knowledge managment operation.

- They facilitate consistency checking. For example, if the value of the 'children' property for the concept 'person' was stated to be 'a set of 0 or more children'[9] (a ClearTalk expression), it is easy to check that a consistent refinement of this in a lower concept 'parent' would be 'a set of 1 or more children'.

[9] This example is most interesting because it is wrong, yet many humans do not see the problem: the children of a person are only children when they are young. This illustrates the type of errors that occur frequently, yet we believe can only be realistically detected by humans.

- They restrict what the user can say, to prevent unclear phrases or statements.

To be interpretable in the sense described above, a value must be expressed in ClearTalk, but not all ClearTalk expressions are interpretable at present[10]. There are several other ways of creating such values, e.g., editing a relation knowledge map (section 3.6) or pasting concepts directly into a value.

We have found that users of CODE4 typically approach formality incrementally; i.e., they start by typing arbitrary strings into values. As more concepts are added, they review statements and convert them into ClearTalk, or at least something close. More details on incremental formalization can be found in (Lethbridge and Skuce 1992) and additional perspectives supporting these ideas can be found in (Shipman 1992) and (Shipman 1993).

2.6 Block Computation and Delegation

In Smalltalk, a block is the equivalent of a closure in Lisp, i.e., a function plus its environment. One can specify a block as the value of a CODE4-KR statement, however we restrict their syntax to a) allow users unfamiliar with Smalltalk to write blocks and b) ensure that CODE4 is not too dependent on the semantics of Smalltalk, to ensure portability.

Such blocks give the following kinds of functionality:

- The ability to define values as functions of the values of other statements.
- The ability to create “rules” like in expert systems.
- The ability to forward chain, i.e., to dynamically re-evaluate if a computed value should be changed, giving CODE4 a behaviour like a spreadsheet.
- The ability to easily define special behaviour such as inheritance and type checking.
- The ability to treat CODE4 concepts as programmable objects, combining ideas from delegation systems such as Self (Ungar 1992) and constraint management systems such as Garnet (Myers, Giuse et al. 1990).

We have had the most experience with the first functionality listed above, which we call ‘delegation’.

An example of a block in the syntax we use is found in the ‘condition’ property for batteries, below. These blocks, kept in facets, run automatically when a value is requested to perform constraint maintenance. Thus if the value of ‘condition’ is needed and the value of ‘voltage’ has been changed, the toCompute block of ‘condition’ will execute. The symbol ‘#’ (read ‘this battery’) refers to a battery of interest (an instance of the type ‘battery’) and the ClearTalk expression ‘the car that # belongs to’ refers to the 'unstartable' property of the concept (a car) pointed to by the ‘belongs to’ property of this battery. ‘thisValue’ is a keyword referring to the value of this property, in this case, the value of ‘condition’.

battery
    belongs to: a car

[10] We intend to enhance the coverage.

```
voltage: from 0 to 15 volts     ; a ClearTalk expression
condition:
        toCheck:                [thisValue isOneOf: {good, bad}]
        toCompute:              [if: the voltage < 10 volts, and
                                 the car that # belongs to is unstartable
                                 then: thisValue := bad]
```

Block expressions could be made to participate in consistency checks (i.e., to ensure that a block in a higher-level concept is more 'general' than that in a lower-level concept).

2.7 Terms

CODE4-KR treats the names of concepts (called terms) as full-fledged concepts themselves. When a concept is first created, it is given a system-created label (e.g., 'instance 12 of car' or 'specialized vehicle') to distinguish it from other concepts and to avoid forcing the user to think of a name. Such new concepts do not have any term associated with them, and the label can change dynamically if the context from which it is derived changes. Once most main subjects and properties are created, users may give them terms by simply typing over the generated label. An instance of the primitive type 'term' is created automatically whenever a user enters a new name for a concept or property.

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). Our experience has shown that this facility, which we believe is unique to CODE, is highly valued by users. 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 is connected.

As far as we know, amongst knowledge acquisition systems, only the Active Glossary system (Klinker, Marques et al. 1993) has similarly treated terms as serious objects. Experience with earlier CODE systems has taught us that the flexibility of CODE4-KR's term system is essential for the type of work we are interested in.

2.8 Metaconcepts

When representing knowledge, it is frequently necessary to describe properties not of the thing T (e.g. a car) a concept represents, but of the concept *itself*, which is different from the thing T (e.g. the concept of a car). The former has weight and a price; the latter does not. Examples of such properties include: the person who entered the knowledge about the concept; the date the concept was invented; declarations about relations between it and its subconcepts, etc. For this purpose, whenever statements are to be made about the concept itself (as opposed to the thing), CODE4 automatically creates (if it is not already there) a unique *metaconcept*, to which it attaches these statements. Metaconcepts are instances of the primitive type 'metaconcept', which is the (most general) subject for such properties as 'English description', 'terms', 'superconcepts', 'graph layout position' etc. These properties inherit to the individual metaconcepts, but *not* to subconcepts of the concept described by a metaconcept.

Figure 5 shows the metaconcept for 'car manufacturer' (i.e., the concept of the concept of car manufacturer). One can see, for example, that there are two terms for this concept, 'car manufacturer' and 'automobile manufacturer'. Terms were discussed above, in section 2.7.

Such a distinction is often made by permitting "thing itself" properties to inherit and others (the "meta" properties) to not inherit. We believe that CODE4-KR provides a more elegant and intuitive mechanism for non-inheriting properties, by putting them on a separate concept since

indeed it is separate. Hence most KR's require that non-inheriting properties (slots) be tagged as such, but CODE4-KR has a uniform rule that all properties inherit. Sometimes it is desired to have statements of metaconcept properties *appear* to inherit (e.g.,. if 'Joe' entered a whole hierarchy of concepts, the 'entered by' property for all the metas would have the value 'Joe'). In other cases, it is desired to have groups of unrelated metaconcept statements have similar values. Delegation can be used to effect this, or else specialized subconcepts of the 'metaconcept' primitive type can be created, which inherit to a certain subset of metaconcepts.

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 second-order metaconcept (e.g., to hold properties describing who updated the first-level metaconcept).

We use an important property of metaconcepts called 'dimensions'. A dimension is a partitioning of a concept into subconcepts based on one point of view or property, for example, the sex of persons. CODE4 has active support for such partitions which we have found are an important guide for organizing and understanding knowledge.

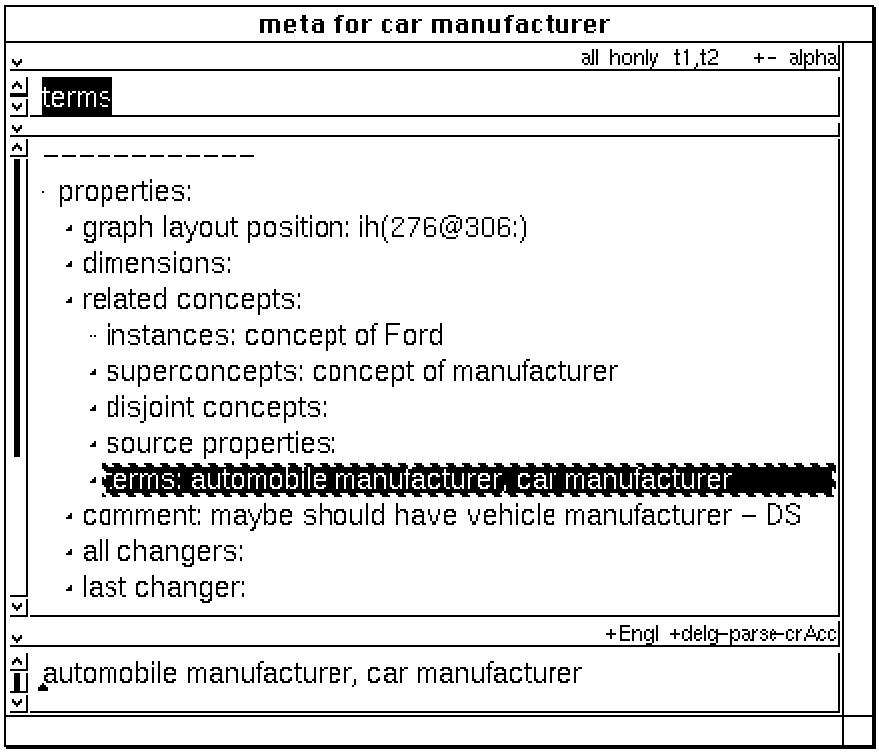

Figure 5: Properties for the metaconcept of 'car manufacturer', showing that the concept 'car manufacturer' has two synonymous terms.

2.9 Persistent Storage; CODE4 as a Knowledge Server

The operations on the CODE4 knowledge base are partitioned into two major sets: Those that update the knowledge base which we call *modifiers*, and those that query the knowledge base which we term *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 *constructor* commands that add concepts and links between concepts, but do

not delete or change knowledge. The *basic constructors* form a minimal set necessary and sufficient to construct any CODE4 knowledge base.

