Looking at the Relations Among Software Objects

Mariano P. Consens* Alberto O. Mendelzon† Arthur G. Ryman‡

November 10, 1994

1 Introduction

What kinds of tools should we build to give software engineers better ways of designing, maintaining, and generally understanding large software systems? We claim that software engineers need (among other things) database facilities that support query, exploration, and visualization of software system structure and behaviour. Furthermore, we believe these facilities should be visual, they should be based on formalisms with precise semantics, and they should concentrate on relationships among software objects not on the objects themselves.

In recent years, much attention has been paid to the virtues of object-oriented databases to support software engineering and other new applications of database technology (see *e.g.* [LH91]). The object paradigm emphasizes *encapsulation*: hiding implementation details behind inviolable interfaces. However, object-oriented systems tend to de-emphasize *relationships* among objects. Relationships among objects are inherently non-local and non-encapsulated, and therefore they are a strong source of complexity in system design. This high degree of interconnectivity happens both in the large, as interrelations among components of large systems, and in the small, where complex data structures are needed to describe object relationships.

Therefore, instead of looking to the OO world for inspiration, we base our approach on the relational model, where relationships are the central concept, and where a universally understood formal model already exists, unlike the chaotic situation in the OO world. Our approach will be to represent software structures with mathematical relations or predicates and to use a formal language to define new predicates and to query the collection of predicates of interest. We impose

*Department of Computer Science, Univeristy of Waterloo, Waterloo, Ontario, Canada N2L 3G1, mconsens@uwaterloo.ca. (Work done while the author was at the Computer Systems Research Institute,University of Toronto.)

†Computer Systems Research Institute, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada M5S 1A1, mendel@db.toronto.edu.

‡IBM Canada Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7, ryman@vnet.ibm.com.

additional structure on these predicates by representing them as hypergraph-like structures called *hygraphs* [Con94]. Hygraphs are similar, but not identical, to Harel's higraphs [Har88]. Objects are still present in hygraphs, but they are relegated to the nodes, while the edges and hyperedges are used to represent relationships among objects.

The notion that relations are a useful construct to think about how objects in an OO system relate to each other is not new; for example, it was explicitly pointed out by Rumbaugh in 1987 [Rum87]. However, it does not seem to have been followed up very much by developers of software-engineering environments.

Expressive Power

The use of a model that is essentially first-order logic and a formal specification language that operates on it is consistent with set-theory based formalisms such as Z [Spi92]. However, the approach to the design of formal specification languages taken by database theory is quite different from the one followed by the formal methods community. In the latter, very powerful languages are used, without much concern for the computational tractability of the specifications written in them. In database theory, the emphasis is on the trade-offs between expressive power and efficiency of evaluation; computational properties are considered quite important, since specifications ("queries") are to be evaluated over potentially large databases. In fact, another advantage of following the database paradigm is this merging of the specification language that is used for defining the relationships of interest with the query and data manipulation language that is used for examining particular instances of these relationships in a given system.

Thus database theory offers a range of choices for the expressive power of the specification/query language. At one extreme are relational algebra and calculus (or their real-world implementation, SQL). However, the expressive power of such languages is very restricted; for example, they cannot express queries that involve transitive closures or path traversals, which are very common in software engineering applications. At the other extreme, we could use a powerful formalism such as set theory with fixpoint or second order logic features; but this method would make it difficult or impossible to use the formal language efficiently as a query language on industrial-scale databases. We choose an intermediate alternative: the *GraphLog* query language [Con89], developed at the University of Toronto as part of the Hy+ project. GraphLog can express all queries based on *linear recursion* and has aggregation and summarization operators that allow computations such as shortest path or critical path [CM93b]. The set of queries expressible in GraphLog strictly contains the set of those expressible in SQL, while the computational complexity of evaluating GraphLog queries remains provably low. It is shown in [Con89, CM90] that all GraphLog queries are evaluable in time polynomial in the size of the database; in fact, the queries expressible in GraphLog are exactly those which are evaluable in non-deterministic logarithmic space in the size of the database.

Visualization

The main purpose of a formal software specification is to explain clearly what a program is supposed to compute, not how the program is to perform the computation. The traditional approach to achieving clarity is to phrase the specification in high-level mathematical concepts, such as sets and relations. We believe additional clarity can be achieved by using formal diagrammatic techniques.

The choice of GraphLog is not only due to its expressive power: it is a *visual* language in which the user specifies queries as patterns that are to be found in the database. The user manipulates both the query specification and result with a graphical editor. In the same way, derived predicates are specified by drawing the patterns that define them. This is another way in which we diverge from most formal approaches to software engineering. Instead of concentrating on formal textual specifications that usually require informal diagrams to be communicated and understood, we make the diagrams formal and complement them, if necessary, with both formal and informal text.

