Expressing Structural Hypertext Queries in GraphLog

Mariano P. Consens
consens@db.toronto.edu

Alberto O. Mendelzon
mendel@db.toronto.edu

ABSTRACT

`GraphLog` is a visual query language in which queries are formulated by drawing graph patterns. The hyperdocument graph is searched for all occurrences of these patterns. The language is powerful enough to allow the specification and manipulation of arbitrary subsets of the network and supports the computation of aggregate functions on subgraphs of the hyperdocument. It can support dynamically defined structures as well as inference capabilities, going beyond current static and passive hypertext systems.

The expressive power of the language is a fundamental issue: too little power limits the applications of the language, while too much makes efficient implementation difficult and probably affects ease of use. The complexity and expressive power of `GraphLog` can be characterized precisely by using notions from deductive database theory and descriptive complexity. In this paper, from a practical point of view, we present examples of `GraphLog` queries applied to several different hypertext systems, providing evidence for the expressive power of the language, as well as for the convenience and naturalness of its graphical representation. We also describe an ongoing implementation of the language.

INTRODUCTION

Hypertext systems are intended to support the organization and manipulation of networks of text nodes (or multimedia nodes, for hypermedia) connected by typed links. As current systems start getting more use, several limitations in the basic approach are becoming apparent. In a recent survey [?], seven key issues are identified as requiring work for the next generation of hypertext systems. The first five of these are:

1. **Search and query facilities.** Current systems are heavily oriented towards browsing and network navigation. They lack powerful query languages that allow the specification and manipulation of arbitrary subsets of the network.
2. **Augmenting the basic node and link model.** The directed graph model is too low level to support complex ways of organizing the information in a network.
3. **Virtual Structures.** Hypertext systems support only manual changes to the contents or structure of a network, making them relatively static in practice. It would be desirable to have

---

The first author was partially supported by the PEDECIBA, United Nations Programme for the Development of Basic Sciences, Uruguay.
Published in *Proceedings of the Second ACM Hypertext Conference*, 1989, pp. 269–292.

dynamically defined structures that can make the network reconfigure itself automatically in response to changes.

4. **Computation over graphs.** Current systems are passive; for example, they do not include inference engines that may actively derive new information from what is explicitly stored.
5. **Versioning.** When hypertext technology is applied to the maintenance of large technical documents, or to computer assisted engineering, it is essential to have mechanisms for managing versions and configurations and to control concurrent access reliably.

In this paper we describe a powerful query language for hypertext, called `GraphLog`, that addresses all points 1 to 5 above. Point 3 is addressed by allowing the definition of virtual links, point 4, by supporting computation of aggregate functions on subgraphs of the document graph, and point 5, versioning, by using `GraphLog` to specify versioning policies. Finally, `GraphLog` can be easily extended to more elaborate object-oriented models that address point 2.

`GraphLog` queries are visually oriented; they are formulated by drawing with a graph editor the patterns that are to be searched for in the hypertext network. Halasz distinguishes in [?] between *structural* and *content based* search. Content based searches will find nodes that contain certain patterns; structural searches look for whole subgraphs of the overall graph that have a certain structure. `GraphLog` emphasizes structural queries, although in the conclusions we suggest the integration of structural and content-based queries in a single language.

An important issue in the design of such a language is expressive power: too little power limits the applications of the language, while too much makes efficient implementation difficult and probably affects ease of use. We have two sorts of arguments for the adequacy of the expressive power of `GraphLog`. Elsewhere, we have used notions from deductive database theory and descriptive complexity to characterize from a theoretical point of view the class of queries that can be formulated in the language[1][?]. In this paper, from a practical point of view, we survey several existing hypertext systems and queries described by their authors and show how they can all be expressed in `GraphLog`.

THE QUERY LANGUAGE

The graph-based query language G^+ [?] provided a starting point for `GraphLog`. We have extended G^+ by adding negation and changing the semantics to make the definition of the language simpler. In `GraphLog`, an interrelated collection of documents – a hyperdocument – is viewed as a directed labelled graph. A *query* is a graph pattern containing one distinguished edge. The effect of the query is to find all instances of the pattern that occur in the hyperdocument and for each one of them define the "virtual link" represented by the distinguished edge. Graph patterns are themselves graphs, and they can be specified by drawing them on a screen.

The formal semantics of the language is given in [?, ?]. Each query is given a precise meaning by associating it with a set of recursive Horn clauses defined on the relations that make up the graph. Instead of giving the full definition of the language here, we will introduce it by a series of examples and informal explanations.

Consider a hierarchical document where there are nodes for each chapter, section, subsection, etc., and edges labelled `contains` relate each part to its subparts. The query in Figure 1 defines a virtual

---

[1] In fact, the name of the language comes from its close relationship to Datalog, in turn a relative of Prolog.

link top that points from each component of the document directly to the top-level component. In this case the pattern is a pair of nodes connected by an arbitrary sequence of contains links such that the second node has no incoming contains link. For each such pattern, the query defines a top link between these nodes. Note the regular expression $contains^+$ labelling a dashed edge in the query. This means the pattern to be found is a path composed of any number of edges, each one labelled with contains. In general, any regular expression may be used; for example, if instead of contains, the graph used several different link types such as has-chapter, has-section, has-subsection, we could replace the regular expression $contains^+$ with (has-chapter | has-section | has-subsection)$^+$. The crossed-out edge in the query means that a node only qualifies as a top node if there is *no* contains edge coming into it.