In-memory knowledge is stored as a network of Smalltalk objects. 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. This supports interaction with both internal and external "applications". When an application interacts with the API, it passes knowledge backwards and forwards in a well defined syntax. For example, CODE4's *knowledge map* layer, which provides high-level abstractions for the user interface and is described in section 3.3, can be considered an application that dialogues with the knowledge engine using the API.

Another major use of the API is the Ckb (CODE4 Knowledge Base) language interpreter. Expressions in Ckb are ASCII representations of modifiers and navigators and provide the means of persistent storage. Figure 6 shows how these CODE4-KR manifestations are interrelated, and figure 7 shows how they are embodied in the CODE4 architecture.

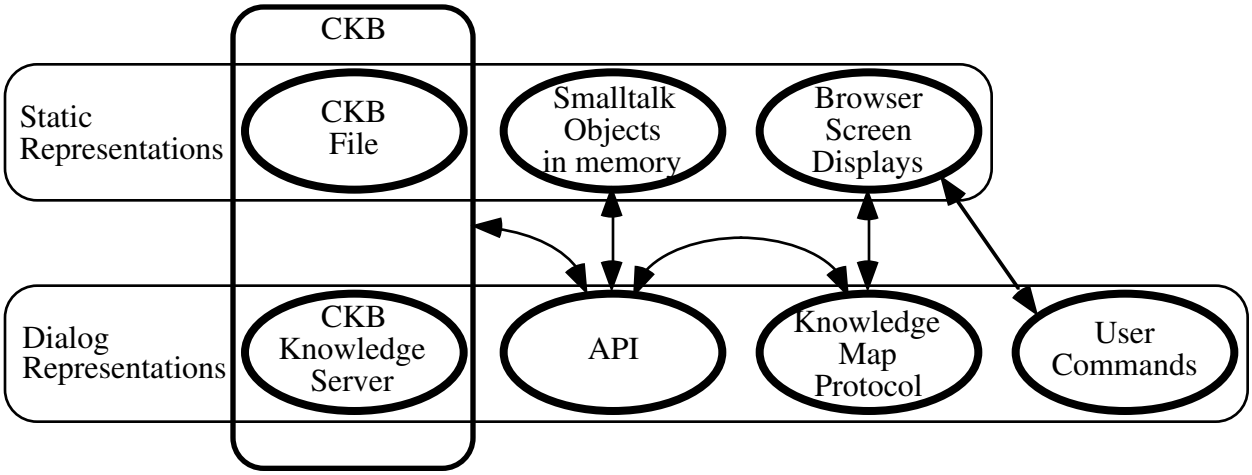

Figure 6: Several manifestations of the CODE4-KR. There are several syntaxes with a common semantics. Arrows show syntax translation paths.

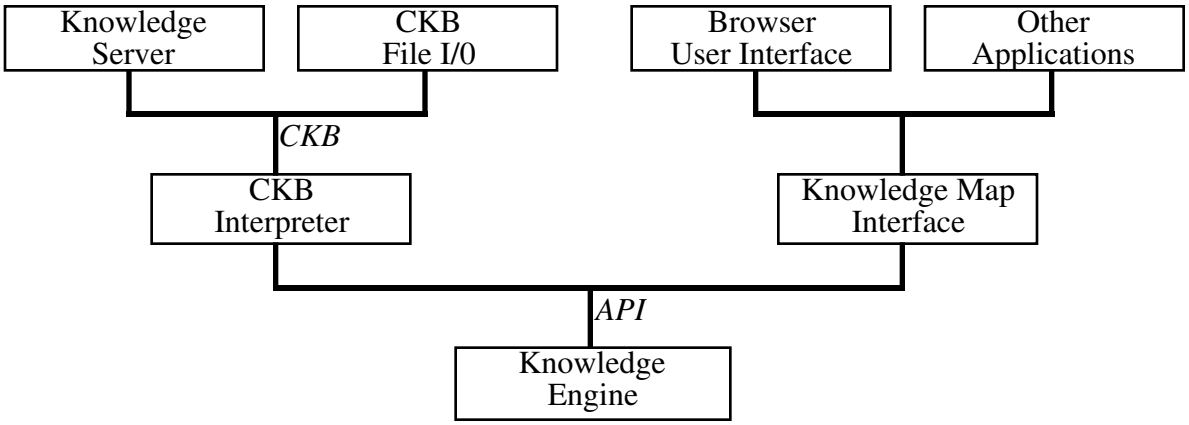

Figure 7: The top-level architecture of CODE4 showing how various internal 'applications' interface to the knowledge engine.

Ckb is used for two major purposes:

- *For persistent storage*: To save knowledge 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. We have used this mechanism for several purposes:

  1) At the National Research Council of Canada (not yet published), Peter Clark has built a Prolog-based expert system called 'Electronic Trader' (ET) that attempts to recommend 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.

  2) We have built two kb translators: One converts a subset of Ontolingua into Ckb and pipes it into CODE4. The other does the same with Porter's KM. We have loaded 3000 concepts of the Botany kb without difficulty.

  3) We are currently experimenting with accessing CODE4 via Mosaic. We have built a Mosiac interface that permits text-only access, though nowhere near as nicely as the CODE4 internal interface. However the Mosaic interface permits multiple users situated anywhere to concurrently interact with a kb. We have not yet begun to study how to control such interaction, but it is high on our agenda. Unfortunately at present, Mosaic is not designed as a user interface tool, and creating input screens is awkward. Future versions of Mosaic should rectify this problem.

Another direction for future research is to enhance the present saving and loading mechanism, which operates on a whole kb at a time, so that it can 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, though this is not a problem in our applications which tend to be hundreds to thousands of concepts; even ten thousand would not be a problem. It would also ease multi-user development of knowledge bases.

The Ckb language is designed to be compact, but human readable. We believe that human-readability is important, as opposed to some binary format, because it simplifies the job of those who need to write conversion programs that read or write Ckb files, and programming remote clients such as our Mosaic interface. However Ckb's readability is limited by its compactness and extreme ease of machine-parsing, which are more important objectives: We want 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 about 350 bytes of memory including the terms, statements, properties and metaconcepts it introduces. On a Sparcstation 10, a 3000 concept knowledge base can be loaded in less than a minute. (We have not seen such figures for other systems published.)

Aside from syntax, there is an important difference between Ckb and Lisp-based knowledge languages like KIF or Ontolingua: Ckb is independant of the natural language terms used, since CODE4 supports renaming. In Ckb, each concept is given a numeric identifier when it is created in memory. When information about a concept or references to the concept are transmitted to other systems, or written to a file, this identifier is used. The name (term(s)) of the concept are never used because they may change or may not exist (see section 2.7). In contrast, most knowledge saving/sharing notatinos require a unique name or symbol to identify concepts. Naming becomes a major problem, and renaming is impossible. We find the Ckb approach more flexible in that people are never forced to come up with potentially poor names merely for uniqueness, and can make changes at will, use multiple sets of terms, or use overloaded terms.

2.10 Semantics and KIF Compatibility

The semantics of CODE4-KR are currently defined operationally in the executable system, and are described semi-formally in its documentation. The core semantics (e.g., concepts, hierarchies, formality/informality, and inheritance) were well established by mid-1991 and have remained virtually unchanged since then. We have found that over 95% of the use of CODE4 involves only this core, and so users have been able to work confidently in a stable environment.

Functioning on top of the core, and subject to greater change, are features such as ClearTalk, specialized facets, combination of inherited values, block computation and language-oriented features of term concepts. Working with the few users who make use of these features, we have been refining them in a series of prototypes. We have found this user-oriented approach has immediate practical value, whereas committing ourselves too early to too much formal semantics could have resulted in a rigidity that eliminates potential applications. We can specify CODE4-KR's semantics by defining mappings into KIF. We have chosen this approach since KIF has a solid formal semantics and is becoming accepted as the de-facto knowledge representation interlingua.

In our preliminary investigations into a formal semantics of CODE4-KR by translation into KIF we have considered the following, e.g.:

- Facet-by-facet inheritance
- Value combination
- The property hierarchy
- Metaconcept and facet hierarchies
- Delegation and block computation

Most of these seem straightforward to translate and we intend to do it. However we are not yet clear on how to deal with, e.g.:

- Informal values
- The independence of concepts from names (terms)

- Natural language-related information
- Graphic (bitmap) knowledge

The challenge of translating from full KIF to CODE4-KR, on the other hand, may prove more difficult, and includes for example the following challenges (our current translator only maps from an Ontolingua subset to CODE4-KR):

- How do we deal with the many computational primitives (Lisp functions) in KIF[11]? We recognized this problem early and have carefully constrained our Smalltalk blocks so that we do not encounter so severe a problem translating *to* KIF. So the question is: should all of Lisp be allowed, and if not, what subset?

- How do we capture the full semantics of KIF's many mathematical features including relations, functions, quantification, etc? So far, CODE4 most users do not use or even know such mathematical sophistication, so we have not focussed on these capabilities. However many of our users require modal quantifiers, which KIF does not support.

