Metrics for Concept-Oriented Knowledge Bases

Timothy C. Lethbridge

School of Information Technology and Engineering
150 Louis Pasteur St., University of Ottawa
Ottawa, Canada, K1N 6N5
tcl@csi.uottawa.ca

Abstract

Metrics are widely researched and used in software engineering; however there is little analogous work in the field of knowledge engineering. In other words, there are no widely-known metrics that the developers of knowledge bases can use to monitor and improve their work. In this paper we adapt the GQM (Goals-Questions-Metrics) methodology that is used to select and develop software metrics. We use the methodology to develop a series of metrics that measure the size and complexity of concept-oriented knowledge bases. Two of the metrics measure raw size; seven measure various aspects of complexity on scales of 0 to 1, and are shown to be largely independent of each other. The remaining three are compound metrics that combine aspects of the other nine in an attempt to measure the overall ‘difficulty’ or ‘complexity’ of a knowledge base. The metrics have been implemented and tested in the context of a knowledge management system called CODE4.

1. Introduction

There has been substantial research into measuring the work products of software engineers. Measurements are taken with several goals in mind, including:

- Predicting and monitoring cost and development time.
- Determining levels of, and improvement in productivity.
- Ascertaining levels of complexity and other aspects of quality.

The same goals can also be used to drive the measurement of the work products of knowledge engineers; however, little work has been done in this area. This paper takes a step towards rectifying this situation: We present some ideas about how we can measure knowledge bases.

1.1 The context of this research

The overall goal of our research program is to develop practical tools and techniques for *knowledge management*. We define knowledge management as a multi-functional process that includes: acquiring, representing, organizing, reviewing, presenting and disseminating knowledge. This term has a somewhat wider meaning than *knowledge engineering*, in that the latter tends not to be concerned with the presentation and dissemination of knowledge.

With this goal in mind, we have developed a series of knowledge management systems that we called Conceptually Oriented Design/Description Environments (CODE). Each successively numbered version of CODE represents a complete redesign. The most widely used versions of the system have been CODE2 [1] and CODE4 [2] [3].

All the CODE systems have been developed with the intent that experts from a variety of fields should be able to build their own knowledge bases. As such, a great deal of emphasis was placed on the following:

- Developing a highly usable interface for the systems. To achieve this we borrowed ideas from such technologies as spreadsheets, software browsers and hypertext.
- Designing a knowledge representation that is flexible and permits incremental formalization. This is important because many users want to sketch their knowledge and massage it many times before adding details involving formal logic.

Numerous people have built knowledge bases using the CODE systems [4] [5] [6]. In order to study how effectively we are achieving our goals of making knowledge management practical, we need to be able to study quantitatively the work products: i.e. we need metrics for knowledge bases.

The metrics discussed later in this paper were developed and implemented in CODE4. Section 6 discusses some tests where we applied the metrics to a series of real knowledge bases.

1.2 The kind of knowledge bases to be measured

This study is concerned with *concept-oriented* (frame-based or semantic-net based) representations such as CODE4, KM [7], CycL [8] and the descendants of KL-ONE [9] (the latter are commonly called description logic languages). Collectively, these are sometimes called object-oriented knowledge representations, however they should not be confused with object-oriented programming languages. Although the design of the metrics in this paper is most appropriate for these types of languages, it does not preclude their use with other forms of knowledge base such as those containing rules or problem-solving methods.

Each concept-oriented knowledge representation uses a slightly different terminology; in this paper we use the terminology of CODE4. For more detailed information, see [3]. Some key ideas are explained below.

The primitive units of knowledge are called *concepts*. In other representations, the word ‘concept’ is also sometimes used (perhaps more restrictively) as are the words ‘frame’ and ‘unit’. A CODE4 knowledge base is composed almost entirely of interwoven networks of various kinds of concepts including :

- A *type concept* represents a set of similar things. A subset of type concepts, that the user explicitly creates and describes, are called *main subjects.*
- An *instance concept* represents a particular thing.
- A *property* represents a relation between concepts (similar to *slots* in other environments).
- A *statement* represents a particular tuple of a property. A statement has a *subject* (a concept), a *predicate* (a property) and a *value* (a reference to another concept). In CODE4, the value can be formal or informal. An informal value is a reference to an as-yet-unspecified concept. No reasoning can be done with informal values.

- A *term* represents a symbol that can stand for zero or more concepts. A concept can have zero or more synonymous terms.
- A *metaconcept* is a concept that represents another concept. It contains metaknowledge. The most important type of metaknowledge is the superconcept-subconcept relation.

2. Important metrics in software engineering

In the field of software engineering the need for metrics has been widely recognized [10]. It is therefore worth trying to learn some lessons from them before setting out to develop metrics for the allied field of knowledge management. Software engineering metrics are used as measures of productivity and as tools in project planning. The following are some of the more widely used:

2.1 Lines of code and other code metrics

Lines of code (LOC) is a very basic and intuitively obvious metric; it is widely used but has many flaws. Major uses of LOC, now frowned upon by many people, are to judge programmer productivity and help estimate cost. The main problems are a) LOC can not be reasonably determined until coding is complete, and b) LOC varies widely with the application, programming language, coding style and intrinsic complexity of the module being programmed.

There have been various other proposals for measuring code, primarily focusing on some aspect of its complexity that is independent of the way it is written. High complexity may be predictive of greater cost in future stages of development; and it may expose situations where designs can be improved. A measure of complexity may also help in accurately measuring productivity, because a small but complex project may take as much work as a large but simple project

The most well known of these is McCabe's cyclomatic complexity [11], which measures the number of possible execution paths through a module. In principle, if there are more execution paths, the software is more difficult to test and the effects of changes will be more difficult to predict. The main flaw with this metric is that it ignores many other possible aspects of complexity.

2.2 System metrics

In contrast with code metrics, *system metrics* (also called design metrics or structure metrics) take into account the interconnections among modules; i.e. the amount and nature of data and control coupling. The best-known such metric is Henry and Kafura's information-flow metric.[12]: This accounts for the number of flows into and out of a module, including access to global data structures. As with code metrics, each proposed system metric only measures one view of complexity.

2.3 Function points: The main specification metric

Function points [13] is becoming one of the most widely used software engineering metrics. It measures *functionality*, and is determined in a semi-subjective manner by counting specific items in a software requirements specification (unadjusted function points), and then factoring in other subjectively-estimated aspects of a project such as the importance of reuse. A major goal is that the subjective perception of the amount of functionality in any project should correlate well with that project's function point count.

The function points metric also allows *a priori* prediction of cost, whereas design and code metrics can only be used for evaluation during and after development.

The biggest flaws with function points are: 1) They require significant training, and cannot be automatically calculated from an informal specification. 2) The formula for computing function points considers many aspects of complexity but weights them all equally, even though there may be huge differences in the importance of these factors. 3) Because the counts are partly subjective, different people can generate different counts for the same specifications.

2.4 Lessons learned from software engineering metrics

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

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

• One should try to make the metric automatically calculable (unlike function points).

• One should try and measure different aspects of the knowledge.

• One should try and develop metrics which can be used as early as possible in the knowledge base development process (akin to function points).

