CODE4: A MULTIFUNCTIONAL KNOWLEDGE MANAGEMENT SYSTEM

Doug Skuce and 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, and 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 interactive knowledge management. The user interface, absolutely 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 and was motivated by experiences in the mid-eighties using state-of-the art expert system tools for capturing knowledge about software. An earlier version of the system, which had a similar philosophy but was much simpler, was described in (Skuce 1993a). 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. 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
- disseminating, transfering, sharing, etc.

of knowledge. The 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. 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 repetory 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 analogical 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 methodology or representation. Large 'integrated' CASE tools provide many facilities, but force you to follow a particular methodology.

The main problems our research addresses are:

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 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) and (Skuce 1993a). CODE4 is programmed in Smalltalk-80 and hence runs without modification on all major platforms.

1.2 Design Philosophy

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 manage knowledge rapidly. The user should be able to jump between tasks at any time.
- A typical desktop-scale environment 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. The experienced user should be able to define new kinds of inferencing.
- 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.

1.3 Related Representations and Systems

To give a better idea of some of our influences, we first note several systems that have some of the features of a KMS that we desire. Most of these have been *knowledge acquisition* systems (Boose 1988), (Gaines 1987). Two of these are somewhat closer to our view of a KMS than most others. KEATS (Motta, Eisenstadt et al. 1988), (Motta, Rajan et al. 1991) is intended for building expert system applications. It uses a frame knowledge representation and has support for natural language. SB-ONE (Kobsa 1991) is also intended for knowledge engineering, with emphasis on building systems that combine both expert system and natural language capabilities. Its knowledge representation is based on KL-ONE, which is not compatible with our philosophy with respect to inferencing and formality.

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. It is a very large and complex system, and assumes that knowledge enterers are highly trained. The Botany Knowledge Base Project and its system, KM (Porter, Lester et al. 1988), is perhaps the closest in spirit to the CODE4 project. It focuses on capturing the kind of knowledge found in university biology textbooks and is intended to be used mainly as a *source* for students.

Although not a direct influence on the early development of CODE4, we would be amiss not to 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 type hierarchies). Both KIF and Ontolingua were designed as stand-alone textual languages, not tools. CODE4 on the other hand was designed as a tool, not a language and this has resulted in an emphasis on abstractions that maximize *usability* .

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*.. 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 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.

A concept represents a 'thing': either the collective idea of a set 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 entities that are

---

[1] Or inside a brain or 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 an 'object', however English speakers use the word some*thing* to refer to anything: e.g. actions: 'We must do something' or properties 'Something bothered me about that man'.

inherited by concepts. In CODE4-KR, *properties* perform this role, but *properties are just another kind of concept!*

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 we use the term 'statement', because the notion is close to the linguistic notion of a statement: a concept is the subject, and the property is encoded in the predicate. Thus a CODE4-KR statement has a subject (some concept), a predicate (some property) and usually more. Statements are discussed in detail below; for now the important idea is that *statements, too, are concepts* .

By virtue of being full-fledged concepts, properties and 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 concepts do in general. CODE4-KR uses two other special classes of concepts called 'terms' and 'metaconcepts' are described in sections 2.7 and 2.8

Any use of these special concepts is represented as an *instance*; i.e. they are instances of the four primitive type concepts that must exist in each knowledge base. 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 all 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'.

