David Soergel
tree@stanford.edu
http://www.stanford.edu/~tree
123 Forest View, Woodside, CA 94062
(650) 725-5436

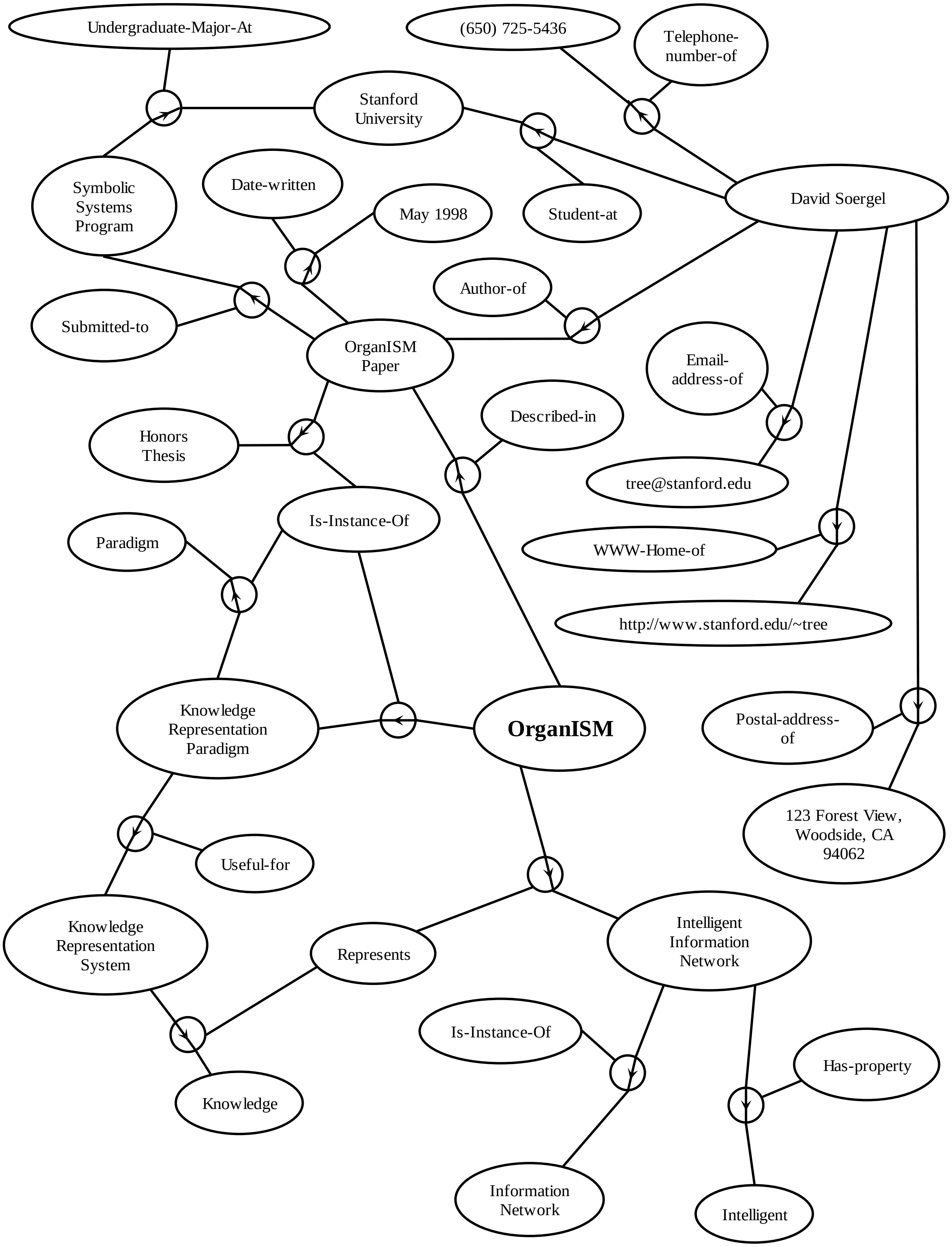
OrganISM

An Abstract Representation Paradigm for Intelligent Information Networks

May 1998

Senior Honors Thesis

SYMBOLIC SYSTEMS PROGRAM
STANFORD UNIVERSITY


Undergraduate-Major-At
(650) 725-5436
Telephone-number-of
Stanford University
Symbolic Systems Program
Date-written
May 1998
Student-at
David Soergel
Author-of
Submitted-to
OrganISM Paper
Email-address-of
Described-in
Honors Thesis
tree@stanford.edu
Is-Instance-Of
Paradigm
WWW-Home-of
http://www.stanford.edu/~tree
Knowledge Representation Paradigm
OrganISM
Postal-address-of
123 Forest View, Woodside, CA 94062
Useful-for
Knowledge Representation System
Represents
Intelligent Information Network
Is-Instance-Of
Has-property
Knowledge
Information Network
Intelligent

Acknowledgements

The original inspiration for this project, and much of the theoretical groundwork, comes from my father, Dr. Dagobert Soergel. I wish to thank him for his continuing support, encouragement, and insightful advice.

I would also like to thank my thesis adviser, Dr. Gio Wiederhold, whose knowledgeable advice pointed me in various fruitful directions and helped me avoid several serious pitfalls. I want to thank him especially for his patience with my inability to limit the scope of the project.

Abstract

This paper proposes a new paradigm for general purpose knowledge representation, storage, retrieval, and inference.

Existing systems, such as databases, information retrieval systems, expert systems, and hypertext systems, are widely heterogeneous in the metaphors and terminology used to describe them, in the transfer protocols and languages they support, and so on. As a result it can be very difficult or impossible to make synergistic use of information from multiple systems, since they cannot communicate with each other.

But in fact these various seemingly different kinds of storage systems are performing fundamentally similar operations. Thus, a system which provides abstractions of these operations should be able to reproduce the functions of all of the preexisting systems simultaneously. Further, by encoding all knowledge in a standardized form, it becomes possible for information from different sources, perhaps even concerning different realms of thought, to be interconnected in new ways that may be very interesting and useful.

This paper describes a framework for such a general knowledge management system, based on an object-oriented network of nodes and links. Following the description of the basic architecture of the system, various issues surrounding its use are discussed, including: queries and inference, uncertainty, truth maintenance, user interface issues, object identification, and time.

Finally, I have begun to implement in software the ideas described in the paper. The current architecture and functionality of the software is covered in the final chapter.

Table of Contents

Introduction

In this paper I describe my intuitions about how knowledge of all kinds can be stored, retrieved, and intelligently processed by computer systems. The bulk of the paper is intentionally philosophical in nature: while I make perhaps extravagant claims about what computers can do in principle, I provide no algorithms or proofs. This project takes a view of computing on a large scale, and attempts to show how different strands of present-day computing, particularly involving object-oriented databases, distributed systems, and component-based software architecture, can be tied together to produce a coherent and elegant paradigm for general purpose computing. To really work out the details of the system I will describe, as several observers have pointed out, would require fifty doctoral theses. For now, I attempt only to explain the general idea as I see it, to make a map of the territory, and to identify a set of issues and ideas for further development.

At the same time, in the interests of rigor and practicality, I have begun to implement the system in software; indeed a large majority of the time I have spent on the project has been in coding. At present, I have running code, written in Java, demonstrating perhaps ten percent of the ideas in this paper. Programming time is the primary limiting factor on the software; no insurmountable theoretical or technical issues have arisen yet. Thus, I expect the capabilities of the software to continue to increase as I have more time to devote to it. The current architecture and capabilities of the implementation are described in chapter 7.

1 Philosophy

1.1 A General Purpose Information Structure Manager

It is extremely frustrating that there exists so much digitized information in the world, and that there is potential for orders of magnitude more, but that this information is stored in such heterogeneous ways that it does not cohere on a grand scale. Due to differences in representation paradigms and languages, transfer protocols, file formats, and so on, it is very difficult to make use of data from different realms in synergistic ways.

As a simple example, consider a physicist interested in gravitational lensing. She is on her way to a conference, and would like to talk to other physicists with similar interests when she gets there. Thus, she would like the computer to provide a list of physicists who are going to the same conference and who are interested in gravitational lensing *or related fields*, along with their photographs.

The list of conference attendees surely exists somewhere in digital form; somewhere else, or in many separate locations (such as personal web pages), the particular interests of each physicist are listed; somewhere else again (at the Department of Motor Vehicles, for example) digital photographs of each person exist; finally there may be somewhere a concept map, in some digital format, of how different fields of physics relate to each other. Clearly there is no way today to make synergistic use of these disparate sources to answer the query.

A recent project at Stanford and Epistemics, Inc., Infomaster, takes steps to solve this problem by introducing an information broker which communicates with diverse preexisting systems. The broker answers queries in a standardized language based on information from all of the connected databases, and is able to make inferences drawing from multiple sources. Thus the Infomaster user is given the illusion of a single large database. For example, if an Infomaster broker knows how to communicate with one database listing financial data for many corporations; a second, separate database (perhaps hosted on a different platform, using a different query language, and so on) containing statistics on industrial pollution; and a third database containing the telephone book, it can tell you the profits of the ten most polluting companies in the 650 area code.

This is clearly a good idea; but I propose a different approach which is more drastic, more long-term, and I think ultimately more powerful. In this paper I describe a completely abstract and general paradigm for storing and retrieving information of all kinds. If all information were encoded in a consistent and accessible way, in spite of perhaps vastly different conceptual content, it would be much easier—or even completely automatic—to answer queries based on information synthesized from many different sources, perhaps involving knowledge from what would normally be considered separate conceptual realms. Rather than developing a glue for connecting puzzle pieces that don’t fit, I suggest an overarching standard for the shapes of puzzle pieces.

I might request a map showing the current geographic density of postmodern philosophers. Right now there is surely somewhere a list in electronic form of postmodern philosophers; somewhere else there is a telephone book giving their addresses; and in a third place there is a geographic information system which can determine the physical location of a postal address. So in principle the query should be answerable today; but it is not in practice because the representation paradigms of the systems containing the three different kinds of information are so different, and they have no mechanisms for communicating with one another.

To complicate matters further, imagine that there is no explicit list of postmodern philosophers available after all. Still it should be possible to answer the above query by first constructing such a list, approximate of course, based on other information. For example, the system I propose might first do a keyword search for books concerning postmodernism, and make a list of the authors. Or it might do a two-step search, first finding journals having to do with postmodernism and then constructing the list of all people who have published articles in such journals; it might even include other authors who often cite sources from these journals (the necessary data is available in the Citation Index).

The very same system should be able—in principle—to tell me the name of the nearest supermarket which sells organic produce and has no complaints pending at the Better Business Bureau, or the total tonnage of ships sunk in World War II (not including submarines, unless they were sunk on Thursdays), or the major differences between Zen Buddhism and Taoism, or the pattern of neural connections between the anterior thalamus and the hippocampus in the brain, or the nearest common root in the etymologies of “tree” and “truth”.

A travel agent recently took a call from a woman who wanted a plane ticket to go to “Hippopotamus, New York”. The agent assured her that there was no such place, but the woman was adamant; only after some time did the agent realize that the woman meant “Buffalo” (Washington Post, May 15, 1998). This is a simple—if whimsical—case in which a single large information structure with a wide range of knowledge could have helped. Presented with this problem, such a system would rank the airports of the world in order of the strength of their conceptual relationships with “Hippopotamus” and with “New York”. Clearly “Buffalo” would be first on such a list (unless the hippopotamus was by coincidence John F. Kennedy’s favorite animal).

Databases (simple, relational, and object-oriented), expert systems, hypertext systems, semantic networks, bayesian networks, neural networks, and even file system hierarchies are all performing fundamentally similar operations concerning storage, retrieval, and inference; they process information with some internal structure. Thus, in spite of the different vocabularies and metaphors used to describe these different systems, I believe that a single abstract paradigm can provide the functionality of all of the above simultaneously—and more.

1.2 Why the Name

The system is for *organizing* information of all kinds.

The system is *organic* and dynamic, being constantly updated and improved through interaction with humans and internal processing. It is a growing and evolving entity—an *organism*.

The system is based on Dagobert Soergel's *Information Structure Management* paradigm.

1.3 Design Goals

1.3.1 Computing paradigm follows human thought, not vice-versa

Computing systems historically have imposed limitations on their users arising from theoretical considerations of computability and complexity, as well as limitations arising from the ways of thinking within which they were designed. While both sets of limitations are clearly impossible to overcome completely, a good information management system should be as flexible as possible; it should be able to represent any human thought which is in principle encodable. Second-order logic can be computationally messy, but human beings can and do often make statements of second (and third, and fourth) order, and are quite good at reasoning about them. The purpose of the sort of system I propose is not to be absolutely rigorous or provably correct about anything; rather it is meant to be extremely expressive and to make a best effort at processing information in ways that are intuitive, interesting, and useful for human users.

1.3.2 General Purpose

Many if not all existing database or knowledge base applications are limited to a certain realm of discourse, since they generally define certain tables and columns, certain inference mechanisms and rules, and so on. In the case of a relational database application, for example, the structure of the representation is always application-specific, and application-specific code exists outside the database and interacts with it.

A more elegant and flexible approach, I think, will be to make the basic storage, retrieval, and inference mechanisms completely abstract, and to encode any application-specific rules, algorithms, display methods, and so on within the knowledge base itself. Thus a single general engine will be able to drive a wide variety of applications, where the particulars of each application are stored just like any other data. The principle here is similar to that of the general Turing machine, which takes an encoded representation of any Turing machine as part of its input and simulates the given machine acting on the remaining input. The OrganISM is in the same sense a generalized environment for storage, retrieval, and inference.

1.3.3 Distributed

The World Wide Web has (in spite of its various failings) demonstrated the power of large-scale distributed effort in providing information. Any system of the scope I envision will clearly need to be widely distributed and decentralized in structure in order to encourage widespread participation, in order to handle the large storage volume and processing load, and in order to be robust against any local failures.

1.3.4 Emergent Order

The system must be constructed in such a way that two conceptually related pieces of information entered by different people at different times will be appropriately linked. Many existing systems are either like a bag of disconnected, loosely connected, or arbitrarily connected bits (like the Web), or they have a static structure (like most databases). The OrganISM is a self-organizing system which dynamically preserves the structure of the information it is given and builds on it in ways that we cannot necessarily predict or encapsulate in static constructs. As such a system acquires more information, its inferences should build on each other to produce a complex model of the given material. The purpose is not only to store information but also to synthesize it.

1.3.5 Subjective Relativism: Author, Time, Context

Claims of absolute certainty or truth are inherently problematic for any large knowledge system containing information relevant to actual human activity. Obviously, different people can have vastly different and contradictory beliefs, even on topics which they classify as matters of fact rather than matters of opinion. The system I propose must accommodate disagreements and contradictions, and must rigorously track the sources of the information it contains. Thus, along with every statement, the system must record at least the author and the time, as well as any other available contextual information which can later be used to qualify the statement, to assess its reliability, to draw inferences about the author or about the context, and so on.