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

2.5 Differences between software and knowledge which affect metrics design

Software has a number of significant differences from knowledge:

1. Knowledge is more declarative than most software.

2. If one represents software as a graph, there are a limited number of well-known node types (e.g. files, routines, statements), arc types (e.g. calls, includes, uses, follows-in-sequence) and ways of traversing the graph (e.g. call-herarchy, flow-chart) that one can use. Furthermore the granularity of the nodes is mixed: Many of the nodes represent entities that are complex and can be represented as graphs themselves. On the other hand, when one represents a knowledge base as a graph, one finds that most nodes are fine-grained in nature and densely connected by many different types of user-defined arcs. Furthermore, there are many different ways of traversing such a graph.

3. Knowledge is intrinsically hard to separate into modules or subsystems; and where this is done, the modules are generally extremely tightly coupled.

4. When building software, one typically starts with an abstract specification, and builds something more concrete. This is rarely the case with knowledge bases. A knowledge base is intrinsically highly abstract. In fact, it may *be* a specification of something more concrete.

5 Knowledge can be expressed in a rough form that is *partially* suited to the task at hand, and then gradually refined and formalized so that it becomes more suited to the intended task. Standard software tends either to work (most of the time) or not to work so there is less scope for incremental development at the detailed level.

These differences lead us to the following observations, as we embark on the process of creating knowledge base metrics:

- The surface syntax of knowledge is largely irrelevant : Any significant knowledge base will have such a complex graph that the only effective way to edit it is to use a tool that selects and pre-formats selected parts of it. As a consequence, metrics analogous to lines of code will probably not be useful. However, some way of determining the number of raw low-level assertions might be a baseline measure of raw size.
- The distinction between code and design does not readily carry forward to knowledge, nor does the distinction between design and specification. However, we may be able to take measurements of partially-complete knowledge bases or 'skeletons' which lack many formal details.

3. Towards knowledge base metrics: Goals and tasks

In this paper we adapt the Goals, Questions, Metrics [14] (GQM) methodology for the development of metrics. This methodology suggests that it is unproductive to develop a metric unless one has a question in mind that one wants the metric to answer. Furthermore, one should have a goal in mind, the achievement of which would be made possible by answering the question.

We prefer to think in terms of *tasks*, instead of questions, as the intermediate step: Thus we will first declare our goals; then we will outline some tasks we need to perform to achieve those goals, and finally we will develop metrics that will enable us to perform those tasks.

3.1 Goals to achieve by developing metrics

There are several goals in attempting to measure knowledge bases, including:

- Permitting knowledge engineers to monitor their work and to provide baselines for its continual improvement.
- Understanding how knowledge bases, users and domains differ in terms of characteristics such as quality, experience and structure.
- Facilitating research into knowledge management systems (both user interface and knowledge representation features) by providing grounds both for their comparison and for the comparison of knowledge bases built using them.
- Providing means whereby research using different formalisms and knowledge engines can be put on a common footing. Currently there is no effective way to compare the size and complexity of a knowledge base represented in, for example, Classic, with one represented in CycL.

3.2 Tasks to perform to help achieve the goals

To achieve the goals listed in the last subsection, we can define a series of measurement tasks we can perform. Many of the tasks have analogs in conventional software engineering, but others do not. The purpose of this section is to further motivate the development of metrics; suggestions for actual metrics and examples of their use can be found in sections 4 through 6.

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

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

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

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

Task A – Assessing completeness

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

In a well-managed standard software engineering project (where requirements are not constantly changing), completeness can be assessed by determining if the software fulfills its specification (e.g. passing appropriate testcases). The degree of completeness for a partially finished project can be estimated based on such factors as the proportion of function points represented by testcases that have been passed.

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

An alternative approach might be as follows: The first step is to recognize there may be no certain state of completeness. Secondly, it should be possible to measure the amount and kind of detail supplied about each main subjects in the knowledge base. Finally it should be possible to ascertain how close this is to the statistical average for knowledge bases that have been *subjectively* judged complete. This method has promise if users develop a skeleton knowledge base and then incrementally add details.

Task B – Assessing complexity

The reasons for measuring complexity of knowledge bases are the same as those of software: More complex knowledge will take longer to develop and will be harder to change.

Adaptations of software engineering metrics such as fan-in, fan-out and hierarchy depth may well apply to knowledge bases, but the special nature of knowledge bases may suggest additional useful metrics.

Task C – Assessing information content

It is unusual to pose the following query to conventional software: How much information is in this system? Rather, one asks: How well does it perform one of its limited number of tasks? With many knowledge bases however, a major goal is that they be queriable in novel ways – deducing new facts using various inference methods. The objective is to be able to uncover *latent* knowledge. The word ‘multifunctional’ has been used for such knowledge bases [15].

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

Task D – Assessing balance

We define two types of *balance* of a knowledge base: 1) the degree to which a group of measurements using related metrics are close to their respective ‘normal’ values, and 2) the degree to which measurements of different parts of a knowledge base using a single metric are close to each other.

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

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

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

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

3.2.2 Task E. Predicting knowledge base development time and effort

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

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

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

In the software engineering world, function points is used for M_s and COCOMO [16] has formulas that fulfill the role of f_p, predicting the number of person-months to complete the project, given M_s.

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

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

3.2.3 Tasks F to I: Comparison

Section 3.2.1 described some ways of measuring a single knowledge base in isolation, perhaps comparing it with a goal or norm. Another task for metrics is to find *relative* differences between knowledge bases so as to indirectly compare their creators, domains, techniques and representations.

Task F – Comparing users

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

Task G – Comparing domains

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

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

Task H – Comparing development techniques

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

Task I – Comparing representation schemata

In a similar manner to comparing development techniques, it is useful to compare representation schemata. It might still be possible to gain useful knowledge about the effectiveness of the abstractions in each schema.

4. Proposals for metrics

This section proposes actual metrics that can be used in the general measurement tasks discussed in the last section. Evaluation of the metrics, including actual measurements of knowledge bases, is deferred to sections 5 and 6. Three classes of metrics are discussed: general open-ended measures of size (subsection 4.2), measures of various independent aspects of complexity (subsection 4.3), and compound metrics (subsection 4.4).

4.1 Open-ended vs. closed-ended metrics

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

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

4.2 Metrics for raw size

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

The total count of all concepts, M_{ALLC}

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

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

The number of main subjects, M_{MSUBJ}

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

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

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

4.3 Independent and closed-ended metrics for complexity

A number of factors contribute to the complexity of a knowledge base. Seven complexity metrics have been developed that are logically independent of the raw size. They have also been designed to be as independent of *each other* as possible. Of course, just because one metric is designed to be independent of another does not prevent correlations from appearing in practice: It may happen that the normal process of knowledge acquisition results in increasing complexity along the different scales in parallel. Correlation is discussed in section 6.2.

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

Relative Properties, M_{RPROP}

This is a measure of the number of user properties, relative to the number of main subjects. It is calculated as the square of the ratio of user (non-primitive) properties to the sum of user properties and main subjects:

$$M_{RPROP} = (M_{UPROP} / (M_{UPROP} + M_{MSUBJ}))^2$$

Where M_{UPROP} is the number of user properties in the knowledge base.

If a user adds no properties, then M_{RPROP} is zero; M_{RPROP} approaches one as large numbers of properties are added to a knowledge base. In the average knowledge base, the number of user properties tends to be slightly more than the number of main subjects (0.55, i.e. people seem to add just over one new property for every main subject), hence M_{RPROP} averages 0.3 (which is 0.55 squared).