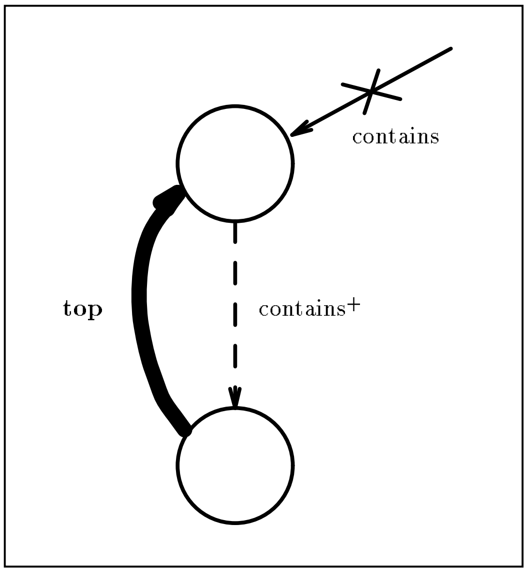

Figure 1: Defining a virtual link.

This simple example already shows that **GraphLog** can express queries that are not expressible in conventional database languages such as relational algebra. The dashed edge, standing for a path of arbitrary length, corresponds to an arbitrarily long sequence of "join" operators, and there is no single relational algebra query equivalent to it, as is well known in database theory [?].

What can we do with a virtual link such as top once we have defined it? We have three possibilities. First, we may wish to treat it simply as the answer to a query, that is, to display it to the user in some form and then forget it. Second, we may wish to incorporate it into the document, treating it as a *snapshot*. In this case, the link as computed by the query will remain in the document, but it will not be affected by future changes. Finally, we may make it into a *view*, meaning that not only does it get incorporated to the document, but it is also kept dynamically up to date, so that if, say, a new leaf node is added to the tree, its top link is automatically inserted. This choice is independent of the query language and would be made by the user interface in consultation with the user.

From the point of view of a user, once a link is created it can be manipulated in the same ways no matter whether it is a snapshot, a view, or a manually created set of edges. As an example of this, suppose that after creating the top link we realize that the network is not really a hierarchy, because there are nodes that belong to more than one document. We now want to link explicitly all top level nodes that share a component document. The query in Figure 2 generates a shares-with link between any two top nodes that have a common sub-document.

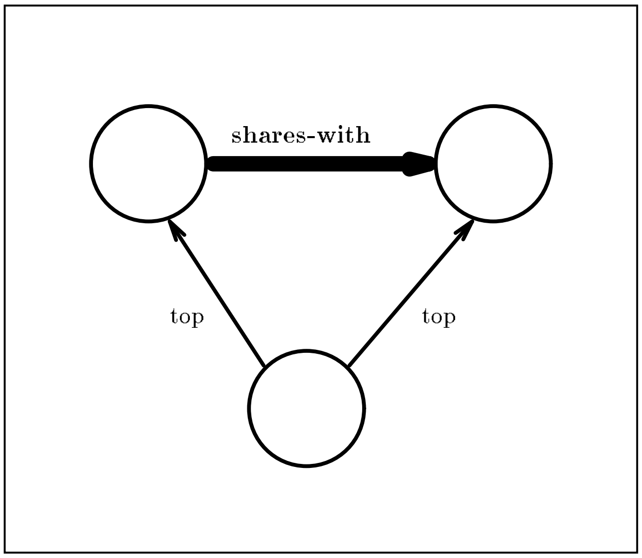

Figure 2: Documents that share a component.

So far we have shown edges labelled only with a property name. This is the simplest case, and the label corresponds to the type of the link. In general, an edge may be labelled with a literal of the form $p(c_1, ..., c_n)$, associating n atomic values with the relationship between the endpoints.

Consider for example a rather different hyperdocument that might be used by a travel agency. The nodes contain textual and pictorial information about cities. One type of link between cities represents flights. We could then have edge labels of the form flight(Airline,Departure,Arrival). Another type of link gives distance information. Edge labels for this kind of links have the form dist(Distance). These values can be used in queries in many ways. Figure 3 shows a query that defines a link ind-dist(D) between Toronto and Vancouver by adding the Toronto-Calgary distance to the Calgary-Vancouver distance. Note that we are modelling node attributes such as city name by edges going to rectangular nodes. This is just to keep the data model as simple as possible; it is straightforward to incorporate node attributes explicitly if we wish.

This last example was somewhat artificial; it would make more sense, if we do not know the distance between Toronto and Vancouver, to estimate it as the *smallest* sum of distances from Toronto to some city C and from C to Vancouver. Figure 4 shows how the *aggregate operator* min can be used in **GraphLog** to express this. The distinguished edge will be labelled by ind-dist(S) where S is the smallest of all sums defined above. The fact that the intermediate node is doubly circled is meant to suggest that we are computing an aggregate over all possible choices of this intermediate node.

