Architecture and Applications of the Hy$^+$ Visualization System

Mariano P. Consens, Frank Ch. Eigler, Masum Z. Hasan,

Alberto O. Mendelzon, Emanuel G. Noik, Arthur G. Ryman, and Dimitra Vista

Abstract

The Hy$^+$ system is a generic visualization tool that supports a novel visual query language called GraphLog. In Hy$^+$, visualizations are based on a graphical formalism that allows comprehensible representations of databases, queries, and query answers to be interactively manipulated. This paper describes the design, architecture and features of Hy$^+$ with a number of applications in software engineering and network management. These examples demonstrate the benefits of using a visualization-based system to carry out tasks of ever-increasing complexity in these areas and others.

1 History and Motivation

Visual presentations are widely considered an effective tool to help manage large and complex collections of data. Researchers in scientific visualization (see, for example, [21]) were the first to exploit computer graphics technology to achieve dramatic improvements in the ability of people to understand the data they work with. The motivation for this is summarized in these words from the "Panel Report on Visualization in Scientific Computing," which appeared in the November 1987 issue of *Computer Graphics*:

> The gigabit bandwidth of the eye/visual cortex system permits much faster perception of geometric and spatial relationships than any other mode, making the power of supercomputers more accessible. Users from industry, universities, medicine and government are largely unable to comprehend or influence the “fire hoses” of data produced by contemporary sources such as supercomputers and satellites.

In other diverse domains, such as software engineering, computer network management, and parallel program monitoring, researchers, developers, and users increasingly consider the importance of visual presentation of data. A related idea is to provide data manipulation tools that are themselves visually oriented. The iconic user interfaces common in today's workstations are examples of these. Visual query languages for databases [2] are more ambitious tools for visual data manipulation.

In this paper, we give an overview of the approach to visual display and manipulation of databases that we have been investigating at the University of Toronto for the past few years. We present the design and architecture of the Hy$^+$ visualization system and its associated visual query language, GraphLog. Moreover, we describe the use of Hy$^+$ and GraphLog in two different application areas: software engineering and network management. In both areas the data have a graph-like structure that can be visualized. Visual queries explore that structure and help the user of Hy$^+$ better understand it.

Section 2 gives a tour of the Hy$^+$ system, using a software engineering application as the example. Section 3 describes the architecture and implementation of the system, particularly of the query processing and graph layout components. Section 4 discusses further applications to software engineering and Section 5, to network management. We conclude with a discussion for further work in Section 6.

2 A Tour of Hy^+

Hy^+ provides a user interface with extensive support for visualizing structural (or relational) data as *hygraphs* [5], an extension of graphs inspired by Harel's higraphs [17]. The Hy^+ system supports visualization of the actual *database instances*, not just diagrammatic representations of the database schema. Given the large volume of data that the system must present to the user, it is fundamental to provide him or her with two fundamental capabilities.

The first one is the ability to *define* new relationships by using queries. This is the traditional way of using database queries: the newly defined relationship either gives a direct answer to a user question, or it provides a new view on the existing data. The derived data can later be presented visually by the system.

The second capability is a way of using queries to decide what data to *show*. The user can selectively restrict the amount of information to be displayed. This *filtering* of relevant information [5] is fundamental for conveying manageable volumes of visual information to the user. Selective data visualization can be used to locate relevant data, to restrict visualization to interesting portions of the data (that is, deciding *what* data to present) and to control the level of detail at which the data is presented (that is, choosing *how* to see the data).

To describe queries, Hy^+ relies on a visual pattern-based notation. The patterns are expressions of the GraphLog query language [4, 9]. Overall, the system supports query visualization (that is, presenting the description of the query using a visual notation), visualization of data that constitutes the input to the query, and visual presentation of the result [5].

We present an example of using GraphLog and its environment Hy^+ for visualizing the structure of the National Institutes of Health (NIH) public domain C++ class library. The

IBM[1] XL C++ compiler was used to extract 13 000 facts from the 35 000 lines of code in the library. In the generated hygraph, the nodes represent classes, functions, and variables. The edges represent relations among them, such as the `subclass` and `friend` relationships between classes, the `mem` relationship between a class and its member functions or variables, the `ref` relationship between a function and all variables referenced by it, and a `calls` relationship between functions.

CASE tools usually provide a static visualization of the architecture of a software system. Hy$^+$ permits a more dynamic approach to the visualization process, allowing different visualizations of the same component to be obtained through different GraphLog queries. In that respect, Hy$^+$ can not only visualize the structure of the class library, but can also explore and better understand that structure.

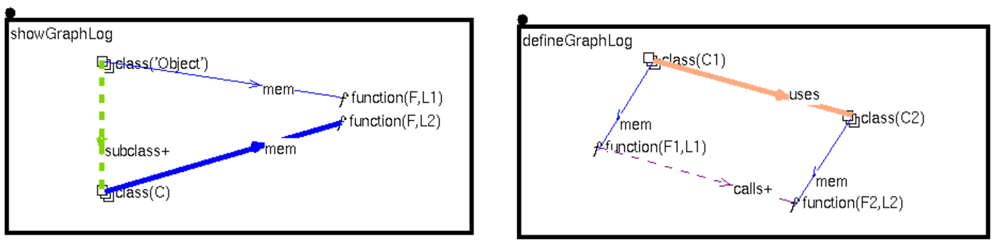

Figure 1: Examples of a filter and a define query

Any visualization can by queried. Figure 1 contains two examples of queries. The first query is what in GraphLog is called a *filter* query: a GraphLog expression enclosed in a **showGraphLog** box. It describes a pattern to be matched in the hygraph designated as the database. As one can see, one of the edges in this pattern is thick, that is, distinguished. This is a visual way of distinguishing edges that the user actually wants to see after the

[1]Trademark of International Business Machine Corporation

match is found. This particular *filter* query searches the database for subclasses `C` of class `class('Object')` that redefine some function `F` defined in `class('Object')`. From all objects that match this pattern, only the relation `mem` between the subclass `C` and the inherited function `F` are displayed to the user. Throughout the paper we use the convention that words starting with an upper case letter denote variables while these with a lower case letter denote constants.

The other query in Figure 1 is a *define query*: a GraphLog expression in a defineGraphLog box. A thick edge here has a different meaning. It represents a relation that is defined every time the pattern is found in the hygraph designated as the database. The *define* query in Figure 1 defines the relation `uses` between a class `C1` and another class `C2`, whenever `C1` contains a function `F1` that calls a function `F2`, which is defined in `C2`.

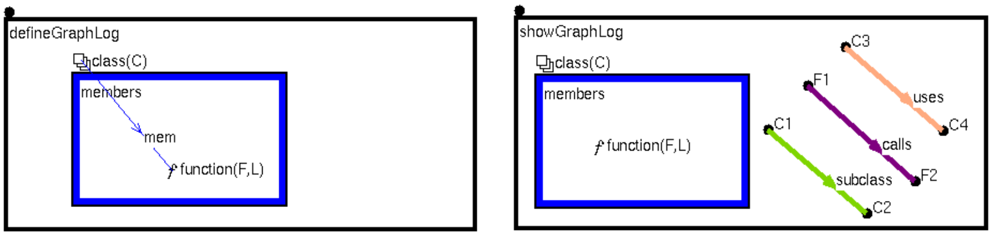

Figure 2: Definition and filtering of `members`

Hygraphs extend graphs by using blobs in addition to edges to represent relationships among nodes. A *blob* in a hygraph represents a relation between a node, called the *container node*, and a set of other nodes, called the *contained nodes*. Blobs are hence generalizations of edges and can be used to cluster related nodes together. Visually they are represented as a rectangular area associated with the container node. The *define* query of Figure 2 demonstrates how we can change the representation of a relationship from edges to blobs.