The metric is squared to reduce a problem that arises due to the deliberate decision not to make it open-ended: For each additional property, the increase in the metric is less. Thus in a 100 main-subject knowledge base, increasing the number of user properties from zero to 100 causes a 0.25 increase, whereas raising it another 100 only causes a 0.19 increase. If the metric were not squared, this problem would be much worse (the first 100 properties would cause a 0.5 increase and the next 100 only a 0.17 increase). In practice these diminishing returns only become severe at numbers of properties well above what have been encountered. Intuitively there appears to be diminishing returns in the subjective sense of complexity as well.

Figure 1 shows how M_{RPROP} varies as additional properties are added to a 100 main subject knowledge base.

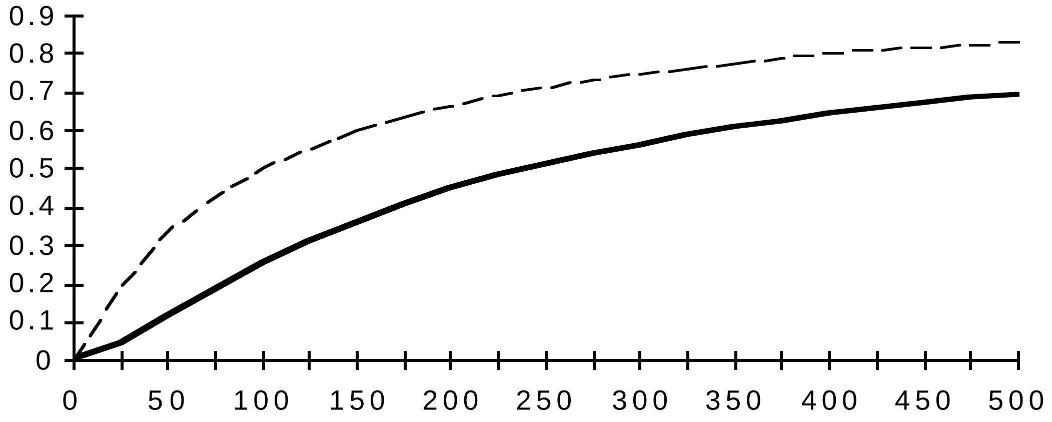

Figure 1: Values of M_{RPROP} when there are 100 main subjects. The x axis shows the effect of an increasing number of user properties. The bold plot is that of M_{RPROP} while the dashed plot shows what the metric would look like if it were not squared. The squaring flattens the curve and thus makes the metric more useful.

Detail, M_{DET}

This metric is the fraction of statements about main subjects, that have a locally specified value. Such statements contrast with those that have an inherited value, or no value at all as can be the case if a property is added and no statements are created using it. If no statement values are filled in at all in a knowledge base, M_{DET} is zero. If every possible statement about every main subject is given a specialized value, then M_{DET} equals one. However should M_{DET} have a value too near one it would indicate that inheritance is not being properly taken advantage of, and thus there is no knowledge reuse with its complexity-reducing effect.

To summarize, the formula for M_{DET} is as follows:

$$M_{DET} = M_{MSSLV} / M_{MSS}$$

where M_{MSS} is the number of main subject statements (statements whose subject is a main subject)
and M_{MSSLV} is the number of main subject statements which have local values

Whereas M_{RPROP} measures the *potential* number of specialized statements , M_{DET} indicates the amount of *use* of that potential. The next measure, M_{SFORM}, goes one step further and measures to what degree that use results in the interconnection of concepts.

Statement Formality, M_{SFORM}

This measures the fraction of values (of statements about main subjects) that contain actual links to other concepts in the knowledge base. If M_{SFORM} is zero, then the user has merely placed arbitrary text in all values (CODE4 and some other systems allow this so the user can sketch knowledge informally before incrementally formalizing it). The higher M_{SFORM} is, the more a knowledge management system would be able to infer additional network structures (i.e. the part-of relation) inherent in the knowledge.

The formula for M_{SFORM} is as follows:

$$M_{SFORM} = M_{MSFS} / M_{MSSLV}$$

where M_{MSSLV} is the number of main subject statements with local values (see M_{DET})
and M_{MSFS} is the number of main subject formal statements, i.e. those with local values containing value items that are other concepts

The denominator of M_{SFORM} is composed of only those statements that form the numerator of M_{DET} – those statements with locally specified values. It thus should be independent of M_{DET} : i.e. regardless of whether M_{DET} is near zero or near one, M_{SFORM} can still range from zero to one. The one exception is when there are no statement values, in which case M_{DET} is zero and M_{SFORM} is undefined.

Diversity, M_{DIV}

While high measurements of relative properties, detail and formality may indicate that substantial work has been done to describe each main subject, that detail may be largely in the form of very subtle differences. There may be a large amount of mere data, expressed as statements of the same set of properties about each main subject. For example, in a knowledge base describing types of cars, hundreds of classifications of cars may be described but only using a fixed set of properties (engine size, fuel consumption etc.) – such a knowledge base would be subjectively judged to be rather simple despite having a lot of ‘facts’ (it would be more like a database). M_{DIV} attempts to quantify this subjective feeling by rating more diverse, or ‘interesting’, knowledge bases higher.

The diversity metric, M_{DIV}, measures the degree to which the introduction of properties is evenly spread among main subjects. If all properties are introduced in one place (e.g. at the top concept) then M_{DIV} is close to zero because the knowledge base is judged to be simpler. Likewise, M_{DIV} is close to zero if properties are introduced mostly on leaf concepts (so there is no inheritance). Maximum M_{DIV} complexity of one is achieved when some properties are introduced at the top of the inheritance hierarchy, some in the middle and some at all of the leaves.

The method of calculating M_{DIV} is described in the following paragraphs; figure 2 is used to illustrate the calculations.

To calculate M_{DIV}, the first step is to calculate the standard deviation of the number of properties introduced at each main subject. This is zero if the introduction of properties is evenly distributed, and some higher number otherwise. The next step is to normalize this standard deviation into the range zero to one. The following formula accomplishes this:

$$M_{CONCEN} = ((\sigma \text{ PI}) / M_{UPROP}) * \sqrt{M_{MSUBJ}}$$

where M_{UPROP} is the number of user properties in the knowledge base
and PI is the number of properties introduced at a main subject
and σ is the standard deviation operator

A simple way to convert M_{CONCEN} into a diversity metric would be to calculate 1-M_{CONCEN}. However, thus results in measurements being too crowded towards zero. M_{DIV} is actually calculated using the following more subjectively appealing formula:

$$M_{DIV} = (1 - M_{CONCEN}{}^2)^2$$

Figure 3 plots the relationship between M_{CONCEN} and M_{DIV}.

Second Order Knowledge, M_{SOK}

The metrics described so far totally ignore the presence of second order knowledge, i.e. knowledge not about the things in the world represented by concepts, but about the representations of those things. M_{SOK} measures the use of two types of second order knowledge:

- The use of metaconcepts, i.e. the concepts that describe concepts themselves
- The use of terms, i.e. symbols that represent things.