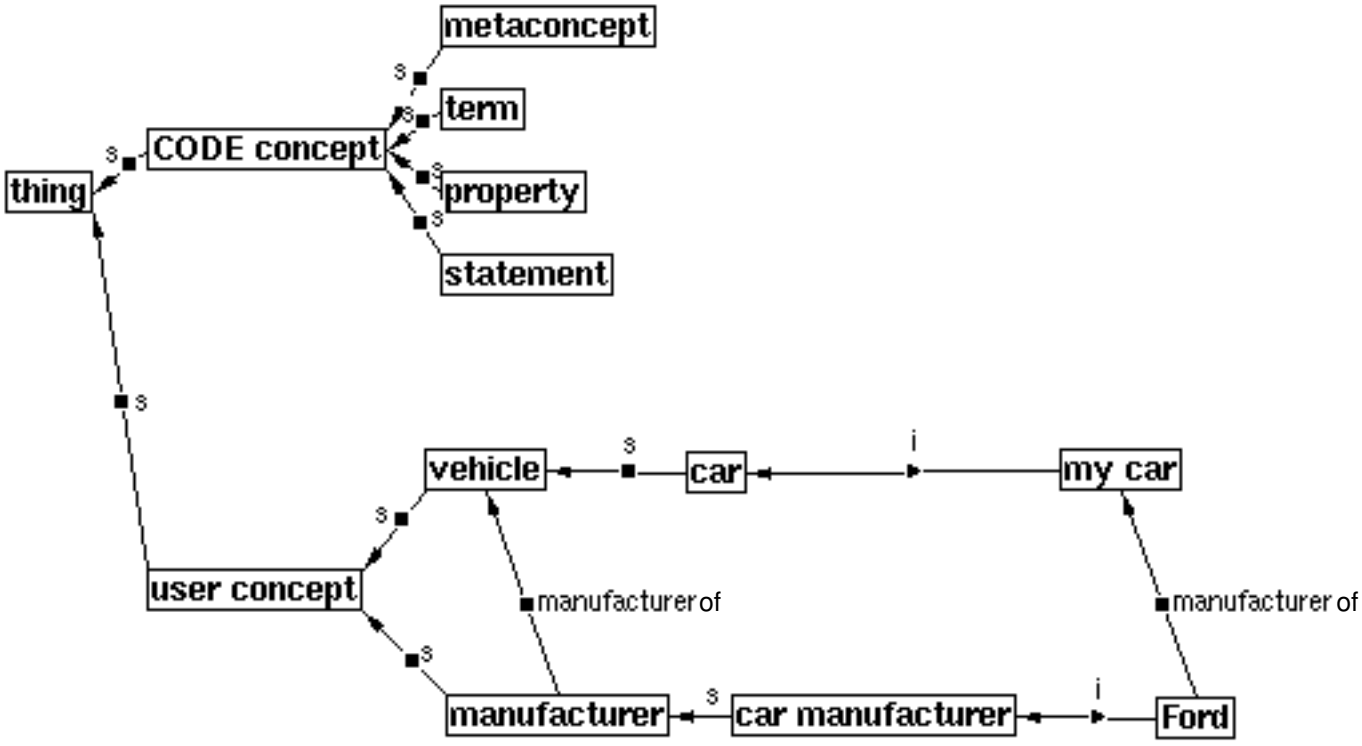

Figure 1: An inheritance hierarchy as displayed by CODE4's user interface, showing a sub-hierarchy of CODE4 primitive concepts (top) and a sub-hierarchy of user-entered concepts (bottom). Users typically use user interface capabilities to hide the primitive concepts. s-links are superconcept links; i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

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 users define, whereas special concepts have certain primitive properties. 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-KR's "knowledge of X". The *meaning* of the concept is determined by these statements.

In Figure 1, a 'manufacturer' is stated to be a manufacturer of vehicles, and this 'manufacturer-of' property is refined for 'Ford'. Note how the display has been designed to make it easy to see various 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.

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

2.2 The Concept, Property, Statement and Relation Hierarchies

Statements link CODE4-KR concepts into a complex network. Out of this network we can extract 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 most can be general directed graphs. These hierarchies form the basis for knowledge maps which are discussed in section 3.3.

Below we describe 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 treated less importantly; relation 'hierarchies' (semantic nets) are not a new idea either, but CODE4-KR's treatment of them uniformly with other 'hierarchies' has advantages for ease of use. Facet hierarchies, a CODE4-KR innovation, are introduced here and described in more detail in section 2.3.

The Inheritance Hierarchy: All concepts participate in the 'inheritance' or 'isa' hierarchy[5], including "empty" ones: one 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(s)). 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). 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'

---

[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.

implies that 'X moves'. No inheritance occurs in the property hierarchy because this would mean that "properties of properties" inherit, and we find few that normally do. (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 make use of this.

Statement Hierarchies: Each concept inherits a sub-hierarchy of the property hierarchy. The statements formed from this sub-hierarchy become a 'statement' hierarchy. A knowledge base has one property hierarchy but many derived statement hierarchies (one for each concept). Systems that lack this capability typically have long, flat and unstructured lists of slots associated with each frame. Figure 2 contains an example statement hierarchy.