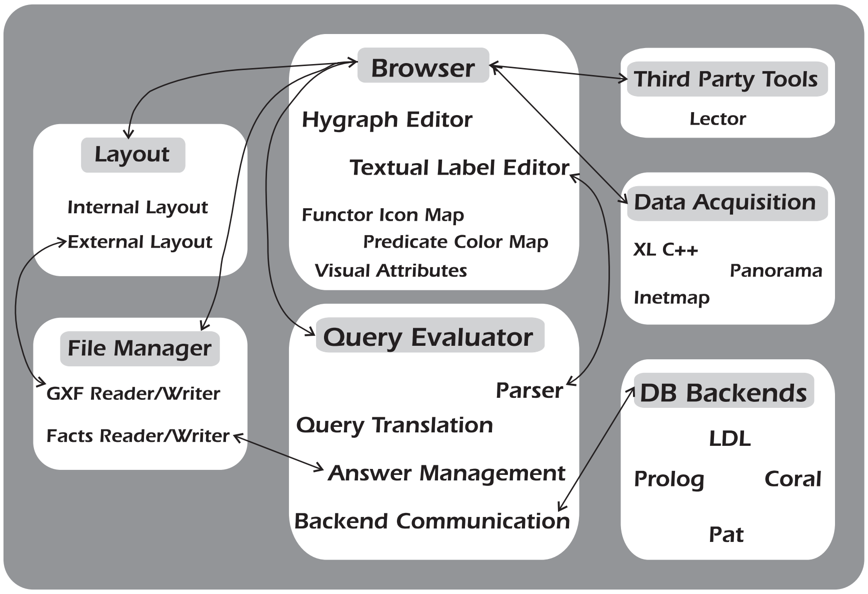

Figure 3: Hy^+ Architecture Overview

The blob `members` clusters the member functions of a class together: a function `F` is enclosed in the blob associated with `class(C)`, whenever `mem(class(C), function(F,L))`. A blob relation can be treated similarly to an edge relation. The second hygraph in Figure 2 is a *filter* query which generates a hygraph that contain all the `members` blobs and all the `subclass`, `calls`, and `uses` edges that exist in the database.

3 System Architecture and Implementation

Hy^+ has evolved from the G^+ Visual Query System presented in [6]. The new system is implemented as a front-end, written in Smalltalk, that communicates with other programs to carry on its tasks, including multiple database back-ends for the actual evaluation of the queries. An overview of the Hy^+ system architecture is given in the diagram in Figure 3.

3.1 Hygraph Browsers

Hy^+ browsers allow users to interact with the hygraph-based visualizations that the system manipulates. They have extensive facilities for interactively editing hygraphs, including copy, cut and paste; panning and zooming; and textual editing of node and edge labels. Icons are automatically selected for nodes according to the functor of the node label (that is, the type of data object represented by the node). Similarly, the colors for edges and blobs are automatically selected based on the predicate in the label (that is, the relationship represented by the edge or blob).

Within Hy^+, all blobs associated with one container node are represented by rectangles contained within a rectangular region that has the container node in the topmost left corner. Blob labels are drawn in the interior of the topmost left corner of the rectangle representing the blob. Control over the level of detail displayed is achieved by interactively hiding and showing blob contents. Blob contents can be interactively hidden to further reduce clutter. When blob contents are hidden, the incoming and outgoing edges are also hidden (but there are options to present summaries of the information carried by the hidden edges).

3.2 Query Processing

Query processing in Hy^+ is performed by translating queries (and data, if necessary) into logic programs suitable for execution by one of two back-ends: the logic programming language LDL of MCC [24] and the experimental deductive language CORAL of the University of Winsonsin [28]. In this section, we give a more precise definition of GraphLog, and we describe how the query processing proceeds within the Hy^+ system.

In GraphLog, a *term* is one of a constant, a variable, an anonymous variable (as in Prolog), an aggregate function $f \in \{$ MAX, MIN, COUNT, SUM, AVG$\}$ applied to a variable, or a functor f applied to a number of terms. An *edge (blob) label* is a *path regular expression* generated by the following grammar, where $\bar{T}$ is a sequence of terms and p is a predicate:

$$E \leftarrow E|E; E.E; -E; (E); E+; E*; p(\bar{T}); \neg p(\bar{T})$$

Database instances are hygraphs whose nodes are labeled with ground terms and whose edges and blobs are labeled with predicates. Database instances of the object-oriented or relational model can easily be visualized as hygraphs. For example, an edge (blob) labeled $p(\bar{X})$ from a node labeled T_1 to a node (containing a node) labeled T_2 corresponds to tuple $(T_1, T_2, \bar{X})$ of relation p in the relational model. No key is associated with the relation p.

Queries are sets of hygraphs whose nodes are labeled by terms, and each edge (blob) is labeled by an edge (blob) label. There are two types of queries: *define* and *filter*, as explained in Section 2. In both, the query hygraph represents a pattern; the query evaluator searches the hygraph designated as the database for all occurrences of that pattern. The difference between the two types of queries stems from their interpretation of *distinguished elements*, explained below.

A hygraph pattern in a define query (which is enclosed in a defineGraphLog box) must have only one *distinguished* edge or blob labeled by a positive literal. The meaning of the *define* query hygraph is to define the predicate in this distinguished literal in terms of the rest of the pattern. The semantics of *define* queries is given by a translation to stratified Datalog [4]. Each *define* hygraph translates to a rule with the label of the distinguished edge or blob in the head, and as many literals in the body as there are non-distinguished edges and blobs in the hygraph. Additional rules may be necessary to define the predicates of non-distinguished edges or blobs that are labeled by regular expressions. The generation of these additional rules is based on the structure of the regular expression. An alternative translation is described in [35].

A hygraph pattern in a filter query (which is enclosed in a showGraphLog box) may have several *distinguished* nodes, edges, and blobs. The meaning of a *filter* query hygraph is: for each instance of the pattern found in the database, retain the database objects that match the distinguished objects in the query. Given a hygraph in a showGraphLog box, for each distinguished edge (blob), we generate a set of *define* queries that *match* the distinguished object, that is, when they are evaluated they determine all instances of the edge (blob) that exist in the portions of the database that match the hygraph pattern. The query evaluator evaluates each of the *define* queries in turn. The results are combined, and the answer to the *filter* query is found.

From a logic programming point of view, a define query corresponds to a conventional set of Horn clauses defining a certain predicate, while a filter query can be viewed as a set of Horn clause *bodies* in which certain literals are retained after each match and the rest are discarded. In a way, define queries generate *theorems*, while filter queries generate *proofs*. A formal definition of filter queries and filtering logic programs can be found in [5].

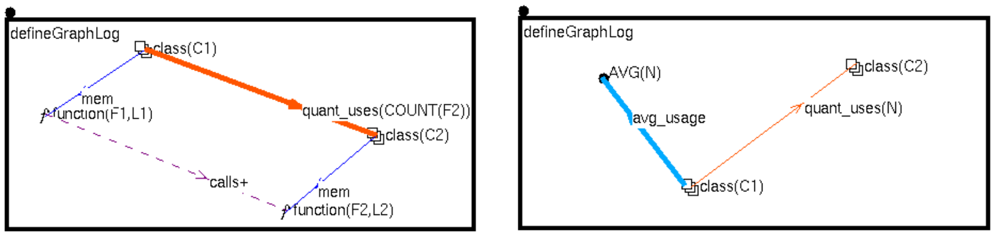

Figure 4: Examples with aggregate functions

GraphLog has the ability to collect multisets of tuples and to compute aggregate functions on them. The aggregate functions supported in GraphLog are the unary operators MAX, MIN, COUNT, SUM, and AVG. They are allowed to appear in the arguments of the distinguished relation of a *define* query as well as in its incident nodes. As an example of the use of aggregation in GraphLog, consider the two defineGraphLog blobs of Figure 4. The first one defines the relation `quant_uses` between two classes. Relation `quant_uses` is different from relation `uses` of Figure 1; the additional attribute in `quant_uses` defines the degree of coupling between the two classes. Thus, `quant_uses(C1, C2, COUNT(F2))` is defined between `C1` and `C2`, whenever `C1` contains a function `F1` such that `F1` calls `COUNT(F2)` functions `F2` in class `C2`. The second pattern defines the relation `avg_usage` to measure the average usage of each class as follows: `avg_usage` is defined between a class `C1` and a number `AVG(N)`, if `AVG(N)` is the average of all numbers `N` that count the number of calls from functions of class `C1` to functions of some other class `C2`. Note there is no explicit `GROUP BY` list (as in SQL). Instead, grouping is done implicitly over all variables appearing in the distinguished edge or its end-points.