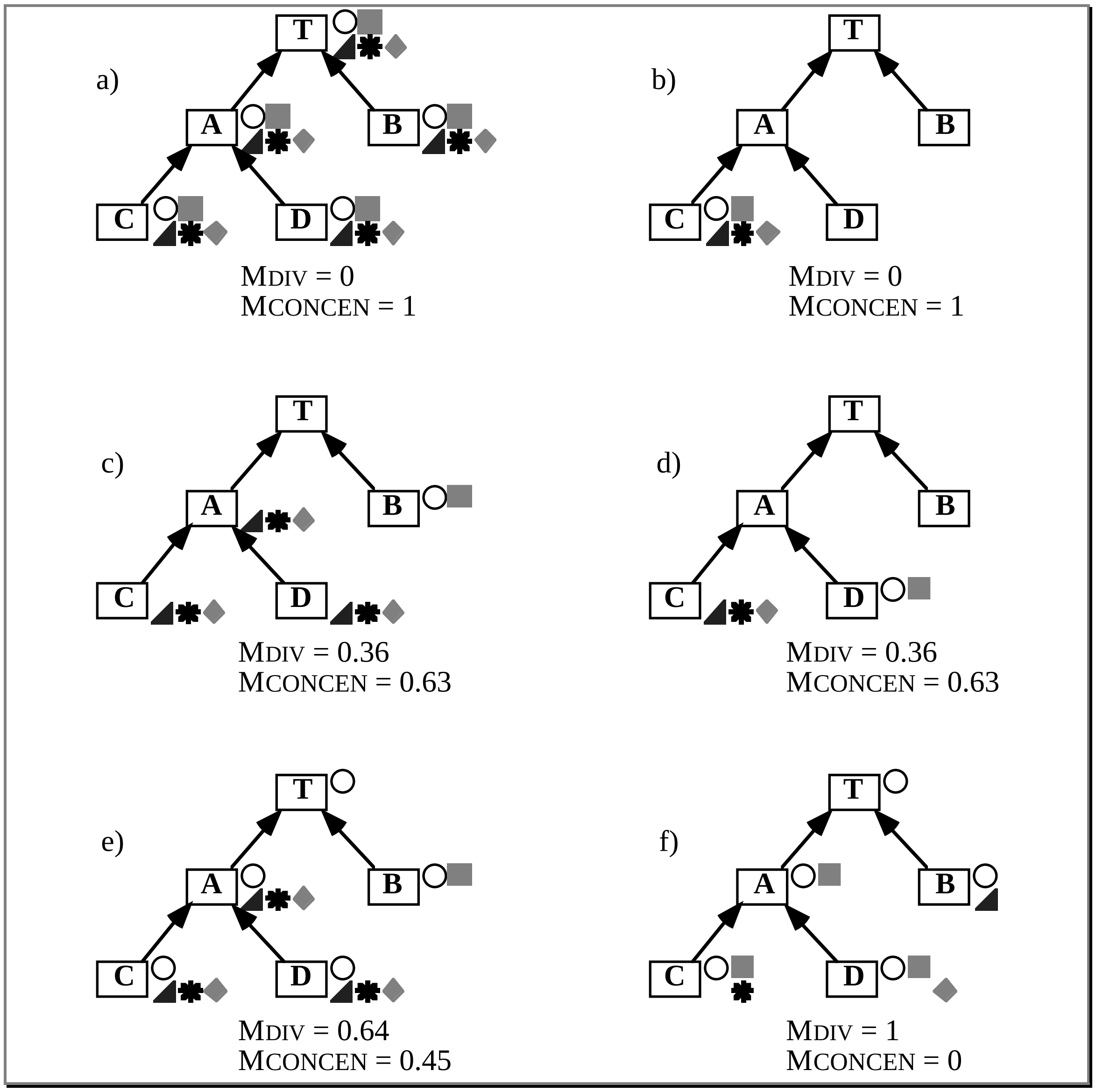

Figure 2: Calculation of M_{DIV}. Each part shows an inheritance hierarchy with five main subjects and five properties (shown as five distinct shapes). Parts a and b show pathological situations where properties are all introduced at a single concept (the top and a leaf respectively) and M_{DIV} is thus zero. Part f shows a well balanced case, where each concept introduces exactly the same number of new properties (one in this case) and M_{DIV} is one. Parts c, d and e show intermediate cases.

Every main subject in principle has a metaconcept, but most are ignored by users. Of interest are those which users have explicitly made the subjects of statements (i.e. those which users have said something about). Similarly, every main subject generally has a term that is used to name it. However, of interest are those cases where extra terms are specified: This indicates that users have paid attention to linguistic issues.

M_{SOK} is zero if the user has neither worked with metaconcepts nor specified synonyms for any concept. The metric gives a measurement of one if every main subject has both multiple terms and metaconcept statements. M_{SOK} is thus calculated as follows:

$$M_{SOK} = (M_{FMETA} + M_{FTERM}) / 2$$

where M_{FMETA} is the fraction of main subjects whose metaconcept has a user-specified statement
and M_{FTERM} is the fraction of main subjects that have more than one term

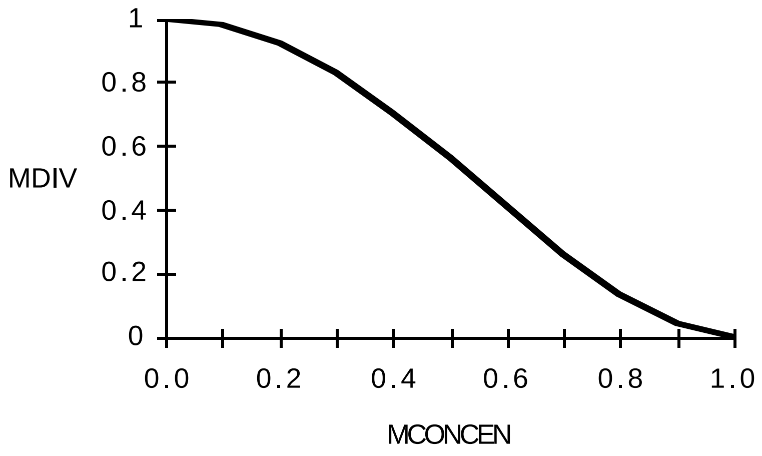

Figure 3: The relationship between M_{CONCEN} and M_{DIV} . M_{CONCEN}, shown in the x axis is a normalized version of the standard deviation of the number of properties introduced at each main subject. This is transformed using a sigmoid function, as plotted in the above graph, in order to obtain a metric that has a good subjective 'feel'.

Isa complexity, M_{ISA}

This metric combines two factors in order to measure the complexity of the inheritance hierarchy. The first factor is the fraction of types that are leaves of the inheritance hierarchy, ignoring instance concepts which are always leaves (i.e. the fraction of types that have no subtypes). This measure is called M_{FLEAF}:

$$M_{FLEAF} = M_{LEAF} / M_{TYPES}$$

where M_{LEAF} is the number of leaf types
and M_{TYPES} is the total number of types

For any non-trivial knowledge base, M_{FLEAF} is a function of the branching factor, M_{BF} (2 for a perfect binary tree, 3 for a perfect ternary tree etc.) however the latter is not used as a metric because it is open ended (i.e. not in a zero-to-one range). The formula relating branching factor to M_{FLEAF} is as follows:

$$\lim_{M_{TYPES} \to \infty} M_{FLEAF} = (M_{BF} - 1) / M_{BF}$$

where M_{TYPES} is the total number of types

Figure 4 shows several simple inheritance hierarchies along with their measures of M_{FLEAF}. M_{FLEAF} approaches 0.5 when the inheritance hierarchy is a binary tree (figure 4, parts c and f). It approaches zero in the ridiculous case of a unary 'tree' with just a single superconcept-subconcept chain (part b) and it approaches one if every concept is an immediate subconcept of the top concept (part a).