As far as such a system is concerned, there is no objective reality; it can answer queries only from explicitly specified points of view, provided by users in the form of reliability ratings for different sources. Thus, different users might get vastly different results for the same query, depending on which sources they trust and what other assumptions they provide.

1.3.6 Efficient balance of Preprocessing Versus Realtime Processing

The system must strike a balance between storage-time processing and query-time processing. Enough storage-time processing must be done to make later queries efficient—e.g. indexing in the simplest case. In general, some set of basic inferences should be stored which connect the new datum at least to the general areas to which it relates, so that it can later be found without searching large amounts of irrelevant material. But there is little point in explicitly storing the results of time-dependent inferences or of inferences which can be rapidly reproduced.

For example, imagine that the system already knows that “all dogs are mammals”. Then, you tell it that “Fido is a dog”. Certainly this new datum must be linked to “Fido” and to the concept of “dog”, but it is probably unwise to explicitly store the inference “Fido is a mammal”, since this can easily be inferred later at search time. Also, the facts may change; we may discover that we were mistaken originally, and that Fido is in fact a frog. In the case that we did store “Fido is a mammal” explicitly, there are truth-maintenance mechanisms (described in section 4.6) which insure that this statement will be appropriately modified to “Fido is an amphibian”. The computational cost of truth maintenance increases with the proportion of inferred statements in the knowledge base, however; so in deciding when to store inferences explicitly, we must consider the cost of making the inferences once, the frequency with which we expect the inferred statements to be used, and the cost of maintaining their truth.

2 The Basic Structure

2.1 Everything is an object

2.1.1 Objects

The fundamental unit of storage in the OrganISM is the object (or node), and the system is structured as a network of objects and links between them. Absolutely everything which is encodable in principle can be encoded fairly intuitively as a set of nodes and links; indeed sentences of natural language essentially establish links between the concepts denoted by their constituent words, so any sentence can be modelled by a network appropriately linking the relevant concepts.

An object may have content in any form—text, images, sounds, movies, files in proprietary formats, compiled code, and so on. The granularity with which content is stored in objects is not bounded above or below by the system's intrinsic mechanisms; but of course choosing a granularity appropriate to some application is an issue, and will be discussed later.

2.1.2 Relations

In the OrganISM, *relations* between objects have semantic meaning (unlike, for example, links in the Web); they are represented as n-tuples in prefix notation. So

```
< “believes-in”, “Kant”, “A Priori Truths” >
```

expresses a particular relation between Kant and A Priori Truths.

Relations between objects are represented as objects in their own right. This makes it possible to make statements of arbitrary order, involving relations of relations, at infinitum. While this approach can quickly introduce extreme computational difficulties, it does mirror the flexibility of human thought and language.

In order to store relations as objects, it is necessary to introduce a lower level of linking them together; this is accomplished through untyped *edges*. So the relation

```
< “teacher-at”, “Plato”, “The Academy” >
```

is itself an object containing an ordered triple of *edges* pointing to the three elements of the relation—in this case, the first edge is to a relation type (a predicate), and the remaining two edges are to the arguments.

Edges should always be bidirectional, so each object has an edge to all the relations in which it participates. Exceptions may be made in cases where this would lead to objects with large numbers of edges of limited usefulness. The “name-of” relation type, for example, will have many, many instances, but searches will rarely be initiated from the “name-of” node. So in this case it will probably suffice to have an edge from each relation of type “name-of” to the “name-of” node and to forego the reverse edge.

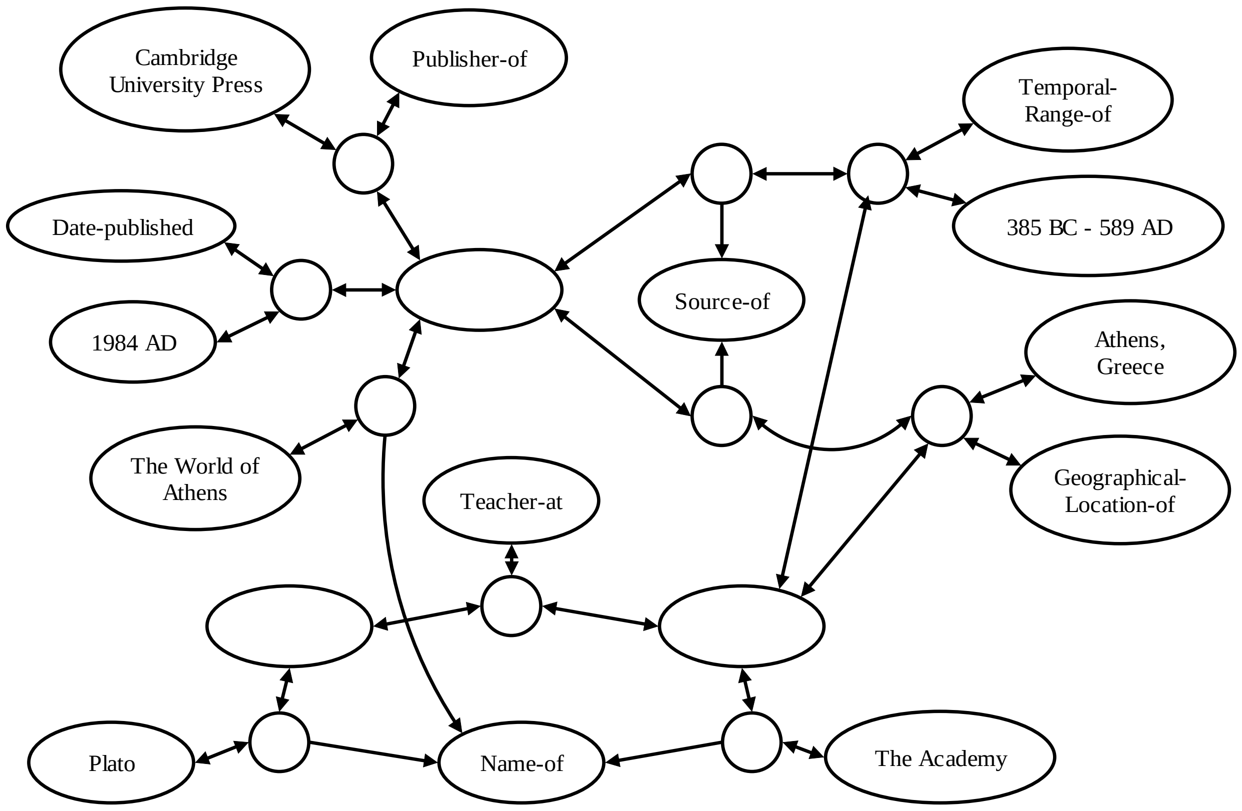


Figure 1. Example of objects and relations.

Figure 1 shows a set of objects linked by relations. The circles represent relation objects. Several of the primary object ovals appear empty because they represent a concept which has no convenient textual representation. The concept of a particular person is entirely different from the person's name, for example, which is just a piece of data about the person.

Thus the above diagram expresses the following: A person called "Plato" was a teacher at a place called "The Academy", which (according to a book called "The World of Athens" published in 1984 by "Cambridge University Press") existed from 385 BC until 589 AD in Athens, Greece.

Note that most edges are bidirectional, except those leading to "Name-of" and "Source-of"—which, being extremely common relation types, are exempt from the bidirectionality requirement. Also note that the "Source-of" relations refer to other relations; they specify the source of the claims represented by the relations. (The relation of type "Geographical-location-of" has an internal structure which allows the arguments of the relation to be distinguished from the edge to the associated "Source-of" relation).

Relations can express predicates with arbitrary numbers of arguments; while the above examples are all of two-argument predicates,

```
< “dead”, “Plato” > and
< “contains”, “Milk Chocolate”, “Cocoa Solids”, “30%”>
```

are equally valid.

2.1.3 Relation Types / Classes

Since everything is an object, relation types are objects; that is, the particular semantic meanings that relations can have are represented by objects. Thus, in

```
< “teacher-at”, “Mark Mancall”, “Stanford”>,
```

the first element in the relation points at an object representing the relation type “teacher-at”. There may be many relations of this type, e.g.

```
< “teacher-at”, “Albert Einstein”, “Princeton” >
< “teacher-at”, “Richard Feynman”, “Caltech” >
```

But there is a single object representing the relation type “teacher-at”, to which each individual instance is connected by an *edge*. Since we can make statements about the “teacher-at” archetype which will apply to all of its instances, that object represents the entire class of “teacher-at” relations, and behaves very much like a class in an object-oriented programming language.

The relation type may itself have relations to other objects which help determine its meaning, such as synonyms, broader and narrower terms, and inference rules (see below). It may have relations to methods which operate on relations of this type. These may include specialized display methods: for example, the “geographical-location-of” relation type may be connected to a map-drawing widget. Also, specialized search methods may take advantage of the characteristics of a particular relation type, so the “geographical-location-of” relation type may be connected to a search program which can consider driving distance based on a road map.

2.1.4 Rules

Since everything is an object, inference rules are objects. Rules are essentially relations between relation types. We might make an inference rule that everyone who is a teacher at a school is employed by the school. This is a relation between the “teacher-at” object and the “employed-by” object, which might be represented something like

```
< “implies”, <“teacher-at”, A, B>, <“employed-by”, A, B>>
```

Other examples:

```
< “implies”, <”area-code-of”, X, “650”>,
      < “geographic-location”, X, “San Francisco Peninsula”>>

< “probably-implies”, <”college-student”, X>,
      < “age”, X, range(17-22)>, “90%”>
```

Relation types present in the system may be quite redundant. Clearly there must be rules encoding the equivalence of synonyms. Also, it may often occur that a certain *chain* between two objects, consisting of relations of certain types in a certain order, implies some direct relation between the two objects. Such a chain can be named and represented as a single relation type, and the appropriate inference rules can link the long form of the chain with its single-relation representation.

For example, imagine we are interested in the general chain type

```
<”parent-of”, A, B> AND <”brother-of”, B, C>
```

We can name this chain by creating a new relation “uncle-of” and an inference rule such that the above statement implies

```
<”uncle-of”, A, C>
```

2.1.5 Meta-Rules, etc.

Since everything is an object—including relations, relation types, and rules—we can make arbitrarily complex structures of relations of relations, relations between rules, rules about rules, and so on. That is, this system can in principle represent logic of infinite order. The conclusions it will be able to draw from this representation are of course computationally limited, but this should not deter us. Natural language can express many questions which humans are unable to answer, and is nonetheless quite useful in practice.

2.2 Inheritance and Probabilistic Reasoning

Just like classes in object-oriented programming languages, relation types can inherit relations from each other, and can override relations that would otherwise be inherited. For example,

```
<”has-body-part”, “bird”, “wings”>
<”provides-ability”, “wings”, “flight”>
<”inherits-from”, “penguin”, “bird”>
```

From this, and the appropriate inference rules, the system could infer that

```
<”has-ability”, “penguin”, “flight”>
```

Since we know this to be false, we must override it with

```
<”lacks-ability”, “penguin”, “flight”>
```

where

```
<”excluded-middle”, “has-ability”, “lacks-ability”>
```

In this case, when a query involving the capabilities of penguins is evaluated, the system will find that penguins both fly (inferred) and do not fly (given), and that the two conditions are contradictory. The probabilities of each statement being true are then evaluated, as in a probabilistic network, and the more likely result is chosen. Explicitly given information is usually more reliable than inferred information, so the inherited inference will be overridden in this case.

But what if the claim that penguins cannot fly comes from a source which is known to be extremely unreliable? In this case the system may conclude that penguins can indeed fly, in spite of the attempted override. Further, depending on the reliability of

```
<”excluded-middle”, “has-ability”, “lacks-ability”>,
```

the system may find that the best solution is to report both possibilities. Obviously, the problem of tracking the probabilities of various statements and inferences can become very messy very quickly, both because there may be many levels of statements, relations, meta-relations, and so on, and because there may be feedback loops. Existing computational methods in probabilistic networks will need to be applied, and new ones developed, to deal with such issues.

Multiple inheritance is certainly possible in principle; but in practice complications will arise which will need to be dealt with—for example, if two inheritance pathways produce contradictory or otherwise conflicting results.

2.3 Object-Relation Duality

One of the features of the OrganISM is that it does away with the distinction between relations and objects (or between predicates and arguments) as much as possible. We have seen how relations are represented as objects and how it is therefore possible to make statements about relations. Conversely, every object is a relation: in the least case, the object embodies the cooccurrence relation between the various relations it is linked to by edges.

The two statements

```
<”tool-for”, “rope”, “rock climbing”>
<”activity-type”, “rock climbing”, “outdoor recreation”>
```

are related to each other by cooccurrence in the “rock-climbing” object, which contains edges to each statement. Thus, “object” and “relation” are really two names for the same thing—an object with edges to other objects—even though they have intuitively different functions, in much the same sense that “wave” and “particle” describe different manifestations of a single underlying physical reality.

Note also that if we add to the above the statement

```
<”tool-for”, “carabiner”, “rock climbing”>,
```

we find that “rope” and “carabiner” have two second-order cooccurrence relations between them: one via “tool-for” and the other via “rock climbing”. I call this “second order” because “rope” is two edges away from “rock climbing”, as opposed to the previous “first order” case in which

```
<”tool-for”, “rope”, “rock climbing”>
```

is only one edge away from “rock climbing”.

There is one apparent distinction to be made between objects and relations, which is that the order of the arguments of a predicate is usually important, while the ordering of edges in a cooccurence relation is arbitrary. But these are really two poles of a continuum. For some relations, the order of the arguments is more important than for others:
`<”believes-in”, “God”, “Nietzsche”>` is obviously very different from its converse, but
`<”married”, “Ronald”, “Nancy”>` is not. Finally, the second and third edges are reversible in
`<”subtractive-color-mixture-result”, “yellow”, “blue”, “green”>`.

Approaching the same point from the other side, the order of the edges in a cooccurrence relation may sometimes be relevant. When searching for documents related to some term, it might be useful to place documents in which the term occurs near the beginning at the top of the result list. If different keywords apply differentially to different documents, it would be best to record that fact explicitly (perhaps via a “percentage” or “rank” argument to the keyword relation); but the order of edges in cooccurrence relations may be useful as a backup if explicit ordering information is not available.

One way to emphasize the equivalence of objects and relations would simply be to create a "cooccurrence" relation type and to make the first edge (the predicate) in every object point to it by default.

At the same time, it will be necessary in some cases to distinguish within a single object between edges which are ordered with respect to each other and edges which are not. In a relation object, for example, the relation type and its arguments form an ordered tuple, which does not include any edges to statements about the relation (such as a "Source-of" relation) that may be present.

2.4 Program Objects

Since everything is an object, programs or scripts are objects. A program object can be run with a single object as an argument (which may contain several logical arguments encoded in some particular text format, or as a list of other objects, etc.) and returns a single object as a result (which may of course also contain arbitrarily much raw data and arbitrarily many edges). Note that, for compiled languages, both source code and compiled code can be objects; the present point concerns the latter, which can be invoked to do something.