GraphLog has higher expressive power than SQL; in particular, it can express, with no need for an explicit recursion construct, queries that involve computing transitive closures or similar graph traversal operations. The language is also capable of expressing first order aggregation queries, as well as aggregation along path traversals (for example, shortest path queries) [3].

3.3 Data Acquisition

The Hy$^+$ system relies on other programs (which are part of the Data Acquisition subsystem) to supply the raw data to be visually manipulated within the system. The File Manager subsystem can directly import files containing logical facts (like the ones produced by the XL C++ compiler for the NIH database of Section 2). These files can also be obtained from relational and deductive databases.

3.4 External Hygraph Representation

Graph Exchange Format (GXF) is a specification for a portable external representation for directed graphs and hygraphs [14]. Hy^+ can read and write graphs in GXF format from and to UNIX[2] files, and uses extensions of GXF to store all layout and display-related information. Processor programs may use these Hy^+-specific extensions to manipulate the visual representation of a hygraph, for example, for layout tasks.

GXF files encode hygraphs in a recursive list structure with a syntax very similar to that of LISP. A GXF file is structured into *lists* that contain one keyword, and multiple *atoms* (numbers and strings), and/or nested sublists. Unparsed comments can also be included. A GXF file consists of one or more top-level GXF lists, at least one of which has the keyword `Graph`, and it contains the encoding of a hygraph. This top-level `Graph` list is decomposed into sublists ("structures") that represent various parts of the hygraph: its edges, nodes, blobs, etc. These structures are decomposed further to describe individual edges, nodes and blobs, and further yet to describe some individual data, such as label, position, and shape. Part of the GXF representation of the graph in Figure 1 is:

```
; Hygraph fileout (a comment)
(GRAPH (ID "fche@rock.db:  Fri Jul 31 12:07:44 EDT 1992" )
        (BOUNDS (RECTANGLE (XY 0 0 ) (XY 1 1 )))
```

[2]Trademark of Unix System Laboratories, Inc.

```
(NODES (NODE (ID "1" )
        (LABEL "class('Object')" )
        (POINT  (XY 0.1 0.8 ))
        (BOUNDS (RECTANGLE (XY 0.1 0.8 ) (XY 0.11 0.81))))
⋮ more nodes)
(BLOBS (BLOB
        (FROM "4" ) ; container node
        (TO "1" ) ; contained nodes
        (TO "2" )
        (TO "3" )
        (LABEL "showGraphLog" )
        (BOUNDS (RECTANGLE (XY 0.1 0.1 ) (XY 0.9 0.5 ))))
⋮ more blobs )
(EDGES (EDGE
        (FROM "1" ) ; from-node
        (TO "2" ) ; to-node
        (LABEL "mem" ))
⋮ more edges ))
```

3.5 Tool Integration

The system has the ability to invoke external programs that, for instance, browse an object being represented by a node in one of the graphs displayed by the system. The Hy$^+$ visualizations can be used as overviews to locate information and then invoke third-party browsers to display the contents associated with the relevant objects. An obvious advantage of this approach over a purely navigational one is the ability to use the convenience and expressive power of GraphLog patterns to retrieve the objects of interest, instead of attempting an often

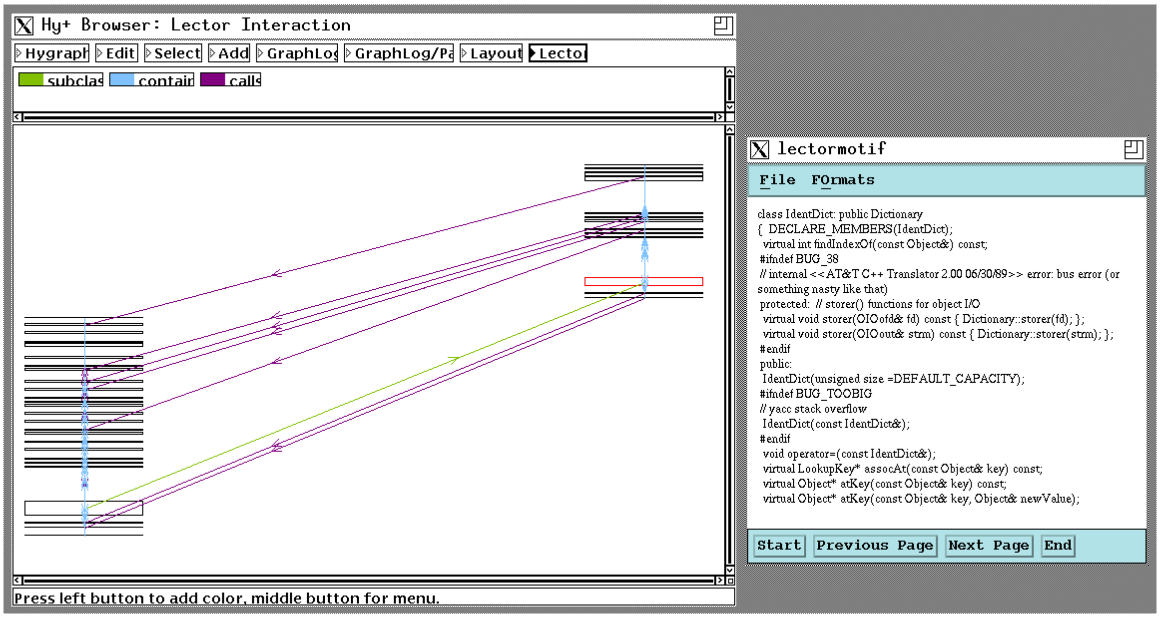

Figure 5: Synchronized Graphical and Textual Browsing of Source Code

impractical brute force search. Of course, this approach is in addition of the use of Hy$^+$ to generate as many specifically tailored overviews as needed. Figure 5 has an example of such an integration [37]. To the right of a specialized hygraph browser there is a Lector[3] [29] window that displays the source code associated with the object selected in the browser (the code for class('IdentDict')). The display synchronization works both ways: when the user changes the page of source code displayed by Lector, the object selected in the Hy$^+$ browser adjusts correspondingly. Furthermore, the query evaluation component has been extended to handle a mixture of traditional and textual queries. The latter kind of queries are handled by the PAT Text Searching Engine [16].

[3] Lector and PAT are trademarks of Open Text Systems Inc.

3.6 Hygraph Layout

When data are imported into Hy$^+$ from non-graphical sources, or when answers to queries are computed, it is necessary to compute a *layout* for the resulting hygraphs, that is, to determine how the abstract topology of the hygraph will be embedded into a 2-D (or 3-D) picture. Unlike many tools that offer a fixed set of predetermined layouts or visualizations, Hy$^+$ provides a flexible and interactive layout mechanism that is essential to allow domain experts to easily experiment with a variety of (possibly unanticipated) visual representations and layouts.