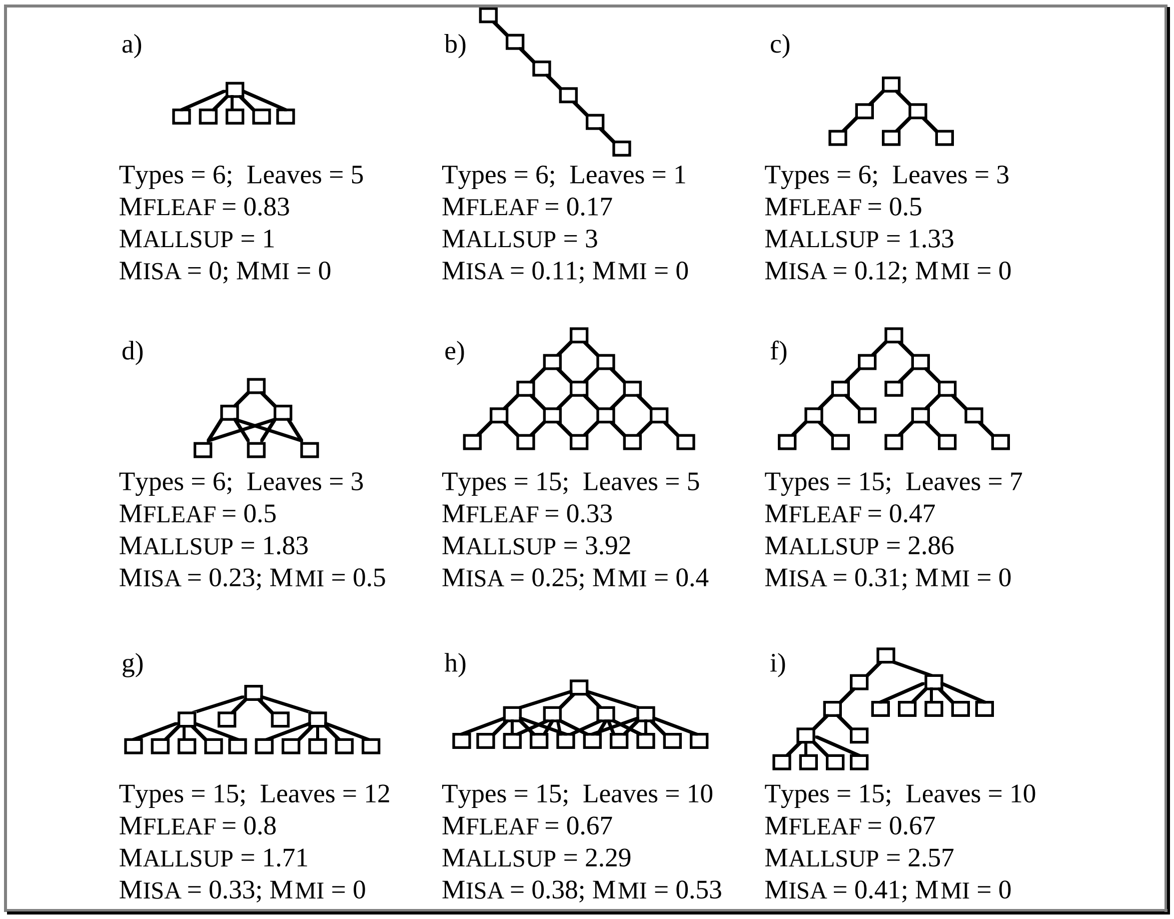

Figure 4: Complexities of various inheritance hierarchies. Parts a through i show increasing complexity using the M_{ISA} metric. Parts a and b show simplistic cases (M_{ISA} would be small regardless of the size of the knowledge base). Parts c and d show the same structure, with variation only in multiple inheritance. Pairs (e,f) and (g,h) also show similar structures with variation in multiple inheritance, however M_{ISA} can be seen to be somewhat independent of the adding of extra parents (such an action may either increase or decrease M_{ISA}).

On its own, M_{FLEAF} has the undesirable property that for a very shallow hierarchy (e.g. just two or three levels) with a high branching factor it gives a measurement that is unreasonably high, from a subjective standpoint (part a of figure 4 illustrates this)

To correct this problem with M_{FLEAF} , an additional factor is used in the calculation of M_{ISA}: the average number of direct and indirect superconcepts per non-root main subject , M_{ALLSUP}. (The root concept at the top if the inheritance hierarchy is not counted since it cannot have parents). This second factor is related to hierarchy depth but depends to some extent on the amount of multiple inheritance.

M_{ISA} is thus calculated using the following formula:

$$M_{ISA} = M_{FLEAF} - (M_{FLEAF} / M_{ALLSUP})$$

M_{ISA} approaches zero in either of the following cases: 1) when there is just a single layer of concepts below the top concept (no matter how many concepts); or 2) when the branching factor is one. M_{ISA} approaches the value of M_{FLEAF} (e.g. 0.5 for a binary tree) as the hierarchy becomes deeper (as the average number of direct or indirect parents per main subject increases).

Multiple Inheritance, M_{MI}

This measures the extent to which main subjects have more than one parent, thus introducing complex issues associated with multiple inheritance such as the combination of values when two inherited values conflict. If just single inheritance is present, this metric is zero.

M_{MI} measures the ratio of *extra* parents to main subjects, thus if all main subjects had two parents (impossible in fact because the concepts directly below the top concept cannot have more than one parent) then the metric would be one; it could also be one if a substantial number of concepts have more than *two* parents. Although the metric, as described, could theoretically give a value above one, it is assumed that this could not be the case in any reasonable knowledge base. Thus a ceiling of one is imposed that the metric cannot exceed, even if there are an excessive number of parents.

4.4 Compound metrics for complexity

In this subsection, the simple metrics for complexity described in section 4.3 are combined into compound metrics in an attempt to give useful overall pictures of a knowledge base. In the previous subsection, the metrics were designed with reasonable confidence that they are generally applicable. Here however, the necessity of *combining* metrics requires finding numerical coefficients that optimize the usefulness of the result. Determining such coefficients requires extensive examination of data. In this research the only data available was that of the user study described in section 6. While the amount of data is quite large for testing metrics, it is not nearly large enough to act as a 'training set' for metric optimization. As a result, the main contributions of this subsection are proposals for *procedures for designing metrics*, rather than proposals for actual formulas (the formulas presented are examples of the application of the procedure).

The generic procedure for designing a compound metric is as follows:

1) Choose the metrics that are to be combined by looking for those with certain desired characteristics.
2) Normalize the metrics so that they are not arbitrarily biased.
3) Weight the metrics based on decisions about which are most important, and how much their values should contribute to the resulting metric
4) Design a formula that combines the metrics.

