BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M17

RelType

Relaxed Typing for Intelligent Hypermedia Representations

by
Dilip K. Barman

RelType

Relaxed Typing for Intelligent Hypermedia Representations

Dilip K. Barman
Brown University
Dept. of Computer Science
dkb@cs.brown.edu
April 5, 1991

Abstract

RelType is a flexible typing scheme, primarily for the links of a hypermedia system, that models hypertext as an object-oriented knowledge representation (KR) medium. The typing is non-intrusive in allowing but not enforcing hypertext link (and node) type specifications. Users continue to have complete linking flexibility, but may choose to use one of a small number of predefined types with associated semantics (e.g., subsumption, explanation) or to define their own types. The ensuing structure, akin to a semantic net, is type checked and specified semantic actions executed. The structure is semantically as rich as the user wishes and could, at least partially, support tractable inference. Tools also allow a RelType system to evolve semantically in a manner reflecting actual usage. Taxonomic and assertional components capture domain knowledge. An object-oriented view supports knowledge acquisition and eases extensibility, particularly through use of linked templates. RelType provides the flexibility of hypertext, combined with some of the representational power of term classification KR systems, in an efficient and usable manner.

Table of Contents

Table of Figures

Introduction

Hypertext is a flexible representational mechanism for storing knowledge about a domain, and typically the resulting structure is manually navigated by users browsing for information. Ideally, a knowledge-based system (KBS) that can reason from a hypertextual base would add inference to a potentially large corpus of information. However, the very flexibility of hypertext makes useful inference difficult. Some systems, such as NoteCards ([Jordan, 1989]), and KMS ([Akscyn, 1988]), suggest elements of semantic nets, but there is no semantic type checking. Resulting systems are neither well structured nor do they afford the user flexibility in writing documents and tying related concepts together in "webs" as they see fit. TEXTNET ([Trigg, 1983]) offers a system with full typing and semantic interpretation, but this system is specifically oriented toward online scholarship and would be difficult and even inappropriate to use in general. [Collier, 1987] describes a system, THOTH-II, which models semantics in hypertext, but really is for structuring information rather than supporting inference, and is at the LISP programming level, hardly a model a writer would adopt. CONCORDE ([Hofmann, 1990]) integrates some semantics with hypertext, but is meant for one application (knowledge acquisition).

The RelType implementation uses the *Hypertext Abstract Machine* (HAM[1]) as a transaction-based server to handle low level details about storage and access of nodes and links. Fine granularity linking is supported by means of offset specification. Because links are viewed as objects unto themselves and are stored independently of nodes (such as Intermedia webs; see, for example, [Yankelovich, 1988]), links are first class objects with their own set of attributes and can be maintained privately by individual users sharing a read-only set of nodes.

The HAM supports the association of attribute-value pairs with objects (*links*, *nodes*, and *contexts*). The RelType user is able to choose optional types that map to the underlying attribute-value system. The user who has instantiated a link (or node) is able to specify one of a small number of semantically defined types, such as *AKO* ("a kind of"; class-subclass subsumption), *ISA* (membership), or *SUPPORTS* (justification for argumentation). This concept of optional types in a hypertext system does not occur in the literature (but see [Hoffman, 1990] for something similar) or existing tools. Users have complete flexibility to continue without typing and the interface (given the appropriate front end) would be a natural hypertext one.

---

[1]see [Delisle, 1986] and page 17 below

Type information is stored in the HAM database as optional object attributes. A type checker ensures integrity so that A_1 *ISA* A_2 ... *ISA* A_k *ISA* A_1 is not allowed, for example; types are defined as a hierarchy maintained as a DAG[2]. The typing of links and nodes enables inferencing so that external knowledge based systems can at least partially use the information in hypertext as representations for reasoning in the spirit of [Clitherow, 1989]'s bridge to Cyc or [Hayes, 1989]'s combination of hypertext and inference for diagnosis. By combining partially typed hypertext and inference, a knowledge base (KB) can support inference over its typed portions and allow reasoning trails to serve as open-ended explanations by giving users the context in which to manually navigate for related information. Prompted query matching allows untyped objects to semantically evolve as they are used in inference paths. Full integration is sacrificed for the importance of keeping the user interface flexible.

A knowledge engineer could use RelType by encoding knowledge in hypertext, possibly including untyped links at leaves of inference for background, browsing reading. As part of the developed system, partially typed *linked templates* could be defined for extensibility so that macro-like chunks of hypertext corresponding to specific knowledge structures and/or documentation could easily be added by an end user[3]. Thus, if a user of such a KBS needs to add a new component that is AKO general component for which a template is defined, the template, with "smart" links so that it would self-integrate, would be instantiated. Frame-like functionality could be included with demons prompting the user for template specifics and watching for new slots to be filled. This is a contribution to knowledge-based systems from the points of view of extensibility, simplification/automation of a large part of maintenance, verification due to enforced system architecture, and knowledge acquisition.

In sum, the value of this work lies in exloring partial typing of links and nodes, inference combined with manual navigation in knowledge structures, and as an investigation for practical knowledge-based system representation tools. While some hypertext systems have typed links, no others have partially typed nodes and links, as well as fine granularity, separately stored links.

---

[2]Directed acyclic graph; such an organization is sometimes called a *tangled hierarchy*

[3][Jordan, 1989] describes structure accelerators for essentially the same purpose in NoteCards.

Rationale

Why Hypertext? Why KR?

RelType provides an information structuring tool and fills a niche that both existing hypertext and knowledge representation systems leave empty. Hypertext is relatively "easy" to use and allows a flexible capturing of information in whatever form seems most suitable to the user. This conceptual (and, in practice, system) interface is a natural one for "sketching" ideas and motivation to support design activities, whether for deliberating on how to build something (e.g., a software engineering project), analysis (e.g., conceiving and weighing alternative feasible solutions to some problem), information transfer (e.g., traditional teaching or "knowledge dumping" of critically skilled personnel in an organization prior to their leaving a position), or other activity involving representing pieces of information that are often ill-defined and may lack clear structure. This is in contrast, say, to updating an existing database with given fields and interpreting a new situation in terms of attributes corresponding to those fields.

Of course, there is a price to pay for such flexibility, and that price is that computationally not much can be done with the resulting structures. Adding content and perhaps structural search capability helps, but the information remains primarily passive.

Knowledge representation schemas address the problem of computationally empowering information from the opposite viewpoint, that of providing primitives (relationships and structures) and well-defined semantics for those primitives. The emphasis is on canonical forms or, as [Wilensky, 1986] describes it, uniformity as a principle of representation; systems such as KL-ONE, Krypton, and Kodiak attempt to map isomorphic concepts to identical representations. By representing their information through a KR formalism, users gain powerful inferencing capabilities and can discover hidden relationships and implications inherent in their data. Rule-based systems, for example, have illustrated the production rule formalism as a powerful paradigm for declaratively storing knowledge and drawing inferences thereof.

The problem with existing knowledge representation systems is flexibility and ease of use. Even some of the more successful commercial rule or frame based systems that include object-oriented principles and graphic interfaces, such as IntelliCorp's KEE or IBM's TIRS, do not provide for capturing information as somehow related to a problem and only later, if at all, letting the user formalize the connection.

This relates to issues three and seven in [Halasz, 1988][4], where "the problem of premature organization" (page 845) constrains a user to force fit his/her ideas into a rigid structure, at loggerheads with user exploration and experimentation. Related to the knowledge acquisition bottleneck for knowledge engineers, it is difficult to translate information and abstractions from a user's mind to a computer-mediated tool. The problem is a fundamental one, but it is compounded by having to structure the ideas in a highly constrained manner. This is a problem with both existing KR formalisms and those hypertext systems which attempt to include semantics in the nodes and links.

Clearly, both hypertext and KR models can contribute to a flexible tool for information design, such as RelType seeks to be. "At a high level of abstraction, hypermedia systems, frame-based systems, and object-based systems present nearly identical data models" [Halasz, 1988 page 847]; the differences lie in emphasis and supported goals (see also [Barman, 1990]). RelType combines the flexibility of hypertext with the semantic structures of KR formalisms and offers function in an object-oriented fashion to system customizers and users.

RelType Goals

An important goal of RelType is to allow a user to easily describe all components of a nascent design, providing as much or as little semantics as desired to afford maximum convenient expressivity in the information space. Both links and nodes are presented to the user as objects with associated operations. Typed link and node support provide semantic constructs for building a subsumptive knowledge base of objects, identifying objects to be executable and specifying how they can dynamically be made, and depicting argumentation used for design rationale. Also, users can specify their own types to suit their specific needs.

The result of using the tool is a hypermedia network with traditional untyped links and nodes, in addition to well-specified objects acting according to defined

[4]See Figure 14 on page 41

semantics. A query interface (along with tools to make querying easier for end users), allows intelligent retrieval from the hypermedia representation. The integration of executable, inferencable, and documentation[5] components provides a unifying representation for continued development or as a target for the posing of intelligent queries. Inference and query operations are performed in time polynomial in KB size.

Review of Relevant Representational Issues

Semantic Networks

A semantic network is a directed graph used to represent knowledge where nodes are concepts, states, events, or attributes. Adjacent nodes are related by the label on the connecting link which could be of varying semantic content depending on the underlying system. Ross Quillian [Quillian, 1966], who wanted to represent the meaning of English and define a model of how English words are represented in human memory, is often cited as the originator of the semantic network, but semantic nets were used since the late 1950s in machine translation efforts[6].

The knowledge representational view is of semantic nets as a form of graphically depicted logic with only unary (*types*) and binary (*attributes*) predicates. Types are organized into a taxonomy having links decorated with attributes such as *ISA* and *AKO*, as well as domain defined attributes. ISA represents instantiation and AKO subtype; for example, *Clyde ISA Elephant AKO Animal AKO Living-Being*. Because a unary predicate is represented by a node containing a "type", nodes may either be types or constants. AKO behavior can be simulated by overloading the meaning of ISA; *const ISA type* represents the normal ISA instantiation, while *type1 ISA type2* is the AKO taxonomic relation.