To compute hygraph layouts, Hy$^+$ uses the graphite utilities developed within our project [26]. The current implementation of graphite consists of several Unix filters. The graph layout filter reads a GXF description of a hygraph from stdin, internally modifies the position and size information of various hygraph elements in accordance to particular layout criteria, and writes the description of the modified hygraph to stdout. The graphite graph layout filter has also been used to generate 3-D graph layouts for a prototype graph browser incorporated into 4Thought [31].

Layout architecture

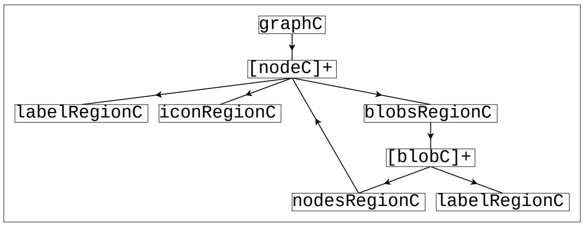

Figure 6: Mapping the hygraph formalism onto the generic container model

To provide more flexible and powerful drawing facilities, we designed every graphite layout algorithm with respect to a generic container model: a container may recursively contain (sub)containers; containers may optionally be linked by directed arcs. The hygraph formalism is mapped onto this model as shown in Figure 6. The Figure essentially says that a graph container may contain one or more node containers, each of which contains a label region, icon region, and blobs region containers. Similarly, a blobs region may contain one or more blob containers, each of which contains a nodes region and a label region container. Finally, a nodes region may contain one or more node containers. All containers are treated uniformly by layout algorithms, thereby allowing the same layout algorithm to be used to position the subcontainers of container types.

Basic layout algorithms

The current version of layout filter is capable of producing a drawing using arbitrary combinations of the following simple parametric layout algorithms:[4]

1. cluster: size-based clustering layout. Containers are divided into clusters according to size and shape similarity. Each cluster is laid out independently, and then resulting drawings are combined to form a composite, similar to the technique described in [18].

2. circle: elliptical layout. Containers are positioned along the perimeter of an ellipse; containers are ordered to reduce (but not necessarily minimize) the number of arc crossings.

[4] Each algorithm supports both two-dimensional and three-dimensional layouts.

3. grid: grid layout. Containers are positioned at the intersections of an $m \times n$ grid. Containers are placed, one-connected component at a time, in top to bottom left to right order, and are ordered to reduce (but not necessarily to minimize) the number of arc crossings.

4. hier: hierarchical layout. The algorithm computes a depth-first spanning forest F of the input graph, and places containers by following tree arcs with respect to F [1], and is similar to several approaches found in the literature [30, 36].

5. nop: no layout. The positions of containers are not altered. This method can be used to provide a form of incremental layout capability.

6. overlap: overlap elimination. Containers are repositioned to eliminate overlaps in a manner similar to the *horizontal shuffle* algorithm [13], except that containers are shuffled along both the x and y axes (and z for 3-D layouts).

7. pack: quadtree-based two-dimensional bin packing. Containers are placed into a small enclosing region by an incremental (and heuristic) quadtree growing approach.

8. random: random layout. Containers are positioned randomly without overlaps.

9. spring: force-based layout. Containers are positioned according to the following analogy to a physical system: containers are treated as charged bodies that repel one another; arcs are treated as springs attached to pairs of containers that pull each member of the pair towards the other. The simulation continues until a low-energy state is reached resulting in the final layout; similar approaches have been described in [12, 15, 19].

10. stack: stack layout. Containers are positioned along the axis that results in the most square (cubical) or compact drawing.

11. 1d: one-dimensional layout. Containers are positioned along one of the specified axes.

Each layout algorithm can be controlled by manipulating a number of generic and algorithm-specific parameters whose descriptions, unfortunately, were too lengthy to be included in this paper. Furthermore, the graphite layout filter can also be used to generate non-geometric fish-eye views of nested graphs [25, 27], that can be used to balance local detail and global context by automatically emphasizing some regions of the layout while simultaneously deemphasizing others.

Composite hygraph layout

Composite layout is a graph layout technique in which the input graph is partitioned into several subgraphs that are laid out independently (by different layout algorithms) and then composed to produce the final drawing [22],[18].

The graphite layout filter is capable of performing composite layout of general graphs at a much finer level of granularity than previous efforts. The filter allows a (possibly unique) algorithm to be associated with each container in the container hierarchy (see Figure 6) induced by a hygraph. If a container is not assigned a layout algorithm, it inherits the algorithm of the parent container. In a composite layout scenario, in addition to the top-level container, a (possibly) unique layout algorithm can be associated with each node, blob, nodes region, and blobs region container, by inserting appropriate layout directives into the GXF file.

4 Software Engineering Applications

In Section 2, we introduced Hy$^+$ and the GraphLog query language by visualizing the structure of a C++ class library and querying that structure during software development to facilitate program understanding. Hy$^+$ has been applied to software structures in [10], [11], and for debugging distributed applications in [8].

In this section, we give additional examples using the same database of Section 2, that is, the NIH C++ library. Once again, the objects of interest include classes, functions, and variables. Relations between these objects include the subclass relation between two classes, the mem relation between a class and its member functions, the friend relation between two classes or between a class and a function, the relation virtual between a class and its virtual member functions, and the relation pure-virtual between a class and its pure virtual functions (that is, functions that are defined as virtual but with no implementation given for them).

In a language with inheritance, it is not always obvious which version of a function will be invoked by a function call. It is useful for a C++ programmer to know where in the inheritance hierarchy a function is defined. Hy$^+$ can group together all "synonyms" of a function and associate the resulting set with the name of the function using blobs. The upper part of Figure 7 demonstrates how. First, the relation fmem is defined to relate a class C with a triple fcl(F,C,L), if class C defines function F at source code line number L (ignoring for simplicity the fact that code comes from multiple files). Then, if class C2 is a subclass of class C1 (or equal to C1) and both classes define the same function F at some line of their code, the triple fcl(F,C2,L2) is placed inside the blob for the function F. The triple fcl(F,C1,L1) is placed when C1=C2 (note the use of the kleene closure). Part of the answer of requesting these synonyms appears in the bottom of Figure 7. Note that, for the function at, both