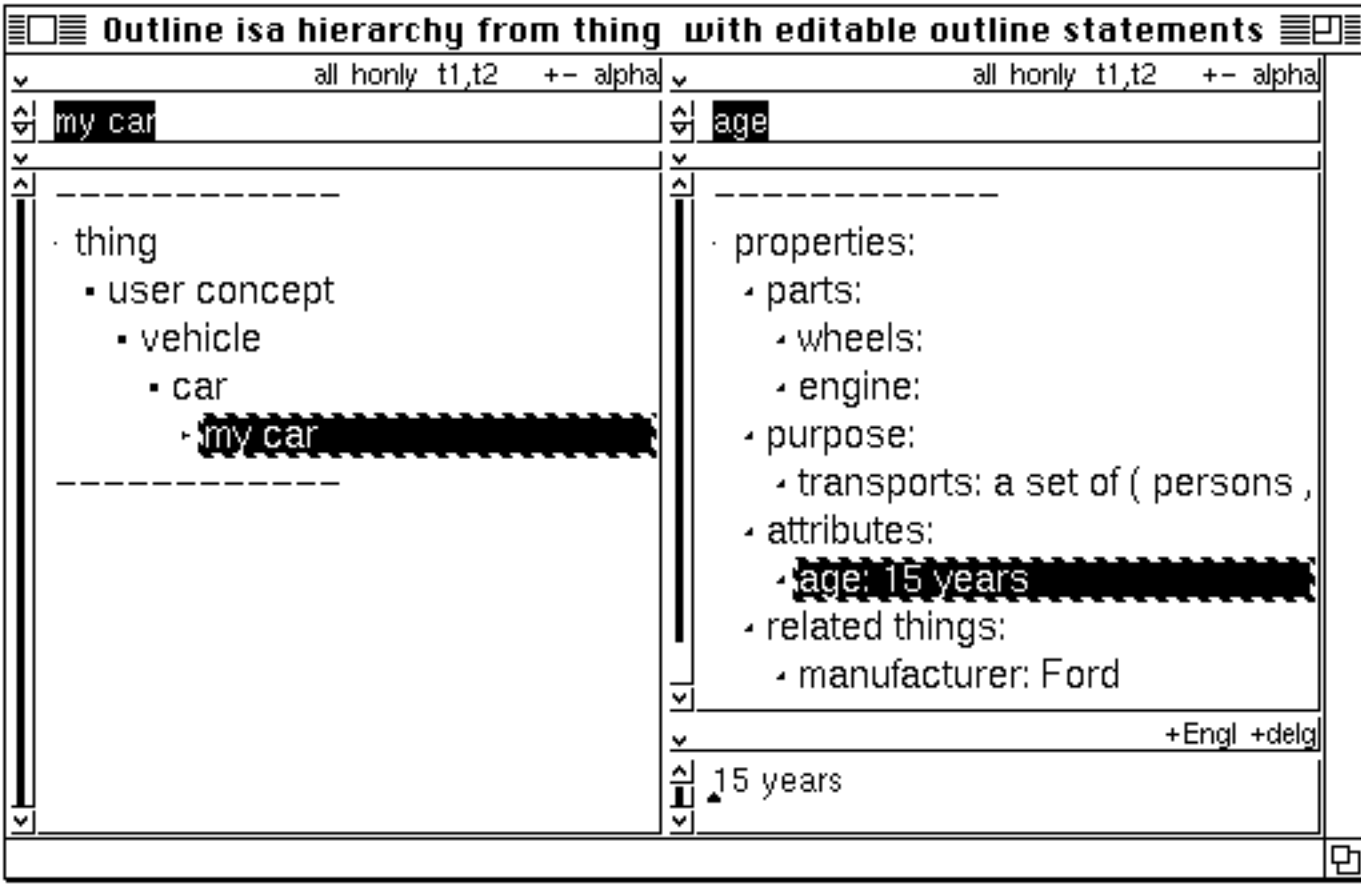

Figure 2: Outline views of an inheritance hierarchy (left) and a statement hierarchy (right) of statements of 'my car'. Components and functionality of the browser are discussed in section 3. The value for the age statement appears in the bottom pane.

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-KR, any relation can be defined thus, assuming it make sense. CODE4 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[6]. There are three main types: 1) Facets corresponding to noun complements following the predicate. 2) Facets for additional information that would be part of a sentence, such as modal, quantificational, temporal and adverbial modification; and 3) Documentation or background information, such as the source and date of the statement.