Besides implicit subsumptive classification (see the section *Subsumption* on page 8), relational semantics may be specified by link attributes. Path-based inference

[5]including design rationale

[6][Masterman, 1961] is in fact the first to use the term *semantic net*.

allows a link to be inferred between two nodes based on the existence of a path between the two nodes. Thus, if we have two assertions, *Sima is a person* and *A person is a living thing*, then we can infer *Sima is a living thing* by using the rule that if an object is an instance of a class C, then that object is an instance of all superclasses of C[7] (see Figure 1). In general, deriving inferences involves including a theorem prover in the net formalism.

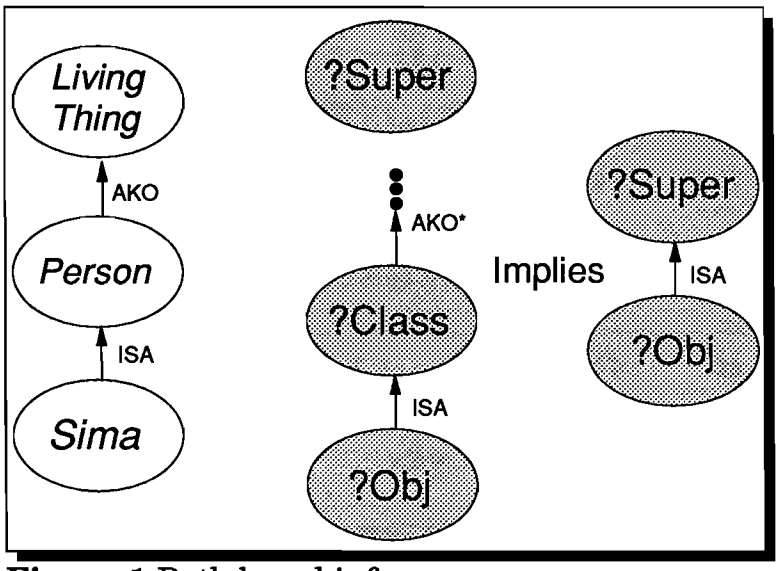

Figure 1 Path-based inference

[Woods, 1975] questions the plethora of systems calling themselves *semantic networks* and points out limitations, such as the difficulty of using attribute-value pairs to represent statements like *John's height is greater than 6 feet* or *John's height is greater than Sue's*. Woods suggests creating an *EGO* link to establish the intensional identity and rationale for creating any given node. Also, he distinguishes assertional links that assert facts and/or relations from structural links that serve a definitional purpose; below we will see this evolve into ABox vs. TBox reasoning components.

Taxonomic vs. Assertional Systems

Mankind has historically used taxonomies to classify and understand the world. The Greek philosopher Porphyry, for example, built a classification tree in the third century based on Aristotle's syllogisms. Figure 2 shows how this organizes

[7]This is an example of a subsumptive inference.

types into subtypes and shows what differential criteria separate the subtypes so that concepts are defined in terms of distinguishing characteristics from their supertype.

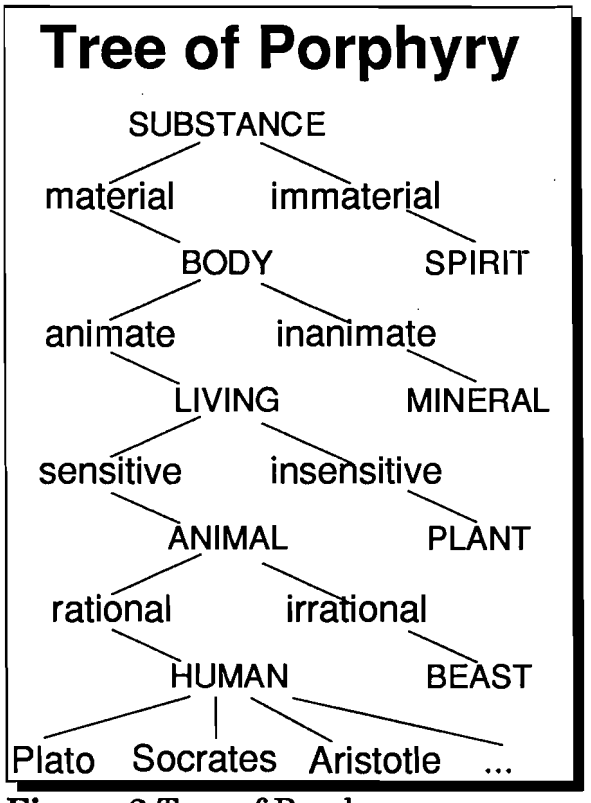

Figure 2 Tree of Porphyry

A taxonomy orders objects, such as concepts, events, states, or Aristotle's ten categories, usually by generality (*type* or *subsumption*[8] *hierarchy*) but possibly by some other principle or relationship such as part-whole (*meronomy*). Because some objects may be incomparable, a taxonomy presents a partial ordering. As in object-oriented languages, a most general type, often called **T**, is provided in a subsumption hierarchy. Also, attributes (e.g., frame slots) can be inherited.

The intent in taxonomy is to classify concepts rather than state constraints or facts of a particular domain. More recently, particularly in this century, methods of formal logic have provided notation and expression for symbolically asserting propositions and manipulating truth values. Alfred Tarski's model theory provides for extensional semantics via an interpretation function that maps terms to their

[8]Concept C_1 subsumes concept C_2 if any instance of C_2 satisfies the definition of C_1 too. In other words, the extension of C_1 is a superset of the extension of C_2.

denotations[9] and Alonzo Church's lambda calculus provides for functional definitions to allow both extensional and intensional descriptions to be stated.

Term classification (TC) (also called *classification-based*) knowledge representations like KL-ONE ([Brachman, 1985a]), begun in the late 1970s, and its descendants (such as NIKL[10], KANDOR, BACK, K-Rep, KL-TWO, LOOM, MESON, and CLASSIC) use concept classifiers to organize concepts (*terms*) into a subsumptive taxonomy[11]. A distinction is made between such classification, or terminological component (*TBox*), and assertional knowledge (the *ABox* component) that provides constraints and facts about a domain. All knowledge is seen as being in either TBox, to describe classes of individuals, or in ABox, to constrain the concept semantics. (For a clear and excellent perspective of TC representations, see [Mac Gregor, 1991].)

Subsumption

[Woods, 1991] feels that the term *subsumption* is sloppily used and has at least five different meanings. *Extensional subsumption* is deductively computed so that it logically follows that the extension of a concept is a superset of the extension of a concept it subsumes. *Structural subsumption* relies on an algorithm (presumably more efficient than deduction) to determine subsumption relations. *Recorded* and *axiomatic subsumption*s are similar; in the first, a static concept hierarchy is given and subsumption is read directly or by computing a transitive closure, and in the second the subsumption relations are axiomatically stated. Finally, *deduced subsumption* is similar to structural subsumption but follows from deduction of the axioms of the domain knowledge.

Many papers have been published showing intractability in worst case for TC systems. [Schmidt-Schauss, 1989] proves that it is undecidable in KL-ONE if one concept is subsumed by another by reducing the word problem in groups (undecidable) to a modest sub-language of KL-ONE. [Nebel, 1988] shows that complete terminological inference is intractable for BACK and KANDOR, a system that was thought to be tractable, and in fact for any system with a nontrivial concept language with subsumption shown to be co-NP hard by transforming the set splitting problem, known to be NP-complete. [Nebel, 1990] demonstrates that subsumption with

---

[9]The denotation of a constant is an individual object, and of a formula is a truth value (T or F).

[10]"New Implementation of KL-ONE"

[11]A concept classifier computes subsumption relations between pairs of concepts. In so doing, it can insert new concepts into an existing taxonomy by placing a new concept with its subsumptive links pointing to the most specific subsuming concepts and being pointed to by the subsumptive links of each most general subsumee.

KL-ONE like concept semantics is co-NP complete. [Patel-Schneider, 1989] shows undecidability of subsumption in NIKL by reduction to the Post correspondence problem for a binary alphabet.

These results should not be surprising for concept languages with power close to that of FOPC[12]. The general case of computing subsumption between two concepts where the concept language has the power of FOPC is computationally as hard as the problem of determining if two FOPC expressions are equivalent, which is undecidable. [Levesque, 1985] was the first of many to suggest that the full expressive power of FOPC leads to intractability so some suitable subset, tailored to the domain of representation, should be chosen to achieve an appropriate tradeoff of expressivity vis-a-vis computability.

[Woods, 1991] and [Mac Gregor, 1991] provide an interesting counterpoint. [Woods, 1991] suggests that research into such results "seems now to have nearly run its course, reaching the conclusion that almost everything of interest is computationally intractable in the worst-case" and that in fact there is no abundance of expressive power that can be traded away for efficiency. Woods feels that most of the work in subsumption is computing transitive closure on recorded taxonomies, so worst case analyses should not be overly stressed and used to argue for limiting expressivity. [Mac Gregor, 1991] downplays these results as well, since at runtime computing concept types is emphasized, not classification. In the tractability analysis below (page 30), we will see that these arguments certainly apply to RelType since RelType in fact uses static recorded taxonomies.

Review of Typing in Existing Typed Hypertext Systems

The classic treatment of types in programming languages is due to P. Wegner and L. Cardelli [Cardelli, 1986] and is set in an OOP context in [Wegner, 1990, pages 33-36]. As [Wegner, 1990, page 34] puns, "the term *type* is heavily overloaded" and is perhaps not a good choice to use in RelType as it shares little with the concept of a programming language type. [Sowa, 1991b] describes philosophical uses of the word and mentions related terms such as *sort*, *isa, set, class, kind*, and *flavor*. Term classification knowledge representation systems use the word "type" in a manner closer to that meant here, as the set of concepts that an individual belongs to. In any case, the notion of "type" in RelType is in consonance with its use in the hypertext

[12]First order predicate calculus

literature to denote constraints or classification for objects in a hypertext system, most commonly nodes and links.