3. User Interface

CODE4 features a very advanced user interface (UI), since we have found that ease of use and flexibility are critical to making such systems acceptable to users. Most CODE4 users cannot appreciate, nor do they need or want, some of the subtleties of formal knowledge representation or inferencing, but they all benefit from and appreciate a good UI. CODE4's UI features are facilitated by Smalltalk-80, in which it is programmed[12].

The main components of the UI are discussed next. We show only the more interesting ones as figures to conserve space.

3.1 The Control Panel

The control panel controls all top-level parameters, and default parameters for various views. It also is the interface for knowledge base actions, such as saving, renaming, merging, opening initial windows, etc. Many knowledge bases can be loaded at once and multiple windows can be opened on each knowledge base.

3.2 The Feedback Panel

The feedback panel tells the user about the result of each command. This is of most use when he or she has done something that CODE4 does not like. Our philosophy toward such checking is the result of four of five years experience with this and previous versions of CODE, i.e., it reflects the kinds of use and users for which CODE4 has been designed. The basic tenets are:

- Users should not be forced to do anything they find unnatural or hard to understand.

---

[11] i.e. there is a danger that KIF-based knowledge can only be interpreted by Lisp-based systems.

[12] Smalltalk was chosen for a) ui flexibility, b) rapidity of development, c) platform independence.

- Users should have control over the system as much as possible, i.e., the ability to cancel or ignore system requests.

- The degree of formality and checking should be under user control and dynamically alterable.

- Inconsistencies should be tolerated except when easily detectable nonsense would result (such as circular hierarchies).

CODE4 announces a number of common semantic errors in the feedback panel, for example, an attempt to delete a concept that is the origin of one or more properties (which would be lost if nothing was done about it). The user is offered several "clickable" solutions. In this example, the user would have the choice to: 1) Move the properties "up" to the superconcept or 2) Delete the concept anyway and lose the properties too.

In normal operation, the execution of *all* user commands results in a notation being added to the feedback panel describing what has changed. In the case where nothing has changed (a failed command or an incomplete command) a list of alternatives is presented. In *no case* is the user forced to pick an alternative, therefore CODE4's user interface can be said to be 'non-modal'. The advantage of non-modality is that the user is always completely in control of the dialog and is never forced to make decisions for which he or she has insufficient information. The user is thus never limited in what it is possible to do next. When the feedback panel presents a set of choices for the completion of a command, the user may perform other operations (e.g., querying the system to gather decision-making information) before making a choice, or may abandon the command entirely.

The feedback panel also provides a history of commands and has a rudimentary 'undo' mechanism.

3.3 Knowledge Maps

A *knowledge map* is a software abstraction that allows allows for the convenient manipulation of a network of concepts. A knowledge map defines the network in terms of some starting concepts and some relations that recursively relate these to other concepts. Knowledge maps are a useful abstraction for the following reasons:

- They allow users to organize knowledge in a context (i.e., how concepts are related to each other in a certain way)

- They simplify the manipulation of knowledge by allowing a few simple commands regardless of the kind of map being displayed

- Users can build maps of their own by specifying interesting combinations of starting concepts and relations

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 (although these graphs use knowledge maps, other user interface components use knowledge maps as well.)

- The cartographical analogy is useful: The user can define the map he or she wants to look at, and can then navigate around the map. The user can use masks (associated with maps) to highlight part of the map.

An example is a knowledge map used to display an 'isa hierarchy'. Its starting concept is the top concept desired, e.g. 'thing'. Its relation is the 'subconcept' relation. A knowledge map that displayed a finite state machine might have several starting states and use the 'outgoing transition' relation, i.e. all properties that are subproperties of outgoing transition. (Relations are a kind of property.)

As figure 7 shows, all displaying and manipulation of knowledge by the CODE4 user interface is mediated by the *knowledge map interface*. This is a layer of software that mediates with a knowledge base using hierarchies or directed graphs described in section 2.2. Whenever a browser view (section 3.4) is opened, a new ‘knowledge map’ is opened; this manages all communication with the underlying knowledge engine.

The knowledge map interface facilitates a simple set of commands for navigating around the kb; adding, moving and deleting concepts; displaying or highlighting particular subsets (see section 3.5); and opening other windows that depend on what is selected in the current one. The commands work identically regardless of whether the user is displaying an inheritance hierarchy, a statement hierarchy or some other structure. For example, there is a generic command to ‘add a child concept to the currently selected concept(s)’. In an inheritance ('isa') hierarchy this adds a new subconcept; in a part-of hierarchy this adds a new concept (as a subconcept of the most general subject that can be a ‘part’) and makes this a ‘part of’ the selected concept by adding the appropriate statement.

3.4 Browsers

In order to display knowledge a user must choose both a knowledge map and a browser type. Three basic browser types are described in this section: Outline browsers, graphical browsers and matrix browsers. Commands that operate on these browsers (especially the first two) are very similar. A new browser can be opened as a separate window or as a new pane in an existing window; pane sizes can also be adjusted.

Browsers allow direct manipulation of concepts graphed as nodes and links, and the issuing of commands to the underlying knowledge map or masks (section 3.5). Sets of concepts may be selected for moving, deleting, enlarging in another view, temporarily hiding, reparenting, renaming, deleting etc.

All kinds of browsers can be dynamically chained so that the what is selected in one browser dynamically determines the contents of the next. In fact, the common concept-property browser (figure 3) is a compound browser where the selected concept (in the left pane) determines which statements are shown on the right.

There is no limit to the number of browsers that may be open at a time, and the consequences of changes made in one browser are immediately reflected in all others.

3.4.1 Outline Browsers

The most commonly used UI component is the outline (or textual) browser. It displays information as lines of text, that behave as in typical outline processors: hierarchical relations are shown by indentation. Figures 3 and 4 (in section 2) are examples of such browsers.

3.4.2 Graphical Browsers

Most interactive knowledge acquisition systems incorporate some type of graphical assistance. Our experience confirms that this is an essential feature, hence the graphical features of CODE4 are highly developed. It is possible to open one or more graphical browsers on any knowledge map. As with other browsers formats, the user may work directly on a graph, adding, deleting, reparenting, etc. Additional facilities allow fine-tuning of the automatic layout, manual layout and control of fonts and node shapes.

Graphs showing non-hierarchical property relationships ("semantic nets") may be drawn, either by adding specified property links on top of an inheritance hierarchy graph (as in Figure 1), or by using a 'relation' knowledge map, as in Figure 9, taken from (Ghali 1993). This process is described in section 3.6.

3.4.3 Property Matrix Browsers

We have found that often a user requires a comparison between two or more concepts, i.e., a comparison of their properties. Usually the *differences* are of interest. A *property comparison matrix* can be dynamically opened by selecting any group of concepts (usually siblings in the 'isa' hierarchy) and then selecting those properties of interest, perhaps all. This feature makes it very easy to compare several concepts, to see how they are similar or different. The matrix shows the value facets (and others if desired) for each concept and property as in Figure 10, where rows show properties, columns show concepts, cells show values, and n/a means the concept does not have the property[13].

A similar display is the *property inheritance matrix*, which shows all values of a property as they change down the 'isa' hierarchy. We find this to be the most useful way of checking for consistency of property values, many of which are expressed only informally and hence cannot be checked automatically. When presented with this view, a user can quickly spot problems.

As with other browser formats, both types of matrices are editable, dynamically track selections made in other browsers, and can be set to show other facets besides the value. Some users prefer to use them as knowledge entry tools rather than just for retrieval.

---

[13] This example is discussed under "Applications"