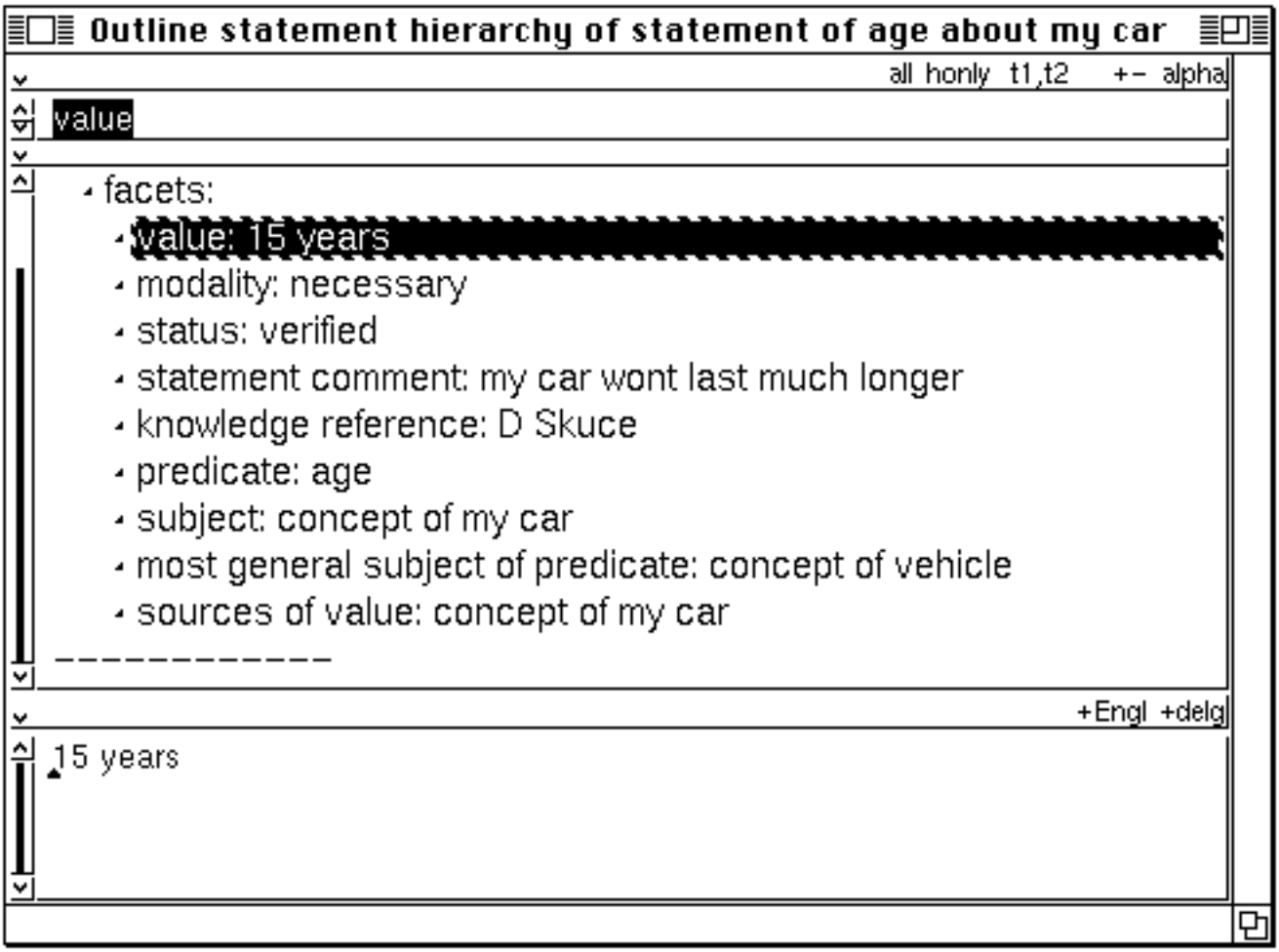

Figure 3: Facets for the statement about 'age' of 'my car'

Figure 3 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"[7]. The value and modality facets par-

[6] Being 'secondary' doesn't in any way reduce the fact that facets are statements, and hence full-fledged concepts.

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

ticipate in this minimal version of the statement. The other facets would be rendered as separate statements about this statement.

Properties can be viewed as logical predicates; most have binary or higher arity, i.e. statements (facets) using them require a value[8]. The value facet automatically inherits; others do not unless specified, though we often make modality inherit. Facets when treated as predicates are almost always binary; i.e. facets always take a statement as their first argument (the subject) and either a reference to other concept(s) or other value as their second argument, termed the 'facet value'[9] (see section 2.5).

A frequently used facet is the *comment*, which is purely informal, i.e. for human processing only. Others include *status* (whether the statement has been verified by someone) and *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). Other facet types are available, such as:

Modality: We find it essential to be able to explicitly record modal information, such as whether a statement is a) necessarily true (in 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 it 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.

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 the ClearTalk 'both of' (a logical 'and' for noun phrases) primitive value combining function:

pets
eat: expensive food

cats:
eat: fish

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

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

[8] 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*.

[9] Facet values are not to be confused with the value of the subject statement, which is a facet whose value is the statement value, another name for the direct object linguistically. All other facets also have values, but these are not "values of the statement".