Clearly, from the computational point of view of being able to interpret a user's hypertext structures, a fully typed system is ideal. Stipulating that all objects be typed would be less than ideal, however, for users, especially in exploratory mode. Full typing would also be of questionable value, as users might be inclined to rely on defaults and thereby side-step semantic issues entirely. Here we review existing type systems in hypertext systems that are typed. As this will bear out, existing type systems are either simplistic labeling schemes to identify targets of search, or are overly restrictive and hamper user flexibility.

gIBIS Types

IBIS (Issue Based Information Systems) is a method, developed in 1970 by Horst Rittel, that aims to structure the search for a design as "fundamentally a conversation among the stakeholders ... in which they bring their respective expertise and viewpoints to the resolution of design issues. Any problem, concern, or question can be an issue" [Conklin, 1988, page 304]. Key *issue*s are identified and *position*s can be expressed which resolve a given issue. Each position may have a set of *argument*s, some supportive and others objecting to it. IBIS works to clarify what's at stake in a design by representing issues, positions, and arguments (and "other" for related material to consider) as node types with typed links in an IBIS network. This can be viewed as a rhetorical model with a set of typed messages (Issue, Position, Argument, and Other nodes) and a set of moves (i.e., valid links) among the message types.

gIBIS, or graphical IBIS, is an MCC implementation of the IBIS method that is being developed as a tool to be used by designers to capture their design rationale. It includes nine link types connecting the typed nodes (abbreviated I, P, A, and O). Followup work on a more general architecture called *Germ* (graphical entity relationship modeling tool) includes thirteen link types in an enhanced gIBIS.

[Fischer, 1989] describes how R. McCall extended IBIS in 1987 by introducing the Procedural Hierarchy of Issues (PHI) which generalizes an issue to include any design question, whether deliberated or not, and changes relationships to *serves* relationships where issue 1 serves issue 2 if answering 2 depends on answering 1 first[13]. This provides further structure for decomposing a problem domain of issues

[13]Subsequent PHI work is described in the [ECHT, 1990] paper "PHIDIAS: Integrating CAD Graphics into Dynamic Hypertext", Raymond J. McCall et.al., pages 152-165.

into subissues. "Author's Argumentation Assistant (AAA): A Hypertext-Based Authoring Tool for Argumentative Texts" by Wolfgang Schuler and John B. Smith in [ECHT, 1990, pages 137-151], describes University of North Carolina work on how people write. AAA is a system supporting PHI argumentation, and here they use the same node types (but *other* is called *fact*; for consistency, we'll stick to *other*), but have nine different link types. Figure 3 lists both (Germ-based) gIBIS and AAA PHI link types.

13 gIBIS Link Types

Generalizes (I→I)	
Specializes (I→I)	
Replaces (I→I)	
Challenges (I→I, I→A, I→P)	
Suggested-By (I→I, I→A, I→P)	
Responds-To (P→I)	these links may evolve to one of the next 2
Resolves (P→I)	a position that is selected as a solution
Is-rejected-from (P→I)	for dropped positions
Supports (A→P)	
Objects-To (A→P)	
Expands-on (I→P)	
Objects-to (A→A)	
Other (O→O)	

9 PHI Link Types

Serve (Ia,Ib)	issue b serves issue a
Replacement (Ia,Ib)	issue b is a replacement of issue a
Suggestion (X, Ia)	issue a is a suggestion/question by X (any node type)
Answer (Ia,Pb)	position b is an answer to issue a
Objection (Pa, Ab)	argument b is an objection to postion A
Support (Pa, Ab)	argument b is a support of postion A
Contributes (PAa,PAb)	pos./arg. b contributes to pos./arg. a
Reference (X,Fy)	fact y is a reference of X (any node type)
Contradicts (X,Y)	X contradicts Y (X,Y any node types)

Figure 3 gIBIS and PHI link types

Neptune CASE Types

[Delisle, 1986] suggests using attribute-value pairs for developing and maintaining code in a CASE environment. It is not clear if they have actually used the following, but in any case they offer these suggestions for structuring the nodes and links in such an environment. No formal semantics is provided, nor is there a type system per se, but the information is expressed through a system of attribute value pairs. The underlying HAM architecture is general, so appropriate applications code could interface with these attributes and values to implement a type or constraint system.

Each node, it is suggested, could have an attribute *Document* with value taken from {*requirements, design, sourceCode, objectCode*} to identify the role a node plays within the software engineering lifecycle. If the language being used for development were Pascal, the code could be structured as a tree with a node for each procedure or function[14]. Modula-2, like Borland's Turbo Pascal, has the ability of providing separate modules and providing a visible calling interface that is separate from an implementation module. As such, the Neptune authors suggest representing Modula-2 project code as a directed graph to show where modules are imported. Nodes might have an attribute *contentType* with values {*text, graphics, Modula-2 source code, Modula-2 object code, Modula-2 symbol table*}. To illustrate functional decomposition, an attribute *codeType* might be defined with values {*definitionModule, implementationModule, procedure*}. Links are suggested to specify relationships between components in the project, perhaps with a link *relation* from {*isPartOf, annotates, references, compilesInto*}.

Textnet Types

The primary consideration of Randy Trigg's Textnet system is to provide an architecture for scientific literature to move online so that all activities from draft and review through publication in electronic journals and reader commentary can be accomodated. All links are typed and the set of types is static - there is no provision for adding new types. The types are oriented toward activities associated with creating and critiquing scientific papers, and the author feels that he has been exhaustive in providing enough primitive types that there should not be a need to add more types. User-defined types would run counter to the standardization that would

[14]But this doesn't handle mutually recursive calls, so a directed graph (possibly with cycles), like a call graph, seems to be more in order.

be required to make this a general and widely used tool, and would result in both reader and semantic confusion.

While links are directional, an interesting distinction is made between physical and semantic directionality. Physical direction represents the progress the author expects readers to make, while semantic direction is dependent on the link type. There is, for example, a link type *Comment* that implies C (source object) comments on T (target); usually, the physical direction is T to C because typically the item being commented on is read before the commentary.

Two classes of types are defined - *normal* and *commentary*. Normal link types are used in development of ideas to connect nodes to nodes, while commentary types are used to connect descriptions of a node to the node itself. There are 29 normal link types and 51 commentary link types. We enumerate his link types below.

In addition to type, other attributes of links described are *author, date, fromobj* (the source object), *toobj* (target), and two tags, *prerequisite* and *must-follow*. If a link goes from object A to object B and is tagged *prerequisite*, A must be read before B because it is required to understand B; if tagged *must-follow*, then a reader of A must at some point also visit B by traversing the link.

The use of types in Textnet is not of much direct relevance to RelType; the goals are different. Textnet's taxonomy of link types is particularly oriented toward producing and critiquing scientific literature, while I am interested in more general use. As such, his taxonomy is both too rigid (cannot be extended) and too specialized. Nevertheless, some of his link types might be worth implementing. It seems that his system, requiring full type specification, would be difficult to use. His notions of semantic vs. physical directionality may be useful to consider in follow-on work to RelType.

Figure 4 enumerates TextNet link types, directly from [Trigg, 1983, page 37]. They are divided into normal and commentary types. Headers for further classifying commentary types appear in *italics*, but do not themselves represent types.

Normal Link Types

Citation	Generalization/Specification	Summarization/Detail
C-source	Abstraction/Example	Alternate-view
C-pioneer	Formalization/Application	Rewrite
C-credit		
C-leads	Argument	Simplification/Complication
C-epon	A-deduction	Explanation
	A-induction	
Background	A-analogy	Correction
Future	A-intuition	Update
Refutation	Solution	Continuation
Support		
Methodology		
Data		

Commentary Link Types

Comment	*Points*	*Data*
Critical	Pt-comment	D-comment
Supportive	Pt-trivial	D-inadequate
	Pt-unimportant	D-dubious
Environment	Pt-irrelevant	D-ignored
E-comment	Pt-redherring	D-irrelevant
E-misrepresent	Pt-contradict	D-inapplicable
E-vacuum	Pt-dubious	D-misinterpreted
E-ignored	Pt-counter	
E-Isupersede	Pt-inelegant	*Style*
E-Irefute	Pt-simplistic	S-comment
E-Isupport	Pt-arbitrary	S-boring
E-Irepeat	Pt-unmotivated	S-unimaginative
		S-incoherent
Problem Posing	*Arguments*	S-arrogant
P-comment	A-comment	S-rambling
P-trivial	A-invalid	S-awkward
P-unimportant	A-insuff	
P-impossible	A-immaterial	
P-ill-posed	A-mislead	
P-solved	A-alternate	
P-ambitious	A-strawman	

Figure 4 TextNet link types

For details on the semantics of the types, see [Trigg, 1983], chapter 4 (*A Taxonomy of Link Types*). For a flavor of the semantics, a few examples follow. *C-pioneer* is a citation to a pioneer in a field. *Environment* types describe how scientific papers exist in an environment of related work; for example, *E-vacuum* indicates a neglect of understanding past work or somehow relating the paper to the field, and *E-ignored* points to some relevant work that was ignored. *P-solved* is a critique that this is a solved problem, and *A-immaterial* states that an argument, while valid, doesn't lead to a desired conclusion.

NoteCards Types

The basic constituents of NoteCards, from Xerox PARC, are *Notecards*, which contain an arbitrary amount of text, graphics, or other editable data, and links. Links are typed and directional, connecting two cards. Granularity of linking is card at the destination and offset within card at the source. The typing, however, is merely a user-chosen label and no provision is made for type inference or other computation of any kind.