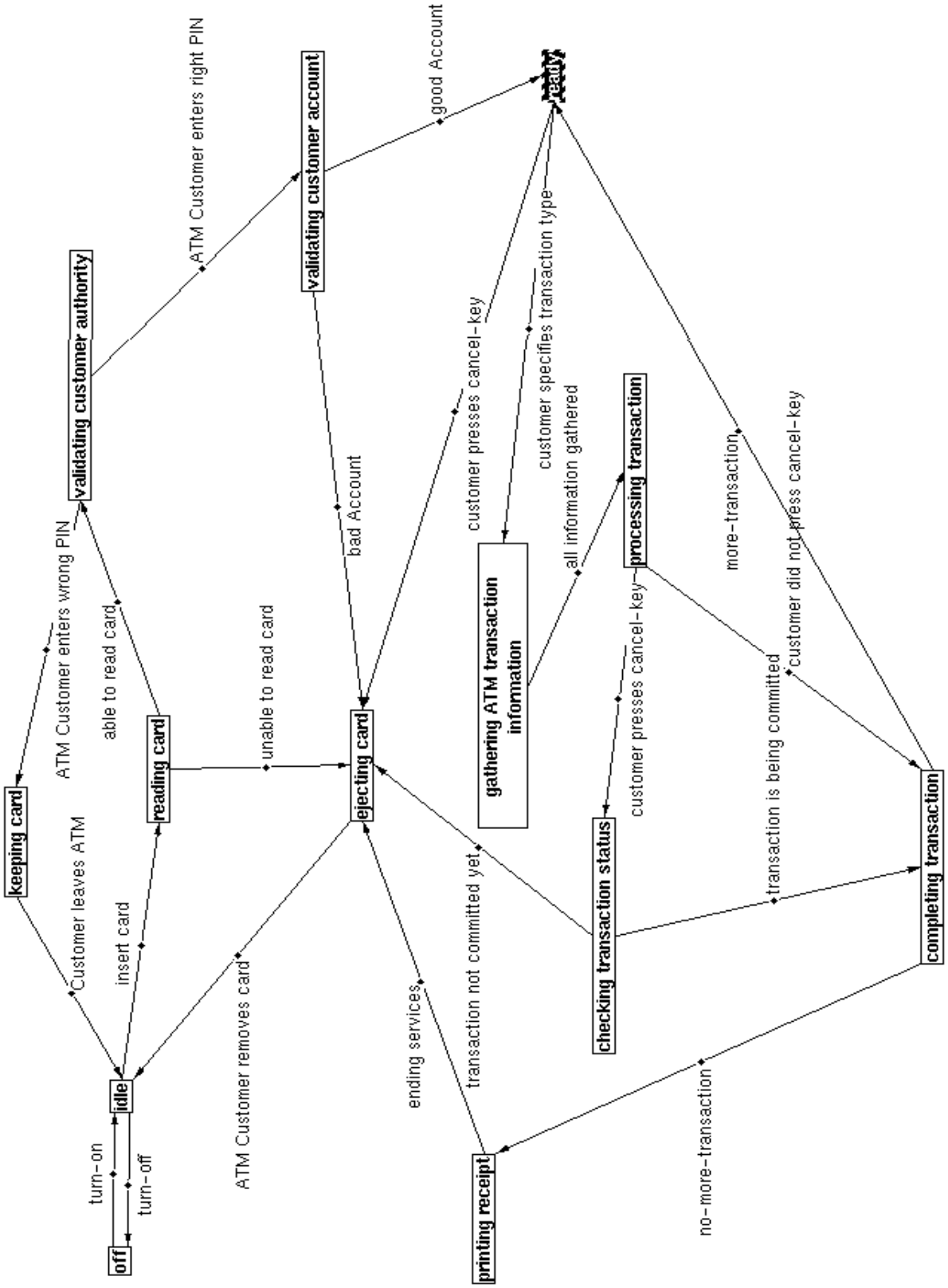

Figure 9: A finite state diagram for an ATM.. Uses a relation knowledge map

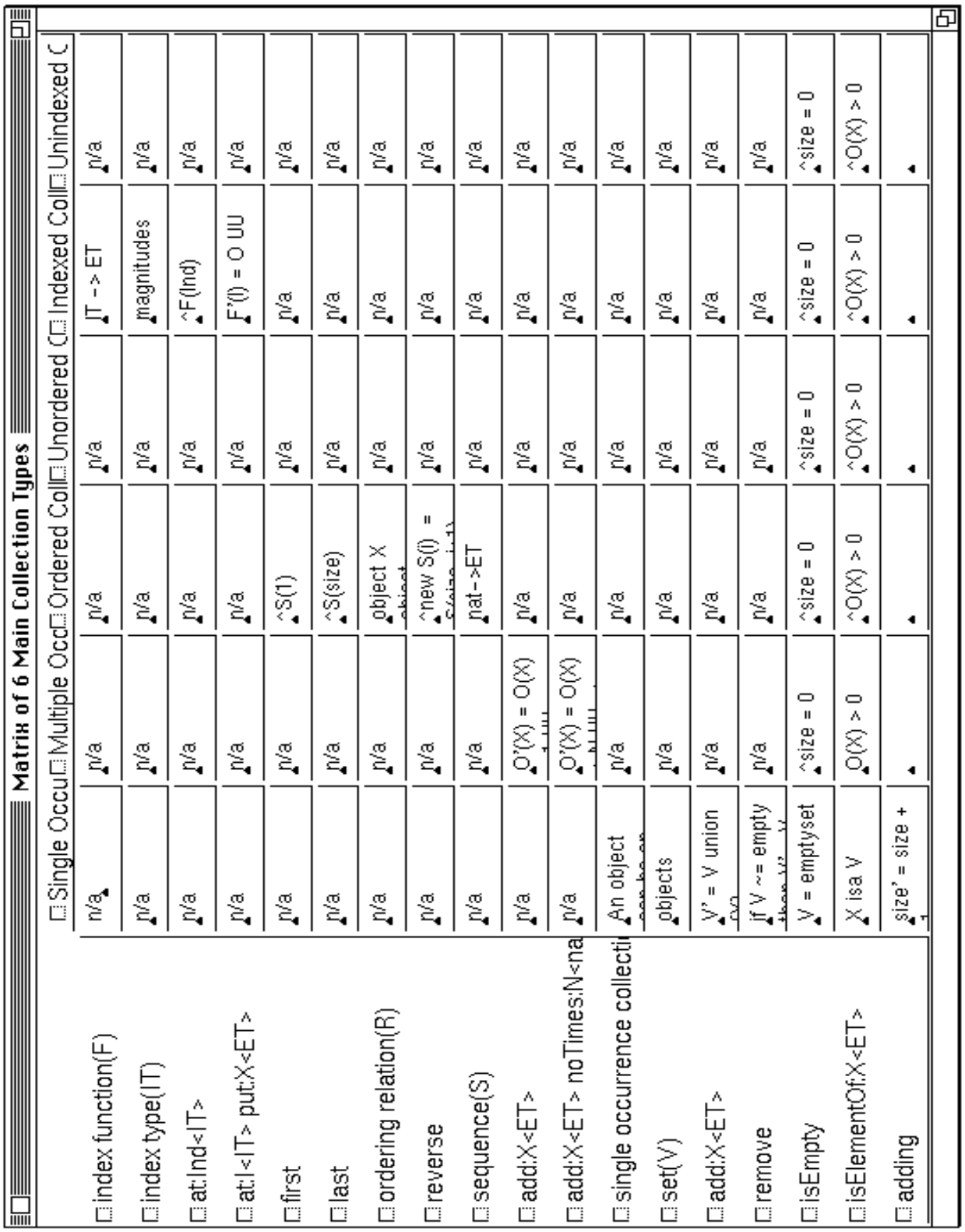

Figure 10: A Property Comparison Matrix

3.5 Semantic Net Graphs

This section describes a common form of browsing: opening a graphical browser on a 'relation' knowledge map. Figure 8 shows the resulting "semantic net" graph. The process the user follows is this:

1. She must previously have opened a browser such as in figure 3, with inheritance hierarchy and statement hierarchy panes.

2. Next, using the mouse, the user selects in these panes a set of concepts of interest and one or more statements (in fact it is the underlying properties that are of interest; e.g., to draw a parts hierarchy one would select the statement for the 'parts' property).

3. Then the "draw graph" command is issued using the menu or a hot key.

4. A result like figure 9 appears, This shows

   - Nodes corresponding to the selected concepts plus other concepts linked to these by the arcs.

   - Arcs corresponding to the statements of the selected properties. These connect the nodes to any other concept that is a value of such a property.

6. Since the graph layout algorithm is not elaborate, the user usually may clean up the display by dragging nodes using the mouse. She can save particular layouts for later recall.

We have found this type of display valuable for providing feedback: For example, software designers frequently want to see so-called entity-relationship diagrams, or finite state diagrams like figure 9. At present, they often draw these by hand, or, for documentation, draw them with draw programs. Of course these and special-purpose design ('case') tools can draw very pretty diagrams, but at the expense of not handling many other knowledge management needs. In CODE4, we seek to integrate such functionality.

3.6 Masks

A mask is a set of conditions that is applied to each concept in a knowledge map as the concept is being considered for display. The mask is either 'true' or 'false' for each concept. Each knowledge map has two masks:

- A *visibility mask* that determines whether the concept will be displayed (true) or hidden (false). The default visibility mask displays the entire map.

- A *selection mask* that determines whether a concept will be highlighted (true) or not. The default mask highlights no concept. Concepts can also be highlighted by 'clicking' on them with the mouse.

The set of conditions in the mask is often very simple, e.g., showing or highlighting the concepts of things whose 'colour' is 'blue'. An expert, however, may create a complex mask combining several conditions into an arbitrary logical expression.

Masks are used for several related purposes:

- To *focus the display* or reduce 'clutter': For example, to only show one or two sub-hierarchies or to show only those concepts that are 'complete' i.e., finished. ('Completeness' would be determined by examining an appropriate 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". One might apply such a mask as a precursor to another operation which then may be applied to the highlighted concepts.