Next, as the reader may expect, we are going to generalize Figure 4 to compute the *shortest distance* between Toronto and Vancouver independently of how many intermediate cities we go through. Instead of simply adding two distances, we now need to be able to add all distances appearing along a path of arbitrary length between Toronto and Vancouver. The dashed edge in Figure 5 between the two cities represents these paths. The distinguished edge will be labelled with the minimum over all paths of the sum of the distances along each path. Note the label on the path between

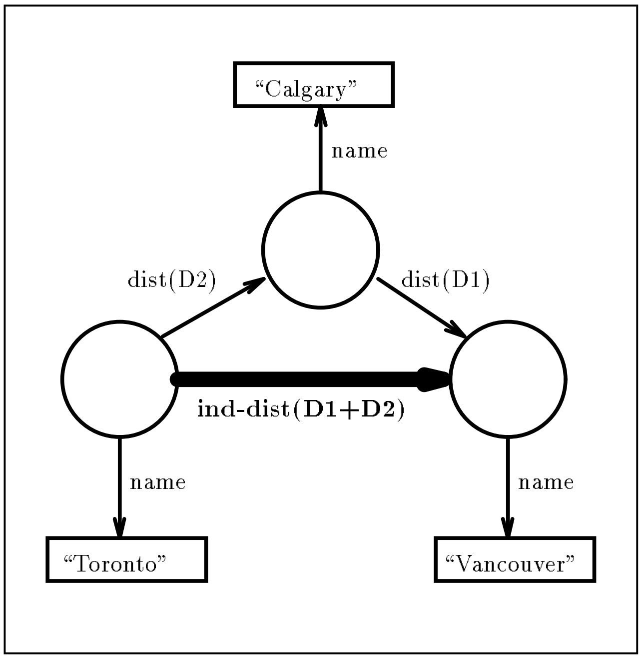

Figure 3: Distance between two cities via a third one.

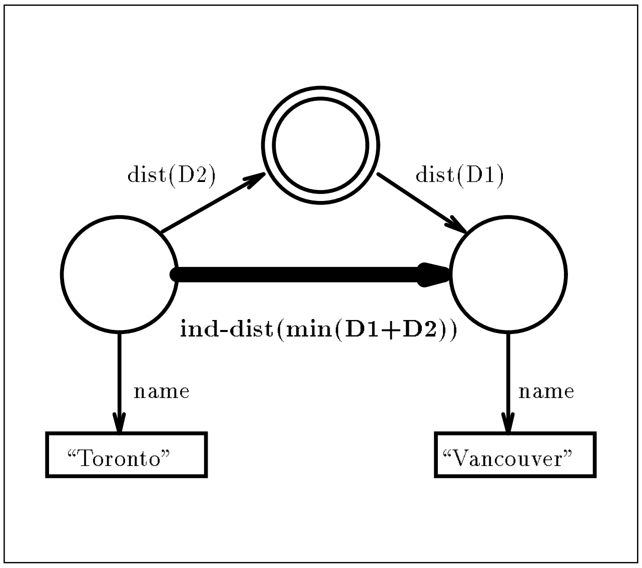

Figure 4: Distance between two cities via any third one.

Toronto and Vancouver is $\mathsf{dist}(\{\mathsf{D}\})^+$, not $\mathsf{dist}(\mathsf{D})^+$. This is because a label $\mathsf{dist}(\mathsf{D})^+$ would mean we are looking for paths such that, for some distance D, all hops along the path are of length D, which is not what we want. The notation $\{\mathsf{D}\}$ is meant to suggest that we want to collect the *set* of all distances because we are going to apply the *path summarization operator* sum to them.

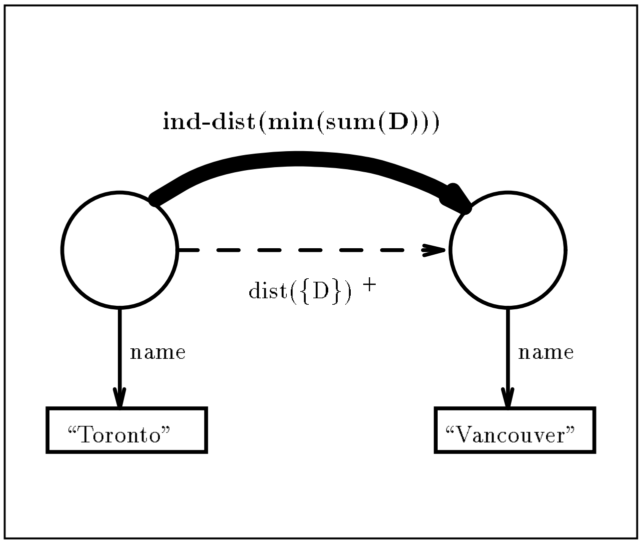

Figure 5: Shortest distance between two cities.

EXPRESSIVE POWER

The expressive power of `GraphLog` can be characterized precisely from a theoretical point of view by relating it to the language of function-free Horn clauses called *Datalog*. Ignoring aggregate operators, `GraphLog` turns out to be equivalent to what is called *piecewise-linear, stratified Datalog*, a version of Datalog in which recursive rules are restricted to use the predicate being defined only once in its definition and negation is allowed in a controlled way [?]. Interestingly, the queries expressible in the language are exactly those than can be computed in space logarithmic in the size of the database. It is also the case that `GraphLog` with aggregate operators is more expressive than the relational algebra and calculus with aggregates of [?]. We will not present these results here; see [?] for the details.

From a practical point, we will now present examples of how `GraphLog` could be used in the context of several different hypertext systems, providing evidence for the expressive power of the language, as well as for the convenience and naturalness of its graphical representation.

NoteCards

NoteCards [?] is an "idea processing" hypertext system. Nodes in NoteCards are the electronic analog of the 3×5 familiar paper notecard. A web of typed links interconnect the note cards in a hyperdocument.

An application of the NoteCards system described in [?] consisted in authoring a public policy research paper. One kind of link used by the author, *supports* links, connected notecards with supporting arguments. The next two examples illustrate the possibilities of `GraphLog` in an idea processing hypertext system.