THOTH-II Types

In THOTH-II, a key element being modeled is the semantic relationship that exists between nodes of text. An explicit semantics is provided by specifying a value link (as opposed to a text link to connect a node to the actual text content or a lexical link to connect a region of text to a node) between two nodes. However, the typing of links is merely a labeling and is so flexible as to have no underlying type system at all. The labels are merely targets for searches. As such, the "explict semantics" is ad hoc and really a way for a user to flesh out his ideas and tag relationships, hopefully in a consistent manner, for later search.

CONCORDE Types

CONCORDE is perhaps the closest existing system to RelType. It is a hypertext system built using Smalltalk-80 to structure information specifically to serve as an aid for all stages of knowledge acquisition. [Hofmann, 1990] mentions that systems exist with typed links or nodes and refer to THOTH-II, gIBIS, and NoteCards. While semantic nets can thereby in principle be supported, they remark that knowledge-based applications of hypertext are in fact rare.

CONCORDE is built to fill the need for an integrated tool for knowledge engineers to use to store knowledge sources, help in structuring such knowledge, highlight errors and places more information needs to be gathered, provide a "knowledge snapshot" for knowledge engineers and experts to both appreciate what the system currently "knows", and for knowledge base maintenance. The basic information-carrying unit is the *card*, and cards are connected via links. As in RelType's theory of relaxed typing, CONCORDE differentiates two kinds of links - *constrained* (typed) and *individual* (untyped). Inherent constraints are expressed by defining valid source and target card classes for a constrained link; predicates (e.g.,

causes, *same_substance*) are also provided for defining explicit constraints. The knowledge engineer can define such predicates as appropriate for the domain being modeled. Although they don't describe this very well, the authors mention "exercises" that are presumably used in verification/validation of an expert system and which they show result in what are called *linked templates* in RelType, or chunks of nodes and links that represent atomic units in the application representation. They hope to add semantic checking in the future.

The ideas in CONCORDE bear a resemblance to those presented here, such as the support of both typed and untyped links, and an application in aiding knowledge engineers. However, the RelType goal is to develop a system that is a knowledge repository that is used after an appropriate KA methodology, perhaps one using CONCORDE, gathers relevant information. RelType seems to be more useful for knowledge base maintenance (instantiating linked templates for user-directed knowledge base growth), possible actual representation (content-rich nodes semantically interpretable and inferencable), and explanation (semantic understanding of link and/or node types and inferencing to a close match to a query to provide a contextual explanation). Perhaps the CONCORDE approach could be combined with RelType to encompass a general hypertext-based knowledge engineering environment to include all phases from initial knowledge acquisition through delivery and enhancement.

Unlike other systems reviewed here, CONCORDE has some real substance to the semantics of link types, at least for constrained links. A constraint system is defined which validates which links can be added by including predicates for checking for conditions. The orientation is for structuring causally related knowledge and for inferring additional links or negating possible links. Again, the utility is best seen in the knowledge acquisition phase of a knowledge-based project, when raw observations and tentative rules are being tested.

The constraint system is not well documented, but from what is provided, it is not oriented toward defining general semantic actions. In particular, conditions as functions of existing objects (nodes, links) can be tested in some restriction of first-order logic, but the resulting actions are not described in the paper except for two examples of deleting or inserting a link. The implication is that actions are meant to modify linkage behavior among existing objects. The semantics of RelType, on the other hand, allow more general actions, such as calling user defined code as procedural attachments to link traversal, and automatic smart linking.

RelType System Architecture

HAM

The Hypertext Abstract Machine is an abstraction that acts as a server to provide callers with generic hypertext functionality for storing, maintaining, and accessing nodes and links[15]. The HAM is transaction-oriented, responding to requests from distributed clients on the LAN, and has recovery, versioning, and synchronization control. It provides both command line and C language calling interfaces. RelType runs with the α-HAM version as its base. The basic objects in the HAM are *nodes*, *links*, and *contexts*. All of the defined objects form a *graph* or *database*.

Contexts are collections of nodes, serving to provide a potentially hierarchic partitioning of objects. Links may connect nodes within same or different contexts; linking is provided at the granularity of offsets into nodes. For any given graph, RelType uses three contexts, one named *Knowledge Base* for the user to develop essentially an A-Box of RelType hypertext structures, another named *Value Taxonomy* for representation of a subsumptive classification hierarchy of values, and a third named *Defined Types* where RelType pre-defined types are represented and which can be extended with user-defined types.

Links and nodes can have arbitrary numbers of attribute-value pairs associated with them. Versioning is provided so that, for example, a link can be made to always dynamically point to the "current" version or be cemented to a specific node version.

At least in principle, demons are supported so that user code can be automatically called upon a specific HAM event, but work on passing environmental parameters to the code was described as in progress. α-HAM, in particular, does not explicitly have demons. Limited demon support is provided in RelType as a semantic primitive (i.e., user-defined types cause specified user code to be called upon traversing the object the type describes).

[15]The HAM is well described in [Delisle, 1986], including an appendix specifying the operations.

Node and Link Types

While the primary object to be typed is the link (since the link establishes a relationship that can be constrained), nodes, representing actual substance, are also typable. Some kinds of relationships only make sense between nodes having certain attributes; a link indicating a dynamic makeable object that is executable should originate from a piece of code, for example. A typical link type will implicitly define the context for interpretation of nodes at either end of the link, and a typical node type will constrain the operations that can be performed or types of inferences that can be drawn on it. Also, typing nodes gives users a way to imply the context and content of a node so that they can find "interesting" nodes and relationships through querying.

There are 2^3=8 possible combinations of partially typed structures for any pair of directly connected nodes. If we represent the triple <x,y,z> by xyz where each variable comes from {U,T} to respectively mean untyped or typed, x and z are nodes, and y a directed link from x to z, then the possible structures are given by {UUU, TTT, UTT, UTU, TUU, TUT; TTU, UUT}.

UUU represents "business as usual" - no types at all; presumably many node pairs will continue to exhibit this structure. The other extreme, TTT, would be useful for linking two nodes of specified semantic content and executing an action or propagating a constraint between the two; an example would be marking the source to be a piece of code in a given language, the target an executable program, and the link the program relationship *compiles Into*.

UTT (and TTU by symmetry) and UTU represent the remaining possibilities of typed links. UTT would be handy to go from a known node content, perhaps an argument, position, or program component, via a *Background* link to a node representing the beginning of a subgraph constituting an area a user can manually navigate to get background information on the target node. UTU could be used in subsumption so that if the source node is AKO (a kind of) the target, we can inherit structural attributes from the target.

Finally, TUU (and UUT) and TUT use no types on links at all. Both are best envisioned as being part of a composite linked template that could intelligently be structurally searched. A knowledge engineer might make a system he/she builds extensible by defining node types T_1 and T_2 and defining a template connecting nodes of types T_1 and T_2 as part of a structure. For example, a structure developed to support an information base on animals in a zoo might include a template T that could be instantiated upon introduction of a new animal. Part of the template might include a node of type *AnimalName*, say, and it would always be connected to a

"scratchpad" for miscellaneous notes that don't fit under any other provided structures or for zoo personnel to initially make notations that only later will be fitted to the rest of the linked template. Arguably, though, this category of node and link types provides little advantage and could be done without. In the zoo example, the application might be better served by labeling the link with a type simply as a label for search, such as *Raw Notes*.

Type Representation

All types, built-in and user-defined, are represented in a subsumptive taxonomy in context *Defined Types*, as shown in Figure 5. Like the attribute-value taxonomy, the types are arranged as a tangled hierarchy (DAG). The most general object is *RelType Type* and it has AKO links from two subclasses, *Link Type* and *Node Type*.

Subclasses of link types are *Transitive Type*, *Reflexive Type*, *Symmetric Type*, *Antisymmetric Type*, and *Nocycle Type*. They correspond to transitivity, reflexivity, symmetricity, antisymmetricity, and acyclicity relations, respectively. For example, the type *ISA* is an instance of *Transitive Type* (A ISA B, B ISA C $\Rightarrow$ A ISA C), *Antisymmetric Type* (A ISA B $\Rightarrow \neg$(B ISA A)), and *Nocycle Type* (cannot have A_1 ISA A_2 ... ISA A_k ISA A_1). Note that a type that can take one of a range of values as an argument interprets noncyclicity in terms of a cycle of exact type plus value; thus if type T can be x or y, then A (T=x) B (T=x) C (T=x) A is a cycle, but A (T=x) B (T=x) C (T=y) A is not.

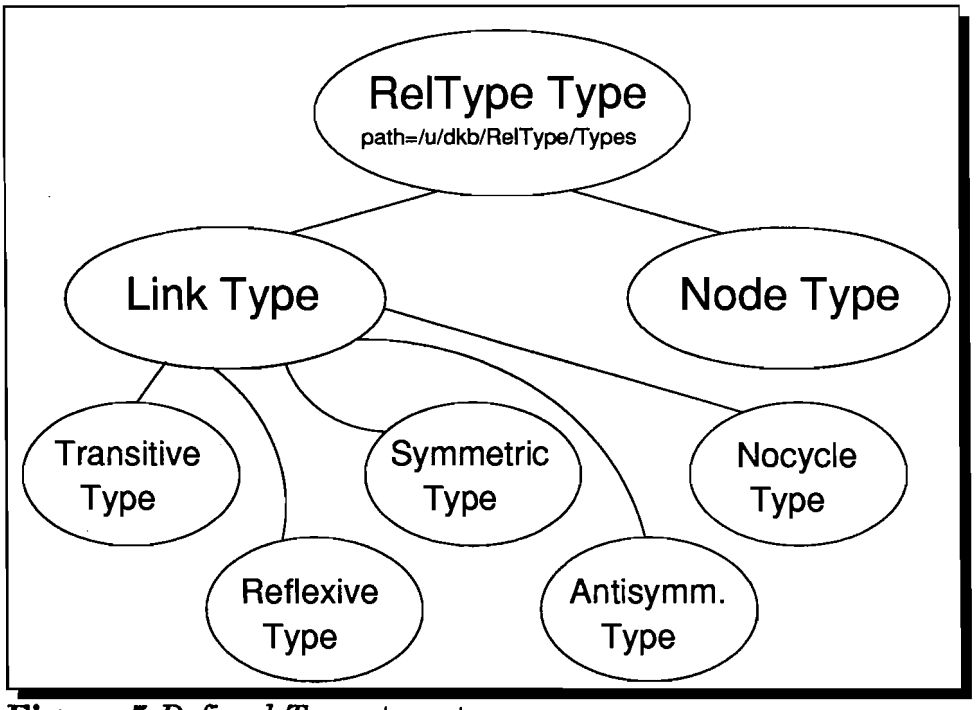

Figure 5 *Defined Types* type taxonomy

A link type with none of these relations can be instantiated directly from *Link Type*. All node types are instances of *Node Type*.