3.7 Document Processor

In many of our applications knowledge bases have been, or could be, built up from information available in documents. CODE4 has a facility to assist with this, shown on the right side in Figure 11. The document appears sentence by sentence in the upper part of the Processor, and the user selects one sentence at a time for processing. Any words not in CODE's dictionary (the list of terms associated with a kb) or the common external dictionary bring up a dialog in which the user defines the part of speech. Compound phrases, which are very common in technical documents, may also be identified. Next, simple rules break the sentence up into useful fragments: every noun and every verb is listed in the middle column, along with the pre and post modifying phrases to the left and right, somewhat like in concordance tools. From this, the user may easily construct a statement to be added to a kb by a series of mouse actions, editing expressions if necessary. We have found that the processor speeds up knowledge capture from documents several times.

The main functions this facility serves are: 1) to discipline the user's thinking so that each noun and verb is given attention (the system keeps track of which have been used, so one can review a document for knowledge not added to the kb); 2) to permit verifying the kb (the system inserts pointers from statements in the kb to the sentences from which they were derived); 3) to eliminate the need to retype many phrases. The user can see the kb additions happening in the browser, open at left in Figure 11 (the screen snap was taken just after adding the statement 'car manufacturers manufacture cars'). At any time, the kb can be browsed to assist in understanding or deciding what to do next.

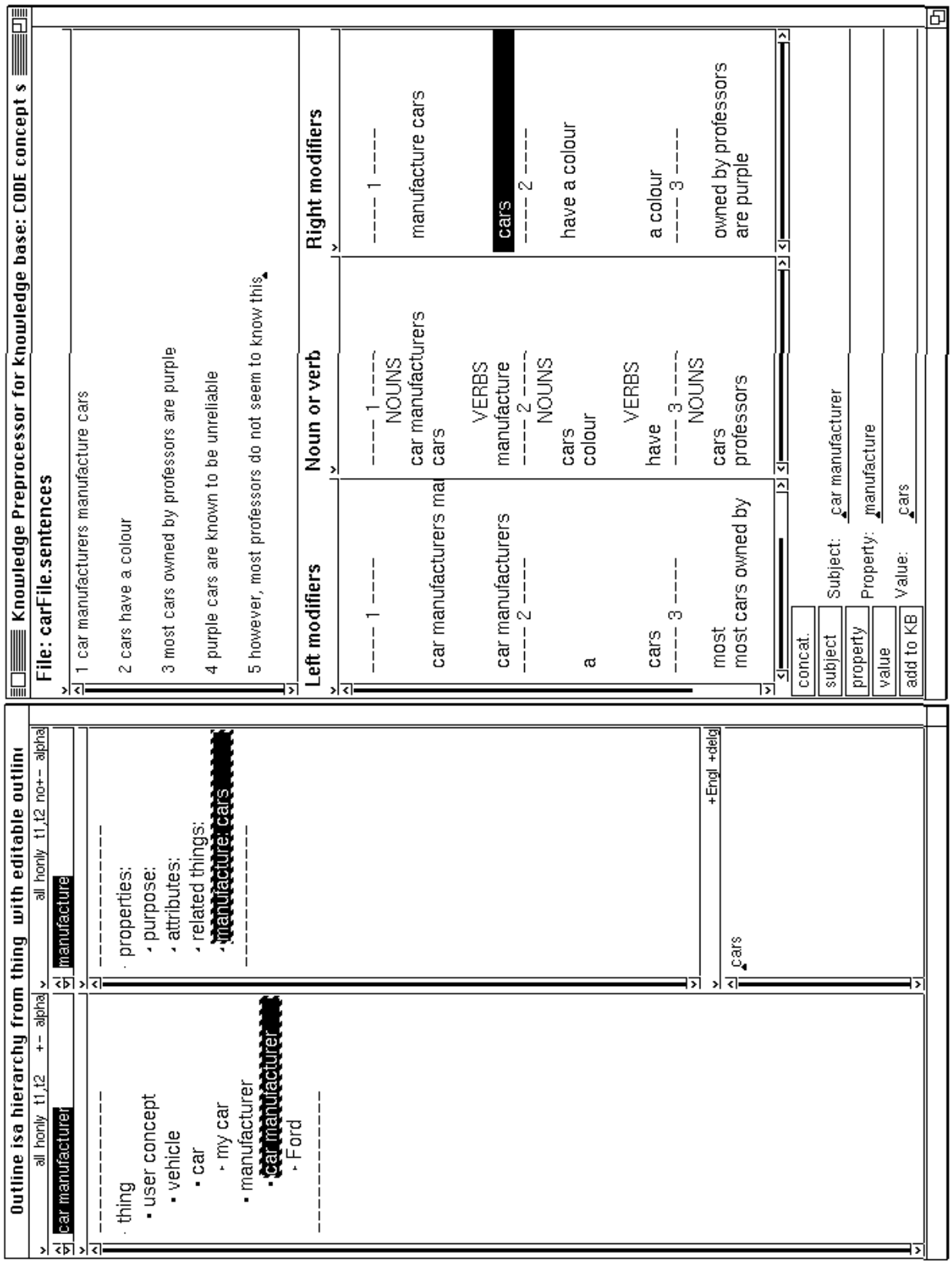

Figure 11: The Document Processor (Knowledge Preprocessor; at left is a normal browser)

3.8 Property Manager

The property manager (which is being designed at the time of writing) addresses a common but difficult conceptual and terminological problem. We have noticed that frequently beginners, and sometimes experts, may create two or more separate properties but intend them to be the same semantically. Distinct properties are created by each execution of the command to create a new property; the name chosen can be the same as an existing name without implying any relationship.

For example, there may be a property ‘length’ on ‘arrays’ and the user could then create another property also called ‘length’ on ‘files’. Or maybe a super of arrays or files already has a 'size' property. These three should be the same. Maybe file and array should have a common super they currently don't have. Thus creating a new property with the same name as an existing one may not be what the user intended: he/she may intend there to be only one ‘size’ property, but that it apply more generally. This usually occurs because the user has forgotten that such a property already existed and/or the other concepts having it are remote from the current one being considered. The common superconcept of these two concepts (‘array’ and ‘file’) may be quite general, such as ‘software object’. Hence there are difficult problems to solve:

- Should ‘size’ be generalized to ‘software object’ - do most software objects have a size?
- Should ‘size’ be otherwise generalized? Does a new concept need to be created for it? If so, how does it relate to ‘software object’?
- Should there be two different properties, one for ‘files’ and one for ‘arrays’, each called ‘size’?
- Should there be two different properties, one for ‘files’ and one for ‘arrays’, but having different names?
- If there are two different properties, should one be a subproperty of the other?

Beginners and even experts experience considerable difficulty with this task, and hence our desire to provide assistance. The most common situation is that the two properties should really be one; distinct but identically-named properties are usually undesirable (we might even want to prevent it, although we would not do this for concepts)[14].

In summary, the property manager is intended to assist the user by:

- Reporting properties that have the same, synonymous, or closely related names.
- Reporting properties that have the same value at their origin (the concept where they are introduced).
- Reporting properties that are hierarchically related, but have the same value.
- Making it easier to see the problem and decide which solution is best.

---

[14] A harder situation to deal with is when the names are different but the properties are intended to be the same ('size' vs 'length') or hierarchically related (e.g. 'spouse' and 'husband' or 'wife'). This would require more sophisticated linguistic knowledge, something we intend to add.

4. Applications

CODE4 and its predecessor CODE2[15] have been used in a number of research and commercial environments, including by Alcoa, Boeing (Bradshaw, Holm et al. 1992), and Bell-Northern Research. CODE4 is also used regularly in graduate courses in knowledge engineering where students practice performing concept analysis on a topic of their choosing. The following sections describe some applications for which CODE4 is currently being used.

4.1 Software Engineering

Software engineering is increasingly being influenced by developments in knowledge engineering (e.g., (Reubenstein and Waters 1991), (Johnson 1992)). We have been experimenting with using CODE4 to capture requirements and design knowledge.

For example, Figure 12 shows a proposed Collection class hierarchy for Smalltalk, i.e., what it would look like if carefully redesigned using CODE4 as a vehicle for describing it both informally and formally. It is well known, e.g., (Cook 1992), that the existing Smalltalk Collection classes have evolved somewhat haphazardly over many years and have a number of anomalies. Hence they are difficult to understand, both because of these anomalies and because the two means of describing them that currently exist, descriptions in textbooks and the Smalltalk browser itself, leave much to be desired. Hence our desire to provide better means of organizing, finding, displaying, and indeed creating such information.

Figure 10 showed a comparison of the six proposed subconcepts of Collection. Here, they can be conveniently compared: the user had asked for a comparison matrix for all subconcepts of Collection and all properties of them, resulting in quite a large matrix. (In more detailed stages of design, probably only certain properties would be displayed.)

[15] CODE3 was a Prolog version abandoned due to insufficient user interface capability.