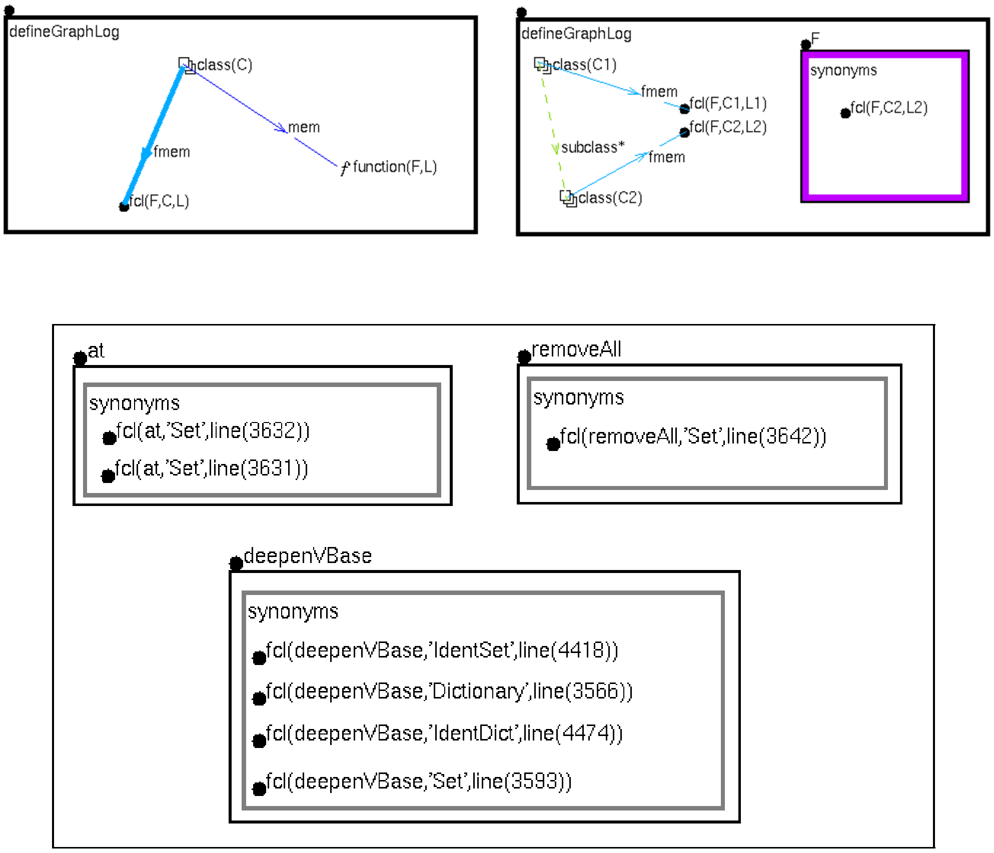

Figure 7: Defining synonyms for functions and displaying some of them

implementations are provided inside the class Set, while the function deepenVBase is defined once in classes Set and Dictionary and twice in class IdentDict.

Another point of interest for C++ programmers is whether a class contains functions that are pure virtual; that is, their interface is available to subclasses of the class, but no implementation is provided by the class. We call such a class incomplete. Figure 8 contains the definition of incomplete classes: a class C is incomplete if it contains at least one pure virtual function. Also, a class C2 that is a subclass of an incomplete class C1 and does not provide the implementation of the inherited pure virtual function is also an incomplete class.

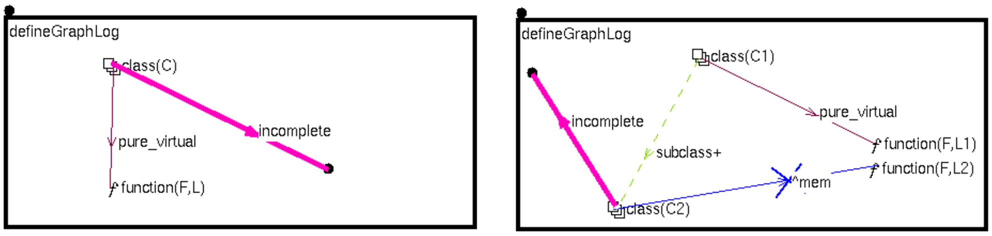

Figure 8: Identifying classes with incomplete implementation

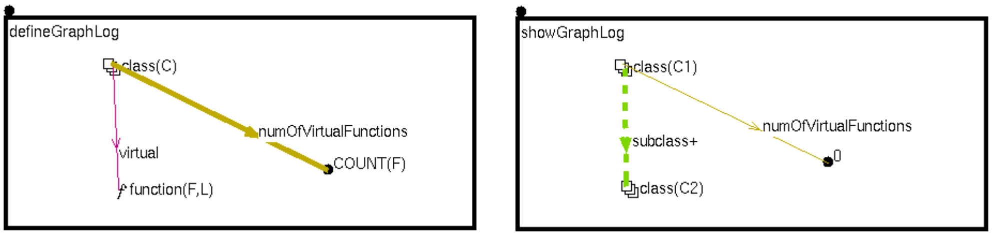

Figure 9: Inheriting from complete classes

The following two examples use Hy$^+$ for determining correctness of the C++ program. Both are taken from [23], which discusses a number of specific ways to improve C++ programs and designs. The first example involves classes that do not have any virtual functions. This means that none of its member functions can be overridden; all subclasses can add new functionality but cannot modify existing behavior, because non-virtual functions impose a mandatory implementation. This can be considered a form of limited inheritance and, according to [23], may be an indication of a design problem. The query in Figure 9 finds such designs: the relation numOfVirtualFunctions is defined between every class C and the number of virtual functions that are defined in C. If that number is zero, the class is complete; hence, subclasses may not modify any of the class's behavior. The filter query in Figure 9 requests to see the part of the hierarchy that uses any complete classes.

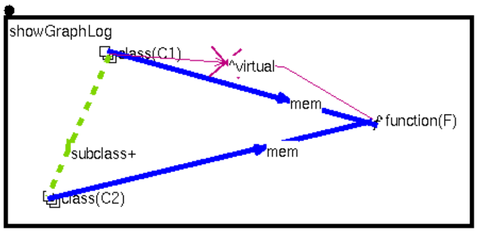

Figure 10: Redefining an inherited non-virtual function

In the same spirit is the second example from [23]. A non-virtual function has both its interface and its implementation specified by the class in which it appears. According to [23], redefining such a function in a public subclass is not good design practice. The query in Figure 10 searches for all classes C1 such that C1 redefines a non-virtual function defined in some superclass C2 of C1.

The above queries can be seen as constraints imposed upon the structure of the database. A software system satisfies these constraints if and only if the answers to the above queries are empty. More interestingly, when the system fails to satisfy a constraint, Hy$^+$ can be used to display the violation in a meaningful context so a person can decide whether the violation is harmful or not.

Technology Transfer: The Use of GraphLog in 4Thought

The 4Thought prototype [31] is a tool whose goal is to explore applications of database visualization technology to software engineering. The visual query language GraphLog is a suitable notation for expressing the type of software engineering queries that 4Thought needs to support.

4Thought was initially targeted at programming-in-the-large tasks such as architecture, design, and performance tuning [10, 32] in which the database being visualized contains structural information about the program. Here GraphLog is used as a visual query language to specify graphical views, such as call graphs and inheritance hierarchies, of the program database. We believe that GraphLog is a very intuitive way to define program views since software engineers are used to think graphically and commonly employ a wide variety of structural diagrams.

Later, 4Thought was applied to programming-in-the-small tasks such as specification [33] in which the database being visualized contains the application data. Here GraphLog is used as a visual specification language to define the computation being performed by the application. We feel that specifications written in GraphLog are much easier to understand than equivalent specifications written in purely textual notations, such as pseudo-code or predicate logic.

An exciting recent development has been the evolution of databases to support new versions of SQL that allow recursive queries. This means that commercial databases will soon be able to efficiently execute GraphLog queries, making tools like 4Thought and Hy^+ useful in application domains such as bill-of-materials and airline routes. We believe that writing queries for these recursive domains is much easier in GraphLog than in SQL, and we are investigating customer requirements for new database application development tools that exploit graphical visualization.

5 Network Management Applications

Managing a large heterogeneous computer network is a complex task. The proposed network management standards for TCP/IP-based Internets and the ISO-OSI network management standards are attempts by standards organizations to reduce the complexity involved. They propose a manager-agent paradigm, where a management station receives status data from network devices and assembles them into a global picture of the network, which is used to monitor and control network operation. Appropriately visualizing this global picture as well as others can be very effective to facilitate the tasks of managing the network. In general three techniques are useful for network management: abstraction, filtering, and visualization.

Abstraction

The complexity of managing networks is reduced by systematically imposing different levels of abstraction onto the network. Doing so allows the functional management to be performed in a structured way. Network objects can be abstracted into higher-level entities based on different criteria set by the network manager, for example, geographical sites, for example, buildings, or administrative responsibilities within the network, e.g., domains. Network objects can also be abstracted according to the ISO/OSI protocol layering, by distinguishing between two different views of the topology map of the network: the *logical map*, which corresponds to the network layer, and the *physical map*, which corresponds to the datalink and physical layers.