Each such node in the taxonomy of types supports attributes that define the type semantics. Attribute *code* is the name of an executable procedure that is to be called whenever an object of that type is traversed. The root object, *RelType Type* has an attribute *path* which indicates a path of directories to be searched for executables; this is inherited by all type objects and can be overriden.

Attribute *range* specifies a list of valid values the type can take on (e.g., T=x above); if *range* is not specified, it must be a type like AKO which doesn't take arguments and if *range=any*, any value at all (including none) may be specified. If attribute *abstract* is present with value *yes* or *true*, that type is comparable to an OOP abstract class and is not allowed to itself be used as a type, but rather serves the purpose of defining subsumees which can be types[16]. Attribute *abstract* is not inherited, because in practice typically only some structurally defined nodes need be abstract.

[16]The basic nodes structuring the taxonomy, all those depicted in Figure 5, are abstract, but are not so annotated here so as not to clutter up the figure.

Link types have attributes *source constraint* and *target constraint* which are query language constraints on what kinds of nodes an instance of this typed link may connect. Of course, the taxonomy is itself a RelType structure[17] and can include arbitrary hypertext links for documentation.

Supported Types

Choosing any one set of types for user support is always going to be a point of debate. Of course, giving users the ability to define their own types helps to overcome any semantic restrictions. The *Defined Types* taxonomy is merely primed with these supported types but new types may be added in the same manner and these existing types even subtyped or used as templates. Furthermore, the issue is largely sidestepped because RelType is not a prototype to support typing in a specialized domain such as argumentation or scientific paper writing/critique, where the specific supported types can be an important issue. Instead, the aim is to show how a well-defined but relaxed type scheme can help in developing general knowledge repositories.

Among the supported types are those necessary for implementing subsumption (*ISA, AKO*) to allow information inheritance. To avoid confusion, these types perhaps should be *"InstanceOf"* (*ISA*) and *"SubType"* (*AKO*), but we will stick to the traditional names for their brevity. Because subsumption calculation is computationally expensive, arbitrary use of ISA and AKO is not currently supported. Rather, as mentioned above, a separate context with the reserved name *Value Taxonomy* acts as a T-Box component to define a global terminological component that a knowledge engineer can build (or the user can build/extend). This acts as a definitional subsumptive domain model that can be used for inference over the assertional hypertext. *ISA* and *AKO* are the only semantically meaningful link types in the Taxonomy contexts, but for documentation purposes, arbitrary hypertext links can be made. End users may annotate links in their knowledge base with *ISA* or *AKO*, but in the knowledge base (assertional component) they only play a role of targets for searches or items that user-defined types can operate on in an arbitrary manner.

One of the ultimate goals is to enable a hypermedia network to act as a representational medium which includes both executable knowledge fragments and concomitant documentation and design rationale. Explicit explanation is supported by link types *Explanation* and *Background*. The use of these links ranges from manual link following (the capturing of design rationale remains as a benefit) to

[17]other than the fact that there can be no type semantics here other than AKO and ISA to avoid recursive definition!

providing intelligent answers to queries. For example, a linked template with a descriptive name like "*Why ... ?*" might follow an explanation link and display the node contents. Appropriate type conventions may be used so that explanation links from "close"[18] nodes may be automatically followed. Eventually, explanation hopefully will be more robust and automatic, mirroring execution paths and presenting more complex inference-driven explanations.

To encourage users to include executable units in their hypermedia systems, RelType includes link and node types to support program development. Nodes can have a type *programComponent* (chosen from {C source, C++ source, PASCAL source, object code, linkable library object code, executable, documentation, design}), and links *programRelation* (from {subModule, definitions for, annotates, compiles Into}) to show how program components, including documentation, relate.

Finally, a limited and simplified subset of IBIS/PHI argumentation is provided, primarily to provide the ability to show design rationale in the development of a RelType information base. Six of the thirteen gIBIS types and two of the nine PHI link types, as well as *Issue*, *Position*, and *Argument* node types, are supported. IBIS types allow arguments to be described with respect to other arguments (*ObjectsTo*), arguments to support or disconfirm positions (*ObjectsTo*, *Supports*), and positions to correspond to, and subsequently resolve or reject, issues (*RespondsTo*, *Resolves*, *Rejects*). The PHI types introduce the serve notion of the resolving of one issue depending on first resolving another issue (*MustResolveFirst*; PHI calls this *Serve*) and the general link type *Contradicts*.

Thus, four node types and thirteen link types are defined. They are listed in Figure 6.

[18]based on type semantics

Node Types

For executables

programComponent ∈ {C source, C++ source, PASCAL source, object code, linkable library object code, executable, documentation, design}; content of node is that component

Argumentation

Issue

Argument

Position

Link Types

Subsumption

AKO specialization

ISA instance

Explicit explanation

Explanation

Background

For executables

programRelation ∈ {subModule, definitions for, annotates, compiles Into}

Argumentation types

RespondsTo (P→I) these links may evolve to one of the next two

Resolves (P→I) a position that is selected as a solution

Rejects (P→I) for dropped positions

Supports (A→P)

ObjectsTo (A→P)

ObjectsTo (A→A)

MustResolveFirst ($I_2 \rightarrow I_1$) $issue_1$ serves $issue_2$ and must be solved first

Contradicts (X→Y) X contradicts Y (X,Y any node types)

Figure 6 RelType's predefined types

User-Defined Types

RelType supports user-defined types by allowing constraints and the specification of user-provided C++ code in the *Defined Types* taxonomy. When a link is traversed, if the link is untyped nothing will happen outside of the traversal. Otherwise, if the link's type is predefined, the semantic action associated with the type will be performed. If the link is typed but not with one of the predefined types,

and if the type is specified[19], the associated actions will be performed. A predefined calling interface gives the user code environmental data such as pointers to the link and source and destination nodes, and the name of the graph environment so that it can directly modify any additional objects it may need. Finally, a non-defined type will not result in any additional action; this might be a way for a user to define a keyword type for searches, in the manner of many of the above reviewed systems which claim to have typed links, and perhaps later specify semantic actions.

Types can cause the execution of actions upon traversal in manual navigation, such as the updating of a database, keeping of statistics, or display of customized help text. For many types, there might be no action at all associated with traversal during browsing, but the semantics may come to play upon query- or program-directed navigation. For example, subsumptive link types would be used in answering a query; if we know that Clyde is an elephant and that elephants have tusks, RelType would be able to find that Clyde has tusks when asked, or to list having tusks as one of Clyde's characteristics.

Subsumption in RelType

RelType supports two tangled hierarchy subsumptive taxonomies, one (whose context is) called *Defined Types* and the other *Value Taxonomy*. They are maintained in contexts separate from each other and the knowledge base. While arbitrary hypertext links and nodes may be present in these taxonomies[20], the only components considered part of the type hierarchy are nodes directly connected to ISA or AKO typed links. Both taxonomies are static and provide no facilities for dynamic classification and introduction of new concepts.

The *Defined Types* taxonomy, as already described, has most general type being *RelType Type*. There is some semantics in the representation, with node attributes specifying pieces of code to be called and inheritance of relationships such as transitivity. Inheritance of attributes is supported.

The attribute-value taxonomy has a most general value, called *Thing*. The nodes in the hierarchy are considered to be named according to the value of their attribute *Name*. Unlike the *Defined Types* hierarchy, there is no real semantics to the concept nodes; the attribute-value taxonomy is meant to be used by a knowledge engineer to model the objects of the problem domain and their relationships. The

[19] i.e., it appears in the taxonomy

[20] for example, to serve as documentation

taxonomy serves as a kind of data dictionary and is used to allow basic subsumption on names. Inheritance does not have any obvious benefit and is not implemented. It would be easy to add inheritance of attributes but this would only make sense if the nodes were true concepts and not just name holders.

Query Interface

An important building block for specifying semantics or for dynamic command-line inquiry is the query interface. Kindly refer to the computer-mediated instruction scenario on page 35 for a worked out example of what a typical query might look like and what it might allow a user to accomplish.

Basically, the query interface is a way of specifying a set of constraints which are applied to the attribute-value pairs of objects defined in the current hypertext. A query consists of a conjunction of selectors which can match objects in the hypertext. Objects matching the last selector are returned as the result of the query.