GraphLog satisfies two needs at the same time: the need for queries and specifications that require path traversal operations not expressible in relational languages or in first-order logic, and the need for diagrams with well-defined semantics that go beyond illustrations. These formal diagrams serve a fundamental purpose: they communicate a wealth of information to the software engineer. After an answer to a specific query is obtained, the human information processing activity starts. Let's say that the query returned a specific set of design rule violations. The software engineer must assess the options available and consider a multitude of factors before re-designing portions of the application to (in many cases only partially) comply with the design guidelines expressed in the query. This sort of decisions currently and for the foreseeable future will rely on human judgment, yet we should strive to provide the best computer-based support for the software development task. It is for this information-intensive problem solving activity that we are in desperate need of visualization tools that support exploration of the problem space.

The Hy+ system [CM93a] is an example of a prototype visualization system that embeds GraphLog; another one is 4Thought [Rym92], under development at the IBM Software Solutions Toronto Laboratory.

Our Experience

As part of an ongoing collaboration between software developers at the IBM Toronto Laboratory and database researchers at the University of Toronto, we have used GraphLog to define and manipulate relationships among software objects in several different contexts, including software specification, design rule checking, software metrics, performance tuning, and reverse engineering. Some of these experiences are reported in [CMR92, Men93]. In the rest of this position paper, we first introduce GraphLog by means of an example that shows its use as a programming-in-the-small specification tool, a kind of visual logic programming. The second example shows GraphLog used to express a well-known design rule for OO programs, the *Law of Demeter* [LH89]. We then illustrate how visualizations of realistic datasets can be explored using the Hy+ system.

2 A Software Specification Example: Nearest Common Ancestor

Consider an application that maintains a set of people and the names of their parents [Rym93b]. The problem is to build a program that determines all of the ways in which any two given people are related. To compute *all* of the ways in which two given people are related, the key step is to determine their set of nearest common ancestors. In a software-engineering textbook, Lamb [Lam88] compares a procedural and a declarative specification for the nearest-common-ancestor problem, arguing that the declarative specification is more reliable. We agree with this point of view, but take it further by claiming that a *visual* declarative specification can be more reliable and more illuminating than a textual one.

First, we visualize the data by drawing a directed graph in which persons are the nodes and parent relationships are represented by the edges. For example, Figure 1 describes a part of the British royal family. Notice that, not only is Charles the son of Philip, but they are also second cousins, twice removed, since they have Christian IX of Denmark as a nearest common ancestor.

Having visualized the data, we use our visual formalism to specify the relationship of interest. The GraphLog definition of nearest common ancestor is illustrated in Figure 2.

GraphLog definitions specify patterns. The nodes can contain constants and variables that represent entities. (We use the Prolog convention in which upper case identifiers denote variables and lower case identifiers denote constants or predicate names). The arcs are labelled by *regular expressions* on the alphabet of relationship names. For example, the arc labelled $parent^*$ from $P1$ to $A1$ matches the snapshot if there is a path from $P1$ to $A1$ that consists of zero or more $parent$ arcs, with the zero case corresponding to node equality. Regular expressions that define paths of unbounded length are indicated by a dashed arc.

The cross-out symbol on two of the arcs indicates the negation of the relationships. For example, the top pattern only matches the snapshot if $P2$ does not descend from $A1$.

The two heavy arcs define the new relationship $nearest\_common\_ancestor$. The arc label $nearest\_common\_ancestor(A)$ means that, for every entity A, $common\_ancestor(A)$ is a binary relationship set. There are two ways in which a person A can be the nearest common ancestor of persons $P1$ and $P2$, corresponding to the two patterns separated by a horizontal bar. The defined relationship holds if either pattern matches.

The meaning of each pattern in the diagram is that the defined relationship $nearest\_common\_ancestor(A)$ holds for every substitution of variables by entity names such that the resulting graph is a subgraph of the data snapshot. For example, Figure 3 shows a

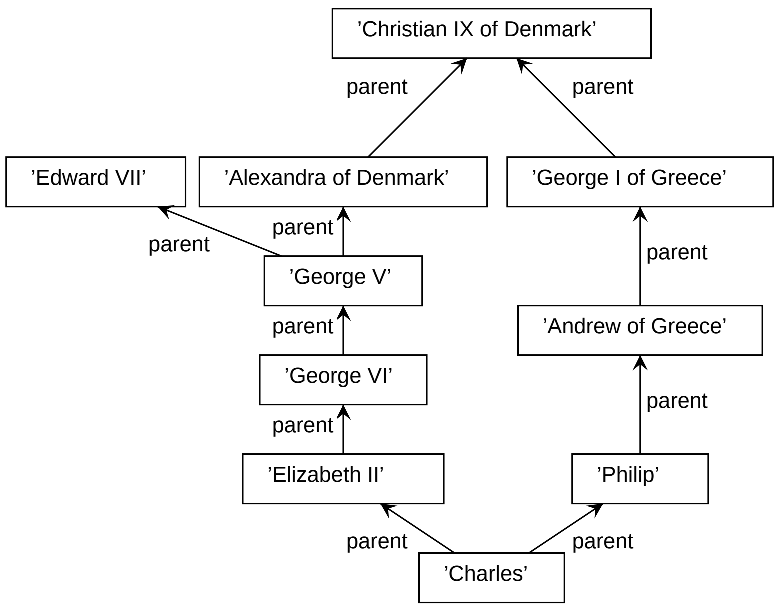

Figure 1: Snapshot of Part of the British Royal Family

match for the top pattern with the substitutions of Christian IX of Denmark for A, Alexandra of Denmark for $A1$, George I of Greece for $A2$, Charles for $P1$, and Philip for $P2$.