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 (although one is never forced to use them):

- 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, at least by the user.
- They facilitate consistency checking. For example, if the value of the 'children' property for the concept 'person' was (incorrectly) stated to be 'a set of 0 or more children'[10] (ClearTalk), 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'.
- 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[11]. There are several other ways of creating such values, e.g. editing a relation knowledge map 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 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 in order to a) permit users unfamiliar with Smalltalk to write blocks and b) ensure that CODE4 is not too dependent on the semantics of Smalltalk.

Such blocks give the following kinds of functionality:

- The ability to define values as functions of the values of other statements (we call this 'delegation')
- 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 (all values on display are always recomputed).
- 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).

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).

---

[10] 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.

[11] We intend to enhance the coverage.

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
        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]
```

2.7 Terms

CODE4-KR treats the names of concepts (terms) as full-fledged concepts themselves. An instance of the primitive type 'term' is created automatically whenever a user enters a name for a concept or property.

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.

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). Terms have properties such as 'part-of-speech', 'plural' or 'French equivalent'. As far as we know, amongst knowledge acquisition systems, only the Active Glossary system (Klinker, Marques et al. 1993) has treated terms as seriously.

2.8 Metaconcepts

When representing knowledge, it is frequently necessary to describe properties not of the thing a concept represents, but of the concept *itself* which is different from the thing. Example 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 a concept (as opposed to a thing), CODE4 automatically creates (if 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.

CODE4-KR has a uniform rule that *all* properties inherit. It attaches to metaconcepts all properties. that would in other systems be specially 'tagged' as non-inheriting. Since metaconcepts are instance concepts, no property inherits from them (although block computation can be used to give such an *appearance*).

2.9 Persistent Storage; CODE4 as Knowledge Server

All operations on the CODE4 knowledge base are performed using a limited set of well-defined commands. The commands are partitioned into two major sets: modifiers and navigators..

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. CODE4's *knowledge map* layer, 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. CKB is used for two major purposes:

- **For persistent storage**: When knowledge is saved to disk, CODE4 generates the minimal set of basic modifier 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 two purposes to connect CODE4 to an expert system and to a module that translates an Ontolingua subset into CKB.

We intend to develop mechanisms using the CKB interpreter further: We would like to permit "groupware" use of CODE4, wherein several users, using separate CODE4 systems, can dynamically interact with the same knowledge base under the control of a master CODE4 server. 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.

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, while intellectually appealing, may result in a rigidity that eliminates potential applications.