The BNF syntax for specifying a query appears in Figure 7. *CAPITAL ITALIC* letters are used for non-terminals, the start symbol is *QUERY*, $\in$ is the empty string, and "id" is a terminal identifier which can be any string of alphanumeric characters starting with an alphabetic character that is not L, N, or C (i.e., [ABD-KMO-Za-z][A-Za-z0-9]*).

```
QUERY       → SEL & QUERY
            → SEL
SEL         → SELID SELSTUFF
            → SELSTUFF
SELID       → id:                    {name selected objects; "id ':'" one token}
SELSTUFF    → WHICHOBJECT SELECTOR
WHICHOBJECT → L | N | C | id         {links, nodes, contexts, or ids}
SELECTOR    → SELECTORS | ∈
SELECTORS   → SELECTORS | CONJ
            → CONJ
CONJ        → CONJ • PAR
            → PAR
PAR         → ( SELECTORS )
            → CRITERIA
CRITERIA    → att = val
            → att is val             {if val is in TBox, att can be a subsumee of val}
            → ne = id                {current object mustn't be id}
            {Node criteria follow}
            → numIncomingLinks RELOP number
            → numOutgoingLinks RELOP number
            → numLinks RELOP number
            → linkedTo = id          {direct conn.; id should be of type N}
            → incomingLink = id      {id should be of type L}
            → outgoingLink = id      {"}
            → link = id              {"}
            {Link criteria follow}
            → source = id            {id should be of type N}
            → target = id            {"}
            → oneEnd = id            {"}
            → oneEnd = id otherEnd = id {both ids N}

RELOP       → = | > | < | >= | <= | <>   {really tokens}


Implementation note: for top-down parsing, need to eliminate left recursion
and replace SELECTORS and CONJ:
 SELECTORS → CONJ DISJ
 DISJ      → | CONJ DISJ | ∈
 CONJ      → PAR CONJS
 CONJS     → • PAR CONJS | ∈
Also, for predictive parsing, need to left factor, replacing QUERY:
 QUERY     → SEL Q
 Q         → & QUERY | ∈
```

Figure 7 Query language BNF syntax

Note that link criteria can be specified as conjoined (“.” operator) or disjoined (“|”), with conjunction taking precedence. Parentheses can be used as well.

For example,

x:N.id=3.color is Brown.numlinks>2.(a=b | c=d) &
y:N.linkedTo=*x*.id=5 &
z:N.linkedTo=*y*.ne=*x* &
lnk:L.source=*y*.target=*z*.id=1 &
N.(incomingLink=*lnk*.att=3.outgoingLink=*lnk*) |
(link=*lnk*.attx=abc)

will assign *x* to the set of nodes with id=3 and color=Brown and more than 2 links and either a=b or c=d. If "Brown" is in our attribute-value taxonomy, then we will also match color to any subsumee of Brown[21]. *y* will be assigned to the set of nodes linked to *x* with id=5. *z* will be set to all nodes, excluding *x*, that are linked to *y* -- i.e., nodes indirectly linked to *x* through *y*. *lnk* will be the set of links whose source is a node in *y* and target is a node in *z* and whose id is 1. The set of nodes with a link among *lnk* that have attx=abc OR those with att=3 and incoming and outgoing links from *lnk* will be returned. In the diagram in Figure 8, the node with attribute *name* (=a) will be returned (and no other objects).

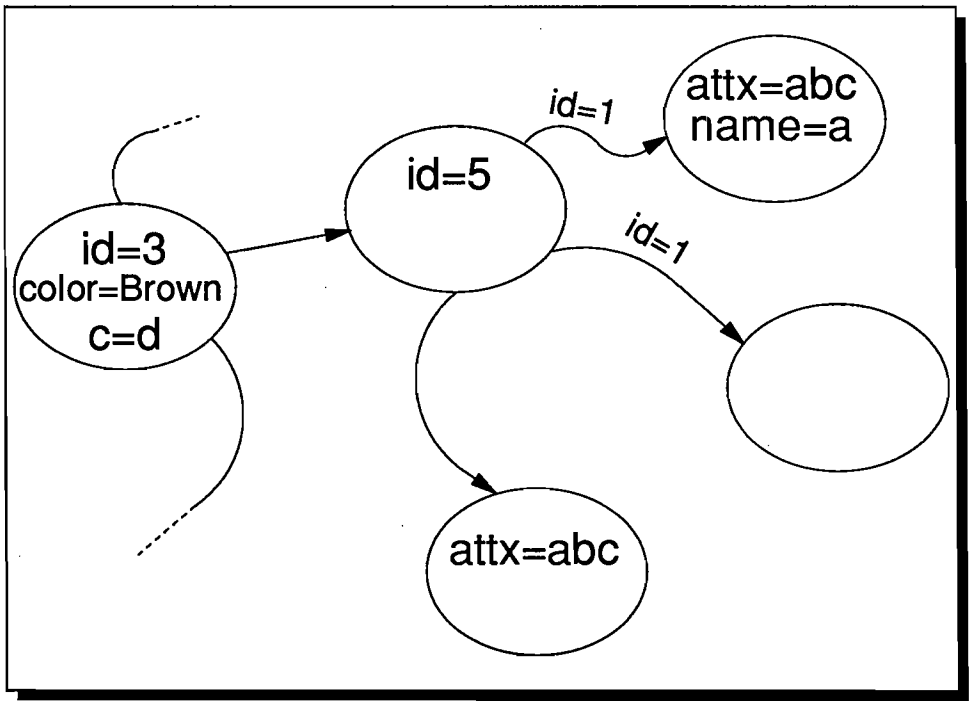

Figure 8 Query example

Formulating queries, admittedly, is "not pretty" for the end user. Undoubtably, a suitable interface could be built to make posing queries more natural language like

[21]So, for example, if *Sepia AKO Brown AKO Dark-Color*, then color=Sepia would match, as well as color=Brown

or graphical. However, even with an "easy to use" interface, query posing could become tedious and perhaps beyond the scope of many users. In the section below, a method is described that hides query details and provides the end user with a node (that constitutes a *linked template*) that can form the origin of a dynamic query.

Linked Templates

To allow a RelType knowledge base to be extended in a principled manner, *linked template*s are provided. These are graphs that themselves are just like hypertext fragments with nodes and (possibly typed) links, but also there are *smart links* that are anchored at only one end. The unanchored ends of smart links are represented by a query-language specification that a candidate node in the hypertext must satisfy in order to be an attachment point for an instantiated template[22]. A system builder could define an architecture for extensibility by providing linked templates for new information that would enforce consistency and, more importantly, preserve semantic meaningfulness of the entire hypertext, when instantiated by a user.

Linked templates provide a nice way to hide details of the query language from end users with understood node or subgraph search needs. A knowledge engineer or database designer could analyze the query needs of the users and create types with the appropriate query language statements being part of the type definition. Any parameters for the query could be prompted for. A toolkit of "smart nodes" could be provided, each of them corresponding to one of these query types and being single node linked templates. Upon activation, an instance of such a node would serve as an anchor for links emanating to all nodes links emanating to all nodes matching the query[23].

Prompted Matching

The whole point of relaxed typing is to allow users to easily use RelType while specifying as much or as little semantic detail as they wish. However, a neglected type can cause an inference to fail that should have been successful. For example,

---

[22]The specification may be null, indicating that the template may be attached anywhere.

[23]No analogue exists for link queries, since links can only point to nodes and not to other links.

x:N.area is ConnecticutTown.type=Argument.topic=*t*.
t is EnvironmentalConcern &
y:N.linkedTo=x.type=Position.topic=*t* &
L.source=x.target=y.type=Supports

will set *x* to the set of nodes with attribute area equal to ConnecticutTown (or anything subsumed by ConnecticutTown), type=Argument, and some topic *t* that is an EnvironmentalConcern or subtype. *y* will be the set of nodes that are linked to *x* with type=Position and the same topic as *x*. The set of links from *x* to *y* will be returned that have type=Supports.

Assume that *Bethel ISA ConnecticutTown* and *Recycle AKO Environmental-Concern*. The structure in Figure 9 almost succeeds, but the user failed to type the link. RelType can prompt the user by displaying the link and asking if it should have type=Supports; if so, the link will be so typed (permanently) and the query will be successful.

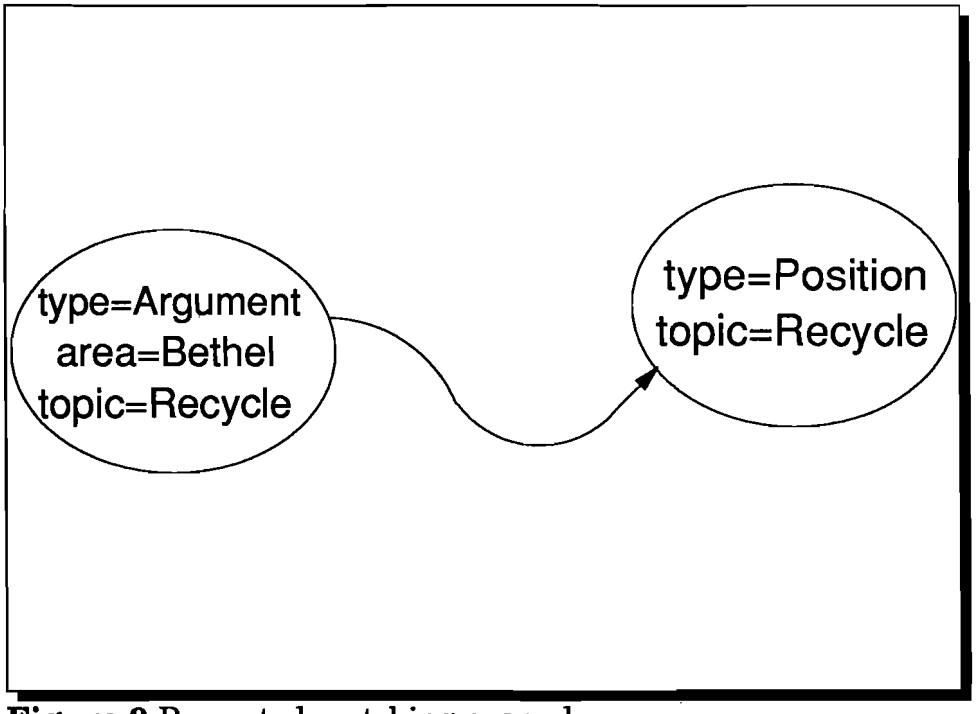

Figure 9 Prompted matching example

There is a tradeoff between making matching very precise on one hand and very flexible by considering the knowledge base objects totally malleable on the other hand. It would be computationally intractable, for example, to relax all constraints in trying to find matches. RelType strikes a reasonable compromise by using this prompted matching mechanism only for objects Ω such that Ω doesn't have a type at

all that the constraint is searching for (including type=null) and Ω is related to an object that has matched its constraints.