Example 1: *Assume that in addition to linking notecards with supporting arguments, the author indicated her belief in the strength of the support by assigning it a number between 0 and 1. The strength could be represented by an edge label of the form supports(S).*

The graphical query in Figure 6 defines a virtual link reliability(R) that connects an argument in notecard N1 to an argument in notecard N2 that supports the argument N1 and is not supported by any other argument (argument N2 is an unsupported evidence for argument N1). The value R is the reliability of the support given by the unsupported evidence N2 to argument N1; R is defined as the weakest link in the chain of arguments from N2 to N1 that is stronger than the weakest link in any other such chain. Making this connection into a virtual link would keep it updated as the author's beliefs change and new arguments are incorporated or old ones deleted.

Note there are two boxes in this query. The first one defines a property of nodes called supported; the bottom one uses this property to find most reliable unsupported arguments. The two queries could have been combined into one box in this case; in general, it is convenient to be able to break a query down into several steps, defining at each step intermediate links that can be used in subsequent steps. The user need not be concerned with the exact order in which the different steps are executed; the system will determine an ordering that ensures each link is computed before it needs to be used, as long as there are no cycles in this ordering. Sets of queries with cyclic orderings are syntactically forbidden. □

Example 2: *Suppose an author is writing a collaborative research paper and need to see what cards have been created recently by her co-authors. Figure 7 shows the query graph that defines a set of nodes with the notecards created by someone other than the author in the last three days (the constant* TODAY *is a "system provided" value for the current date). Note that, instead of defining any new links, we simply highlight one of the nodes in the query to indicate that we just want the resulting set of nodes as the answer.* □

gIBIS

The gIBIS hypertext system has a specific objective: "to provide a systems design team with a medium in which all of their work can be computer-mediated and supported" [?]. It provides a hypertext environment for the IBIS design methodology. IBIS supports the constructive discussion of the issues that arise during the design process by presenting positions that respond to the issues and arguments that support or object to the positions.

Nodes in gIBIS are of three kinds: they hold either an issue, a position or an argument. There is also a fixed number of link types. For example, a position responds-to an argument and an argument either supports or objects-to its position.

Example 3: *Figure 8 shows the graphical query that finds the issues with at least two positions without arguments (another example query from [?]).* □

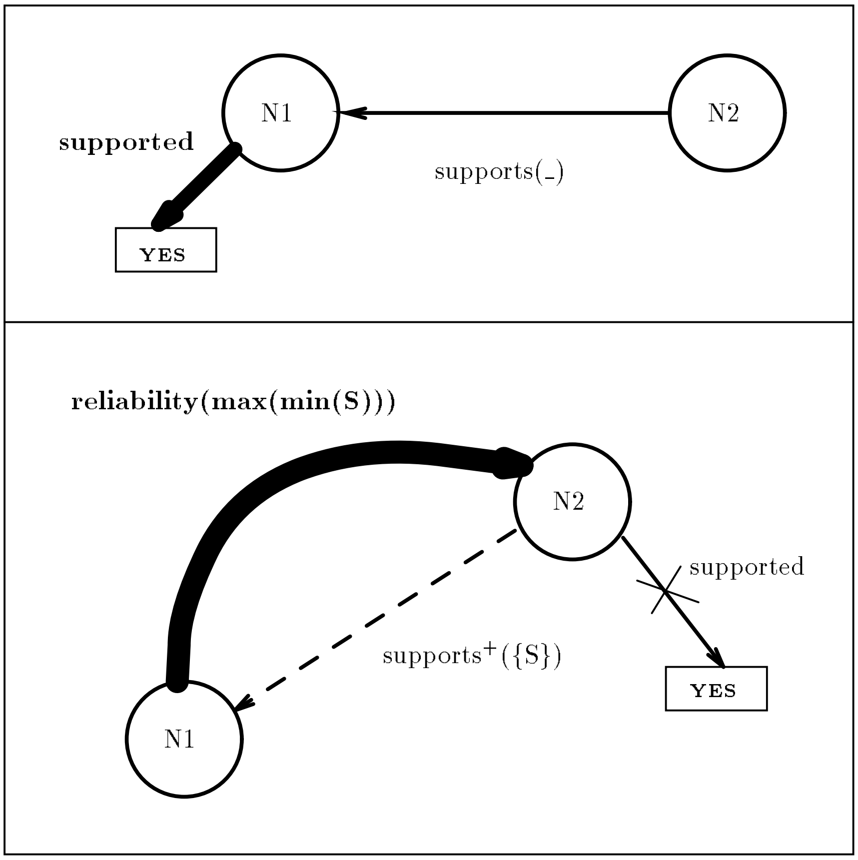

Figure 6: Creating a virtual link with the reliability of unsupported evidences.

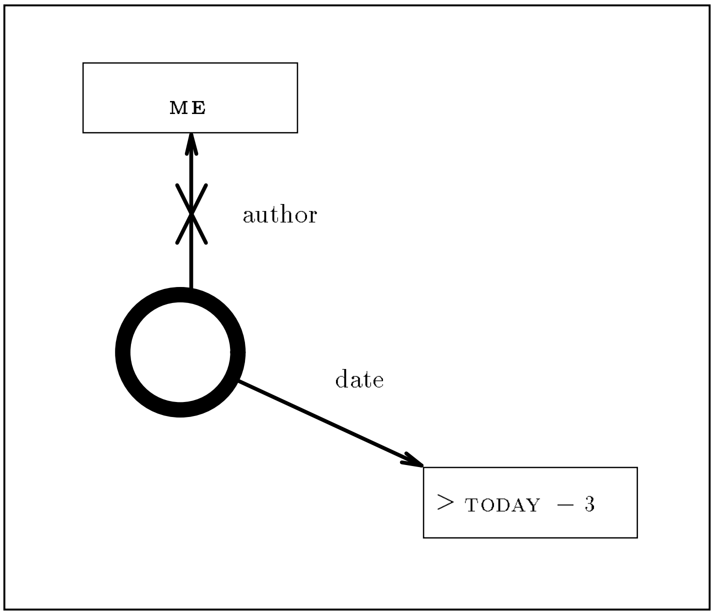

Figure 7: Nodes created by someone else in the last three days.