Filtering

The limitation in the size of the presentation media makes it difficult to visualize the status of the entire network. When a problem occurs in the network, it would be helpful to see only the relevant areas of the network. It would be nice to be able to specify what we want in a simple and declarative way, filter out the problem area, and visualize it in terms of the topology of the network and at different levels of abstraction.

Visualization

In presenting the data to the human manager, it is widely believed that graphical displays are essential; all commercial network management tools make an attempt to communicate information visually. These visualizations tend to be hardwired into the software, with limited control available to the manager. A flexible system where the what and how of visualization is under user control has significant potential for advancing the state of the art in network management software.

5.1 Network Management in Hy$^+$

In this section, we show how we can obtain different views of the network using the querying capabilities provided by the Hy$^+$ system. The example network management database contains information about the topology of the network and about its constituents. This information is obtained from MIBs (management information bases [20]). By executing the queries shown in Figure 11, we get the logical network layer map, shown on the right of the figure. Similar queries can produce the physical topology map.

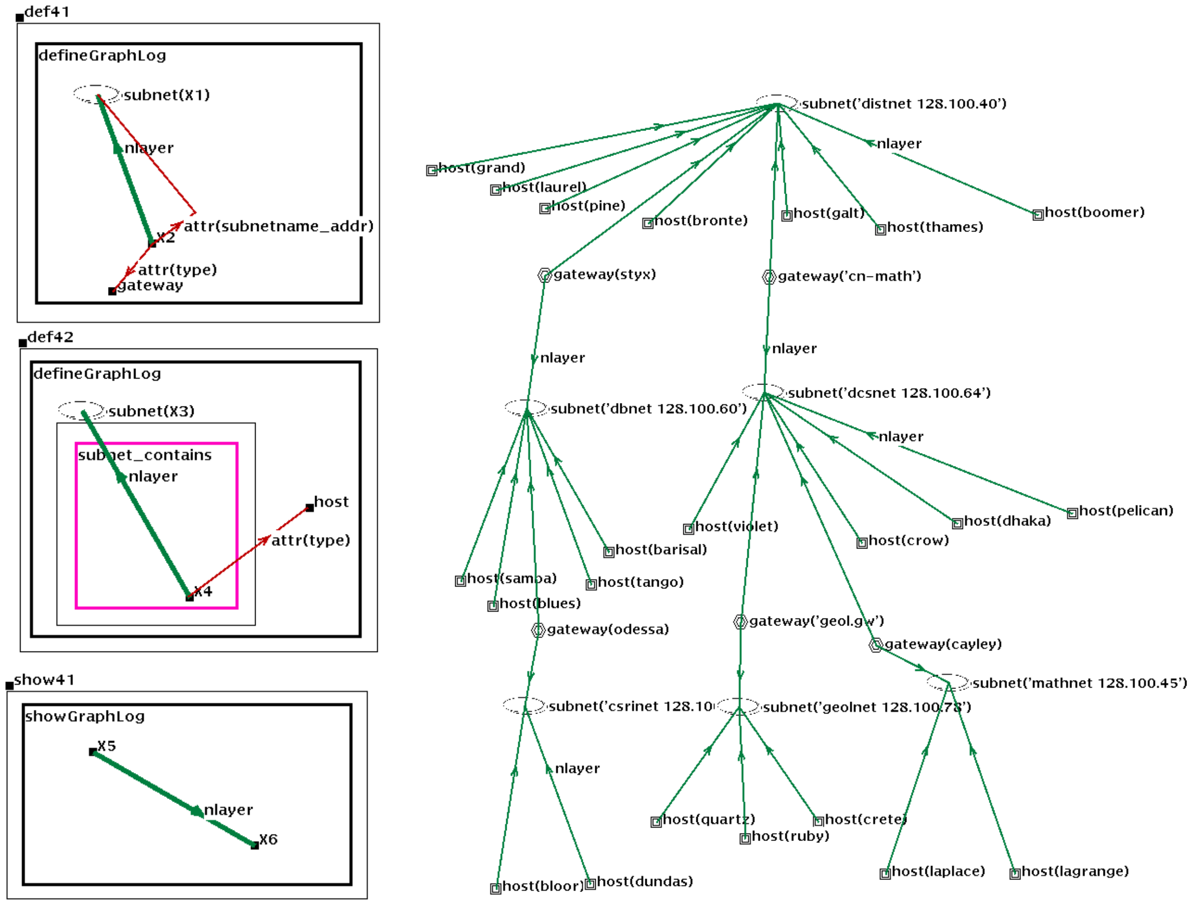

Figure 11: Defining and displaying the logical network layer map

To handle performance bottlenecks and faults, a network must be monitored continuously. The result of monitoring is reflected in *alerts*. Defining alerts and correlating them requires sophisticated and flexible alert definition facilities. By considering the network as a distributed database and the MIB as the schema of the database, alerts can be defined as declarative GraphLog queries.

Suppose that we monitor the utilization of a server called samba. For this purpose, we poll the tcpInSegs and tcpOutSegs TCP MIB objects of the server. These two variables store total number of TCP segments flowing in and out of the polled server, as calculated from a

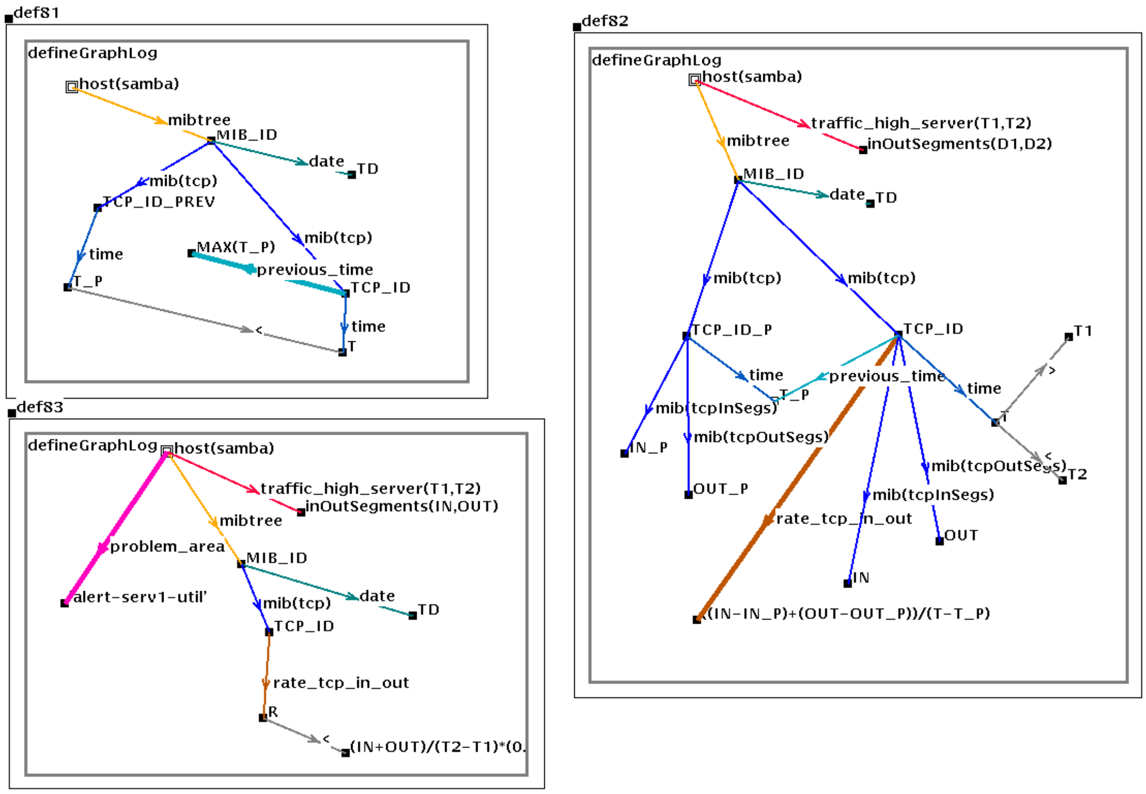

Figure 12: Defining an alert for possible problem symptoms

defined epoch. The values returned are kept in the database as history traces.