Nevertheless, in the near future we plan to specify CODE4-KR's semantics formally 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. Major research issues when defining a formal semantics for CODE4-KR include dealing with informal values, the independence of concepts from names and English-like rules.

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 or 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.

CODE4 announces in the feedback panel a number of common semantic errors that users make on the fly, for example, an attempt to delete a concept that is the origin of one or more properties (which would be lost if nothing were done about it). The user is offered several easy 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 property 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'. 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.

3.3 Knowledge Maps

A *knowledge map* is a software abstraction that allows allows for the manipulation of a network of concepts. The knowledge map defines the network in terms of some starting concepts and some relations that recursively relate the starting concepts to other concepts. The word 'map' is used instead of 'directed graph' which may be preferred by some mathematicians to prevent confusion with the graphs drawn by the user interface (although these graphs use knowledge maps, other user interface components use knowledge maps as well.).

An example is a knowledge map that displays the entire 'isa hierarchy'. Its starting concept is the top concept, 'thing'. Its relation is the 'subconcept' relation. A knowledge map that displayed a subtree of the isa hierarchy would have a different starting concept. A knowledge map that displayed a finite state machine might have several starting states and use the 'outgoing transition' relation. Knowledge maps are implemented for the kinds of 'hierarchies' described in section 2.2.

The knowledge map interface presents a simple set of commands for navigating around the nodes; adding, moving and deleting nodes and links; 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 the kind of knowledge map.

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

---

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

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 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, deleting etc. Individual concepts may be selected for renaming. 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.

All kinds of browsers can be dynamically chained so that the what is selected in one dynamically determines the contents of the next. In fact, the common concept-property browser (figure 2) is a compound browser where the selected concept (in the left pane) determines which statements are shown on the right.

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 2 and 3 (in section 2) are examples of such browsers..

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, 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 5, taken from (Ghali 1993).

Property Matrix Browsers: We have found that many questions to which the user seeks answers require 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 6, 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.

3.5 Masks

A mask is a set of conditions that is applied each concept in a knowledge map as the concept is being prepared 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.

[13] This example is discussed under "Applications"

- 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.

3.6 Document Processor

In many of our applications, knowledge bases have been, or could be, built up from information available, at least partly, in documents. CODE4 has a facility to assist with this, shown on the right side in Figure 7. The document appears sentence by sentence in the upper part of the Processor, and the user selects sentences 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 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 construct a statement to be added to a KB by a series of mouse actions, editing expressions if necessary. This facility 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 7 (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.