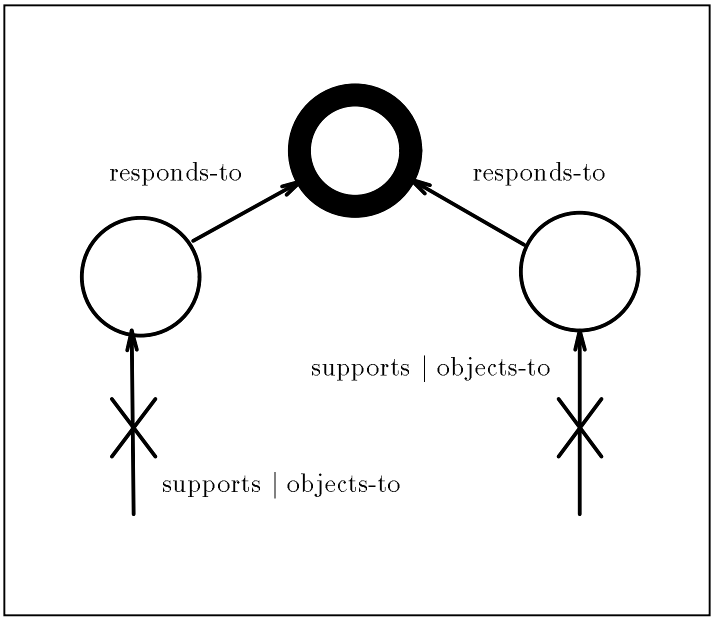

Figure 8: Issues with at least two positions without arguments.

Example 4*: This example illustrates the use of in a typical "ad-hoc" query: how popular are an author's positions in a gIBIS hyperdocument? We will break this query down into three steps. The first two associate with each position* P *two links called* count-s *and* count-o *that point to nodes containing respectively the total number of arguments that support* P *and the total number of arguments that object to* P*. The third step computes the average of the support-to-objection ratio for the author's positions and stores this average as the "popularity" of the author. Figure 9 shows the graphical query that results.* □

In gIBIS, a user can also group an issue together with its positions and corresponding arguments into what constitutes an IPA (issue-position-argument) composite node. This IPA composite node is used to record the decision reached on an issue. `GraphLog` would allow a flexible selection of what nodes should be grouped in a composite.

History Mechanism

A very useful aid to avoid disorientation when browsing a hyperdocument is the history mechanism. Systems that support a history mechanism (like ZOG/KMS and HyperCard) provide the user with virtual links that connect in sequence the last nodes visited.

A slightly more sophisticated history mechanism would create attributes for the nodes in the history trace, recording for each node the time at which the user opened and closed the node. The following example illustrates a `GraphLog` query that uses the history mechanism to locate previously neglected relevant information.

Example 5*: After a couple of hours of working within a hyperdocument, a user realizes that while*

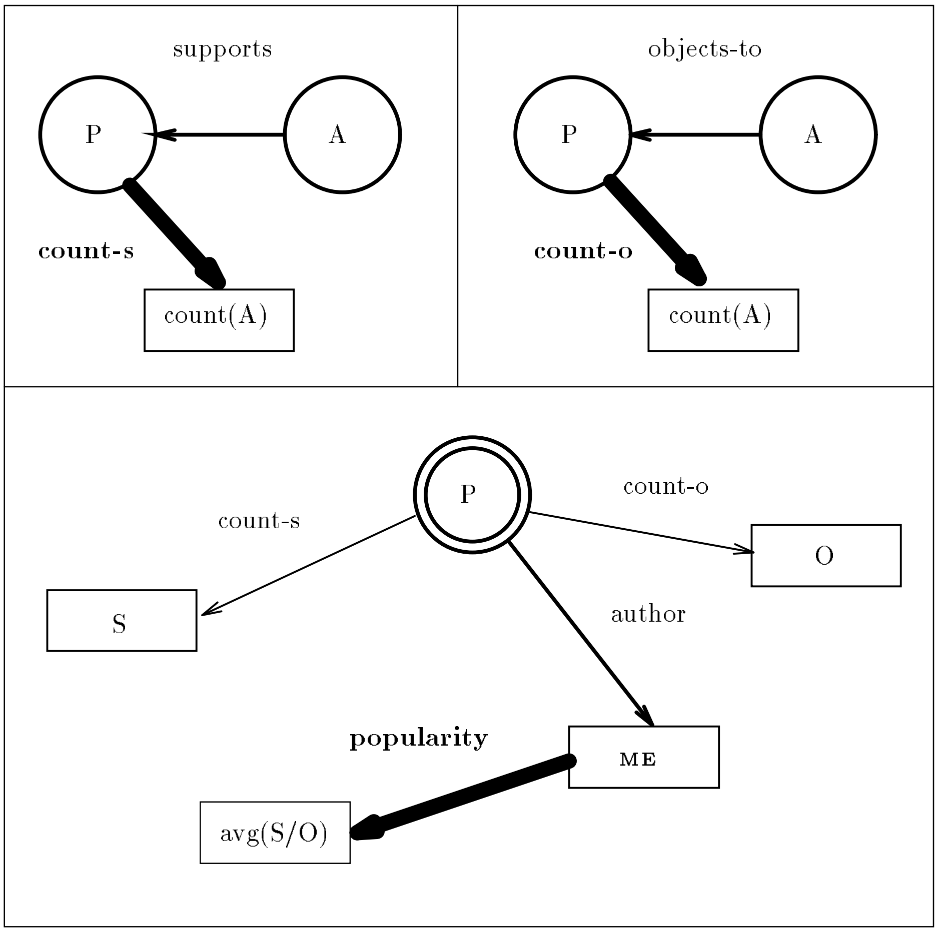

Figure 9: Finding out how popular are an author's positions.