This approach has several advantages. First of all, it encourages highly modular, component-based program development. In place of monolithic applications, I envision a more dynamic network of small, manageable parts. The component approach has already been applied in systems such as OpenDoc and JavaBeans, and is very much related to the idea of remote program execution or method invocation available through CORBA and Java.

Second, this approach allows the representation and execution of fully object-oriented systems. Although an object can take only one argument, the argument is itself an object which can contain the name of a method to be invoked and a list of arguments to the method. Given an object which represents my display screen, for example, I might send a message containing "DrawWindow" and an edge to a window object, resulting in the appearance of the window on my screen. This of course requires that the screen object really has an appropriate DrawWindow method; unless I have written such a method specific to my particular screen, the method is most likely inherited through the previously described inheritance mechanisms from a general "screen" class.

Third, since everything concerning a given application—including its programs—is an object, the entire application resides within the system itself. This is in contrast to conventional databases, for example, where the application-specific code which actually processes the data usually resides outside the database. This can make it more difficult to write, less portable, and so on. In the case of the OrganISM, the only code which resides outside the database is the OrganISM engine itself, which is completely abstract with regard to content. Anything content-specific is inside. In this sense the system is an operating environment within which programs may be run.

Finally, many core components of the OrganISM may themselves be stored inside the system. In particular, multiple different search engines may exist as objects, as may display methods for different kinds of content. In the same sense that most components of an operating system are stored on a disk which is made accessible by the relatively small booting portions of the OS, most of the components of an OrganISM system can be stored within the system itself.

It is important to note that the loading and execution of program objects is completely dynamic; there is no hard linking or early binding. Also, it is not important where the program runs (e.g. locally on the client machine, remotely on the machine where the object resides, or on a third server); ideally all the machines in the system together provide one big pool of computing resources. Efficiency considerations alone should dictate where and how fast a program runs. This approach supports and encourages parallelism, and many known algorithms for parallel computing (e.g. for load distribution, synchronization, and so on) can be applied to the implementation of the OrganISM.

2.5 Display Components

In keeping with the object-oriented philosophy of the system, objects should be able to display themselves. This is accomplished by a relation to a display method (program, object, component), which is often inherited or inferred. Arbitrarily many different display methods may exist to handle a certain kind of content, and different objects of the same type may specify different display methods.

For example, an image P of a penguin may have a relation
`<”mime-type-of”, P, “image/jpeg”>`.
Given the appropriate inference rules and a relation between “image/jpeg” and an image display component, the system can display the penguin image.

Another image might require a more specialized display component, however. For example, scientists studying volcanic activity on the moons of Jupiter may want to pass images of the moons to a more powerful viewer with options for false color, magnification, and various kinds of image processing.

3 Queries and Inference

3.1 Query, Query Engine, and Result Objects

Since everything is an object, queries and query results are objects. A query is passed to a query engine (which is itself a program object as described above). The query must of course have a format appropriate for that particular engine—some engines might parse English, others might implement SQL, some might search for images similar to a query image. Depending on the nature of the content and the searching possibilities, there may be a specialized tool for each engine which provides a user interface for creating queries. A search engine with a geographical component, for example, may have associated with it a program which generates queries in the appropriate format based on user clicks on a map.

To fulfill the promise of object orientation, query objects should themselves be executable. This requires only a relation between the query and the query engine, and a "run" method which causes the query to pass itself to the engine. Both can be inherited from an engine-specific query class or inferred from other known properties of the query.

The query engine produces a single object as a result. That result object may contain the familiar list of hits in the case of a search (perhaps sorted, color-coded, etc.), but can in principle have any kind of content—perhaps graphs representing statistical results, or graphical maps or trees representing conceptual relationships, or another query intended for a different engine. (Types of queries and the results they return are discussed in greater detail in section 3.7).

In addition to the result object, the query engine creates a relation between the query and the result, including any relevant contextual information (such as the time). This makes it easy to run the same query at different times, or on different engines, and later to compare the results—since they are all related to the query object.

The need to store every query and every result (and possibly even every intermediate result for some complex query engines) as an object gives rise to the concern that large masses of transitory material will overwhelm the primary content of the system. It may be possible to mark these objects as temporary and to delete them regularly, but this is not always the best approach; section 5.6 concerns the issues surrounding the relative permanence of different objects.

3.2 Query Types

Since query engines are just program objects, arbitrarily many different kinds can be created. Engines might differ in their algorithms, in their specificity with regard to content, in what inputs they expect, and in what sorts of output they produce.

There are, however, a few basic query types which are fundamental to the operation of the system. These involve searches concerning a simple structure consisting of two objects and a *chain* of relations and other objects connecting them.

The following query types have different intuitive meanings for the user, although they can all be executed by a single underlying search mechanism.

1. Given two objects, find all chains of relations connecting them (up to a given chain length).
2. Given an object and a relation type, find all objects related to the given object by a relation of the given type.
3. Given a relation type, find all tuples of objects between which a relation of the given type holds.
4. Given an object, find all objects to which it is related, and the types of the relations connecting them (up to a computationally feasible chain length or inference depth).

3.3 Inference

These queries seem conceptually simple enough at first, but become much more complex and computationally expensive when inference rules are taken into account, as they must be. Now, every relation involved in a query must be expanded to include all of its equivalent representations, within computational limits. For example, if I request a list of all the uncles of A, the system should return not only those objects which have an explicit “uncle-of” relation to A, but also those connected by a “father-of” – “brother-of” chain, providing an implicit “uncle-of” relation through an inference rule.

Since an inference rule generally has the form `<”implies”, <A, …>, <B, …>>`, and since edges are generally bidirectional, one can quickly do a one-level expansion from B to the set of all A’s which imply B. That is, if we want a list of some person’s uncles, we can run the inference rule backwards to discover that a “father-of” – “brother-of” chain would suffice to establish uncle-hood. Knowing that, we can now search not only for explicit “uncle-of” relations but also for implicit ones. Of course, each element of the first-level expansion may itself be deducible from some other inference rule; for example

```
<
      ”implies”,

      <”mother-of”, A, C> AND <”mother-of”, A, D> AND
      <”father-of”, B, C> AND <”father-of”, B, D> AND
      <”male”, D> AND <”not-equal”, C, D>,

      <”brother-of”, C, D>
>
```

So any given statement can have an arbitrarily deep tree of sets of other statements which would imply it. When searching for structures of objects and relations matching some pattern, the system should consider as many levels of expansion of the sufficiency tree as are computationally feasible. Since the expansion cannot be arbitrarily deep in practice, inferences are made only on a best-effort basis; there may be implications present in the structure which the system does not find because they are too complex.

Upon making an inference, the system may explicitly store the result for future reference. In this case, we must somehow record the fact that the statement was an inference made from certain premises via a certain rule. This is accomplished by establishing a relation between the objects in question. The relation type is simply the inference rule itself, since the rule expresses the semantic relationship between the premises and the conclusion. After the initial edge to the rule, the next n edges in the relation point to the n arguments of the rule—the premises. The final edge points to the derived object.

Inference rules provide a further example of the fundamental equivalence between different ways of thinking of objects, like the object-relation duality previously described. In this case, an inference rule is simultaneously 1) a relation between relation types and 2) a relation type for inference relations.

If the results of many inferences are stored explicitly rather than left to be rederived at search time, then the effective depth of backwards inferences will be increased, or the speed of queries will be increased, or both. For example, imagine that we can expand five levels of backwards inference from some node in reasonable time. If inferences are often stored explicitly, we may find relations at level five which were themselves generated from five-level inferences. Thus the effective depth of the current inference is ten, at the expense of a lot of storage space and truth-maintenance computation time for explicitly stored prior inferences.

We can see now that inheritance is just a special case of inference, accomplished through the basic rule

```
<”implies”, <”inherits-from”, X, Y> AND <R, X, Z>, <R, Y, Z>>.
```

It is easy to specify that an object inherits some relation types but not others. This may be particularly useful for avoiding conflicts in cases of multiple inheritance. The associated inference rule is

```
<”implies”, <”inherits-from”, R, X, Y> AND <R, X, Z>, <R, Y, Z>>.
```

3.4 Conceptual Grouping

3.4.1 Optimal Query Results

Some queries may return more results than a human user can deal with. In this case the system should be able to group the results into conceptual categories which are as different from each other as possible, and display a characteristic member of each category. Depending on the application, selecting one such member might expand a tree and show characteristic members of subcategories, and so on. In many cases the optimal number of results to return will be seven plus or minus two—the number of slots in human short-term memory (Gio Wiederhold, personal communication).

There are two problems here: first, how should the system determine conceptual distances between objects? Second, how can it find a set of seven (or *n*) which are maximally different from each other? The first problem is particularly complicated by the fact that different measures of difference will be applicable to different kinds of content, and worse, that different measures of difference will be interesting to different people even for the same kind of content.

In choosing a rental apartment, for example, one user might be most concerned about location, a second might be most concerned about price, and a third might be most concerned about noise level (e.g. nearby railroad tracks). Rather than doing a standard sort with several levels of keys, however, the system should find a set of maximally different objects which takes a number of dimensions into account simultaneously, with different user-specified weights for each dimension.

This problem can be modeled as a geometric problem in an n-dimensional Euclidean space, where each dimension corresponds to some variable which is to be used in calculating distances between objects. Thus the problem of finding maximally different objects is strongly related to a number of problems which have been studied extensively, and a number of known algorithms may be useful in solving it—such as shortest-path algorithms, graph layout algorithms, principal components analysis, and the Kohonen map algorithm.

3.4.2 Common and Distinctive Characteristics

The system should also be able to answer the reverse question: given a set of objects, what characteristics best describe the set as a whole? The question has two variants. First, what characteristics do the objects in the set have in common? Second, which of these characteristics are distinctive?

A query of the first sort seeks the smallest set of properties which are related to the largest number of objects in the set; it seeks a description of the set which bounds it above, so there may be many nodes which fit the description but are not in the given set.

A query of the second sort seeks the smallest set of properties which describes the largest number of objects in the given set while describing as few objects outside the set as possible. This query seeks a description of the set which bounds it below.

3.5 Pipes and Scripts

Since queries take objects as input and produce objects as output, they can be easily strung together with pipes. In addition to the standard query types, the system should provide some simple utility functions which may be useful in processing query results. One such function, for example, would take a chain object as input and produce a list of the objects on the chain; another would filter a list based on some criteria; another might flatten a list of lists.

A complex query object consists of a script which calls other query engines and support functions. Such a script might, for example, make a list of all objects related to A by a relation of type R, then find all chains between the resulting nodes which do not include A, and return a list of all objects on such chains. (A and R are specified in the input to this script). When A is a businessman, and R is “business associate”, then this query returns a list of objects having to do with ways in which A’s business associates are related to each other (e.g. two of them might work for the same firm, another two might be members of the same club, a third pair might be cousins, and so on).

3.6 Neighborhood Searches

A “neighborhood” is simply a set of objects. In practice the objects in a given set will usually have some conceptual relationship to each other; indeed the set may often be defined by a concept search of some sort to begin with (e.g. the set of all rock bands in San Francisco, or the set of all chapters, sections, and paragraphs of a book).

3.6.1 To-Neighborhood Search

Dagobert Soergel (1998, p.34) describes the “to-neighborhood search” in which the target of a search is an entire neighborhood rather than an individual object. For example, a search for all books concerning both “metallurgy” and “plastics” must return not only those books for which both keywords are listed but also books which have one chapter on metallurgy and another on plastics, but which do not list the two keywords explicitly at the book level. Each entire book is thought of as a neighborhood, and since the search is a “to-neighborhood” query, it seeks books which have relations to both keywords somewhere in the neighborhood, even from different chapters. This is different from a search for books which contain at least one chapter which concerns both topics.

The need for the to-neighborhood concept is particularly evident in searches on the World Wide Web. It is not possible with current search engines to find an entire web site as a unit which matches two keywords appearing in different pages. Search engines catalog only on the page level, not realizing that pages are often grouped into conceptual and organizational units. For example, a search for “casper AND socrates” should return http://www.stanford.edu, since both terms appear—on different pages—one level below the home page (Gerhard Casper is the university president, and Socrates is the name of the library system’s online catalog). Sadly, none of the search engines I tried found Stanford.

In the OrganISM, this functionality is covered by the search and inference mechanisms. First, there should be inference rules which cause keywords to propagate up containment hierarchies, e.g. from paragraph to section to chapter to book. Thus, when the search is done for books concerning two different keywords, the inference mechanisms take care of the upwards inheritance to the neighborhood level (in this case, the book as a whole).

It may be useful in this case for keyword relations to have a strength which diminishes as the relations are passed up the hierarchy by inference, but which are appropriately added when several subsections of a node have the same keyword. Thus a book with a whole chapter on metallurgy may be more highly ranked in a sorted result than a book with only one paragraph about it; and a book with three chapters on metallurgy will be more highly ranked still.

The search for books concerning “metallurgy” and “plastics” is essentially a search for books which occur on chains between “metallurgy” and “plastics” such that the chains traverse only relations of type “covers” (or some other keyword-like type) and “contained-in”.

```
<”covers”, “(MMS Chapter 4)”, “metallurgy”>
<”covers”, “(MMS Chapter 6)”,  “plastics”>
<”contained-in”, “(MMS Chapter 4)”, X>
<”contained-in”, “(MMS Chapter 6)”, X>
<”is-instance-of”, ”book”, X>
<”name-of”, X, “Modern Materials Science”>
```

So there is a matching chain:

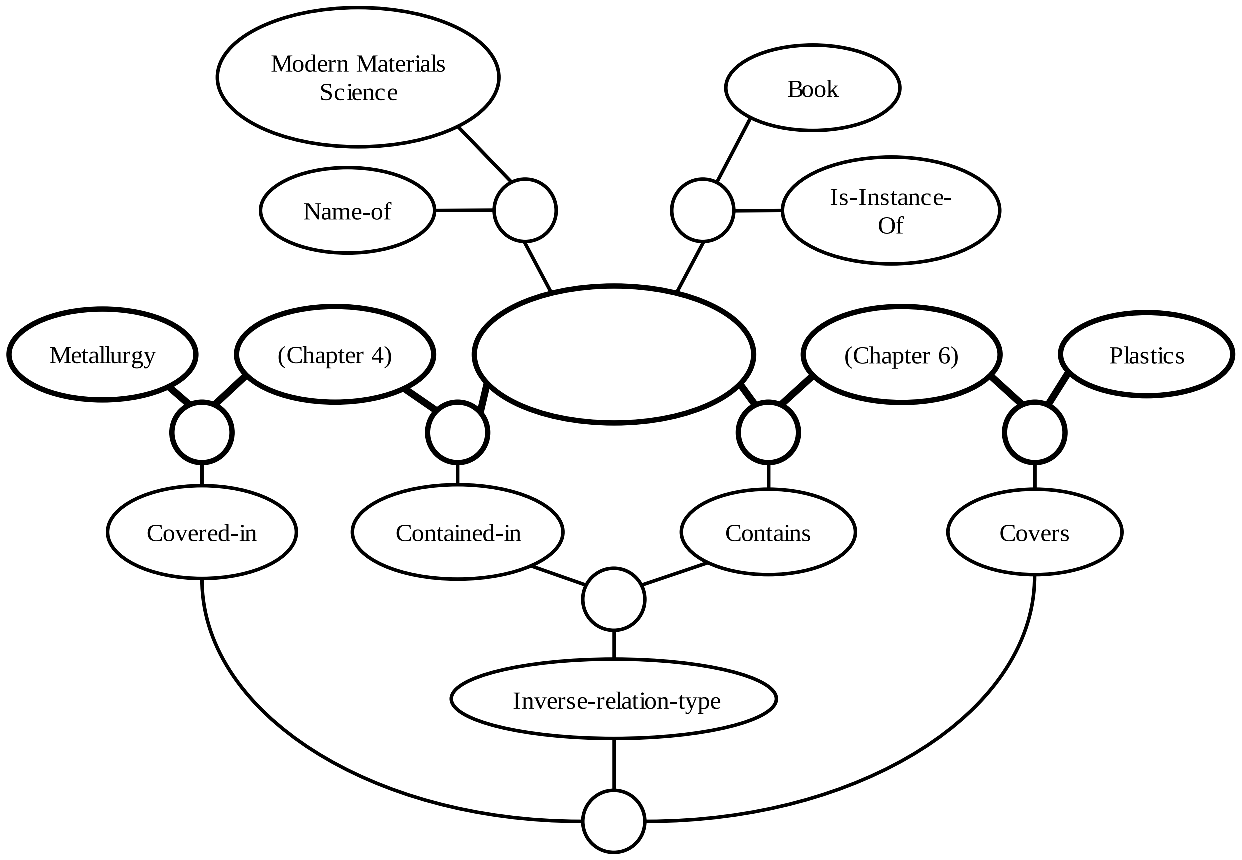


Figure 2. A chain matching a query for books concerning both metallurgy and plastics.