An alert can be generated if the server utilization falls below a certain threshold in a particular period of time, for example, during an expected high-traffic period. If the alert goes off, it indicates a possible problem symptom. The queries in Figure 12 define the alert alert-serv1-util. Query define81 finds out the time (pointed by the distinguished edge previous_time) when the previous poll of samba was taken. Query define82 computes the rate of tcpInSegs and tcpOutSegs traffic in the latest poll interval during high-traffic time. Query define83 checks whether the rate computed in the previous query falls 80% below the expected traffic kept on

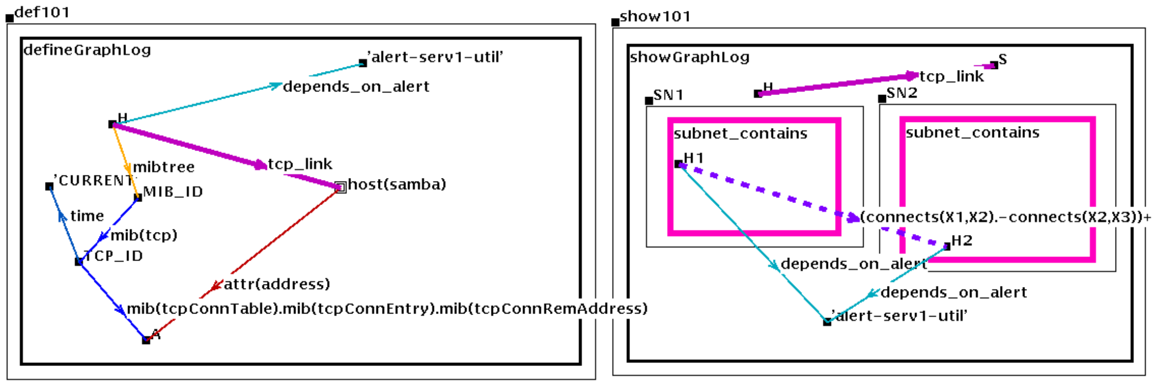

Figure 13: Defining and showing TCP links

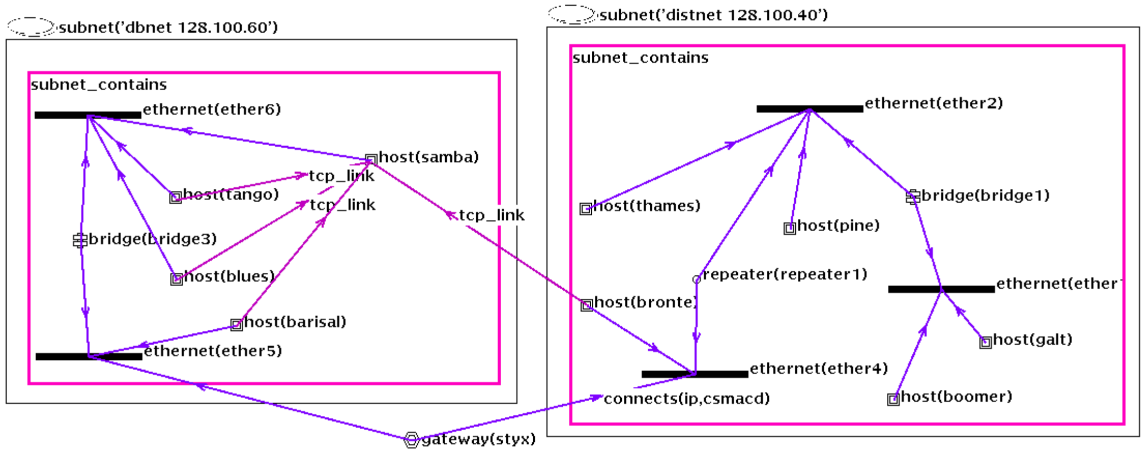

Figure 14: TCP links superimposed on the physical topology map

the **traffic_high_server** edge of **samba**. If the condition is satisfied, an edge called **problem-area** is created from the node labeled **samba** to the node labeled with the name of the alert.

Assume that the above alert has gone off, and the problem area is the server **samba**. The alert may fire for various reasons: congestion in gateways between the server and the clients or a hardware fault. Assume that we first hypothesized that the cause of the alert was congestion, but the investigation (see [7]) showed no congestion, although it found that the clients of samba are not generating the expected traffic to the server. Our second hypothesis is that the cause of the alert is a *hardware problem*. In this case, the map showing physical details like repeaters, bridges, datalink layer protocols, and so on is brought up. Before looking for the physical causes of the problem, we superimpose the currently active TCP links on the portion of the physical map that depends on the alert.

Query define101 of Figure 13 can be interpreted as follows: create an edge called tcp_link between the hosts H that depend on the alert alert-serv1-util and the server samba, if the IP address found in the tcpConnRemAddress of H's TCP connection table matches the address of samba (which is stored in the attribute attr(address) of the configuration database). Query show101 produces, as a result, the visualization in Figure 14 showing the portion of the physical topology map that depends on the alert, together with the previously defined tcp_link edges superimposed on the map.

The visualization of TCP links allows us to pinpoint the problem area at the portion of the network beyond ether4. The problem could be at the repeater repeater1 or at the ethernet segments ether2. This belief follows from noticing that no TCP connection originates at ether1 or ether2. The SNMP manager could not access the current TCP objects from that part of the network.

The example shows how, when isolating faults in a network, it is advantageous to view network maps at different abstract levels, while proceeding in a structured manner to pinpoint the problem area.

6 Conclusion

We have described the architecture and some aspects of the implementation of the Hy$^+$ visualization system and related software. We are encouraged by the ease with which we were able to apply these ideas to a wide variety of applications, including various aspects of software engineering, network management, and distributed and parallel debugging. Traditional data management tools have not been very successful in tackling these kinds of domains; we believe the unique combination of visualization and deductive database features of Hy$^+$ and GraphLog give us the edge here.

Work is proceeding to extend Hy$^+$ in several directions, including dynamic visualizations, temporal queries, three-dimensional diagrams, incremental query and layout, and new emphasis techniques for large visualizations. We hope this work is contributing to achieving David Harel's vision[17]:

> We are entirely convinced the future is "visual." We believe that in the next few years many more of our daily technical and scientific chores will be carried out visually, and graphical facilities will be far better than today's. The languages and approaches we shall be using in doing so will not be merely iconic in nature [...] but inherently diagrammatic in a conceptual way [...]. They will be designed to encourage visual modes of thinking when tackling systems of ever-increasing complexity, and will exploit and extend the use of our own wonderful visual system in many of our intellectual activities.

Acknowledgments

The University of Toronto project leading to the development of the Hy$^+$ system has had a close relationship with the IBM Canada Centre for Advanced Studies (CAS) since its early days. In 1989, we presented what was then called the G$^+$/GraphLog system at a technical vitality seminar at the IBM Toronto Lab, hosted by Shahram Javey. Arthur Ryman, a lab member working on an advanced software-development environment called 4Thought[31], saw the potential of GraphLog as a query and manipulation tool in such an environment, leading to a collaboration within one of the first projects of the fledgling CAS. Soon afterwards, we started interacting with Jacob Slonim, by then the head of CAS, on the possibilities of using the G$^+$/GraphLog approach for visualizing various aspects of distributed systems, and this led to the data visualization component of the CORDS project[34].