If the link above had type=ObjectsTo (or type=*t* for any $t \neq$ Supports, including null), it would not be considered or prompted for. The second condition of being related (if a node, connected to a matching link or linked to a matching structure of nodes; if a link, connected to a matching structure of nodes) will keep RelType, in our example, from prompt matching all nodes that don't have type= specified. In other words, the first constraint must be exactly matched.

Note that the semantics of each line must stand alone in defining relatedness to already matched objects. Without partial matching, the following query is equivalent to the one above.

x:N.area is ConnecticutTown.type=Argument.topic=*t*.
t is EnvironmentalConcern &
y:N.type=Position.topic=*t* &
L.source=*x*.target=*y*.type=Supports

But since part of the *y* constraint does not specify that it is connected to *x*, it will not be match prompted. This is another illustration that end user query needs should be understood by the system architect and provided for via linked templates, rather than having end users craft their own queries.

Thus, we provide a definition of "almost" match as an object associated with an otherwise matching sub-structure that could match if an unspecified type were given some range of values. To avoid re-querying of objects in a later constraint, if a user specifies that there is no value for a given type, the object is assigned a value for that type of null.

This approach helps users to specify a minimum of semantics and have the system ask intelligent questions to annotate the knowledge base for inferential paths that arise in practice. Thus, a partially typed representation can evolve to include stronger semantics.

Tractability

In a TBox with the expressive power of first order predicate calculus, computing a subsumption relationship between two concepts is computationally equivalent to determining if two FOPC expressions are equivalent, an undecidable problem. RelType, by "cheating" on this issue, avoids the computational problem,

since it provides static axiomatic recorded subsumption hierarchies and includes very simple expressivity of ISA and AKO links only. An object can be tested for taxonomy membership in constant time and precomputed transitive closure subsuming and subsumee concepts can be returned.

Because RelType does not support dynamic object addition to its TBox components, it does not have a classifier. In the attribute-value taxonomy, classification simply would not make sense since there are no inherent semantics that could be used to make decisions as to where to add a new object in the taxonomy. It might be worth considering classification in the type taxonomy, however, to allow dynamic addition of user created types. Computing subsumption and taxonomy membership would then have to be done for each value access in the knowledge base. However, the concept nodes still have no semantics, so the cost would be that of computing transitive closure, which can be done in time $O(n^3)$, where n is the size of the taxonomy.

Navigation by query in RelType is tractable, being linear in the knowledge base size when the knowledge base size is reasonably large. In particular, a query consists of N_l constraint lines, each of which has some number of constraints, the largest of which is N_{Cmax}. If N_o is the number of objects (links and nodes) in the knowledge base, then simple constraint matching can take time up to $N_l N_{Cmax} N_o$. If we include attribute-value and (static) type subsumption, we can identify all subsumees in constant time, and can potentially check all of the subsumees against the constraint. If the size of the value taxonomy is N_v and that of the type hierarchy is N_t, matching a query can take time up to $N_l N_{Cmax} N_o N_v N_t$. The KB size, N_o, is the dominant term for all but tiny systems where the system would in any case be quite efficient. For significant N_t with dynamic user created types, the cost is still modest, being of order $N_o N_t^3$.

Without RelType constraints, prompted matching would be exponential in the knowledge base size with all possible combinations of objects meeting the topological requirements of a given query being considered. However, RelType limits prompting to objects already being otherwise considered and that "almost" meet specifications, and elminates vacuous reprompting by nullifying types. Thus, prompted matching introduces a constant factor times the number of KB objects of additional work.

So even in the worst case RelType's computations are tractable. Efficiency can be improved over the analysis presented here by appropriate database organization and retrieval so that time $O(\log N_o)$ may be realizable to match and retrieve objects against a query. In any case, computations are decidable in time polynomial in knowledge base size.

Scenarios of How RelType Can Be Useful

Maintaining a Knowledge Base

RelType allows a new approach to knowledge engineering, providing a unified representation for both executable rules and immediately connected and browseable supporting documentation. In principle, rules could be supported and the entire network thereby set to play the dual role of execution and explanation, but at least in this version, only procedural and some object-oriented programming coding support are provided. Explicit support for a frame-language or an environment for rules could be built atop the provided language tools or, better, an integrated rule language could be built-in in the future. It would be expected that link and node types for supporting executables would have to be extended, in any case.

Nevertheless, even in this version, some degree of integration is indeed supported. The subsumptive link types, for example, allow attribute inheritance. Use of untyped links encourages any kind of user documentation, while use of provided documentation types makes it easy to interface an application layer to automatically relate documentation to a given node.

A knowledge engineer could provide linked templates for user extensibility of a knowledge base[24]. The builder of the system could provide users with templates of hypertext with "smart" typed links that could only fit into an existing knowledge base in "correct" places and perhaps associated demons to keep databases dynamically updated. *Class linked templates* could introduce another level of type checking, by ensuring that a template could fit only into any instance of a given node N[25]. N could be viewed, in an OOP fashion, as a class, and any constraints in that class (as well as those of the class instance to which the template is anchored) could be inherited to the new addition.

---

[24]assuming the hypertext structure, at least potentially, is considered to be inferencable plus browseable, the name *knowledge base* (KB) is used

[25]This introduces an additional layer, so that N becomes the grandfather of the node linked to in the template.

Even without explicit rule-based support, RelType provides frame-like representation with typed links and demons, and includes untyped as well as typed links for contextual browsing. Templates with smart links allow the knowledge engineer to provide macro-like building blocks to enable end users to evolve their systems. This has payback as a knowledge acquisition and explanation tool, and helps by leveraging the knowledge engineer's design so that users can take on a large degree of the maintenance task.

A scenario that illustrates the value of this work with respect to KB maintenance is provided on the accompanying figures which show a segment from a knowledge base on office supplies. This is suggestive of a generic KB, and could be envisioned as a maintenance or diagnostic manual for some domain of operations. The mixed typed/untyped network allows a taxonomy of office supplies to be built, while flexibly providing for users to link, via user-defined types and simple hypertext links, to information that they feel is relevant.

Replacement is an example of a user-defined link type that is simply a value constraint; an instantiation of this template would not allow any but the given values to be specified. Another applications layer could be built to perhaps graphically depict possible choices and ask the user to select one. *Cost* is another user-defined type; it has associated semantics to ensure the cost is in a specific range and to invoke code for database updating. In Figure 11, the AKO link is a "smart link" and ensures that the template can only link to a node called *Supply*. Demons[26] handle type checking values associated with the template, and may even provide a helpful interface for specifying the values. Demon0 is associated with the instantiation of the template, and causes the updating of a database.

[26]User code defined in the type specification and called upon object instantiation

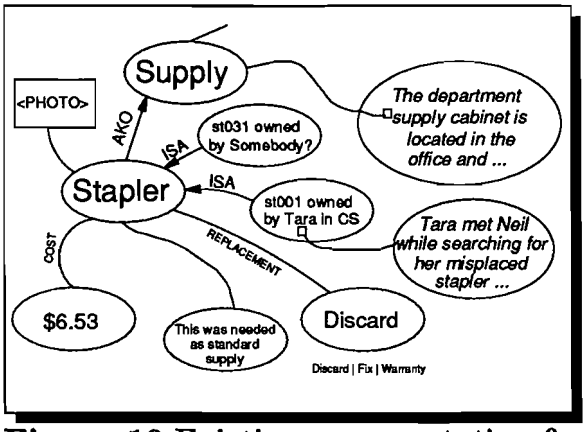

Figure 10 Existing representation for an office supply KB

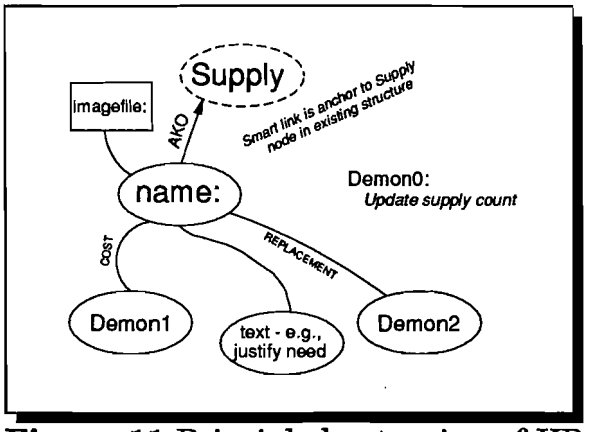

Figure 11 Principled extension of KB through linked templates

Software Engineering

Another good application has already been seen in Neptune, and that is for helping to manage the complexity of multi-file, multi-module, software projects with all their concomitant executables and non-executables. RelType can provide some of the basic functionality of Neptune, and has the added strength of its relaxed typing.

Nodes can be marked with attributes indicating the role of their contents as source code, documentation, or header files that can be linked into an application. Relations supporting modular decomposition such as subModule will be handy, and explicit programming semantics for constructing modules are specifiable through customizing relations such as *compiles Into* with system-specific details. All life cycle constituents, including design rationale, can thus be integrated by using RelType. Design rationale could very well be represented by argumentation subnetworks; this could be a major advantage for builders and extenders of large systems, so that design tradeoffs could be made explicit and not have to be rediscovered in maintenance or system extensions.

An example scenario might be of a user developing applications code. Given a "slick" interface, s/he might even choose to use RelType in the initial system design phase to built entity-attribute-relationship diagrams, using types just as labels and the tool as a graphic aid. This could then be put into its own HAM context and the context given an attribute *contentType* with value set to *design*.