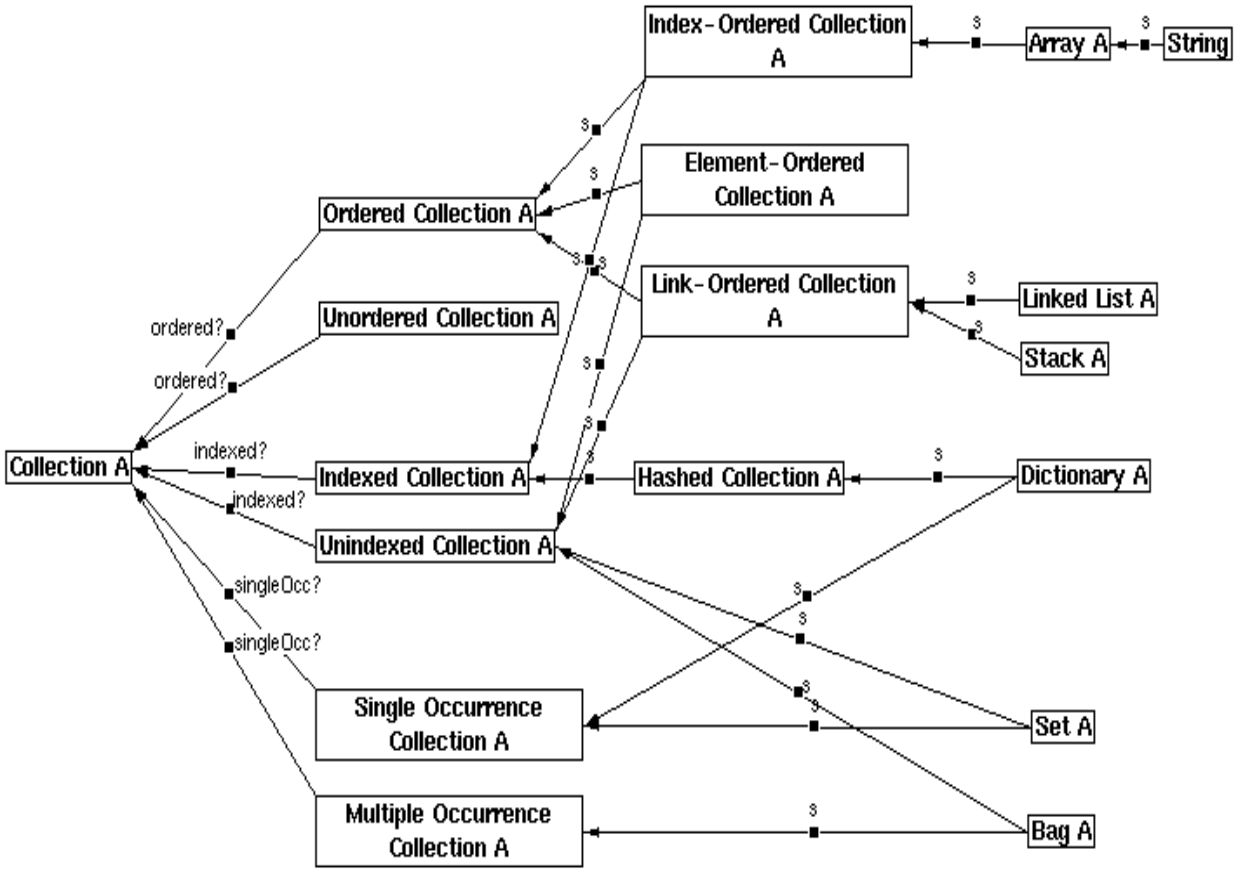

Figure 12: Proposed Collection Classes for Smalltalk

Some of the properties are qualitative and descriptive; others are formal, based on a simple denotational semantics. Given such a description, the creator and others may participate in a structured review of these properties, browsing first the general knowledge (i.e., to see if they agree on Collection or its immediate subconcepts) and, when satisfied with this, browsing the refinement of it in the lower concepts, some of which will correspond to actual classes to be implemented. Thus without ever entering a programming environment or writing a line of code, the conceptual picture can be clarified and agreed upon solely within the knowledge management system. It would be not difficult to automatically create a Smalltalk class hierarchy from these definitions: what is missing is the method bodies which, to be sure, is a lot. These would have to be supplied and somehow verified or tested to assure conformance with the specifications. This is outside the scope of our present research.

The above example illustrates a common design situation where skilled software experts need to communicate amongst themselves. Another common situation involves communication with a client who is not a software expert. Figure 8 is part of a requirements analysis for the well-known Automatic Teller Machine problem, showing the proposed behaviour of the ATM in terms

understandable to the client, presumably some banking personnel[16]. By using CODE4, a proposed design can be specified to an appropriate level of precision and subjected to a rigourous review process by the client. The concepts can be described more exactly than if only natural language documents are used (still the norm today), but CODE4 permits better validation by the client. In this figure, we seek validation by the client of the proposed behaviour of the machine, after having built a kb describing all the parts of the machine, the associated concepts such as customer, bank account, types of transaction, etc. All such knowledge can be validated by the client, using CODE4's facilities to make this process as painless as possible; for example, this diagram has been drawn automatically from the kb.

In (Ghali 1993), a three-*viewpoint* approach was described, which is supported by mechanisms in CODE4 (e.g., one can show only a certain viewpoint, which is knowledge needed by a certain type of user or for a certain purpose).

In the first (*application* or *requirements*) viewpoint, only knowledge about the ATM and banking *per se* is permitted, i.e., only knowledge that banking personnel can understand. This knowledge must be clearly established before the system can be designed. It can be considered *requirements* knowledge, and agreement on it is an essential part of a contract.

In the second viewpoint (*design*), a proposed design is developed, usually within some context (e.g., other reusable designs, or certain design methodologies). For example, a particular object-oriented technique may be used, and perhaps a certain type of hardware or database system would be required to situate the design. Here then, the application concepts are mapped into design concepts, reusing existing designs as much as possible, and integrating descriptions of how the design will interact with all other players at delivery time. This knowledge is of course all in the form of concepts in the kb. Thus a proposed design can be developed reusing existing design knowledge, as in the example above. The banking people cannot understand these technical details of course: their view is limited to the first viewpoint. CODE4 can show either or both viewpoints and maintains links between them. For example, a design class 'bank account' would be linked to the application concept 'bank account' and probably each property of one would be linked (by explicit facet information) to the appropriate property(ies) of the other. One can then easily answer questions such as "why does bank account have three instance variables?" or "how are the two kinds of bank account dealt with in the design?" This viewpoint can be seen as a major generalization and improvement over what is possible using the well-known CRC technique (Wirfs-Brock 1990).

When it is agreed that the design is ready, the third viewpoint, *implementation*, can be driven from the design. Here, a commitment to an actual system is made (i.e., some particular implementation language and operating system) whereas the design viewpoint remains generic[17]. The concepts in this viewpoint correspond to each class and method as they are actually implemented, i.e., a knowledge-based approach to documenting the code. One could (and should) have the KMS system intimately linked to the programming environment, which is particularly easy in the case of programming in Smalltalk (Ghali 1993 describes a first step in this direction).

---

[16] This example is taken from (Ghali 1993).

[17] Such distinctions are not required by CODE4, hence a user might chose to merge the design and implementation viewpoints, or have more than three as appropriate.

Thus the KMS would be used to capture all information about the implementation, both the programmer's comments and automatically accessible information such as the collaboration patterns of classes. Comparing the implementation against the design is thus greatly simplified, and maintenance becomes much easier since the KMS environment is much richer in knowledge content and much easier to explore. For example, if a maintainer did not realize there were, say, five kinds of bank account, he/she might have difficulty understanding some code, and could make a mistake.

4.2 General-purpose Ontologies

Developing ontologies is an important unsolved problem, made all the harder if they are to be shared by diverse users (Skuce and Monarch 1990); (Skuce 1993d), (Gruber 1990, 1993); (Neches, Fikes et al. 1991). They must be understood by many people if there is to be agreement leading to standardization. And without standardization, there can be little effective knowledge sharing and certainly no knowledge base integration.

In any broad domain, the top 50 or 100 concepts and their properties (so-called 'top-level' ontologies) are the most problematic, since lower-level ontologies must conform to them, and these elusive concepts are extremely hard to pin down with any precision (e.g. "thing", "entity", "property", "event", "state", etc). At the moment, every proposed ontology known to the authors looks very different: there is virtually no agreement on what concepts are most general or, more fundamentally, what their properties are, or what to call them. Dictionnairies do not help: definitions of such words are often vague and frequently circular. CODE4 can function as a useful tool for either experimenting with a proposed ontology, or examining one developed elsewhere with a view to critiquing, modifying, or adopting it. The hard part of course is to establish some agreed-upon methodology by which consensus may be reached (Skuce 1993d).

Skuce has developed a top-level ontology for use in CODE kb's, and possibly for shared use. Figure 13 shows a part of this ontology. This ontology was developed over a period of several years by continually adding or modifying concept descriptions based on ideas gleaned from other ontologies (such as Cyc's (Lenat and Guha 1990), or the Penman ontology (Bateman, Kasper et al. 1990)), from the linguistics literature (e.g., (Frawley 1987)), or from the psychology literature (e.g., (Smith and Medlin 1981)).