Although both step 2 and step 3 involve finding coefficients that modify the original values of the metrics, they are logically separate steps. After step 2, accidental bias is removed. Such bias may be caused by the fact that the metrics might have quite different expected values or ranges. In step 3, deliberate weighting is added. For this initial research, step 3 is ignored since it would require a very large amount of data to justify an unequal weighting. The most important step in this research is step 4.

Apparent completeness, M_{ACPLT}

This metric combines those metrics that, intuitively, should steadily increase as a project progresses. The idea is to create a metric that ranges between 0 and 1 so that users can obtain an impression of how much work needs to be done on a knowledge base to make it 'complete' (i.e. with a reasonably high percentage of detail, formality etc.). See section 5.3.1 for a discussion of what it means for a knowledge base to be complete.

The metrics chosen to compose M_{ACPLT} are:

• M_{RPROP}, indicating the extent to which properties have been added
• M_{DET}, indicating the proportion of potential statements with actual values, and
• M_{SFORM} indicating the extent to which the knowledge base has been formalized.

The possibility of adding M_{SOK} was considered, but it was decided not to do this because the amount of second order knowledge might be heavily dependent on the domain, whereas the extent to which statements are filled in and formalized appears much more likely to measure the state of completeness, independent of domain.

To remove accidental bias from the metrics, actual knowledge bases were studied to see how the component metrics range in practice (see section 6). None ever approached a value of one, and the maximums for well-worked out knowledge bases were all about 0.7. It seems reasonable that none of these metrics would ever reach one for the following reasons:

- M_{RPROP} can only asymptotically approach 1 as ever larger numbers of properties are added. A measurement as high as 0.7 already indicates that there are over five times as many properties as main subjects. Experience has shown that it is unlikely for a knowledge base to have a much higher proportion of properties than this.

- If M_{DET} were to approach 1, it would mean that hardly any statements are inherited; all would be locally specified (i.e. they all override inherited values). This seems unlikely to happen in most knowledge bases.

- If M_{SFORM} were to approach 1, the user has been able to formalize all statements. This seems unlikely in a normal knowledge base.

It was thus decided that prior to combining the metrics to create M_{ACPLT} , all of them should be normalized so that when a measure using the normalized metric has a value of about 1, it is considered to indicate that the particular aspect of the knowledge base is reasonably complete. It is purely coincidental that the 0.7 is used for all three: All three component metrics all had measured ranges of 0-0.7.

The coefficient used to normalize the metrics is the reciprocal of 0.7, i.e. 1.4. As a result of this, M_{ACPLT} can give measurements greater than one – that would merely indicate that more detail has been provided than in a normal complete knowledge base.

The following points explain how it was decided to combine the metrics:

- M_{RPROP} must dominate the calculation for the following reason: If there are few properties, then high measurements of detail and formality mean little (because there can only exist a few statements with potential for having detail and formality). Thus M_{RPROP} should be a multiplicative value for the whole metric formula – if M_{RPROP} is zero then M_{ACPLT} should be zero.

- For similar reasons, M_{DET} must dominate M_{SFORM}. If there are few statements, then a high measurement of formality means little because there are only a few statements to *be* formal.

The basic formula for combining the three metrics is therefore:

$$M_{ACPLT} = C * M_{RPROP} * (W + C * M_{DET} * (X + Y * C * M_{SFORM}))$$

Where C is the constant 1.4 used to unbias the metrics; i.e. to convert their 0 to 0.7 range to a range of 0 to 1. If this factor were missing , the resulting metric could only yield measurements that would approach 70%.

And where W, X and Y are coefficients used to weight the metrics. As discussed earlier it was decided to weight the metrics equally. These three coefficients should all be 0.33 then. That would mean that if all three input metrics approached one, the result would be 0.33 * 3 = 1.

Simplifying the above, where Z is used in place of W, X and Y, gives:

$$M_{ACPLT} = M_{RPROP} * (C * Z + M_{DET} * (C^2 * Z + C^3 * Z * M_{SFORM}))$$

After application of the constants, the derived formula for M_{ACPLT} is thus as follows:

$$M_{ACPLT} = M_{RPROP} * (0.47 + M_{DET} * (0.65 + 0.91 * M_{SFORM}))$$

An interesting derivative use of M_{ACPLT} would be to apply it to different subhierarchies of a knowledge base in order to determine which areas need work, or alternatively, which areas contain more useful knowledge.

Pure complexity, M_{PCPLX}

The objective of this metric is to combine all the independent complexity measures into a size-independent metric for the ‘difficulty’ or ‘sophistication’ of a knowledge base. It was decided to use a mechanism similar to that used to calculate function points (see section 2). M_{MACPLT} is used as the analog for ‘unadjusted function points’. The four metrics not included in the calculation of M_{MACPLT} are then used as ‘complexity adjustment factors’. After the application of each to either increase or decrease the unadjusted metric, the resulting metric M_{PCPLX} is an analog of ‘adjusted function points’.

To calculate adjusted function points, the first step is to measure fourteen complexity adjustment factors using simple scales and then to sum the measurements (without weighting). The sum can be called the ‘compound adjustment factor’. In the second step, a formula involving the compound adjustment factor results in the multiplication of the unadjusted function points by a factor ranging from 0.65 to 1.35. In the following paragraphs, a similar two-step approach is applied to knowledge base metrics.

Step 1: Scaling and summing. The four metrics not included in M_{MACPLT} have theoretical ranges of zero-to-one. However, upon examining data from user studies, it was determined that measurements of three of the four metrics rarely approach the top of that range (M_{SOK} is the exception).

Thus it was decided to apply multiplicative scaling factors to M_{SOK}, M_{ISA} and M_{MI} in order to unbias them (so that each metric contributes fairly to the result). The scaling factors (2.5, 1.6 and 1.2 respectively) were determined by examining the means and ranges of the test data. These scaling factors are the least objective aspect of the calculation of M_{PCPLX} – extensive data analysis would be needed to fine tune them; however since this research is intended to propose a method for developing metrics rather than a definitive formula, these figures are considered adequate. The resulting compound adjustment factor has a range of zero to four (see formula below).

Step 2: Creating the adjusted metric: It was decided to adjust M_{MACPLT} so that the resulting metric is M_{MACPLT} multiplied by between 0.25 and 1.85. This is a wider range than that used for function points, but it was decided the four adjustment factors should have a larger influence than those used for function points. Again, more extensive data analysis would be needed to arrive at a more objective range.

The derived formula for M_{PCPLX} is thus as follows:

$$M_{PCPLX} = M_{MACPLT} * (0.25 + 0.4 * (M_{DIV} + 2.5 * M_{SOK} + 1.6 * M_{ISA} + 1.2 * M_{MI}))$$

where the second line is the compound adjustment factor

Overall complexity, M_{OCPLX}

The overall complexity measure is simply M_{MSUBJ} multiplied by M_{PCPLX}. In other words the count of the number of main subjects is adjusted using the measure of 'pure' complexity of the knowledge base. M_{OCPLX} might be considered to measure 'fully specified main subjects'. It is intended to serve as a measure of productivity or information content.

5 Desirable qualities of the metrics

This section discusses various useful qualities of the metrics presented in the previous section· The following criteria are used to judge the quality of the metrics: 1) Does each metric have a use that is independent of the others? 2) Is each metric understandable? And, 3) does each metric have a reasonable mapping onto the subjective phenomenon (including behavior at extremes)?