The following is a much looser search:

“Give me a list of all the authors whose works might be relevant to both neural networks and music.”

To execute such a query, the system might find all the chains connecting neural networks to music (up to a computationally feasible length), flatten the chains into lists of objects, combine the lists, and select out all of the authors (where “author”, like anything else in a query, is subject to expansion by backwards inference to include, say, anyone who has ever published a text document).

3.6.2 From-Neighborhood Search

A user might also want to do a search starting from a neighborhood. For example, rather than performing a simple search for documents explicitly matching a keyword, a user may want to search from the neighborhood of narrower terms of the keyword (Dagobert Soergel, 1998, p. 36). Thus a classified ad for an “amplifier” will be returned by a search from the narrower-term neighborhood of “musical equipment”.

This functionality again is covered by the inference mechanisms, since
```
<”is-instance-of”, “amplifier”, “musical equipment”>
```
(or “narrower-term”, etc.)

Thus a search for classified ads for “musical equipment” will automatically return ads for “amplifiers”, though these may ranked lower than ads that are explicitly for “musical equipment”.

3.7 Display of Query Results

The type of result that a query produces is arbitrary, but there are a few common types which deserve mention.

3.7.1 List response

Many queries will return lists of hits. In the case of a search for all of a person’s uncles, for example, the result will be a list of objects (encapsulated, as any query result, into a single result object). List display components may have various options for sorting, color-coding based on given criteria, and so on.

The main function of the list display component is simply to call an appropriate list-item display method for each element in the list and to lay out the resulting panes in a container. Thus, if a query returns both text objects and image objects, for example, the list can show both text items and image thumbnails.

A search for all chains between objects A and B will return a list of objects, each representing a distinct chain; similarly a search for all tuples of objects between which some relation holds will return a list of objects representing the tuples. In either case, the user is probably not interested in the encapsulated representations, but rather wants to see each chain or tuple fully specified. This is accomplished by the list-item display methods associated with chain and tuple objects: these components should display some appropriate representation of the contents of the node on which they are called.

In some cases the user may want not only a list of hits but also some information associated with each hit. For example, I might request a telephone directory of people who work in my building. The relevant search is for people who work in my building, but I want their telephone numbers included when the list is displayed. This functionality should also be handled by the list and list-item display components.

3.7.2 Tree response

Some query responses will be best displayed as a tree. Different display components may display the tree differently. One component might show a fan chart or pedigree chart; another might represent the tree in the collapsible-list form familiar from file managers in Windows or MacOS.

The simplest tree that might be constructed is the tree of all edges starting from a given node. The first level would show the relations in which the node participates; the second level would show the objects participating in each relation, the third level would show relations of each of those, and so on. Such a tree can be arbitrarily deep and must therefore be constructed on the fly by the display component as the user expands or contracts branches. There would be little point in repeating a node in every list that occurs two levels below it (because it is obviously a member of each of its relations); this is a special case that should be filtered out.

The display component should be able to filter which children of each node it displays by relation type. For example, I might have an object representing my hard drive; I might ask for a tree representation of its contents (e.g. filtering for relations of type “contains”) , or I might ask for a tree showing the derivation of the drive. Depending on how “derived-from” is defined, this might include the serial number, the batch number, the model number, other model numbers from which aspects of the design of this drive were borrowed, the factory, and the manufacturer. The tree might also include the construction company which built the factory, the engineers who designed the drive (and their ancestors!), the computers on which it was designed, the manufacturers of those computers, and so on. That is, this request could produce the tree of all factors which eventually contributed to the existence of this hard drive.

Similarly, a filtered tree request starting from some body of water could show a tree of all waterways which contribute to it and of all factories in the watersheds of those waterways; this might be useful in tracking down potential sources of a pollutant.

One issue with displaying trees in this way is that the structures being displayed will rarely be purely hierarchical; thus, nodes might appear repeatedly in different branches. For example,

```
Synergy House
 |<"Geographical-location", "Synergy House", "Stanford University">
 |   |Geographical-location
 |   |Stanford University
 |
 |<"Responsible agency", "Synergy House", "Housing and Dining Services">
 |   |Responsible agency
 |   |Housing and Dining Services
 |   |  |<"Department-of", "Housing and Dining Services", "Stanford University">
 |   |  |  |Department-of
 |   |  |  |Stanford University
 |
 |<"Political bias", "Synergy House", "left">
 |   |…
 |
```

"Stanford University" appears in two different places in the tree. This is not problematic, but the display component should probably make some indication (by color-coding, for example) that the node is displayed more than once.

3.7.3 Graph response

For many query results, and for navigation in general, a graphical display of nodes connected by lines may be appropriate. Laying out such a graph in a sensible way is difficult, but there are algorithms available which do reasonably well for small numbers of nodes.

As in the case of trees, it will usually be useful to display only relations of certain types. Further, relations of certain types can be color-coded or otherwise emphasized. The display component should offer navigation options (e.g. zooming in and out, redrawing the graph around a new node, etc.), and should allow the user to rearrange the graph by dragging nodes around the screen.

3.7.4 Statistics response

Some queries will provide statistics of some sort that can be displayed in tables, pie charts, bar graphs, and so on. Obviously there are very many options for the display of such charts which the statistics display components might implement.

Most statistics queries will probably concern the *content* of the system; for example, they might involve counting the nodes meeting some criteria. The generation of a chart showing the ethnic makeup of a university student body would require counting the students in different groups.

Some statistics queries will concern the *structure* of the OrganISM itself; for example, "show a graph of the distribution of number of edges per node among nodes in the set S".

3.7.5 Animated response

Some programs, such as dynamical systems simulators, may produce a time sequence of result objects. Indeed many query engines may return sets of result objects varying along one or several continuous dimensions, of which time is only one possibility. An animation display component would be useful for showing such results. Animations may be most useful when displaying statistics or sequences of related images (other result types, such as lists of objects, are not easily animated).

3.7.6 Combined responses

The response types given above are basic examples; many more complex results can be imagined. Simply by combining the basic types above one might produce a graph where the color coding of nodes and links varies over time with some statistic on each object. A display component which did this might be useful, for example, in monitoring the load on computers and network links in a distributed computing environment. The same component could be used to display the relative success of different populations in an ecosystem over time, where relations between species encode the many interdependencies in such a system.

4 Features and Benefits

4.1 Expressivity

I believe that the system I have described is capable of expressing any thought which is in principle encodable in way which lends itself to computation while remaining intuitive for human users.

```
<”believes”,
      ”David Soergel”,
      <”for-all”, X,
            <”implies”,
                  <”has-property”, X, “encodable in principle”>,
                  <”there-exists”, Y,
                        <”expresses”, “OrganISM”, X, Y>
                        AND <”has-property”, “computable”, Y>
                        AND <”has-property”, “intuitive”, Y>
                        >
                  >
            >
      >
```

Thus the system follows the first design principle (Section 1.3.1).

4.2 Granularity

There is no theoretical upper or lower bound on the granularity with which objects are represented in the OrganISM. (In practice, the upper bound on the size of an object is imposed by the available storage space, and the lower bound is one bit). For example, the text of an entire book may be stored in a single object, or the text of each chapter may be stored in objects which are “contained-in” the book object. The paragraph or even the sentence level is probably most appropriate for books, though an exhaustive index would essentially require word-level granularity. In principle, an entire book could be represented simply by an appropriately structured network of relations starting from the letters of the alphabet; but this would obviously be highly inefficient.

As far as the storage and inference mechanisms are concerned, in any case, the granularity level is arbitrary, so it can be chosen by human users to be appropriate to the type of content and the intended use. Further, granularity can vary widely within the knowledge base: the system may simultaneously store one book in a single object and another with word-level grains. For queries at the book level (e.g. involving author, keywords, and so on) these will behave equivalently. In the fine-grained representation, given appropriate inference rules, keywords may propagate up from paragraphs to the book object; and this is transparent to the user searching for books with certain keywords. For queries at the paragraph level, however, the coarse-grained book cannot be considered, since it contains no paragraphs as far as the information structure is concerned.

4.3 Normalization

Whenever two objects have identical content (not taking their relations into account), they are considered to be equivalent. When this occurs, the two objects must be related by an equivalence relation. Two objects with identical content but without an equivalence relation are not permitted (empty objects—that is, objects with relations but without primary content—are exempt).

Multiple equivalent objects with different relations are logically treated as a single object which has all the relations present in the set. This is accomplished by an inference rule

```
<”implies”, <”equivalent”, A, B> AND <X, …, A, …>, <X, …, B, …>>
```

In principle each concept should have only one object representing it, but in practice the sort of redundancy described above may be useful for efficiency reasons. In a widely distributed system with network bottlenecks, for example, one might want to keep multiple copies of commonly accessed objects in geographically disparate locations (e.g. like the internet’s nameserver system). Further, it may turn out that some algorithms for navigating the information structure can be made more efficient if there is a maximum branching factor—that is, a requirement that every object have at most n edges. In this case, an object with more than *n* logical edges can be split into multiple objects related by equivalence, each carrying a subset of the edges.

The required equivalence of objects with identical content enforces normalization of the information structure to second normal form (2NF). Whenever two objects have a property in common, they must both be related to the same object which represents the property, or to two objects representing the property which are related to each other by equivalence. While redundant objects may exist, their redundancy must be explicitly recorded. Thus it is not possible to introduce errors by updating some object without updating redundant copies of the same information.

Normalization to third and fourth normal forms is encouraged, but is not necessary or enforceable. Indeed efficiency considerations may sometimes favor a “degeneration” to 2NF, since this can shorten the inference chain needed to establish some often-used statement. For example, in a database of sales transactions, it may be desirable for reasons of speed to explicitly record the total price of each transaction, even though this could be calculated on the fly from the numbers and prices of the items sold, sales tax, and shipping information. In this case the primary data is still in 4NF, and the additional relations which make the structure act in some ways like 2NF are just explicitly stored derivations.

4.4 Subjectivity and Uncertainty

Whenever an object is stored in the knowledge base, the system tracks, via a relation, the source of the information. In the case of primary data entry, there is simply a relation to an object representing the user. A login is required so that these relations can be established automatically; data entry without a user record is not permitted. In the case of derived statements, sources are tracked as previously described.

In addition to making statements in the positive, users can explicitly negate statements simply by entering their converses. Thus it may be that some statement would have been considered true by default, due to some inferences or inheritances, but that the user overrides this default. The OrganISM does not make the closed-world assumption, so disagreeing with some statement is quite different from having no explicit opinion about it.

In cases where the OrganISM is used to duplicate the functionality of database and knowledge base systems available today, all statements will generally be considered to be very reliable (e.g. if the inventory database claims that there are 54 sprockets in the warehouse, there is a 95% chance that this is true). But ultimately it may be used to describe much larger and more complex realms in which there are large uncertainties, differences of opinion, and so on. The system accommodates disagreements, contradictions, and paradoxes, since it stores any information it is given without judgement. Nonetheless, information about the source and reliability of each piece of knowledge may allow the system to provide useful results.

When using the system, users must specify which sources they trust and to what degree. This allows the system to sort and filter results based on their reliability according to each individual user’s premises, according to probability-network calculations. There are no absolute truths; knowledge is always viewed through a certain lens, from a certain perspective, and the OrganISM requires that this perspective be explicitly specified.

Of course, it is not practical for each user to rate the reliability of each other user or source. To overcome this problem, a user might subscribe to a particular reliability rating service; she may use a weighted average of the views of her friends to establish reliability measures; she may use an average over all users who have given an opinion on a certain source; and so on. Conversely, if a user explicitly enters reliability measures for some sources, and makes these publicly readable, other users may ask the system to take her views into account in answering their queries.

There are political and moral dangers lurking here. For example, a small group of reliability rating services may end up defining the lenses through which most people see the world, in precisely the way that governments and mainstream media do today. Conspiracy theories may arise and propagate, in the form of a set of reliability ratings such that any statement supporting some theory is thought to be reliable while any statement contradicting it has zero or negative reliability. But these dangers, and many more, exist in the real world; the OrganISM simply allows us to construct models of them. The system aims to be maximally democratic and tolerant of diversity by allowing each user to specify (and publish) his or her own worldview.

4.5 Ranges of Reference and Ranges of Validity

4.5.1 Time range of reference

Time is represented just like any other knowledge, with relations to standard calendar objects representing certain times or ranges of time.

```
<"time-event-occurrence", "Reunification of Germany", "October 3, 1990">
<"contained-in", "October 3, 1990", "October 1990">
<"contained-in", "October 1990", "1990">
```

The same calendar is used for recording contextual information about the objects themselves, e.g.

```
<"object-created-by", "Reunification of Germany", "David Soergel">
<"time-object-created", "Reunification of Germany", "May 10, 1998">
```

Some time relations might encode recurring events:
`<"time-event-occurrence", "Reunification Day",`
`"October 3 in years following 1989">`, where we have defined the object called "October 3 in years following 1990" to be a class containing individual days, e.g. `<"instance-of", "October 3, 1991", "October 3 in years following 1990">`. (The class might also be defined by an inference rule, so that each member need not be specified explicitly).