In addition to the support from IBM Canada, parts of the work described here were supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of the Province of Ontario, and the Institute for Robotics and Intelligent Systems, a network within the Networks of Centres of Excellence Programme of the Federal Government of Canada.

We also thank Sergio Faria, Yanni Jew, Christine Knight, Carlos Mendioroz, Toomas Toomsalu, and Annie Yeung for their contributions to the development of Hy$^+$.

About the Authors

Mariano P. Consens is a Research Assistant Professor in the Department of Computer Science at the University of Waterloo. His interests are in the theory and practice of database

systems and data visualization systems. Specific areas of interest are text databases and applied databases, for application domains such as software engineering and network management. His email address is mconsens@uwaterloo.ca.

Frank Ch. Eigler is a visiting student at the Centre for Advanced Studies and an undergraduate in the Department of Computer Engineering at the University of Toronto. His interests are in the areas of programming languages, deductive databases, operating systems, and computer networks. His email address is fche@db.toronto.edu.

Masum Z. Hasan is a Ph.D candidate in the Department of Computer Science at the University of Waterloo. His research interests are in the area of distributed systems, network management, and active temporal databases.

Alberto O. Mendelzon is a Professor in the Department of Computer Science at the University of Toronto. His interests include the theory, practice and applications of databases and knowledge bases, and in particular the data visualization and data manipulation interfaces for them. His email address is mendel@db.toronto.edu.

Emanuel G. Noik is a PhD candidate in the Department of Computer Science at the University of Toronto. His interests include the design and implementation of user interfaces, human-computer interaction, relational data visualization, graph layout, 3-D user interfaces and virtual reality, and hypermedia systems.

Arthur G. Ryman is a Senior Development Analyst in the Application Development Technology Centre (ADTC) of the IBM Toronto Software Solutions Laboratory. He is currently leading an advanced development project that is incorporating the GraphLog visual query language into 4Thought, an interactive graphical tool applicable to software engineering and database prototyping.

Dimitra Vista is a PhD candidate in the Department of Computer Science at the University of Toronto. Her interests are in the area of query processing, query optimization and incremental query evaluation, especially as they apply to visual query languages and database visualization. Her email address is vista@db.toronto.edu.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[2] C. Batini, T. Catarci, M.F. Costabile, and S. Levialdi. Visual Query Systems. Technical report, Universita degli Studi di Roma La Sapienza, March 1991.

[3] M. P. Consens and A. O. Mendelzon. Low Complexity Aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):379–394, 1993.

[4] M.P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, January 1989.

[5] M.P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns.* PhD thesis, Department of Computer Science, University of Toronto, February 1994.

[6] M.P. Consens, I. Cruz, and A.O. Mendelzon. Visualizing Queries and Querying Visualizations. In *ACM SIGMOD Record*, pages 39–46, 1992.

[7] M.P. Consens and M.Z. Hasan. Supporting Network Management through Declaratively Specified Data Visualizations. In H.G. Hegering and Y. Yemini, editors, *Proceedings of*

the IEEE/IFIP Third International Symposium on Integrated Network Management, III, pages 725–738. Elsevier North Holland, April 1993.

[8] M.P. Consens, M.Z. Hasan, and A.O. Mendelzon. Debugging Distributed and Parallel Programs by Visualizing and Querying Event Traces. In A.O. Mendelzon, editor, *Declarative Database Visualization: Recent Papers from the Hy+/GraphLog Project*. CSRI, University of Toronto, June 1993. Technical Report CSRI-285.

[9] M.P. Consens and A.O. Mendelzon. GraphLog: a Visual Formalism for Real Life Recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[10] M.P. Consens, A.O. Mendelzon, and A.G. Ryman. Visualizing and Querying Software Structures. In *14th. Intl. Conference on Software Engineering*, pages 138–156, 1992.

[11] M.P. Consens, A.O. Mendelzon, and D. Vista. Deductive Database Support for Data Visualization. In *Proceedings of the 4th International Conference on Extending Database Technology*, pages 45–58, 1994.

[12] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1990.

[13] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the Mental Map of a Diagram. Technical Report IIAS-RR-91-16E, Fujitsu Laboratories, August 1991.

[14] F.Ch. Eigler. GXF: A Graph Exchange Format. Technical report, CORDS, Computer Systems Research Institute, University of Toronto, January 1993.

[15] T. Fruchterman and E. Reingold. Graph Drawing By Force-Directed Placement. Technical Report UIUCDCS-R-90-1609, Dept. of Comp. Sci., U. of Illinois at Urbana-Champaign, June 1990.

[16] G. Gonnet. PAT 3.1: An efficient text searching system. Technical report, UW Centre for the New OED, University of Waterloo, 1987.

[17] D. Harel. On Visual Formalisms. *Communication of ACM*, 31(5):514–530, 1988.

[18] T.R. Henry and S.E. Hudson. Interactive Graph Layout. In *ACM UIST '91*, pages 55–64. ACM, 1991.

[19] T. Kamada and S. Kawai. A General Framework for Visualizing Abstract Objects and Relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[20] K. McCloghrie and M. T. Rose. Management Information Base for Network Management of TCP/IP based Internets - MIB-II. RFC 1213, Hughes LAN Systems, Performance Systems International, March, 1991.

[21] B.H. McCormick, T.A. DeFanti, and M.D. Brown. Visualization in Scientific Computing. *SIGGRAPH Computer Graphics*, 21(6), November 1987. Entire Issue.

[22] E.B. Messinger, L.A. Rowe, and R.R. Henry. A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):1–12, 1991.

[23] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[24] S.A. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases*. Computer Science Press, 1988.

[25] E.G. Noik. Exploring Large Hyperdocuments: Fisheye Views of Nested Networks. In *Hypertext '93: ACM Conf. on Hypertext*, pages 192–205, Seattle, WA, November 1993.

[26] E.G. Noik. Graphite: A Suite of Hygraph Visualization Utilities. In A.O. Mendelzon, editor, *Declarative Database Visualization: Recent Papers from the Hy+/GraphLog Project*, pages 108–126. Technical report CSRI-285, University of Toronto, July 1993.

[27] E.G. Noik. Layout-independent Fisheye Views of Nested Graphs. In *VL '93: IEEE Symposium on Visual Languages*, pages 336–341, Bergen, Norway, August 1993.

[28] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of International Conference on Very Large Databases*, 1992.

[29] D. Raymond. Flexible Text Display with Lector. *IEEE Computer*, 28(8):49–60, 1992.

[30] E. Reingold and J. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.

[31] A.G. Ryman. Foundations of 4Thought. In J. Botsford, A.G. Ryman, J. Slonim, and D. Taylor, editors, *Proceedings of the 1992 CAS Conference, Volume I*, pages 133–155, Toronto, Ontario, November 1992.

[32] A.G. Ryman. Constructing software design theories and models. In David Alex Lamb and Sandra Crocker, editors, *Proc. of the Workshop on Studies of Software Design*, May 1993.

[33] A.G. Ryman. Illuminating software specifications. In Ann Gawman, editor, *Proceedings of the 1993 CAS Conference*. IBM Canada Centre for Advanced Studies, October 1993.

[34] J. Slonim, M. Bauer, P. Finnigan, P. Larson, A. Mendelzon, R. McBride, T. Teorey, Y. Yemini, and S. Yemini. Towards a New Distributed Programming Environment (CORDS). In *Proceedings of the 1991 CAS Conference*, Oct. 1991.

[35] D. Vista and P. Wood. Efficient Visual Queries for Deductive Databases. In *Proceedings of the Workshop on Programming with Logic Databases*, 1993.

[36] C. Wetherell and A. Shannon. Tidy Drawings of Trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, 1979.

[37] A. Yeung. Text Searching in the Hy$^+$ Visualization System. Master's thesis, Department of Computer Science, University of Toronto, October 1993.