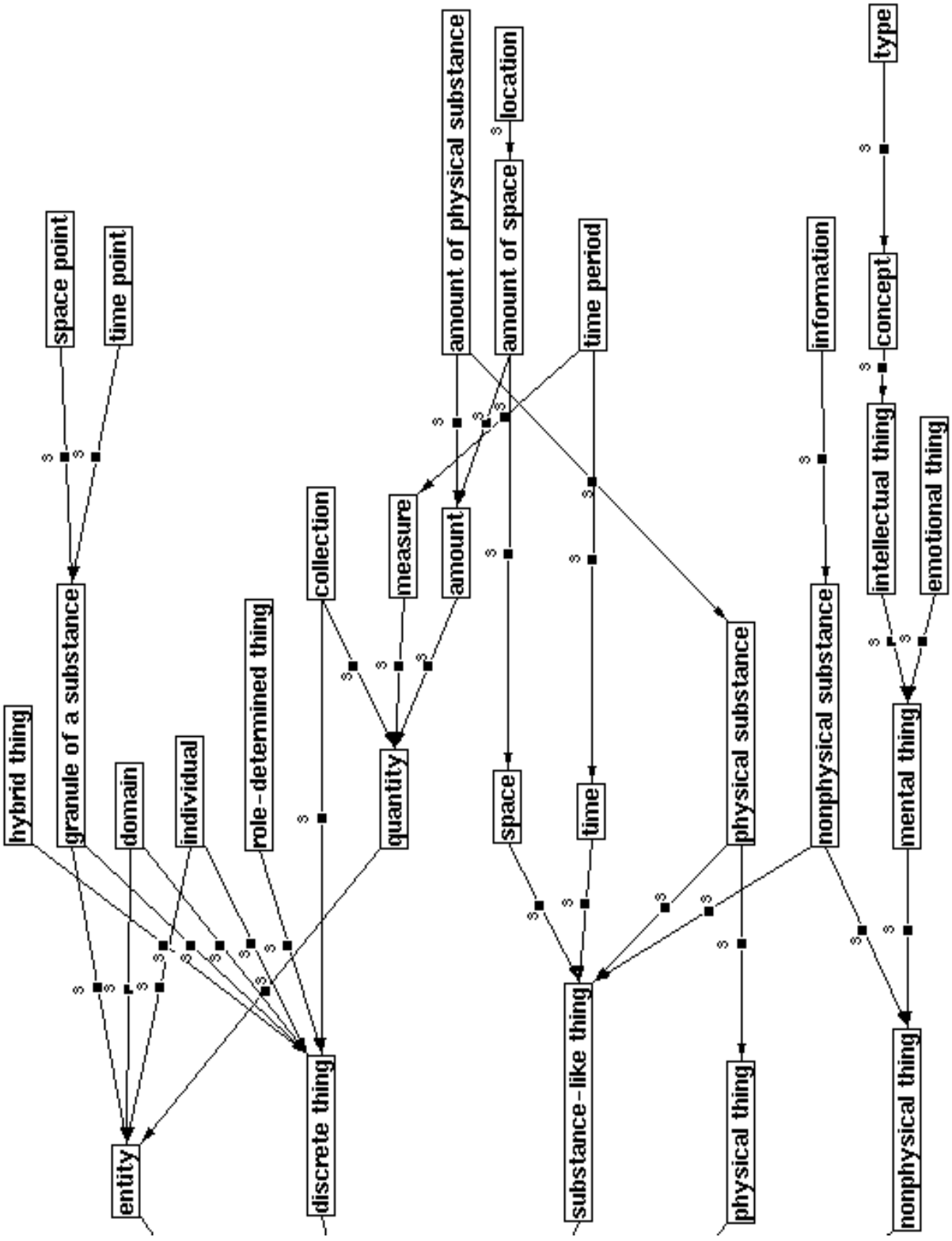

Figure 13: Part of a Proposed Ontology

Using CODE4, it was possible to keep track of changes in a series of kb's, several of which could be viewable at once, or to make changes and explore the ramifications of a change ("what-if" experiments). For example, assuming there is one all-inclusive top-level category (we call it 'thing', having only a few properties such as 'can exist' and 'can be referred to by') then what should be at the next level? Some ontologies offer only two concepts at the second level (e.g., 'entity' and 'property', our current choice); others have many. Yet others claim there is no single "top" concept.

We hope to cooperate closely with other ontology-building efforts (e.g., (Porter, Lester et al. 1988); (Gruber 1990, 1993); (Knight 1993)), importing their ontologies into CODE4 to carefully study them.

The problem of how to reach agreement on shared ontologies, much discussed at the Banff Knowledge Acquisition for Knowledge-based Systems Workshop in 1994, will probably remain unsolved for some time. At bottom, we believe it to be largely a problem in terminology , i.e., in establishing a meaning for a set of agreed-upon terms. This is the problem that terminologists face daily, yet to date, the knowledge engineering world has not made use of their expertise (Meyer and Skuce 1990), (Skuce and Meyer 1991), (Skuce 1993d). We work actively with terminologists in this direction. In our approach, each proposed property and concept term will somehow have to be given as precise a definition as possible, and the chosen terms should be based on empirical evidence, not individual preferences. We are investigating how statistical techniques based on corpus analysis (Aarts and Meijs 1990) may be able to contribute to making this process of establishing word meanings less subjective.

4.3 Terminology

We have just noted the need for links between knowledge engineers and terminologists. In fact, documentors, terminologists, and knowledge engineers have many common problems and needs, mainly, to gather and clarify knowledge, to define the meaning of terms, and to present knowledge in such a way that others can easily make use of it.

Currently these groups of professionals do not communicate as closely as they might. Terminology is sometimes not adequately supported in the documentation process, and outside the documentor's community, documentation is sometimes not appreciated as being very important, despite the fact that it is still our main means of collecting and disseminating knowledge. On the other hand, probably few people working in documentation are aware of developments in knowledge engineering research, while knowledge engineering conferences rarely have papers related to terminology or documentation. Yet similar tasks are faced by documentors, terminologists and knowledge engineers: 1) initially gathering and storing knowledge, 2) validating and perfecting the knowledge, and 3) packaging or formatting it for use. For the former, the final "product" is usually some form of document or terminology bank, so far nearly always only for human use. For the latter, the final product usually requires that the knowledge be embedded in executable software, i.e. not for humans.

We believe that repositories of terminological data (traditionally called *term banks*) are in fact evolving into knowledge bases in that they may contain a large amount of *encyclopedic* knowledge, i.e., knowledge about the concepts per se. For this reason, one component of our research, called *COGNITERM*, is to use CODE4 to build a prototype knowledge-intensive term bank (Meyer, Skuce et al. 1992), i.e. a *terminological knowledge base.*

In this project, several Masters theses ((Eck 1993); (Miller 1992); (Bowker 1992)) have explored how CODE4 could be used to build a kb that would provide more knowledge than a terminologist's basic needs, indeed, going far beyond what is currently available in existing term banks. CODE4 assisted the concept analysis in the chosen domain (optical storage), traditionally

a difficult process for terminologists who must struggle to understand and organize the terminology and concepts in fields unfamiliar to them with little or no help from experts. Hundreds of terms were collected from articles and their meanings explicated by creating concepts for them in CODE4. Such conceptual knowledge is usually missing or very haphazard in the best terminological databases available today. The purely linguistic information (e.g., part-of-speech, spelling variations, example usages, or translations into another language), which is all that most of these databases provide, was also captured in CODE4 by specially-formatted property structures. CODE4's extensive browsing and comparison facilities make it easy for terminologists or anyone else seeking to understand meanings of terms to find the desired information. It is our belief that terminology research and the engineering of knowledge bases, which are almost always distinct fields of research today, will progressively merge in the next decade or so (Skuce and Meyer 1991, 1993).

5. Concluding Remarks

In CODE4 we have attempted to combine some of the most desirable aspects of various types of knowledge representation/acquisition/management systems, and have built upon several years experience using its predecessors. In some other systems, the knowledge representation was the driving concern at the expense of other desirable features; in others, support for knowledge acquisition dominates but with weak knowledge representations, or no support for other knowledge operations such as retrieval. In CODE4, we have tried to balance these desiderata, with the specific bias that we wanted to assist people in managing the kind of knowledge that otherwise would probably be placed in documents.

The knowledge representation itself is a hybrid, combining ideas from frame-based systems, ideas from object-oriented systems, and ideas from hypertext systems. Its semantic behaviour reflects the desire to accommodate the needs of many users who either cannot or prefer not to have to follow a rigid formalism, thereby forsaking some automatic inferencing to gain expressiveness and ease of use. For those who need it, formal syntax and semantics can be incrementally added in the form of ClearTalk rules, mappings into KIF, and the ability to program in Smalltalk. Our experience, involving thousands of concepts created by more than seventy users, has confirmed to our satisfaction that for these applications, such a trade of automatic, logic-based inferencing for expressiveness was the appropriate choice. (Lethbridge and Skuce 1994; Lethbridge 1994) discuss the experiences of some of these users solicited by an extensive questionnaire intended to discover patterns of use and shortcomings in the system.