Some objects or relations may refer to a certain time period. This can be recorded through edges or relations to the calendar, as appropriate:

```
<"holds-public-office", "Bill Clinton",
      "President of the United States", "1992", "1996">
<"holds-public-office", "Bill Clinton",
      "President of the United States", "1996", "2000">
```

4.5.2 Time range of validity

Some objects may be valid only during a certain time period. That is, regardless of the time to which an object refers, its truth value may change over time: it may take effect at one time, and expire at another time. Keeping track of the time range of validity for each object will allow us to ask the system to evaluate a query as it would have been evaluated at any previous time. That is, this allows us to ask not only "What are the health effects of cholesterol," but also "What would the system have told me about the health effects of cholesterol if I had asked in 1993?"

By default, objects take effect when they are created. An object's "time-takes-effect" relation may refer to a time after the creation of the object, in which case the system will ignore the object in the interim; and it may refer to a time before the creation of the object, indicating that the meaning expressed by the object was true in the world before it was encoded. Similarly, a "time-expires" relation means that the object no longer holds after the given date. A "time-expires" relation should never refer to a date before the creation date of the relation; objects cannot be expired retroactively. Objects should never be deleted, but they may be marked "expired" as of the current time.

Imagine that a NASA probe discovers next year that, contrary to previous evidence, there is life on Europa (a moon of Jupiter) after all. Let

```
X = <"believes", "NASA", <"has-property", "Europa", "lifeless">>
```

and

```
<"time-object-created", X, "May 10, 1998">
<"time-takes-effect", X, "1979">
<"time-expires", X, "August 12, 1999">
```

This means that Europa had been believed lifeless since 1979, when Voyagers I and II looked at it, but that Galileo's observations prove this to be false on August 12, 1999.

The time range of validity associated with a statement specifies the time during which the statement is valid; so the time range during which a person believes a statement can be specified via the validity range on the belief relation. Thus, in the example above, the existence of life on Europa did not change in 1979 or in 1999; only NASA's belief in it changed. Presumably the life itself existed, if at all, for millions of years.

Specifications of time range of validity may be useful in conjunction with relations tracking multiple different versions of an object. If the evolution of the object is purely linear, then the time validity relations will be sufficient to establish the order of the versions; but if the evolution of the object has a tree structure, independent version specifications will be required. In any case, these relations allow the user to request, for example, the Macintosh version of some piece of software as of some date.

It is important to note the distinction between 1) the time range an object refers to and 2) the time range when the object is valid. The former is used to answer queries about the state of the world at some time according to *current* information; the latter is used to answer queries about the state of the world according to the content of the information structure *as of a given time*. Although Bill Clinton will stop being the president in the year 2000, the fact that he was president from 1992 through 2000 will continue to be true. So if we ask, many years from now, "Who was president in 1998", the system should easily respond "Bill Clinton". But if we wanted to know who would be president in 1998 *according to the information available in 1990*, the system could provide a wild guess at best.

4.5.3 Ranges on other variables

The same distinction can be made concerning geographic ranges of reference and of validity, or indeed ranges of reference and of validity on any variable (e.g. demographic groups, members of certain organizations, and so on).

The geographic reference of

```
X = <"North-of", "Oklahoma", "Texas">
```

is obvious. The range of validity of this statement is “everywhere on Earth except in Guyana”, since access to maps is strictly controlled by the Guyanese government.

```
<”valid-in-geographic-area”, X, “Earth”>
<”invalid-in-geographic-area”, X, “Guyana”>
<”valid-for-group”, X, “Government of Guyana”>
```

Thus, *according to the information available in Guyana*, the geographic relationship between Oklahoma and Texas is unknown, except to members of the government.

Similarly, the sentence

> “Schrödinger’s equation for the electron pairs in a superconductor gives us the equations of motion of an electrically charged ideal fluid.”

is valid only for physicists (and only after a certain date). No one else is equipped to judge the truth of the statement, in just the same way that no one was equipped in 1990 to judge the truth of the statement “Bill Clinton will be president in 1998.” Learning sufficient physics to evaluate the above sentence is analogous to waiting a few years to find out who will be president: the point of view moves from the range where the statement is not valid into the range where it is.

4.6 Truth maintenance

Recall that, whenever an inferred object is stored in the knowledge base, its derivation is stored as well. Because all of the relations in question are bidirectional, it is easy to start from a given object and find all derived objects which depend on it. That is, we can simply follow any inference relations (relations whose type is an inference rule) recursively to construct the dependence tree. Thus, whenever we delete or modify an object, the system can check, and delete or update if necessary, any derived statements which depend on it.

We must also take into account the time range of validity of the premises in determining the truth of an inference. The conclusion is valid in the time range given by the intersection between the ranges for all of the premises. This fact can be expressed by an inference rule; so, like any other inference, the times of validity for the conclusion can be inferred when needed, or can be explicitly stored. If the validity range is changed for some object, the standard truth maintenance mechanisms update the range for any derived objects.

The subjective nature of truth presents a seemingly more difficult problem. If one user disagrees with a statement entered by another user, then by implication he disagrees with anything derived from that statement as well, unless otherwise indicated.

But this is really not a truth-maintenance problem at all; and the existing mechanisms for subjectivity and inference cover this case. In disagreeing with an existing object, the user does not modify the object, but rather adds new objects encoding his belief that the object is false. Thus the truth-maintenance mechanisms are not activated. When the user runs a query, it must be executed from his specified point of view. The usual probability and inference mechanisms will recognize that objects derived from an object with which the user has disagreed have a low probability of being true according to that user.

It may occur, on the other hand, that a user disagrees with a premise in some inference but nonetheless believes the conclusion. When a user registers his truth judgement of an object, the user interface might show him the tree of derived statements, giving him the opportunity to assign independent truth values to the conclusions.

4.7 Semantic Redundancy

The system may contain several objects representing the same concept, or several objects which, though not identical, are very highly related. There is no prohibition against redundancy, but all semantic relationships should be encoded so that the inference engine can take them into account.

Since relation types are just objects, the system requires no controlled vocabulary, and users can add new relation types at any time. Thus, there may be several redundant relation types present—for example, one user may use a relation type “constructed-by”, while another may prefer “built-by”.

In order for queries to work properly, the system must know when objects are equivalent. There are two ways of accomplishing this: we must either create two inference rules saying that each object implies the other, or there must be a “synonym” relation between the terms and an inference rule specifying how synonyms work. There may of course be more complex redundancies which must be appropriately modelled—between “husband”, “wife”, and “spouse”, for example.

Although the system can in principle model these relationships, the question arises of who will enter them. If I create a new relation type “constructed-by”, how can I discover that “built-by” is already in the system? There are several possibilities here.

First, the system may consult an authoritative thesaurus and dictionary to find a list of possible synonyms and related terms; it can then present these lists to the user with the suggestion that some semantic relations should be established.

Second, once the new term has been used in a number of relations, the system should be able to analyze the structure to find terms that are often used in similar contexts; it may then suggest to the user that these terms might have a semantic relationship. For example, once I have used my new term “constructed-by” in a few relations, I can request a list of structurally related terms. In response, the system will first construct a list of pairs of objects related via “constructed-by”; it can then return a list of other relation types which relate the same pairs. That is, it seeks terms which directly correlate with “constructed-by”. Next, the system can follow the “isa” or “instance-of” hierarchy upwards, seeking correlations at every level. Thus it can observe that every pair of objects related by “constructed-by” consists of a physical object and some sort of manufacturer; since many preexisting relations between physical objects and manufacturers have the type “built-by”, the system can suggest that “built-by” and “constructed-by” are semantically related.

This process becomes more difficult if a term has several different meanings or is used in multiple contexts. For example, we can say in English that an argument or proof is "constructed-by" a person. While we can model the analogy between physical construction and conceptual construction, this sort of thing can rapidly lead to intractable complexity if we are not careful. On the other hand, this is just the sort of calculation that might produce interesting semantic results. (For example, Doug Lenat's Cyc system can make inferences like "the father of a family is something like the king of a country".)

Even when two words are related by a synonym relation, they are still separate objects which may differ in other ways. This is important for two reasons. First, different sources might not agree on the synonymy of two terms; we can track this easily since the synonym relation itself must be related to its source. Second, different terms which are generally accepted to be synonymous may nonetheless express subtly different shades of meaning, or may be used in different linguistic contexts. In an alternative representation, one object representing an abstract concept would have relations to the various words expressing the concept; but this structure makes it difficult or impossible to track sources of claims of synonymy or subtle differences between words. So it is preferable to use a structure which treats each word as a separate entity.

Most instances of semantic redundancy will be much more complex than synonymy: some words are synonyms of each other some percentage of the time, or to a certain degree, or only under certain conditions; and some relations will be synonymous with chains of other relations. These cases are covered by the inference mechanisms previously described. Through inference rules we can express, for example, that in European culture the word "uncle" refers to the brother of a parent 90% of the time, and to an older male family friend 10% of the time, but that in Chinese usage the percentages are closer to 40% and 60%, respectively.

4.7.1 Multiple Languages

We can take advantage of the mechanisms for dealing with semantic redundancy to enter knowledge in many different languages: so the system may contain both "railroad station" and "bahnhof".

```
<"synonym", "railroad station", "bahnhof">
<"language-of", "railroad station", "english">
<"language-of", "bahnhof", "german">
```

4.8 Data vs. Metadata

Metadata is information used in locating data sources. The distinction between data and metadata concerns only how the information is used; there is no inherent difference in the structure of the information or in how it is stored.

In the OrganISM, query processing involves navigating around the object network in order to find objects meeting some criteria. Thus, any objects which are accessed in the course of answering this query, except for the results themselves, are being used as "metadata": they direct the inference engine (and the user) towards the results. But the same objects are probably interesting in their own right, and will be used as “data” in other contexts. Therefore everything is simply data related to other data, with no distinction of levels.

4.9 User preferences

The system should contain an object representing each of its users. As with any object representing a person, there may be links to various knowledge the system has about the person—contact information, relatives, interests, affiliations, publications, personal philosophy, and so on.

Since user preferences concerning the OrganISM itself are simply facts about the person in question, these are stored in just the same way as any other knowledge.

As much as possible of the look and feel of the user interface should be stored within the knowledge base, in the form of display objects, layout managers, and so on. A user may prefer one particular image-viewing component over others, in which case this is recorded as a relation of an appropriate type (e.g. "preferred-image-viewer") between the user and the component.

The ability to use different display components based on user preferences allows easy integration of access technologies, such as magnified text and image viewers, text-to-speech converters, and the like.

Further preferences might be:
```
<"preferred-screen-layout", "David Soergel", L>
```
(where L is an object containing a set of window types, positions, and sizes).
```
<"preferred-screen-font", "David Soergel", "Times 12">
<"preferred-autologout-time", "David Soergel", "10 minutes">
<"preferred-email-notification-time", "David Soergel", "immediate">
```
and so on.

In addition to the general advantages of treating all knowledge in the same way, this approach allows the user's preferences to be effective regardless of which terminal the user uses in a widely distributed network.

In addition to general preferences, a user may store viewing preferences associated with particular objects. When viewing a collapsible tree, for example, the pattern of expanded and collapsed branches may be stored so that the tree will appear the same the next time it is opened (file listings on the Macintosh behave this way). Also, a user might view a two-dimensional network representation of some set of objects, rearrange it by dragging the objects around on the screen, and save the arrangement as a new object (related to the set of objects being displayed and to the display component). Thus, the next time the user views the same set of objects through the same viewer, they will be properly arranged.

Such arrangements might be used in more complex ways: for example, if the user views a graph of a superset of a set of objects which he has previously arranged, then the previous arrangement should be taken into account in laying out the new graph. Further, two two-dimensional arrangements, in which the objects are grouped by different criteria, might be synthesized to produce a three-dimensional representation.

4.10 System settings

Like user-level preferences, server-level settings can be stored inside the system. These settings might include a schedule for automatic backups, the size of the server's object cache, a list of administrators to contact in case of a problem, and so on.

4.11 Security

Knowledge about security and privacy of objects is just like any other knowledge, and consists primarily of relations between objects and users or groups expressing various permissions. The OrganISM server is responsible for enforcing these security measures.

```
<"permission-read", "David Soergel's email", "David Soergel">
<"permission-read", "David Soergel's publications", "Everyone">
<"permission-write", "Project XYZ paper", "XYZ Project Team">
```

These relations mean, respectively, that I have read permission on an object representing my email; that everyone may read my publications; and that write access to a paper about Project XYZ is granted to (members of) the XYZ Project Team.

Of course it would be tedious to specify permissions explicitly for each object; but, like any other relations, permissions can be inherited from other objects, overridden if necessary, and so on. So, for example, any object which inherits from a private object is also private unless otherwise specified.

All forms of access to an object should be denied by default, so permissions must be given in the positive. It will nonetheless be useful to explicitly deny permissions in some cases—in order to override an inherited permission, for instance.

The security mechanisms can be used to implement information hiding. At the program level, this is done simply by making certain objects executable but not readable to (most) human users. Thus users might run a query engine using the publicly available interface, without being allowed to examine the program object itself. Of course source code for many programs is unavailable today; but in this case, even the compiled code is unavailable (this may be useful in a commercial setting to avoid hacking and reverse-engineering). The same principle operating at the method level allows an object to have a public interface while keeping its internal structure hidden to other methods—except perhaps "friends" and children in a class hierarchy.

4.12 Self-organization and emergent order

We can imagine that there is some average probability that some random two pieces of information are fairly closely related. Thus, when we add a new object to the system, the number of preexisting objects which are related to it is—on average—proportional to the size of the knowledge base. Assuming that some proportion of these relations are actually established when the object is added, then, the ratio of relations to primary objects increases as the size of the knowledge base increases.

Imagining for the moment that we add morsels of knowledge to the system in no particular order (e.g. rather than one topic at a time), the information structure initially consists of a number of disconnected chunks with no relations between them. But as we add more objects, the ratio of links to nodes increases; and as the ratio passes 1/2—that is, when there is one relation for every two primitive objects—an important phase transition occurs. At this point, the size of the largest internally connected subset of the network increases rapidly and begins to approach the size of the entire network. (Kauffman 1993)

Thus, when the network becomes large enough, there is a high probability that a chain exists between any two given objects. The critical network size at which this occurs depends on the probability of a relation between any pair of objects. This is just a mathematical fact about systems of nodes and links, and is applicable in many realms; indeed Stuart Kauffman has suggested that life originated when this gelling process occurred in chemical reaction networks.

The crucial question in the case of the OrganISM is that of how to actually establish all the relevant relations upon adding an object. Automated addition of relations via inference rules helps once humans have established some initial relations, but not before. In order for the network to gel, we must establish relations between groups of objects which were previously unrelated. Since inference rules depend on preexisting relations, they cannot do this. Inference rules serve only to increase connectivity within groups that are already internally connected.