avoiding distractions by not following links named **digression** *he probably missed some important point. He wonders:*

> *"Where did I see a link labelled "digression"? I remember that it was between 2 and 3 hours ago."*

Figure 10 shows the query graph that helps the user to trim down the search. The constants HERE *and* NOW *are "system provided" values for the currently open node and the current time, respectively. Note how the arithmetic comparisons "<" and ">" are represented by links like any other relationship between nodes, although both the links and the nodes to which they point are virtual in this case.* □

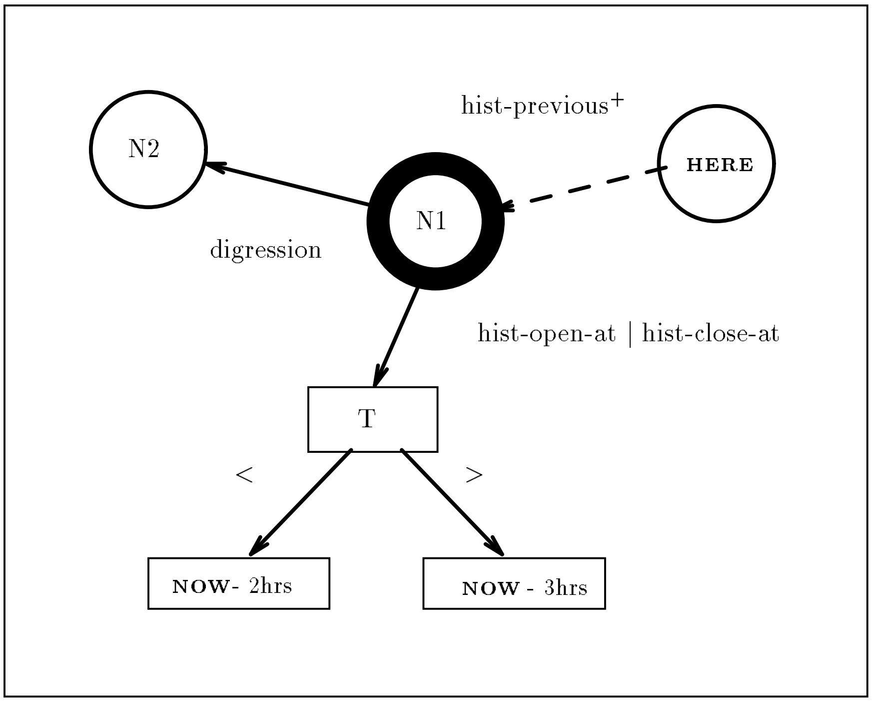

Figure 10: Searching back the history mechanism.

DynamicDesign

DynamicDesign [?, ?] is a CASE (Computer-Aided Software Engineering) environment for the C programming language implemented as a front-end to the HAM hypertext storage system.

The nodes store all the components of a software engineering project. A node attribute, project-component, takes values that indicate the kind of component stored in the node: requirement, specification, object-code, source-code (one C function is stored per node), library, comment, dictionary (i.e., symbol table), etc.

The links are used to relate the different components. A link attribute, relation, describes the kind of relationship between nodes: calls (between functions stored in source code nodes), refers-to (from a function to a dictionary; it describes the C variable referred to in the function and has an additional attribute with the name of the C variable), in-library (from functions to libraries), implements (from functions to specifications), follows-from (describing the linear order between nodes when printed or compiled), etc.

DynamicDesign answers some queries on the structure of the hyperdocument of a software engineering project:

- What does a function do? (follows comments, implements)
- Who (directly) calls this function? (follows calls)
- What is this variable used for? (follows comments)
- Who (directly) uses this variable? (follows refers-to(V))

The importance of these queries is clear. It is also clear that there are several other relevant queries (e.g., who directly or indirectly uses this function?), and that not all of them can be anticipated and "built into" the environment.

Adding **GraphLog** to DynamicDesign allows dynamic specification of queries that were not anticipated by the system designers. The different values of the relation link attribute can be used to define **GraphLog** virtual links calls, follows-from, in-library, implements and refers-to(V), where V is a C variable name. The example below illustrates one application of **GraphLog** in the DynamicDesign hypertext system. It also introduces a new feature of the language, link inversion.

Example 6*: Figure 11 shows the query graph that finds the functions F1 that share a variable with some function F2 implementing (directly or indirectly) the io-spec specifications, but not belonging to the syncio library nor calling any function in it. The refers-to(V) links from nodes F1 and F2 to node D means that variable V, defined in dictionary D, is referenced in both nodes. Note the "– calls" label on the path from F2 to io-spec. We are looking for a path from function F2 to some specification whose name is io-spec. The path may be a direct one, in which case we do not use the "– calls" part, or it may be that F2 is at the end of a chain of functions* G_1, G_2, …, G_n, *such that* G_1 *implements io-spec and each* G_i *calls* G_{i+1}, *and* G_n *calls F2. The edges from* G_i *to* G_{i+1} *go in the opposite direction to the path; this is the purpose of the inversion operator "–". Similarly, the in-library link is inverted in the path from F2 to syncio. Notice that the node labelled with variable D is not strictly necessary; it could have been omitted by using a link from node F1 to node F2 labelled "refers-to(V) – refers-to(V)".* □

The HAM versioning mechanism is particularly useful for a CASE application. The next example illustrates a query that uses a link next-version to locate a specific piece of code.