3.7 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, will create two or more separate properties but meant 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 'size' on 'arrays' and the user can then create another one also called 'size' on 'files'. 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 the problem here is to choose between:

- Should 'size' be generalized to 'software object' - do most software objects have a size?
- Should 'size' be generalized so that it inherits to both? 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, should one be a subproperty of the other?

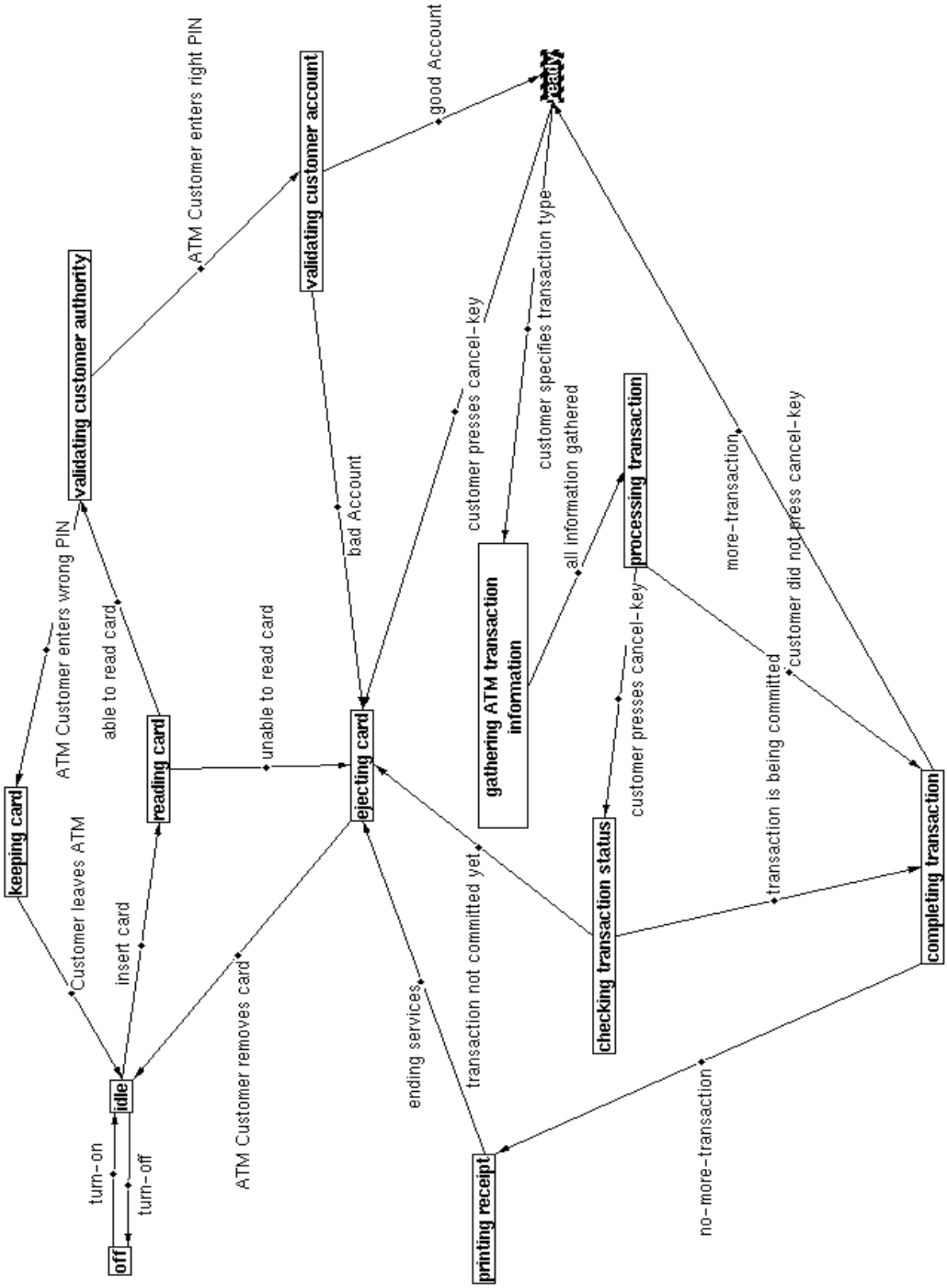

Figure 5: A finite state diagram capturing customer requirements. A graphical view of a relation knowledge map.

Figure 6: A Property Comparison Matrix

Figure 7: The Document Processor (at right; at left is a normal browser)

Beginners and even experts experience considerable difficulty with this task, and hence our desire to provide assistance. The most common situation is that two properties should be one; the same name is chosen and distinct but identically-named properties are usually undesirable (we might even want to prevent it, although we would not do this for concepts)[14]. Hence 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 (concept where they are introduced).
- Reporting properties that are hierarchically related, but having the same value.
- Making it easier to see the problem and decide which solution is best.

4. APPLICATIONS

CODE4 and its predecessor CODE2[15] have been used in a number of research and commercial environments, including by Alcoa, Boeing, and Bell-Northern Research. Here are some applications that CODE4 is currently being used in the following applications:

A software engineer's "assistant", focussing on the conceptual stages of software design: Software engineering is increasingly being influenced by developments in knowledge engineering (e.g. (Reubenstein and Waters 1991), (Johnson et al. 1992)). We have been experimenting with using CODE4 to capture requirements and design knowledge. For example, Figure 8 shows a proposed Collection class hierarchy for Smalltalk, i.e. what it would look like if CODE4 were used as a vehicle for carefully describing it both informally and formally. It is well known, e.g. (Cook 1992), that the existing Collection classes, which evolved somewhat haphazardly over many years, have a number of anomalies and are difficult to understand. Figure 6 shows a comparison matrix of the subclasses of Collection, enabling design proposals to be evaluated.

Figure 5 is part of a requirements analysis for the well-known Automatic Teller Machine problem, showing the proposed behaviour as a kind of semantic net in terms understandable to a banker. By using CODE4, a proposed design could be specified and confirmed by a client at a level he/she can understand. When the informal design is accepted, one could add formal properties for checking using external systems. When it is agreed that the design is ready, implementation could be driven and documented by having the CODE4 system intimately linked to the programming environment, particularly easy in the case of programming in Smalltalk. CODE4 then would be used to capture all information about the implementation, both programmer's comments and automatically accessible information such as class collaboration patterns. Visual inspection to compare the implementation against the design thus is greatly simplified. (Ghali 1993 explores these issues.) We have not yet persued more "automatic" programming since that is not our primary interest.