Once a human has established a single bridge between two previously separate groups, inference rules may be able to tie together many other parts of the groups, based on the initial connection. (Humans are sometimes fortunate enough to have this experience, in which the initial insight of a single connection between two fields or realms of thought rapidly leads to the realization that there are many connections between the two fields, or even that what seemed like two separate realms were really two sets of terminology for the same thing all along).

A significant part of the motivation for the design of this system arises from a faith that a large knowledge base will in fact gel in this way. The hope is essentially that, when a new object is added, a relatively small number of explicitly created relations from it will suffice to establish the object's place in the information structure, and that many relations can then be derived automatically which will make the object really well-connected and hence useful.

For example, if I record the single fact that the body sizes of male and female squirrel monkeys are roughly equivalent, the system should be able to infer that squirrel monkeys are a pair-bonding species, and hence that they are likely to exhibit a whole host of particular social behaviors. If I later request a list of primate species in which males often care for children, then, squirrel monkeys should be on the list (ranked by some measure of the probability of this behavior). This should occur even if no one has explicitly created or described the category "pair-bonding species", since the system should automatically take into account strong correlations between sexual dimorphism and male parenting behavior in what it already knows about primates.

Thus the design of the system is intended to encourage self-organization, emergent order, and “information resonance”.

5 Interesting & Difficult Issues

5.1 Object Identification

5.1.1 Object Uniqueness

In general, objects must be identified by some set of criteria concerning either their content or their place in the information structure (e.g. by their relations). The problem is that there is no guarantee that a given set of criteria will return only one object. While objects do have unique identifiers, these are internal to the system and should not be visible to users. For a user to uniquely identify an object, he may need to provide several pieces of identifying information. A real-world example of this problem occurs at the Stanford Medical Center, when patients ask for "Dr. Weiss". In fact there are two doctors called "Eric Weiss" who both work in the Emergency Department. They are generally distinguished by their middle initials (A or L), their beardedness, or their hair color.

There should never be two objects in the knowledge base which are completely indistinguishable based on their contents and relations. In general, the position of the object in the information structure will be enough to distinguish it by description. Also, objects in certain realms may be given unique identifiers—such as social security numbers, ISBN numbers, and UPC codes.

Even when two objects have the same contents and are related by an equivalence relation, each individual object must have a distinguishing relation of some sort. In some cases, equivalent objects will reside on different servers, in which case a "stored-on-server" relation could be used to distinguish them. Otherwise, a simple index relation would suffice, analogous to the "copy 3" labels found on library book call-number stickers. These relations will be propagated through the equivalence relation, like any other relation; but the probability mechanisms will cause the local distinguishing relation of each object to have a higher certainty than any inferred relations.

5.1.2 Object Names

"The name of the song is called 'Haddocks' Eyes.' "
"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. The name really is 'The Aged Aged Man.'"
"Then I ought to have said, 'That's what the song is called'?" Alice corrected herself.
"No, you oughtn't: that's another thing. The song is called 'Ways and Means': but that's only what it's called, you know!"
"Well, what is the song, then?" said Alice, who was by this time completely bewildered.
"I was coming to that," the Knight said. "The song really is 'A sitting on a Gate': and the tune's my own invention."
-Lewis Carroll, "Through the Looking Glass and What Alice Found There"

One potentially confusing consequence of full normalization is that several objects may have the same name, simply because multiple objects in the real world have the same name. "Name" is just another piece of metadata about an object. So we might, for example, have two completely different people where

```
<"name-of", X, "John Miller">
<"name-of", Y, "John Miller">
```

In this case each person is represented by an object (e.g. X and Y) which probably has no primary content at all but has a number of relations representing facts about the person. A name is simply a property that a person or thing might have.

How then can we deal with statements like
`<"citizen-of", "John Miller", "United States">`?

The "name-of" relation is special because it is frequently used to identify objects; thus it may be convenient for the system automatically to dereference names when evaluating queries, unless otherwise specified. That is, when the expression evaluator comes across "John Miller", it should locate the object containing the text "John Miller", follow any "name-of" relations, and return the set of named objects. If we wish to make a statement or query concerning the name itself, rather than the named person or people, we might use a special notation which suppresses the name dereference, such as ""John Miller"".

This automatic dereferencing of names is just a shorthand for a more complex query or statement. Thus,
`<"citizen-of", "John Miller", "United States">` is equivalent to

```
<"name-of", A?, ""John Miller"">
<"name-of", B?, ""United States"">
<"citizen-of", A, B>.
```

This statement means: find objects called "John Miller", find objects called "United States", and make the former citizens of the latter.

`<”citizen-of”, “John Miller”, “United States of America”>` would establish the same relation(s), since “United States” and “United States of America” are just alternative names for the same country.

Depending on the implementation, the above assertion might result in a new “citizen-of” relation for every John Miller in the system, when we probably intended to affect only one object or a small set of objects. If a name refers to multiple objects, how can we uniquely identify one of them? In this case the user must provide additional information. The system can help by finding distinguishing characteristics and asking, for example, “Do you mean John Miller, the bricklayer from Portland, or John Miller, the stockbroker from Chicago?”

The system may also make assumptions based on the logical distances between the objects in question and the user, since the user is more likely to make statements about objects which are somehow related to him or her, or about which she has made other statements recently. For example, if the user is married to someone named John Miller, then the system can safely assume that this user’s references to “John Miller” and even to “John” are meant to refer specifically to her husband, unless otherwise specified—unless of course someone else in the user’s conceptual neighborhood (e.g. her son, her dentist, etc.) is also named “John Miller”.

5.2 Deletion

Objects should be deleted only rarely if at all; in general it will be preferable to leave the object in the knowledge base but to mark it obsolete, as described in section 4.5.

In cases where it is really necessary to delete an object, considerations of referential integrity require that the deletion be carried out very carefully. Most importantly, how deep should the deletion be? If an object has relations specifically describing it, then those relations may become useless when the object is deleted—so perhaps direct relations should be deleted as well.

But in some cases these relations may contain information which is useful even without the deleted object, and this information should be salvaged. For example, suppose that we have a relation

```
<”gave-to”, “Fred”, “Barney”, “Car”>
```

and that we delete “Barney” from the knowledge base. Even without the edge to “Barney”, the relation expresses the interesting fact that Fred gave his car away—information that would be lost if we automatically deleted every relation involving “Barney”.

In other cases, a deletion invalidates not only the direct relations but even objects several relations away. For example, deleting a document probably involves deleting all of its constituent parts (but not necessarily!). Certainly any derived objects, which may be arbitrarily distant, will need to be updated or deleted via the truth maintenance mechanisms.

Thus, the appropriate depth of deletion is completely dependent on the particular content, and it will generally be necessary to ask the user how much to delete.

5.2.1 Salvage by Substitution

One possible method for salvaging information surrounding a deleted object is simply to keep all of its relations but to replace edges to the deleted object with edges to one or several of the object's parents in the "isa" or "instance-of" hierarchy. So, for example, if we delete the "cat" object, we can retain all of the relations concerning cats, replacing the reference to "cat" with a reference to "mammal".

There is a danger of overgeneralization here, however: it may be necessary to distinguish universal from existential scope in the resulting relations. Given a relation
`<"makes-sound", "cat", "meow">`,
deleting "meow" results in `<"makes-sound", "cat", "sound">`, which seems reasonable. But deleting "cat" results in `<"makes-sound", "mammal", "meow">`. The original statement about cats simply meant "all cats meow", but the new one seems to mean "all mammals meow". Since "cow" inherits from "mammal", the inference mechanisms will now conclude that cows meow, simply because we deleted "cat". This is an example of the difficulties that arise when the quantification of relations between classes is ambiguous, as discussed in section 5.4.

5.2.2 Salvage by Placeholders

To avoid possible problems associated with substituting instance-of parents for deleted items in relations, while keeping the relations, one option is simply to substitute a meaningless placeholder. The carries less information, but is also less likely to result in erroneous inferences. This would give rise to relations like
`<"gave-to", "Fred", -----, "Car">`.

5.2.3 Salvage by Inference

Using placeholders is really not ideal, however. If the above relation really contains interesting information, and if the system is properly normalized, then there should be a relation which encodes the meaning without empty arguments:
`<"gave-away", "Fred", Car">`.
As long as "Barney" was present in the knowledge base, this fact was derivable via an inference rule, but it must be explicitly stored before "Barney" is deleted.

Thus, whenever an object is deleted, all inference rules should be found in which the object plays a role in the premises but not in the conclusion; and the results of invoking these rules should be stored. Assuming that we have a rule
`<"implies", <"gave-to", X, Y, Z>, <"gave-away", X, Z>>`,
we should invoke it and store the result every time an object is deleted which fills the Y place in the rule.

5.2.4 Inheritance

Inherited relations can be salvaged by the above mechanism. If A inherits properties from B, and B is deleted, then we must explicitly record that A has the relevant properties. This is accomplished by the "salvage by inference" process operating on the basic inheritance rule
```
<"implies", <"inherits-from", X, Y> AND <R, X, Z>, <R, Y, Z>>
```

Since X occurs in the premises but not in the conclusion, this rule should be invoked and the results stored every time an object is deleted which fills the X place in an inheritance relation. This will cause inherited properties to be salvaged when the inheritance parent is deleted.

5.3 Copy

There are several issues surrounding the duplication of objects. First, when an object is copied, it must be determined how deep the copy should be. If I make a copy of a book, does this involve only a new object with relations to all the existing chapters, or must the chapters be copied as well? Second, is the copy simply a duplicate, expressing the same meaning, or is it to be modified in some way to represent a different concept?

5.3.1 Copy as Equivalent

Because objects with identical content are considered equivalent, and must be related by an equivalence relation (as discussed in section 4.7), a copy of an object with primary content will automatically be equivalent to its original. It may be desirable to make geographically distant copies of this sort, for example, for efficiency reasons. While the copy must have an equivalence relation to the original (and a copied-from relation, to facilitate source tracking and truth maintenance), the relations of the original object need not be duplicated or referred to in the copy, since they can be reached through the equivalence relation.

The relations concerning an object are just objects themselves, so the same efficiency considerations which lead to the copying of some object may apply to its relations as well. If a network bottleneck is the motivation for a copy, for example, copying an object without also copying its relations may produce little benefit, since the relations will still need to be retrieved from the original source through the network. (On the other hand, copying only one object may produce a great benefit, e.g. if the object is very large but its relations are very small). Further, it may be desirable to copy an arbitrarily large set of objects surrounding the first copied object. The OrganISM system may suggest a certain copy depth based on efficiency calculations, but ultimately the decision must be left up to the user.

A good user interface can assist the user in determining the depth of copy. For example, a collapsible tree representation could be used, in which any object selected for copying can be expanded to allow selection of the objects pointed to by each of its edges. Thus, starting from some object, the user could choose relations on the first level, related objects on the second level (where there will be at least one—the relation type—per relation, and probably more), relations of related objects on the third level, and so on.

When relations are copied along with an object, they must be modified to refer to the copied object rather than the original, and the copied object must be modified to refer to the copied relations rather than the original ones.

5.3.2 Copy as Template

In some cases the intent may be not to create equivalent objects but rather to make a copy as a template to be modified. When an object is copied for this purpose, some of its relations will probably need to be copied as well, since the new copy will be orphaned without them; as before, the user must select which relations should be kept in the copy.

If a user attempts to update the contents of an object which has equivalency relations, the system must ask whether it should update all equivalent objects as well, or whether it should sever the equivalencies. The latter choice allows the user to copy an object for use as a template, modify it, and save it without affecting the original. Even if the object has no primary content, adding, deleting, or modifying its relations should result in the same question: should the change apply to all equivalent objects, or only to this one (in which case the equivalencies must be severed)?

An equivalence relation is essentially a bidirectional “inherits-from”, and thus allows relations to propagate implicitly through the inference rule
`<”implies”, <”inherits-from”, X, Y> AND <R, X, Z>, <R, Y, Z>>.`
As a result, severing them (either by marking them obsolete or by outright deletion) can be dangerous, since this can cause relations that should hold for an object to become inaccessible. The “salvage by inference” process (section 5.2.3) solves this problem, however: when an equivalence relation is severed, any implicit relations which would otherwise be lost are explicitly propagated.

When a copy of any object is made, the copy should be related to its original by a “copied-from” relation. This allows proper source tracking by distinguishing objects created by a user from objects merely copied by the user, and more generally by distinguishing originals from copies.

5.4 Quantification in Relations between Classes

Apparently there are implicit quantifications in English such that, in a statement “A verb B”, where A and B denote classes or categories, A is interpreted as universal (e.g. for all A), but B is interpreted as existential (e.g. for some B), resulting in different quantifications for active and passive voice. So “music is written by composers” means “for every piece of music there exists a composer”, while “composers write music” means “for every composer there exists a piece of music”. Which does `<”writes”, “composer”, “music”>` mean? Apparently “writes” and “written-by” are really not strictly inverse relation types when they concern classes rather than individuals.

These different quantifications could be explicitly modeled with inference rules; for example, instead of `<”writes”, “composer”, “music”>` we could say either

```
<”forall”, X,
      <”implies”, <”instance-of”, “composer”, X>,
            <”there exists”, Y,
                  <”instance-of”, “music”, Y> AND <”wrote”, X, Y>
            >
      >
>
```

or

```
<”forall”, Y,
      <”implies”, <”instance-of”, “music”, Y>,
            <”there exists”, X,
                  <”instance-of”, “composer”, X> AND <”wrote”, X, Y>
            >
      >
>
```

or both, depending on which we mean. But this is obviously much more complex and tedious, so it would be nice to find a reasonable default behavior for
`<”writes”, “composer”, “music”>`.

5.5 Negation

When working with any large database or knowledge base, the implementation of the Boolean “not” operator (applied in the context of the entire universe) is impractical, for two reasons. First, it can easily return enormous result sets—often approaching the entire contents of the system (e.g. there are one million people who live in San Francisco, and six billion people who don’t). Second, while it is easy to navigate relations that exist explicitly or can be inferred, it is computationally expensive to navigate based on relations that don’t exist, because this can require testing large numbers of objects for the existence of the relation.

In the OrganISM, “not” is supported only in the sense of subtraction, e.g. “A and not B”; it is assumed that this will suffice in practice. One must have a tractable set at hand in order to subtract things from it, so the result will always be of tractable size as well.

It is worth pointing out that the evaluation of an expression involving subtractions can be speeded up by algebraic manipulation of the expression to minimize the sizes of the sets being subtracted. For example, since subtraction of an m-element set from an n-element set is o(nm),
the time for (A-B)-C is `size(A)size(B) + size(A-B)size(C)`, while
the time for (A-C)-B is `size(A-C)size(B) +   size(A)size(C)`, and
the time for A-(B+C) is mostly `size(A)size(B+C)`.

Which of these alternatives will be fastest depends on the relative sizes of the three sets and on their degrees of overlap. For operations on large sets, it may be worth doing a preliminary calculation to estimate the overlaps in order to choose the best subtraction method.