Example 7*: Suppose a programmer has to find the code for a particular function, and she gives the following information:*

> *"I am looking for the last version of a function that implements the security-spec; the first versions were done by myself; then Dennis took charge of it; I do not remember the authors that followed him; maybe Dorothy or Chris were in charge before Jeff, who I am sure wrote the current version."*

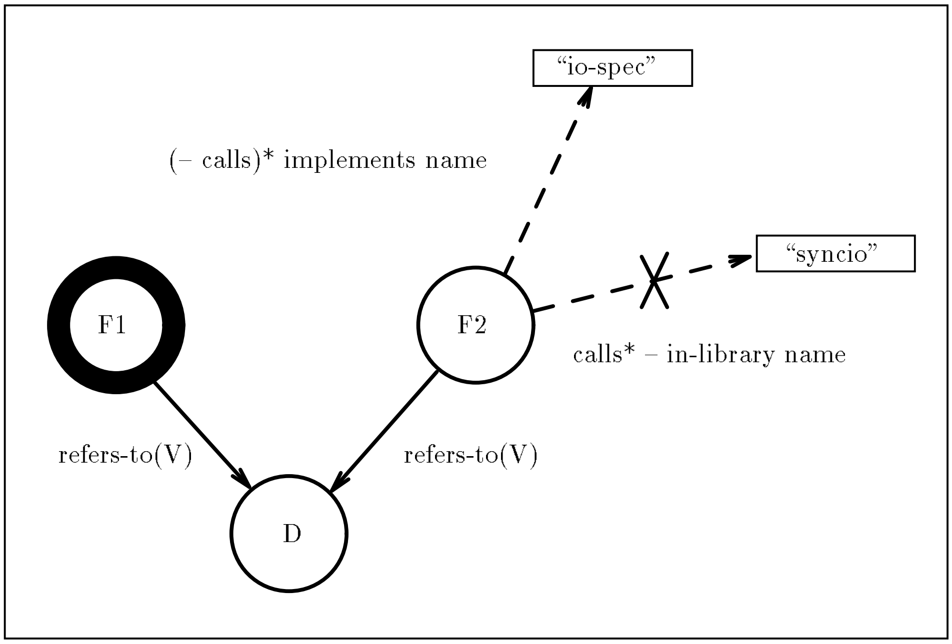

Figure 11: Finding code in hypertext CASE.

The graphical query of Figure 12 finds the functions satisfying the above description. The first graph simply adds to the existing link between two successive versions a new link indicating who is the author of the second version. The second graph looks for the current version of functions F1 that implement, directly or indirectly, the security-spec specification and are preceded by a chain of versions satisfying the rather vague criteria the user has in mind. Note that, even though the system does not support approximate search, the flexibility of regular expressions does provide some of the power of approximate matching. In particular, the occurrence of an underscore in a regular expression involving closure means that we are looking for an arbitrary sequence of values along a path. □

HAM Versioning

Versioning is an important feature in hypermedia systems [?, ?]. The specific details of versioning mechanisms differ from system to system. An advantage of a query language like `GraphLog`, that has the capability to describe revision sequences, is that it can specify quite clearly these details as well as implement variations of the versioning policies originally provided by the hypertext system.

Example 8*: In the HAM, a link may or may not keep linking the most recent versions of the nodes it connects. This is a user's choice, selected by setting to* YES *or* NO *a keep-up-to-date link attribute defined by the system. Keeping links up to date is the default policy, while not doing so is useful to retain links between previous (fixed) node versions.*

Figure 13 describes the "keep up to date" policy for link versions in the HAM. To simplify the graphical query, current-version is defined simultaneously for links and nodes in the first query graph.