5.1 How subjectively useful are the metrics?

Each metric must perform some useful task. There should be some reason why a user might want to use the metric independently of the others. The following paragraphs indicate that each metric apparently has a valid use by showing what the user can learn by using it.

- **All concepts: M_{ALLC}**: This gives an idea of the amount of memory and disk space used by the knowledge base. It also gives an idea of the number of discrete facts in the knowledge base.
- **Main subjects: M_{MSUBJ}**: This indicates to the user the number of things being talked about in the knowledge base, and hence is a natural measure of size that is independent of complexity. If the user follows a methodology of sketching out the inheritance hierarchy before filling in details, this metric can help him or her estimate the eventual size of the knowledge base and hence the amount of work that might be required to create it.
- **Relative Properties: M_{RPROP}**: This indicates to the user whether a knowledge base has a reasonable number of properties, relative to the number of main subjects. A low measurement might indicate that more properties should be added.

- **Detail: M_{DET}**: This indicates whether a knowledge base (or portion thereof) has a reasonable number of statements. A low measurement might suggest that additional statements should be added.
- **Statement Formality: M_{SFORM}**: This indicates whether a knowledge base has a reasonable number of formal links. A low measurement indicates that the user would be unlikely to be able to generate sophisticated graphs of relations.
- **Diversity: M_{DIV}**: This indicates the degree to which knowledge is focused at the top or bottom of the inheritance hierarchy. A low value might indicate that inheritance is not being properly used and that many concepts are merely placeholders.
- **Second Order Knowledge: M_{SOK}**: This indicates the extent to which knowledge is included about concepts themselves as opposed to the things represented by concepts. A low measurement indicates that there is probably significant knowledge missing.
- **Isa Complexity: M_{ISA}**: This metric indicates the degree to which the inheritance hierarchy has a non-trivial pattern of branching. A low measurement indicates that many concepts do not have siblings and thus relatively few distinctions are being made.
- **Multiple inheritance: M_{MI}**: This indicates the extent of the complexity added due to multiple inheritance. A low measurement indicates that little multiple inheritance is being used.
- **Apparent completeness: M_{ACPLT}**: By combining those metrics that should logically increase as a knowledge base approaches completion, this metric gives the user an idea of how close to completion the knowledge base might be.
- **Pure complexity: M_{PCPLX}**: By combining all the complexity metrics this gives the user an overall value of the ‘difficulty’ of the knowledge base, independent of size.
- **Overall complexity: M_{OCPLX}**: In combining pure complexity with size, this metric is intended to give the user an idea of the information content in a knowledge base.

5.2 How intuitive or understandable are the metrics?

To be preferred are simpler metrics, i.e. ones where users can easily understand the reasons for the calculations.

The methods of calculating M_{ALLC} and M_{MSUBJ} are the simplest, being mere counts. M_{DET}, M_{SFORM} and M_{MI} have a slightly lower level of understandability since they are simple ratios of reasonably understandable counts. M_{RPROP} is lower again in understandability since it is squared in order to give it better responsiveness to the underlying phenomenon. The other metrics all have relatively complex formulas.

5.3 How good is the mapping between the metric’s function and the subjective phenomenon?

The function described by the metric (the objective phenomenon) must correspond well throughout its range with the subjective phenomenon in the mind of the metric’s interpreter. If the correspondence is poor, it might be because some objective factor is omitted or incorrectly weighted.

One test of correspondence for metrics with a zero-one range is as follows: The metric should yield a value of 0.5 when the subjective phenomenon is at about the half-way point, i.e. the subjectively ‘normal’ point. Another test is to ensure that the endpoints correspond with what one would intuitively expect. Table 1 gives interpretations for the endpoints and centre points of the closed-ended metrics and shows that they behave well in these respects. The only possible exception is M_{RPROP} whose middle point indicates that there are 2.5 properties per concept – however this occurs in order to ensure that equal *deltas* of the metric correspond more closely with equal changes in the subjective phenomenon.

Metric	**Meaning approaching 0**	**Meaning near 0.5**	**Meaning approaching 1**
M_{RPROP}	No properties	2.5 properties per concept	Very many properties per concept
M_{DET}	No values specified	Values specified on half of statements	Value specified wherever possible
M_{SFORM}	No formal values	Half of values are formal	All values are formal
M_{DIV}	Properties introduced on one concept	Properties introduced on half of all concepts	Properties introduced evenly on all concepts
M_{SOK}	No second order knowledge	Metaconcept detail and extra terms on about half of concepts	Both metaconcept detail and extra terms on every concept
M_{ISA}	Each concept has about one parent - very simple	Binary tree	Very bushy tree - very complex
M_{MI}	No multiple inheritance	Half of concepts have an extra parent	Very high degree of multiple inheritance

Table 1: Interpreting metrics – meanings of various values. Rows show metrics defined that are closed-ended. Columns indicate the interpretation of measurements at the extremes of that range and in the middle.

5.4 Summary of desirable qualities of the metrics

Table 2 summarizes how well the metrics appear suited to the various tasks described in section 3.

The following are some other general observations about the metrics:

- The most useful metrics appear to be M_{OCPLX}, M_{DET} and M_{SFORM}. User studies (section 6) showed that the overall complexity metric correlates reasonably well with time-to-complete, whereas M_{DET} and M_{SFORM} give clear indications of where work needs to be done in a knowledge base.
- The metrics that have the best combination of understandability and usefulness are M_{MSUBJ} and M_{DET}.
- Most metrics correspond well with subjective phenomena.
- All the metrics with a zero-to-one range have good meanings for the ends of the ranges

Task	**Suitable Metrics**	**Observations**
A to D. Assessing the present state of a knowledge base		
Completeness	M_{ACPLT}	The compound metric of apparent completeness.
Complexity	M_{PCPLX}, M_{DET}, M_{SFORM}	Pure complexity, and two of its most important components: detail and statement formality
Information content	M_{OCPLX}	The overall complexity of the knowledge base
Balance	M_{DIV}	The diversity of distribution of properties; other balance metrics could be created.
E. Predicting	M_{MSUBJ}	The number of main subjects can typically be determined earlier than other metrics
F to I. Comparing	all	

Table 2: Measuring tasks and how well they can be performed. At the left are the measuring tasks listed in section 5.3. The middle column lists some of the metrics that may be useful in the performance of the task.

6. Use of the metrics during the development of knowledge bases.

This section summarizes the results of preliminary tests using the knowledge base metrics introduced in section 4.

For this work, the following procedure was used:

1. 12 people who were interested in building knowledge bases in CODE4 were selected as participants. These included 11 graduate students and one professor.

2. The participants were trained to use the system. They also had access to a 100 page user manual .

3. They created 25 knowledge bases, covering such topics as:
   - Computer languages and operating systems.
   - Other technical topics (such as optical storage, electrical devices, geology and matrices)
   - General purpose knowledge (people, vehicles etc.)

   The total amount of work involved in creating the knowledge bases was about 2000 hours, i.e. an average of 80 hours per knowledge base.

4. The participants were asked to complete a questionnaire about their experiences. This questionnaire contained 55 main questions and, among other things, asked for the participants’ subjective impressions of the complexity and completeness of their knowledge bases.

5. The resulting knowledge bases were measured using the simple complexity metrics developed in this paper. The same knowledge bases were also used to calibrate the compound complexity metrics.