Some of the most novel innovations in CODE4 are in its user interface features, a *sine qua non* for our vision of a highly interactive KMS. There has not yet been much discussion in the literature of the importance of UI design for such systems, and we plan a paper specifically on this. Our experience has convinced us that, particularly for unskilled users, the UI is the biggest challenge in building this type of KMS, but the existing "UI-building" tools (so-called "visual" programming) are too rudimentary: they lack sufficient support for facilities such as hierarchical or matrix displays and graph-drawing, the core components of our interface.

A serious issue is CODE4's genericity. Certainly, systems designed specifically for a particular application may function better in that application, but do nothing for another. For example, a tool specifically for software development may be superior to CODE4 in doing what it is designed specifically to do (at least in CODE4's present state with no enhancements specifically for this application.) However consider two different applications of KMS technology, for example software development and terminology management. At present, these two applications are seen to be totally independent, and we feel sure that the existing tools that apply to one would be of no use to the other. But CODE4, even in its present generic state, has been demonstrated to be useful *both* to software developers *and* to terminologists, because it addresses the common

knowledge management problems they both have. With relatively little additional programming, features could be added to assist either further. But is there any connection between terminology and software engineering? Absolutely! In fact we believe it is critical, for software engineers ought to use terms precisely, yet usually do not at present. And these are only two possible applications among many. Our point then is that by starting with the generic system, we can in the long run provide knowledge management to a wider audience at less cost than by developing separate systems that have a lot of overlap in functionality or worse, cannot interoperate.

There are two main thrusts to our research at present. First, we intend to work in the direction of adding (possibly external) modules to CODE4 that will permit building kb's semi-automatically from existing texts. The idea is that by scanning large volumes of text on a particular subject, certain statistics can be accumulated that will suggest terms, concepts, properties, to be automatically collected into what might be termed a *proto*-kb. The hope is that then it should take less effort to manually correct this kb than to construct an equivalent one entirely from scratch. It could then be used to construct an actual CODE4 kb semi-automatically. To our knowledge, no one has yet reported on such an undertaking. Second, we intend to explore how CODE4 can be used cooperatively by, e.g., a design team, using possibly interfaces such as Mosaic so that everyone need not have a copy of CODE4 nor become experienced with its complex user interface. An application that ties these two together has been started: building a kb for teaching purposes from the text of a well-known undergraduate computer text, and delivering the kb to students via Mosaic.

Availability of CODE4

CODE4 is available for academic or commercial use under certain conditions. It runs on all standard platforms (that support Smalltalk-80). Contact the authors for more information.

Acknowledgements

Many of the ideas in CODE4 go back to ideas of Yves Beauvillé on earlier versions. Ingrid Meyer and her students made many useful suggestions. Earlier versions of this paper have benefited from comments from her and Peter Clark. He and Jeff Bradshaw have been enthusiastic users and supporters of the research. This research has been supported by Bell Northern Research; Cognos, Inc.; Boeing Corp; the Natural Sciences and Engineering Research Council of Canada, and the URIF program of the Ontario government.

References

Aarts, J. and W. Meijs (1990). *Theory and practice in corpus lingusitics.* Amsterdam, Rodopi.

Anjewierden, A. and J. Weilemaker (1992). Shelley - computer-aided knowledge engineering. *Knowledge Acquisition* **4**, 109-125.

Bateman, J., R. Kasper, J. Moore and R. Whitney (1990). A General Organization of Knowledge for Natural Language Processing: the Penman Upper Model. USC/Information Sciences Institute.

Boose, J. (1988). A Survey of Knowledge Acquisition Techniques and Tools. *Proceedings of the 3rd Knowledge Acquisition Workshop*, Banff, Alberta.

Boose, J., J. Bradshaw, C. Kitto and P. Russo (1990). From ETS to Acquinas: Six Years of Knowledge Acquisition Tool Development. *Proceedings of the 5th Knowledge Acquisition Workshop*, Banff, Alberta.

Bowker, L. (1992). Guidelines For Handling Multidimensionality In A Terminological Knowledge Base. Masters thesis, School of Translators and Interpreters, University of Ottawa.

Bradshaw, J., P. Holm, O. Kipersztok and T. Nguyen (1992). eQuality: A Knowledge Acquisition Tool for Process Management. *FLAIRS 92*, Fort Lauderdale, Florida.

Cook, W. (1992). Interfaces and Specifications for the Smalltalk-80 Collection Classes. *OOPSLA* 92.

Eck, K. (1993). Bringing Aristotle Into the Twentieth Century: Definition-Oriented Concept Analysis in a Terminological Knowledge Base. Masters thesis, School of Translators and Interpreters, University of Ottawa.

Frawley, W. (1992). *Linguistic Semantics.* Lawrence Erlbaum, Hillsdale, NJ.

Gaines, B. (1987). An Overview of Knowledge Acquisition and Transfer. International Journal of Man-Machine Studies. **26**, 453-472.

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. Masters thesis, Department of Computer Science, University of Ottawa.

Gruber, T. (1990). *The Development of Large Shared Knowledge Bases: Collaborative Activities at Stanford*. Knowledge Systems Laboratory, Stanford University.

Gruber, T. (1993). A Translation Approach To Portable Ontology Specifications. *Knowledge Acquisition* , **5**, 199-220.

Johnson, W., Feather, M., and Harris, D. (1992). Representation and Presentation of Requirements Knowledge. *IEEE Trans. SE ,* **18** (Oct).

Klinker, G., D. Marques and J. McDermott (1993). The Active Glossary: taking integration seriously. *Knowledge Acquisition* , **5**: 173.

Knight, K. (1993). Building a Large Ontology for Machine Translation. *Proc. ARPA Workshop on Human Language Technology.*

Kobsa, A. (1991). Utilizing Knowledge: The Components of The SB-ONE Knowledge Representation Workbench. *Principles of Semantic Networks.* Los Angeles: Morgan Kaufman. 457-486.

Lenat, D. and R. Guha (1990). *Building Large Knowledge Based Systems*. Reading, MA: Addison Wesley.

Lethbridge, Timothy. (1994) *Practical Techniques for Organizing and Measuring Knowledge. PhD dissertation, Department of Computer Science,* University of Ottawa.

Lethbridge, T. and D. Skuce (1992). Informality in Knowledge Exchange. *AAAI-92 Workshop on Knowledge Representation Aspects of Knowledge Acquistion*. San Jose, CA, pp. 10.

Lethbridge, T. and D. Skuce (1994). Knowledge Base Metrics and Informality: User Studies with CODE4. *8th Knowledge Acquisition for Knowledge-based Systems Workshop.* Banff, Alberta.

Meyer, I. and D. Skuce (1990). Computer Assisted Conceptual Analysis for Terminology: a Framework for Technological and Methodological Research. *4th International Congress of EURALEX*, Malaga.

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.

Miller, D. (1992). Toward Knowledge-Base Systems for Translators. Masters thesis, School of Translators and Interpreters, 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.

Myers, B., D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish and P. Marchal (1990). Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, **23**(11 (Nov)): 71-85.

Neches, R., R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator and W. Swartout (1991). Enabling Technology for Knowledge Sharing. *AI Magazine*, Fall 1991: 36-55.

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.

Reubenstein, H. B. and R. C. Waters (1991). The Requirements Apprentice: Automated Assistance for Requirements Acquisition. *IEEE Transactions on Software Engineering*, **17**(3): 226-240.

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, Alberta.

Shipman, F. M. (1992). *Supporting Knowldge-base Evolution using Multiple Degrees of Formality*. Dept. of Computer Science, University of Colorado at Boulder.

Shipman, F. M. (1993). *Formality Considered Harmful: Experiences, Emerging Themes, and Directions*. Dept. of Computer Science, University of Colorado at Boulder.

Skuce, D. (1989). A Generic Knowledge Acquisition Environment Integrating Natural Language and Logic. *IJCAI Workshop on Knowledge Acquisition*, Detroit.

Skuce, D. (1993a). A Multifunctional Knowledge Management System. *Knowledge Acquisition*, **5**, 305.

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.

Skuce, D. (1993d). Your Thing Is Not The Same As My Thing: Reaching Agreeement On Shared Ontologies. *International Conference on Formal Ontology in Conceptual Analysis and Knowledge Representation*, Padova.

Skuce, D. and I. Meyer (1991). Terminology and Knowledge Acquisition: Exploring a Symbiotic Relationship. *6th Knowledge Acquisition for Knowledge Based Systems Workshop*, Banff, Alberta.

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, Alberta.

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, Alberta.

Smith, E. and D. Medlin (1981). *Categories and Concepts*. Cambridge, MA: Harvard University Press.

Ungar, D., Smith, R., Chambers, C., Holzle, U. (1992). Object, Message, and Performance: How They Coexist in Self. *Computer,* Oct: 53-64.

Wirfs-Brock, R., Wilkerson, B., and Weiner, L. (1990) *Designing Object-Oriented Software*. Englewood Cliffs NJ, Prentice-Hall.