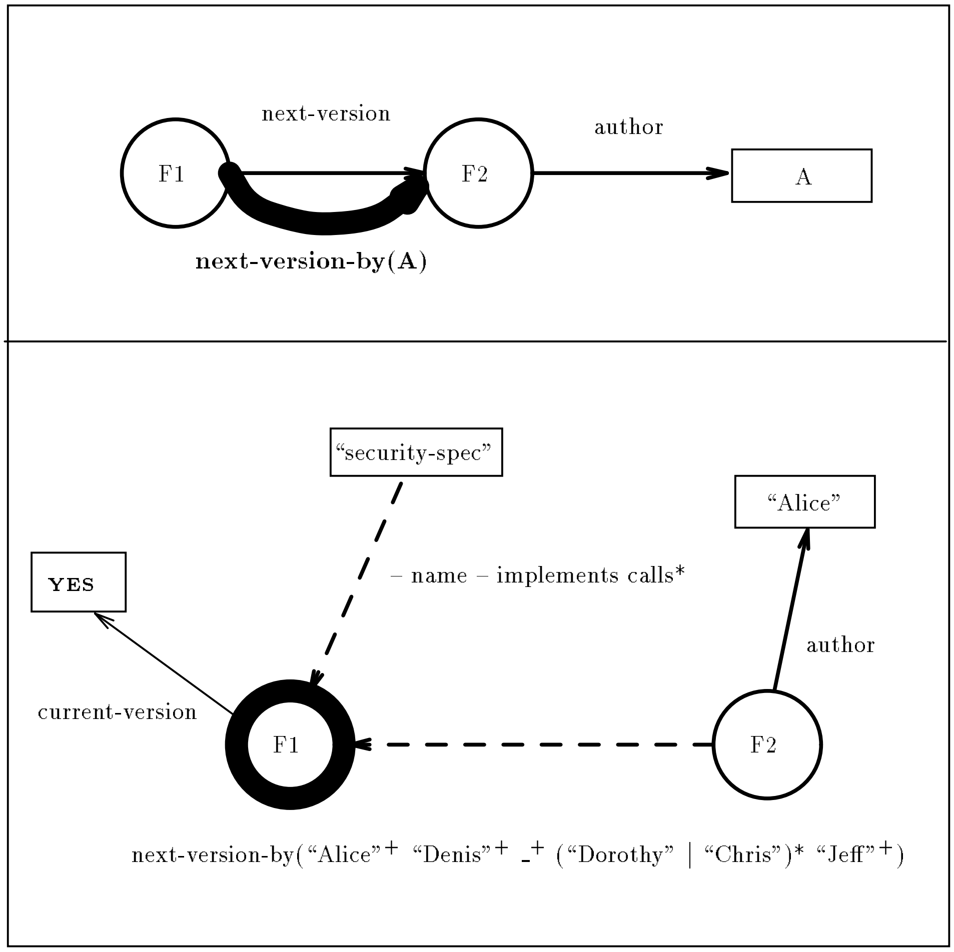

Figure 12: Querying the versioning mechanism.

The second query graph defines the from *relation between a link and its start-point node to be kept up to date with the last node (and link) version if the* keep-up-to-date *attribute is set to* YES. *The third query graph contemplates the situation where no "keeping up to date" is desired.* □

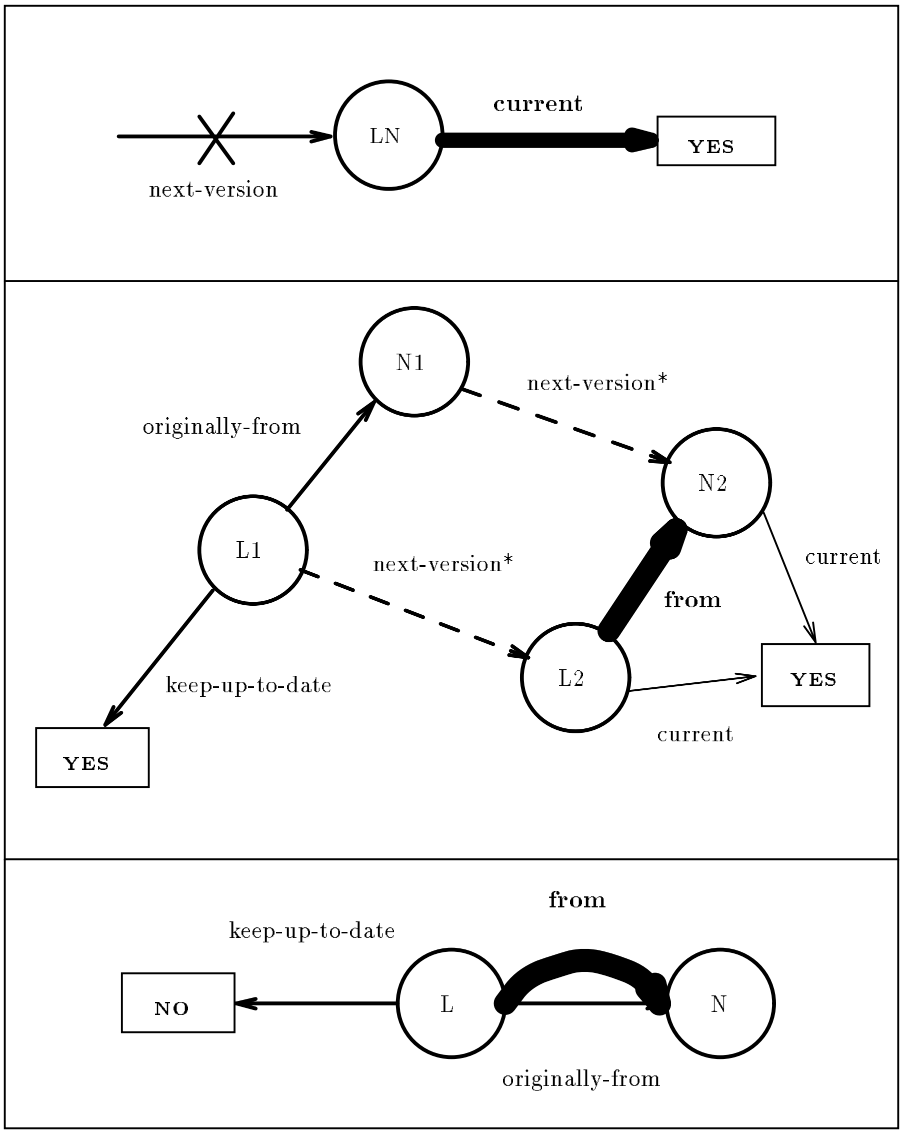

Figure 13: Keep up to date link version policy.

Dynamic Medical Handbook

In the Dynamic Medical Handbook [?, ?] a content-based query is given as a set of keywords and the sectioning structure of the hyperdocument is used to help find the best starting points (i.e., either a chapter, section, subsection, and so on) for the interested reader. This search mechanism constitutes