5.6 Temporary Objects

A total prohibition against deleting objects will rapidly lead to an overwhelming amount of junk in the knowledge base, since every query, every query result, and perhaps many intermediate results produced by query engines will be saved. So the prohibition must be relaxed somewhat to exempt objects which are reproducible.

There are various reasons why some queries and query results should be stored, however. If a query operation is very computationally intensive, frequently used, and always produces the same result, then of course the result should be stored explicitly out of efficiency considerations. In addition, users should have the option of conferring permanence on any query or query result.

For some query types, the execution of a preexisting query object should result in the immediate display of the stored result from the previous execution of the query, with an indication of the time when it was executed (e.g. "valid as of 15 hours ago"). Only if the user requests an updated response does it need to be computed again. If prior results are stored for a given query, this also allows the system to rapidly answer the query as of a previous execution time, without recalculating the result using the time-dependent validity machinery (Section 4.5).

Rather than making derived objects either completely fleeting on the one hand or completely permanent on the other, we can assign varying lifetimes to objects, depending primarily on the computation time needed to reproduce them and the frequency with which they are accessed. Thus an object X might have a relation

```
<"time-to-delete", X, "December 15, 2000">.
```

The way in which these lifetimes are calculated will determine the balance between query speed and storage space, and can be adjusted as necessary. A garbage collection mechanism can delete objects when they come to the end of their specified lifetime.

6 Example Applications

Movies

```
<”director-of”, “The Passionate Adventure”, “Hitchcock”>
<”year-released”, “The Passionate Adventure”, “1924”>
```

“Give me a list of all movies such that members of the cast later married each other”.

Geography

```
<”lat-and-long-of”, “Cupertino”, “ 37’21 N, 121’57 W “>
<”contains”, “Santa Clara County”, “Cupertino”>
```

“What is the distance from Cupertino to New York City?”

“What is the telephone number of the U.S. Congressional Representative for Cupertino?”

“What is the nearest national park to Cupertino with an average visitor density of less than one person per square mile?”

Foods

```
<”ingredient-of”, “Milk Chocolate”, “Cocoa Solids”, “30%”>
```

“I’m allergic to curry in concentrations higher than 10 ppm. Which Indian restaurants within a 30 mile radius of my home serve at least five entrees that I can eat?”

Philosophical argument maps

```
< “believed-in”, “David Hume”, “Empiricism” >

A = <”instance-of”, “brain”, “computer”>
B = <”implies”,
      <”has-capability-sometimes”, “brain”, “consciousness”> AND A,
      <”has-capability-sometimes”, “computer”, “consciousness”>
    >

<”believes”, “Zenon Pylyshyn”, A>
<”believes”, “Zenon Pylyshyn”, B>
<”disbelieves”, “John Searle”, A>
<”believes”, “John Searle”, B>
```

“If I believe both Immanuel Kant and Michael Polanyi, on which points do contradictions arise?”

Note that the same mechanisms can be used to have an ongoing intellectual discussion between multiple users of the system, not just to encode arguments which have already occurred.

Encoding arguments in this way should have the effect of focussing the discussion on the root issues of a disagreement, and allows immediate reference to primary sources, along with reliability measures of those sources.

Mathematical theorems

```
<”derived-from”, “Theorem A”, “Theorem B”>
<”implies”, “Theorem C”, “Theorem D”>
```

“I have disproved theorem X. What other theorems have I thereby disproved by implication?”

Email and newsgroups

```
<”message-from”, X, “Josh Peterson”>
<”message-to”, X, “Robin Hughes”>
<”message-subject”, X, “Shocking Gossip”>
<”message-text”, X, “blah blah blah”>
```

“Show me any messages I haven’t read yet if they’re from my closest friends or concern topics that I’m particularly interested in”.

Linguistics and natural language processing

```
<”instance-of”, “adjective”, “red”>
<”syntax-substitution-rule”, “noun”,
            <”syntax-block”, “adjective”, “noun”>>
```

“Is the sentence ‘Colorless green ideas sleep furiously’ well formed?”

```
<”implies”, <”has-property”, X, “colorless”>,
                  <”lacks-property”, X, “color”>>
<”instance-of”, “green”, “color”>

<”instance-of”, A, “ideas”>
<”has-property”, A, “green”>
<”has-property”, A, “colorless”>
```

“Is the sentence ‘Colorless green ideas sleep furiously’ sensible? What does it mean?”

```
<”date-occurred”,
      <”said”, “Jeffrey”, “I’m going to buy some milk”>,
      “August 24, 1998” >
```

“Is there milk in Jeffrey’s refrigerator on August 23?”
“Is there milk in Mary’s refrigerator on August 25?”

(Maybe; the system might know that Mary and Jeffrey share a refrigerator).

History and Culture

```
<”date-event-occurred”, <”invaded”, “France”, “England”>, “1066 AD”>
<”headed-movement”, “Gandhi”, “Satyagraha”>
<”influenced”, “Zoroaster”, “Nietzsche”>
<”instance-of”, “Allan Ginsberg”, “Beat Poet”>
```

“What relationships might have existed between Georges Braques and futurist architects?”

Politics

```
<”received-contributions-from”, “Newt Gingrich”, “Philip Morris, Inc.”>
<”friend-of”, “Bill Clinton”, “Vernon Jordan”>
<”causes”, “clearcutting”, “erosion”>
```

“Who are the ten most influential people in determining American policy concerning economic aid to Israel?”

“For what reasons might Representative Peter DeFazio in particular be opposed to NAFTA?”

“What arguments might I use in lobbying against the sale of U.S. Forest Service lands to logging companies? What evidence can I present to bolster my arguments?”

Chemical reaction networks

```
A = <”chemical-reaction”, “iron”, “oxygen”, “iron oxide”>
<”catalyzes”, “water”, A>
```

“What series of reactions might have produced saccharine, given the materials available in my lab?”

Molecular biology

```
<”binds”, “saccharine”, “amino acid sequence S”>
<”contains”, “Protein Q”, “amino acid sequence S”>
<”instance-of”, “Protein Q”, “ligand-gated sodium channel”>
```

Gene regulation networks

```
<”encodes-protein”, “Gene K”, “Protein K”>
<”activates”, “Protein K”, “Gene L”>
```

Neural networks

```
<”synapse”, neuron 358, neuron 2044, 0.38>
```

Physics

```
<”related-to”, “gravitational lensing”, “dark matter”>
<”origin-of”, “gamma ray bursts”, “unknown”>
<”maybe-origin-of”, “gamma ray bursts”,
                        “coalescing binary neutron stars”>
```

“What are the current major open questions in general relativity? Who is actively working on them?”

“What subjects would I need to study in order to understand the latest developments in quantum electrodynamics? What texts cover these subjects?”

Operating systems

Just as software processes and hardware devices are represented as files in UNIX, they can be represented as objects in the OrganISM. Thus priority information, ownership and security restrictions, and so on can be represented just like any other information.

File system hierarchies and GUI environments

```
<”icon-for”, my todo list, salamander icon>
<”contained-in”, my todo list, “Personal files”>
<”preferred-window-size”, my todo list, “250”, “300”>
```

Programming

One could use the OrganISM to make maps of programs during development: not just class hierarchies, but containment hierarchies, reference nets, which methods call which other methods, possible paths for exceptions to travel up the stack, links everywhere to relevant documentation, and so on.

Prolog

Prolog can be implemented fairly straightforwardly in the OrganISM. The representation and inference mechanisms are already in place, so this is mainly just a question of parsing Prolog syntax.

7 The Implementation

7.1 Overview

I have begun to implement a working prototype of the ideas described in this paper. The implementation has a three-tiered structure, consisting of a storage server, a mediator, and a client. The server simply stores and retrieves objects, much like a web server. The mediator is responsible for all of the interesting computations. The client provides the user interface, like a web browser.

The three components communicate through network sockets, and the connections are many-to-many. That is, a server can simultaneously serve many mediators; a mediator can communicate with many servers; and a mediator can serve many clients. (A client needs to communicate with only one mediator at a time.)

The code is written entirely in Sun's Java 1.1 programming language, and uses the Swing 1.0.1 library for the user interface. In principle all three components should be able to run on any machine with an appropriate Java virtual machine. Unfortunately, there are slight incompatibilities between different vendors' VMs; as a result, the client currently runs only under the Sun VM. Problems of this sort should evaporate as the Java language matures.

7.2 Object Identifiers

An object is uniquely identified by the server it resides on and an index number. An object identifier follows the URL convention and has the form

```
      istp://servername/objectnumber
 e.g. istp://satori.stanford.edu/25934
```

the protocol "istp" (for "Information Structure Transfer Protocol") is described below.

These identifiers should not be visible to human users. Users should identify objects by their place in the information structure (e.g. by name or some other attribute), as described in section 5.1. Thus, users can identify objects in a location-independent way.

But clearly the system must have a lower level of location-dependent identification, in order to be able to actually locate the objects; that is what these object IDs are for.

7.3 Object Structure

The current implementation can deal only with plain text objects. Multimedia capabilities will be added later.

At the lowest level, an object is stored as a text string of the form

```
<ISML>
<HEAD>
head
</HEAD>
<BODY>
body
</BODY>
</ISML>
```

The head can contain edges of the form

```
      <Link URL>
 e.g. <Link istp://satori.stanford.edu/25934>
```

The body can contain arbitrary text, but angle brackets and backslashes must be escaped with a backslash. When multimedia capabilities are added, the body will contain an arbitrary binary blob.

The body must contain all of the object's edges (and may associate them with particular words, areas of a picture, and so on, as in HTML). The edges are redundantly stored in the object head for efficiency: inferences and other calculations based only on the topology of the information structure can be done based only on the head, without needing to load or parse the object body. The contents of the body are authoritative; whenever the body is changed, the head must be regenerated from it.

In the current implementation, the object body consists of plain text with interspersed links of the same form as the links in the head.

7.4 Server

The server listens for connections on a network socket (number 7878 at the moment). Once a connection is established, the server allows simple operations (read, create, update, delete) on the local object store.

In the current implementation, the local store is simply a file with objects stored in it sequentially. An index allows rapid access by object number. When an object is deleted, its space is reclaimed.

There is a working JDBC interface for the server, so any database for which a JDBC driver exists (e.g. any reasonably popular database) may be used for backend storage. Because of the extra cost of creating SQL queries and parsing their results, however, the direct file approach seems to be faster for now.

In principle the server need not store the objects in any particular format, so long as it conforms to the transfer protocol.

7.4.1 Information Structure Transfer Protocol (ISTP)

The transfer protocol currently supports: echo, close, read, create, update, and delete. There is no "write" command; writing a new object is accomplished by creating an empty object and then updating it. In addition to these, there should be a command (e.g. "readheader") for reading only the head of an object without the body, but this is not yet implemented.

The syntax is as follows:

Echo: Returns the given string.

Query: `echo \n` *`string`* `\n \0`

Return: *`string`* `\n \0`

Close: Closes the network socket.

Query: `close \n \0`

Read: Returns the full text of the requested object.

Query: `read \n` *`idnumber`* `\n \0`

Return: *`objectstring`* `\n \0`

Create: Creates an empty object and returns the number.

Query: `create \n \0`

Return: `created` *`idnumber`* `\n \0`

Delete: Deletes the identified object and returns a verification.

Query: `delete \n` *`idnumber`* `\n \0`

Return: `deleted` *`idnumber`* `\n \0`

Update: Updates the identified object and returns a verification.

Query: `update \n` *`idnumber`* `\n` *`objectstring`* `\n \0`

Return: `updated \n` *`idnumber`* `\n \0`

In the event of an error, the server may return an exception:

Exceptions:

Return: `exception notfound \n \0`

Return: `exception permissiondenied \n \0`

Return: `exception generalfailure \n \0`

7.5 Mediator

The mediator is responsible for queries, inference, and other computations on the contents of the system. Requests for these computations are given in the Information Structure Scripting Language (ISSL), which the mediator must interpret.

7.5.1 Cacheing

The mediator keeps a model of the universe (e.g. all of the data on all of the servers) in memory and operates on this model. Of course, only a limited number of objects are actually cached at any given time; whenever an object is needed to answer a query, the mediator automatically retrieves the object from the appropriate server. Thus, the user of the mediator has the illusion of interacting with a single server which contains all of the data in the entire system, even though the data is in fact stored on many different servers.

Some objects are used for alphabetic, temporal, or other indexes, and so have special status: these objects are stored in a separate cache and are not pushed out by other objects. Since index searches occur frequently, it is beneficial to have as much as possible of the index cached in memory in advance.

It is possible to preemptively cache objects which are likely to be requested—that is, objects which are connected by edges to objects which actually have been requested (to some given depth). This could provide significant speed improvements for the user, at the expense of increased network activity and memory allocation. Preemptive cacheing has not yet been implemented.

7.5.2 Tight bind to server or client

Although the mediator can communicate with many servers over the network, it may happen, especially during the prototyping phase, that a particular mediator communicates primarily with a single server running on the same machine. In this case, it is possible to combine the two into a single program, bypassing the network step completely and thereby increasing speed.

In other circumstances, the user may benefit from using a client combined with a mediator.

The current prototype code supports both combinations, as well as the separated three-tiered model.

7.5.3 Alphabetic Index

The mediator supports an alphabetic index encoded in the information structure itself. Whenever an object is updated, the mediator automatically reindexes it based on the initial text contained in the object. The index is case-insensitive, and tags (e.g. anything in angle brackets) are ignored.

The index has a tree structure, where each branch has a relation type associated with a letter of the alphabet. Thus, to look up an object beginning with "Electricity", the mediator begins from the index tree root and follows a relation of type "IndextreeAlpha Relationtype E" to the object "e". From there, it follows another relation of type "IndextreeAlpha Relationtype L" to the object "el". The process continues until the mediator reaches the most granular level of the index (perhaps at "electri"), at which point it follows all of the "IndextreeAlpha Leaf" relations to find the desired object.

The root of the index tree, as well as the alphabetic relation types, have predefined ID numbers which are hard-coded in the mediator.

The number of leaves from a given node in the tree is limited to 32; an attempt to add a 33rd leaf causes the node to be decomposed—that is, the index tree is expanded by one level from that node, and all of its leaves are reindexed.

If there are more than 32 equivalent objects (e.g. containing the same text), then this decomposition process will recurse forever. This is a bug which needs to be fixed.

7.5.3.1 Full text search

The alphabetic index described above allows searching only on the initial text of each object. It must be expanded to allow full text search, but this has not yet been done.

One way of accomplishing this is to make a distinction between "IndextreeAlpha Begins-With Leaf" and "IndextreeAlpha Contains Leaf", where the latter relation type links each final index branch with every object that *contains* a word beginning with the given string. This approach obviously requires enormous storage overhead. Each word, with an average length on the order of six or eight bytes, requires at least one indexing relation, with an average length of perhaps sixty or eighty bytes.

Thus it is probably necessary to rely on standard text search methods, though an index tree of limited depth (e.g. two or three levels) might help significantly in selecting which objects to do a full search on, without producing overwhelming overhead.