Developing general purpose ontologies for knowledge sharing: Developing ontologies is an important unsolved problem, made all the harder if they are to be shared by diverse users (Skuce and Monarch 1990); (Gruber 1990); (Neches, Fikes et al. 1991); (Skuce 1993b). They must be examined and understood by many people if there is to be agreement leading to standardization. CODE4 can function as a useful tool for either experimenting with a proposed ontology, or examining one developed elsewhere with a view to critiquing or adopting it. Skuce has spent considerable time developing a top-level ontology[16]. We also hope to cooperate closely with other

---

[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.

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

[16] The top 50 or 100 concepts are the most problematic, since all else must conform to them, and these elusive concepts are extremely hard to pin down and agree on with any precision.

ontology-building efforts (e.g. (Porter, Lester et al. 1988); (Knight 1993)), importing their ontologies into CODE4 and using its highly developed UI features to carefully study them.

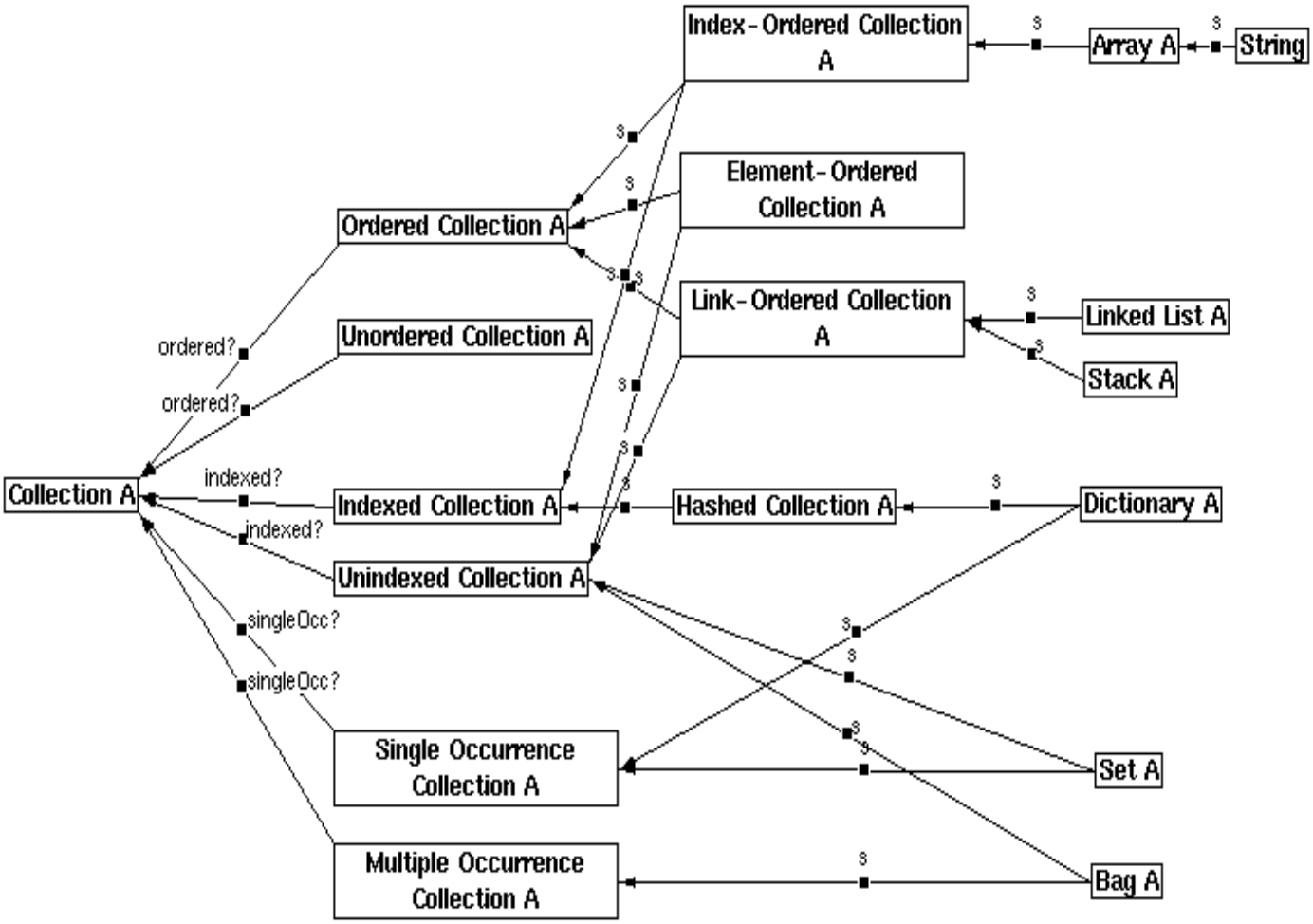

Figure 8: Proposed Collection Classes for Smalltalk

Developing terminology databases: We have worked closely for a number of years with terminologists, who share many of the same interests and problems as knowledge engineers (Skuce and Meyer 1991) (Eck, 1993). Our work assumes that repositories of terminological data (traditionally called *term banks*) are in fact evolving into knowledge bases in that they contain a large amount of *encyclopedic* knowledge, i.e. knowledge about the concepts per SE. For this reason, one component of our research has been to use CODE4 to demonstrate a prototype, knowledge-intensive term bank (Meyer, Skuce et al. 1992), called *COGNITERM*.

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 of these 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. In CODE4, we have tried to balance

these desiderata. One specific bias, probably absent from the minds of many other system designers, is that we want 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) discusses 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, and existing "UI-building" tools do not help: 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.)

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. Jeff Bradshaw has been an enthusiastic user and supporter 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

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

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,

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.

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, University of Ottawa, Dept of Computer Science.

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**(18 (Oct)): 853869.

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, T. and D. Skuce (1992). Informality in Knowledge Exchange. AAAI-92 Workshop on Knowledge Representation Aspects of Knowledge Acquisition. 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,

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,

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**(xx): 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,

Shipman, F. M. (1993). Formality Considered Harmful: Experiences, Emerging Themes, and Directions. Dept. of CS, Univ. 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). Your thing is not the same as my thing: reaching agreement 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,

Skuce, D. and I. Monarch (1990). Ontological Issues in Knowledge Base Design: Some Problems and Suggestions. 5th Knowledge Acquisition for Knowledge Based Systems Workshop, Banff,

Skuce, D., S. Wang and Y. Beauvillé (1989). A Generic Knowledge Acquisition Environment for Conceptual and Ontological Analysis. 4th Knowledge Acquisition for Knowledge Based Systems Workshop, Banff,

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