6.1 Measurements of several knowledge bases

Table 3 summarizes measurements taken of the knowledge bases prepared by the participants in this research.

Metric	Theoretical Range	Mean	Minimum	Maximum	Standard Deviation
Raw size metrics					
All concepts: M_{ALLC}	(40-∞)	842	224	2825	612
Main subjects: M_{MSUBJ}	(0-∞)	91	21	278	61
Independent complexity metrics					
Relative Properties: M_{RPROP}	(0-1)	0.33	0.05	0.70	0.17
Detail: M_{DET}	(0-1)	0.18	0.03	0.69	0.14
Statement Formality: M_{SFORM}	(0-1)	0.16	0.00	0.67	0.19
Diversity: M_{DIV}	(0-1)	0.71	0.00	0.99	0.27
Second Order Knowledge: M_{SOK}	(0-1)	0.06	0.00	0.41	0.11
Isa Complexity: M_{ISA}	(0-1)	0.40	0.19	0.59	0.11
Multiple inheritance: M_{MI}	(0-1)	0.19	0.00	0.87	0.23
Compound complexity metrics					
Apparent completeness: M_{ACPLT}	(0-2)	0.20	0.03	0.39	0.11
Pure complexity: M_{PCPLX}	(0-5.6)	0.19	0.02	0.38	0.10
Overall complexity: M_{OCPLX}	(0-∞)	16	2	56	14

Table 3: Statistics about knowledge bases created by the participants. Each row corresponds to one of the metrics discussed in section 4.

The following are some general observations about table 3:

- The knowledge bases differ substantially in size. When measured using M_{ALLC} and M_{MSUBJ} the ratio of largest to smallest is about 13:1. When measured using M_{OCPLX}, however, a more realistic ratio appears, i.e. 28:1.
- The knowledge bases vary widely according to all of the independent complexity metrics, in particular according to M_{DIV} and M_{MI}. Of these, M_{DIV} is probably the most interesting since it seems to be more of an indicator of the individual style of the knowledge base developer than the other metrics.

6.2 How independent is each complexity metric from the others?

Metrics should be as independent as possible because it would be redundant to make measurements using two metrics that are dependent on each other. The simple complexity metrics were *designed* to be independent of each other – all involve separate aspects of a knowledge base. However, the only way to really test for independence is to calculate correlation coefficients.

As table 4 indicates, for the most part success has been achieved. The biggest exception is the reasonably strong negative correlation between the isa complexity and the amount of multiple inheritance. This can be accounted for theoretically because if there are more parent concepts to be multiply inherited (including by leaves), then the proportion of leaf types should decrease.

	Multiple Inher. M_{MI}	Isa Complex. M_{ISA}	Second Order M_{SOK}	Diversity M_{DIV}	Statement Formality M_{SFORM}	Detail M_{DET}
Relative Properties: M_{RPROP}	0.15	-0.23	0.11	-0.13	-0.17	0.17
Detail: M_{DET}	0.12	-0.28	-0.21	-0.06	-0.16	
Statement Formality: M_{SFORM}	-0.19	0.04	-0.02	0.34		
Diversity: M_{DIV}	-0.18	0.07	-0.08			
Second Order Knowledge: M_{SOK}	-0.21	0.14				
Isa complexity: M_{ISA}	-0.58					

Table 4: Coefficients of linear correlation among the seven complexity metrics. The data used in calculating these coefficients was obtained from the knowledge bases prepared by the participants.

7. Summary and conclusion

The primary purpose our research into knowledge management systems is to make knowledge management practical. It is very hard to manage something, however, unless one can quantify it.

This paper has discussed several metrics that can be applied to frame-based knowledge representations. The metrics can be divided into three classes: 1) raw measures of size; 2) measures of various attributes of complexity, and 3) compound measures intended to help users assess their productivity.

With the help of metrics, users can better do such things as the following:

1. Estimate completeness of a knowledge base, or a component thereof. Using a metric like M_{ACPLT}, a user can decide how much work might need to be done and where.
2. Judge the overall volume of knowledge in a knowledge base, using M_{OCPLX}.
3. Obtain a rough idea of how difficult a knowledge base might be to navigate or modify; M_{PCPLX} can help with this.
4. Compare subjectively ‘complete’ knowledge bases to see how domains differ, in order to help in the estimation of future knowledge base development tasks.

The main scientific contribution of this paper has been to point out several kinds of things that one might wish to measure in a knowledge base. It is hoped that the work will stimulate people to actually measure the products of knowledge engineering and to perform further research into metrics.

References

1. D. Skuce, “A Multifunctional Knowledge Management System”, *Knowledge Acquisition,* **5** (1993), 305-346.

2. D. Skuce and T. C. Lethbridge, “CODE4: A Unified System for Managing Conceptual Knowledge”, *Int. J. Human-Computer Studies* **42** (1995), 413-451.

3. T.C. Lethbridge, *Practical Techniques for Organizing and Measuring Knowledge*, Ph.D. thesis, Department of Computer Science, University of Ottawa, 1994.

4. I. Meyer, K. Eck and D. Skuce, “Systematic Concept Analysis Within a Knowledge-Based Approach to Terminology”, in *Handbook of Terminology Management*, Eds. S.E. Wright and G. Budin, John Benjamin, 1994.

5. J. Bradshaw, P. Holm, O. Kiperztok and T. Nguyen. “eQuality: A Knowledge Acquisition Tool for Process Management”, Proc. *FLAIRS 92*, Fort Lauderdale.

6. M. Longeart, G. Boss and D. Skuce, “Frame-based Representation of Philosophical Systems Using a Knowledge Engineering Tool”, Computers and the Humanities **27** (1993), 27-41.

7. B. Porter, J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker and T. Jones, *AI Research in the Context of a Multifunctional Knowledge Base: The Botany Knowledge Base Project*, Technical Report, The University of Texas at Austin.

9. D. Lenat and R. Guha, *Building Large Knowledge Based Systems*, Addison Wesley , 1990.

9. R. Brachman, D. McGuiness, P. Patel-Schneider, L. Resnick and A. Borgida, “Living with CLASSIC: When and How to Use a KL-ONE Like Language”, in *Principles of Semantic Networks*, Ed. J. Sowa, Morgan Kaufmann, 1991, pp. 401-456.

10. M. Shepperd and D. Ince, *Derivation and Validation of Software Metrics,* Oxford, 1993.

11. T.J. McCabe, “A Complexity Measure”, *IEEE Trans. Software Eng.* **2** (1976), 308-320.

12. S.M. Henry and D. Kafura, “Software Structure Metrics Based on Information Flow,” *IEEE Trans. Software Eng.*, **7** (1981), 510-518.

13. G. C. Low and D. R. Jeffrey, “Function Points in the Estimation and Evaluation of the Software Process”, *IEEE Trans. Software Eng.* **16** (1990), 64-71.

14. V.R. Basili and H.D. Rombach, “The TAME Project: Towards Improvement-Oriented Software Environments,” *IEEE Trans. Software Eng.* **14** (1988), 758-773.

15. L. Acker, *Access Methods for Large, Multifunctional Knowledge Bases*, TR AI92-183, Ph.D. Thesis, Department of Computer Science, University of Texas at Austin, 1992.

16. B.W. Boehm, *Software Engineering Economics,* Prentice-Hall, 1981.