7.5.4 Time Index

A similar index tree should be constructed for temporal information; the same mechanisms for indexing, decomposing, and retrieving will apply. The time index is not yet implemented.

7.5.5 Mediator Protocol

The protocol for communication between the client and the mediator is similar to that for communication between the mediator and the server. The mediator protocol supports additional commands for retrieving links and for executing ISSL scripts.

The syntax is as follows:

Echo: Returns the given string.
Query: `echo \n` *`string`* `\n \0`
Return: *`string`* `\n \0`

Close: Closes the network socket.
Query: `close \n \0`

ReadText: Returns the body of the requested object.
Query: `readText \n` *`objectid`* `\n \0`
Return: *`objectstring`* `\n \0`

ReadLinksAsHtml: Returns a list of the object's links, formatted in HTML.
Query: `readLinksAsHtml \n` *`objectid`* `\n \0`
Return: *`linksAsHtml`* `\n \0`

Create: Creates an empty object and returns the number.
Query: `create \n \0`
Return: `created` *`objectid`* `\n \0`

Delete: Deletes the identified object and returns a verification.
Query: `delete \n` *`objectid`* `\n \0`
Return: `deleted` *`objectid`* `\n \0`

Update: Updates the identified object and returns a verification.
Query: `update \n` *`objectid`* `\n` *`body`* `\n \0`
Return: `updated` *`objectid`* `\n \0`

RunISSL: Runs the script contained in the given object and returns the id of the result object.
Query: `runissl \n` *`objectid`* `\n \0`
Return: `ranissl` *`resultid`* `\n \0`

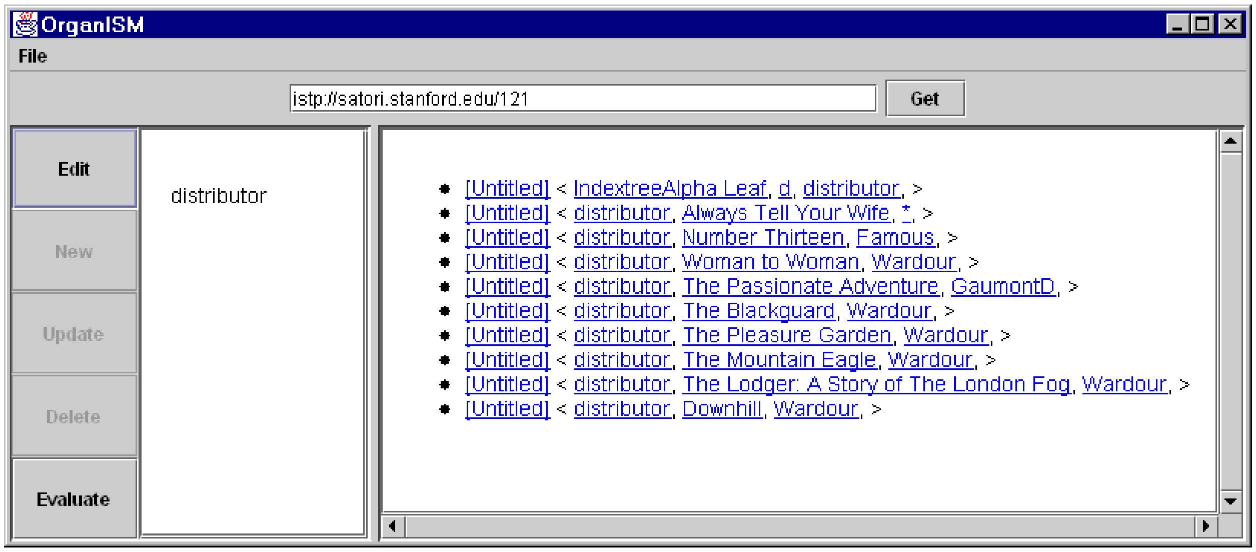
In the event of an error, the server may return an exception:

Exceptions:

Return: `exception notfound \n \0`
Return: `exception permissiondenied \n \0`
Return: `exception generalfailure \n \0`

7.6 Client / User Interface

The user interface is currently rather rudimentary. It allows the viewing, creation, updating, and deleting of text objects, as well as execution of scripts.



- [Untitled] < IndextreeAlpha Leaf, d, distributor, >
- [Untitled] < distributor, Always Tell Your Wife, *, >
- [Untitled] < distributor, Number Thirteen, Famous, >
- [Untitled] < distributor, Woman to Woman, Wardour, >
- [Untitled] < distributor, The Passionate Adventure, GaumontD, >
- [Untitled] < distributor, The Blackguard, Wardour, >
- [Untitled] < distributor, The Pleasure Garden, Wardour, >
- [Untitled] < distributor, The Mountain Eagle, Wardour, >
- [Untitled] < distributor, The Lodger: A Story of The London Fog, Wardour, >
- [Untitled] < distributor, Downhill, Wardour, >

Figure 3. A view of the “distributor” relation type, which has edges to relations encoding the distributors for various Hitchcock movies.

When viewing objects, the interface shows two windows: the body of the object appears in one window, and its relations appear in the other. The text of the relation object itself is shown (this is usually “[Untitled]”, since relation objects are usually empty), followed by the names of the related objects in angle brackets. Whenever an object has a “name-of” relation, its name is shown; otherwise the text of the object itself is shown. The user can click on any object shown in a relation to redraw the screen around that object; so the system can be browsed like the web.

If the user presses the “evaluate” button, the client asks the mediator to execute the contents of the object being viewed as an ISSL script, and displays the results.

The client currently runs as an application but not as an applet due to technical snafus; again, it is hoped that the maturing of Web browsers and their associated Java virtual machines will alleviate this problem in the near future.

There is not yet a good graphical interface for establishing relations between existing objects; the best way to do this currently is through ISSL. It is also possible to manually establish edges between objects by typing them in the edit window; but this allowed only temporarily for testing, since the object identifiers should eventually be invisible to users.

7.7 Information Structure Scripting Language (ISSL)

Note: at the time of this writing, the working version of the language is somewhat less capable and has a slightly different syntax than the version described here. Changes in the code to account for the improved language are in progress.

7.7.1 Grammar

The grammar for ISSL is as follows:

```
S -> expression ;\n

expression ->

expression -> variableAssignment

      variableAssignment -> variable = expression
      variableAssignment -> variable = new

expression -> expression expressionremainder

      expressionremainder ->
      expressionremainder -> + expression   (plus/or)
      expressionremainder -> - expression   (minus)
      expressionremainder -> # expression   (intersection)

expression -> .*
expression -> "expression"
expression -> (expression)
expression -> [expression]
expression -> {ntuple}

      ntuple -> expression , ntuple
      ntuple -> expression

expression -> evaluate(expression)
expression -> variable

variable -> $.*

pattern -> expression  (containing $_ or $1, $2, $3 etc.)

expression -> foreach(expression, pattern)
```

7.7.2 Description

The interpreter evaluates one line at a time. A line consists of an expression (which may have any number of subexpressions), and is terminated by a semicolon.

An expression always returns an unordered set of object IDs. Many expressions need return only one ID, but they return it in the form of a set of size one.

An expression may be a variable assignment. There is only one variable type: like an expression, a variable represents a set of object IDs. The return set of any expression can be assigned to a variable; the assignment operator itself also returns the assigned set. When the reserved word `new` is used in an assignment, a new, empty object is created, and its ID is returned as the single member of the ID set.

Since expressions return sets of IDs, they can be added, subtracted, or intersected.

If an expression consists purely of text, it returns the set of objects matching that text exactly. Thus `Fido` returns a set of all objects whose text is exactly Fido.

The quotation marks return the set of objects named by the objects in the argument set. That is, the interpreter starts from the objects in the set inside the quotation marks, follows all “name-of” relations forwards, and returns the resulting set. Thus `“Fido”` returns a list of all objects *named* Fido.

Parentheses allow grouping of expressions.

Square brackets encapsulate the contained set. That is, they represent the contained set in a single object, with relations of type “set-element” to each of the elements. The return value is a set with the new container object as the only member.

Curly brackets represent ntuples (e.g. relations). The elements of an ntuple are separated by commas. An ntuple is written in ISSL as a list of expressions—that is, a list of sets of object IDs. But in fact a relation consists of an ordered list of IDs, not of sets of IDs. Thus, when presented with an ntuple, The ISSL interpreter evaluates it for *every possible combination* of IDs in the given sets.

An expression containing no unknowns is an assertion. The interpreter does everything necessary to make the assertion hold in the knowledge base, including creating new objects.

```
{instance-of, dog, mammal};
```

causes the interpreter to locate objects matching `instance-of`, `dog`, and `mammal`, or to create them if they do not already exist, and to create a new object establishing the relation between the three. It then returns a set containing the ID of the new relation.

The `evaluate()` function causes the contents of the objects in the argument set to be interpreted as ISSL.

Variable names must begin with `$`.

A pattern is an expression containing the special variables `$_` and/or `$0`, `$1`, `$2`, and so on. In most contexts, a pattern acts as a query, and the special variables are considered unknowns. If the query is for a list of single objects matching some criteria, the pattern consists of an expression in which `$_` represents the unknown. Thus

```
{instance-of, $_, mammal};
```

returns a list of all mammals.

Some queries will be more complex, however, and will return a list of ntuples rather than a list of single objects. In this case the variables `$0`, `$1`, `$2`, and so on represent multiple unknowns, and specify their order in the resulting ntuples. So

```
{lives-in, $1, ocean} # {instance-of, $1, mammal} # {feeds-on, $1, $2};
```

returns a list of pairs specifying the feeding habits of marine mammals. Apparently there is some relation among the elements of the tuple, but the interpreter has no way of knowing (yet) what the relation is, since `$0` was not used in the pattern. Thus, it represents the results as relations of type "unknown", such as `{unknown, whales, krill}`, and returns a list of these relations. (It might be worthwhile to represent the pattern as an object in its own right before evaluating it; then the pattern could act as the relation type for the results).

The foreach() operator evaluates its second argument once for each element of the first argument. The second argument is a pattern, but it acts as a template for substitutions rather than as a query. If the second argument contains `$_`, the ID of the current element is substituted. If the second argument contains `$0`, `$1`, `$2`, and so on, the *n*th edge from the current element is substituted. Thus it is possible to provide a list of ntuples as the first argument, and to make some assertion about the elements of each ntuple in the second.

For example,

```
foreach(
      {parent-of, $2, $1} # {brother-of, $3, $2},
      {uncle-of, $3, $1}
      );
```

causes the interpreter first to construct a list of ntuples matching the appropriate parent-of and brother-of relations, such as `{unknown, John, Dennis, Stan}`, since the first argument contains a pattern query. Then, for each element in this list, the interpreter extracts edges 3 and 1 and establishes a new relation between the objects they point to. Finally, it returns a list of the newly created relations. The interpreter is responsible for avoiding redundancy: it does not create a new uncle-of relation where one already exists.

Note that this is not a search-time inference rule; it is a storage-time inference, which causes the resulting relations to be explicitly established. Search-time inference is not yet supported; when it is, it will require inference rule objects containing a convenient representation of `{"implies", pattern, pattern}`.

7.7.3 Examples

```
{instance-of, dog + cat, mammal + pet};
```

establishes four separate relations, one for each combination of elements from the argument lists.

```
$ShipSinkEvents =
      {event-part-of, $1, World War II} # {sunk-in-event, $2, $1}
      # {tonnage-of, $2, $3}
      - {instance-of, $2, submarine} + {event-day, $1, Thursday};

foreach($ShipSinkEvents, {unknown, $2, $3});
```

returns a list of all ships sunk in WWII, along with their tonnages—except submarines, unless they were sunk on Thursdays.

```
$this = new;
{instance-of, $this, movie};
{title, $this, The Pleasure Garden};
{name, $this, The Pleasure Garden};
{year-released, $this, 1925};
{director, $this, Hitchcock};
{producer, $this, Balcon};
{studio, $this, Gainsborough and Emelka};
{distributor, $this, Wardour};
{cinematographer, $this, Ventimiglia};
{based-on-book, $this, Oliver Sandys};
{screenwriter, $this, Eliot Stannard};
```

Establishes a new, empty object to represent a movie and makes a number of assertions about it.

7.8 Importing Data

The best way of importing existing data is to translate it into a set of assertions in ISSL; the resulting script can then be saved as an object and executed. Assuming that the data is a reasonably structured form to begin with, the translation is quite easy to do in Perl. This method was used to import existing data concerning movies, including information about directors, studios, actors, academy awards, and so on, as in the example above.

Conclusion

In this paper I have presented an abstract paradigm for storage, retrieval, and inference, and have shown how it can be applied to general purpose computing. The paradigm described is significantly different from existing systems, in its internal structure, in the level of its abstraction, and above all in its expressivity—that is, in the ability to represent and process any encodable human thought in an intuitive way.

There are clearly great technical difficulties involved in making such a system work. I have demonstrated, with running code, that the basic structure for storage and retrieval is feasible, but there are very many features which remain to be implemented. Further, the current prototype runs fairly slowly, and has been tested only for small data sets; so there is much work to be done to increase the speed of the system and to insure its scalability over many orders of magnitude.

I am confident that, when it is more fully implemented, the way of thinking about computing presented in this paper will prove to be very powerful. The development of systems based on these ideas will benefit users by providing intelligent and intuitive access to large knowledge bases that are both widely distributed and arbitrarily broad in conceptual scope.

References

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S. "The Object-Oriented Database System Manifesto." Proc. DOOD 89, Kyoto, Japan, December 1989.

Dittrich, K.R., Dayal, U., and Buchmann, A.P. (Eds.). *On Object-Oriented Database Systems*. Berlin: Springer Verlag (1991).

Genesereth, Michael. http://infomaster.stanford.edu (1997)

Kauffman, Stuart. *The Origins of Order: Self-Organization and Selection in Evolution*. New York: Oxford University Press (1993).

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

Loomis, Mary. *Object Databases: The Essentials*. Reading, Massachussetts: Addison-Wesley (1995).

Soergel, Dagobert. *Organizing Information: Principles of Data Base and Retrieval Systems*. San Diego: Academic Press (1985).

Soergel, Dagobert. "Information Structure Management: A Unified Framework for Indexing and Searching in Database, Expert, Information Retrieval, and Hypermedia Systems." in Fidel et al., ed. *Challenges in Indexing Electronic Text and Images*. Medford, NJ: Learned Information (1994).

Soergel, Dagobert. "SemWeb: Proposal for an Open, Multifunctional, Multilingual System for Integrated Access to Knowledge about Concepts and Terminology." in *Advances in Knowledge Organization v. 5*. 4th American ISKO Conference, Washington, D.C., July 1996.

Soergel, Dagobert. "Design of an Integrated Information Structure Interface: A Unified Framework for Indexing and Searching in Database, Expert, Information Retrieval, and Hypermedia Systems." Unpublished (1998).

Wiederhold, Gio. *Database Design*, 2nd edition. New York: McGraw-Hill (1983).

Wiederhold, Gio. *Movies Database*. http://www-db.stanford.edu/pub/movies/doc.html (1998)