The real use comes when s/he uses RelType to specify the application at lower levels of design and finally down to implementation. Design would ideally be specified with some sort of argumentation model so that design rationale and alterna-

tives could be captured. Tying components together becomes a graphical way for showing dependencies and would provide enough information to the system that it could rebuild the application by checking version times and applying semantic rules, for example for converting a C++ source module into object code and linking it to library code. Follow-on work could actually attack the system build problem from the opposite end to allow conventions to suggest defaults[27], and essentially allow a predefined make file to implicitly annotate the links and nodes. Functionality could also be added to provide intensional links to a data dictionary or from program components to their documentation.

The power of the RelType approach is that artifacts of all phases of the software engineering lifecycle could be easily stored together. RelType could become a very general software engineering tool and could provide an environment as open or restrictive as the user wants. It could interface to design tools and integrate all the data associated with a project in using a given design methodology, for example. On the other hand, a developer could simply use the implicit make facility (or even not and explicitly create his/her own executables) by following a given set of conventions and simply annotate the architecture, design, and code with arbitrary, non-typed hypertext links.

Computer-Mediated Instruction

In the past twenty years, quite a variety of educational software has been developed with a range of claims of benefit to instructor and student. While much of the resulting software has been less successful than advocates of computers in the classroom had hoped for, many educators still feel that computers can be useful. Hypertext systems have met with some success in teaching college-level classes; see, for example, [Landow, 1988] for a description of the use of Intermedia in teaching literature and literary criticism.

RelType provides a flexible environment for application builders to use as a platform for delivering educational software. Teachers could provide a hypertext network to students apropos to a topic being studied, and define a set of conventions for typing links. Students could work with their own copies of the network or with a common copy at disjoint time intervals; version control through appropriate use of contexts could also be used. After the exercise, the collective work of the students could be evaluated and simple search on the agreed upon types could be used to find who found which relationships. A "fat link" view could be provided that would

[27] e.g., if content of a node points to a file of type ".CC", it is a C++ source file

coalesce all targets of the same link types[28]. Thus, while concurrency control is not provided, "groupware" or class collaboration for joint learning is nevertheless encouraged.

For example, a high school class on the civil rights movement in the U.S. might start off with a hypertext network with a small set of teacher-defined link types such as {*source of prejudice*, *perceive prejudice*} and node types {*government action*, *authority-supported action*, *ignored-by-authority action*, *minority spokesman*, *minority community*, *non-violent protest*, *violent protest*, *community standard*}. Students would be given reading assignments or asked to do library research and to add information they gleaned, using this set of types, as well as including untyped material. By simple search, all sources of prejudice, for example, that the students document could be seen. Figure 12 depicts RelType structures that might result from such an assignment. Discussion could resolve any conflicts and help the class integrate their research into a common model.

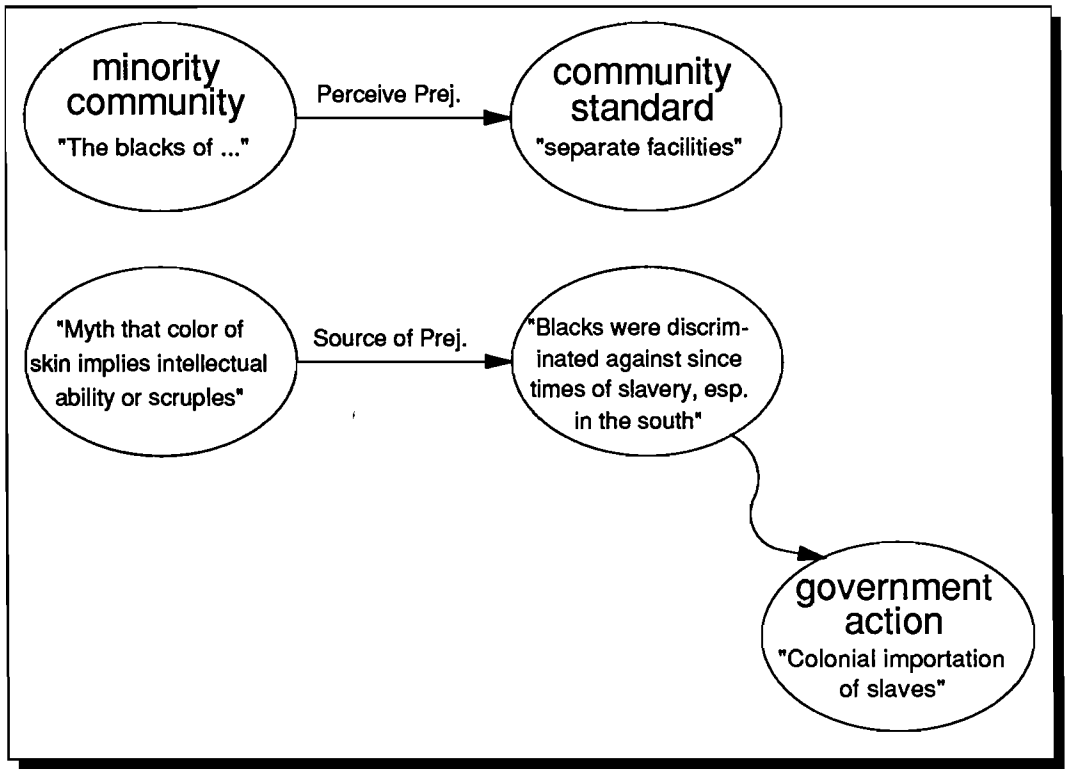

Figure 12 Initial ideas for typing in a high school civil rights scenario

[28][Shapiro, 1991] suggests exactly this paradigm of a fat link, which is called a *cable*, in the SNePS semantic network; a cable is a set of like-labeled arcs coming out of a node and ending at different nodes.

Linked templates could also be used. Perhaps after a preliminary excursion as described above, once the students and teacher all had a better problem domain understanding, the teacher could fashion a template for filling in further details. Continuing the development of the civil rights example, perhaps an important element would be confrontations challenging existing traditions.

Before asking students to particularly research confrontations, perhaps a template as in Figure 13 would be designed, so that students would need at least this much information to document a confrontation. Prior to detailing confrontations (or, better, iteratively), the class might take a number of positions that they feel might resolve the problems associated with achieving equal rights for all Americans. The open boxes for position nodes indicate intensional attachment points for smart links. Thus, all documented confrontations would be required to be related to the theoretical positions and to show one position the action supports and one it is at odds with. The students' research would not be in a vacuum and would be tied to evolving class theories. Students may find such an investigative and collaborative approach to make their study very exciting and would show that problems are multi-faceted and may have no obviously superior solutions.

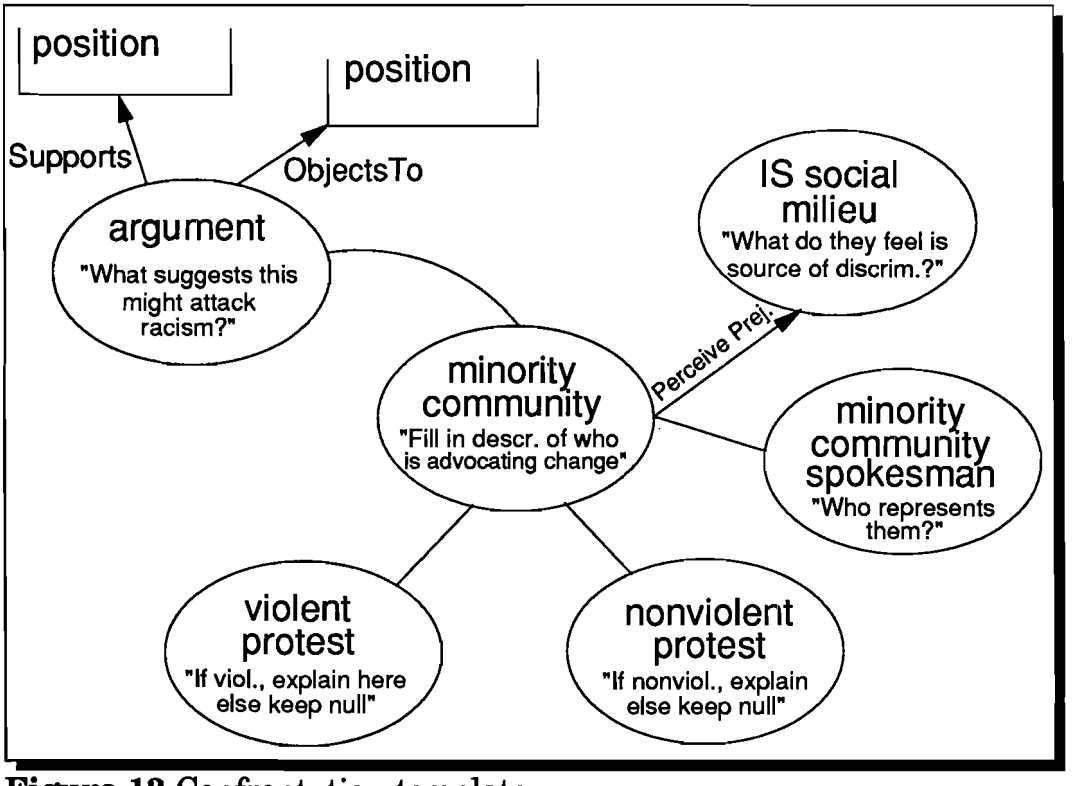

Figure 13 Confrontation template

[Woods, 1991]
Woods, William A. "Understanding Subsumption and Taxonomy: A Framework for Progress". Chapter 1 of [Sowa, 1991a].

[Woods, 1985]
Woods, William A. "What's in a Link: Foundations for Semantic Networks". In [Brachman, 1985b], pages 217-241. Originally in Bobrow, D.G. and A.M. Collins, eds. *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press, 1975, pages 35-82.

Other References

[Cardelli, 1986]
Cardelli, Luca and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". *Computing Surveys*, vol. 17, number 4 (December 1985), pages 471-522.

[Wegner, 1990]
Wegner, Peter. "Concepts and Paradigms of Object-Oriented Programming". *OOPS Messenger* (ACM SIGPLAN quarterly), vol. 1, number 1 (Aug. 1990).