Our experience has been that many people find it much easier to reason about the meaning of GraphLog patterns than about the corresponding logical formulas. A good example of this ease of understanding occurred the first time these techniques were presented in a seminar. Instead of the correct definition that appears in Figure 2, the incorrect definition shown in 4 was used. After studying the definition for a few minutes, one of the members of the audience remarked that it did not match the example. According to the definition in Figure 4, Christian would not be a nearest common ancestor of Charles and Philip since Charles does descend from George I of Greece, thereby contradicting the crossed-out arc labelled $parent^*$ from $P1$ to $A2$. These types of errors are generally harder to detect in textual specifications.

3 A Design Rule Example: The Law of Demeter

The Law of Demeter [LH89] proposes restrictions on how member functions call each other. These restrictions are intended to reduce coupling between classes and improve the

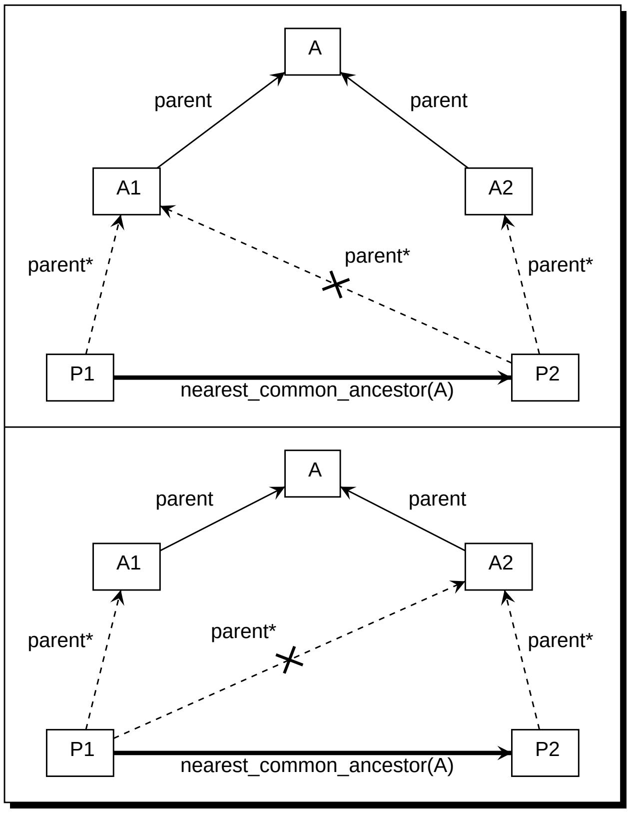

Figure 2: GraphLog Definition of Nearest Common Ancestor

understandability of member functions. The law is language-independent, but in the following discussion we will use C++ terminology and examples.

Informally, the law states that a member function should only call other member functions that belong to the classes of its member variables, its arguments, the objects it constructs, and the global variables it accesses.

We use the example program given in [LH89], shown in Figure 5. The IBM AIX[1] XL C++ compiler [IBM92] generates static analysis information, encoded as Prolog facts. Using this compiler, we can generate from the example program a graph that can then be queried with GraphLog [Rym93a]. Javey *et al.* do something similar with Prolog, and give Prolog definitions of the Law of Demeter in [JMN$^+$92].

Figure 6 shows a GraphLog query and the resulting snapshot of the database. This figure

[1]IBM and AIX are trademarks of International Business Machines Corporation

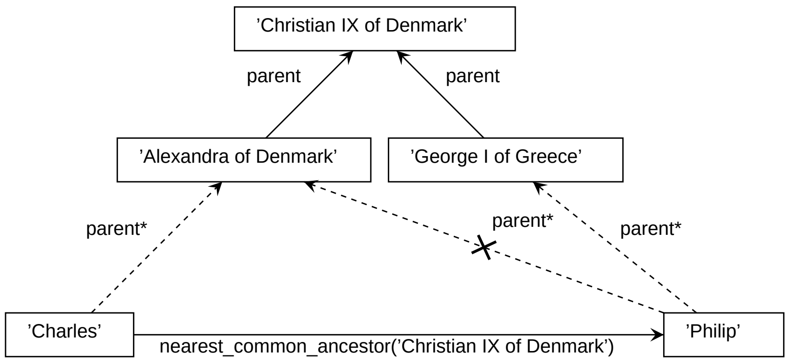

Figure 3: Christian is a Nearest Common Ancestor of Charles and Philip

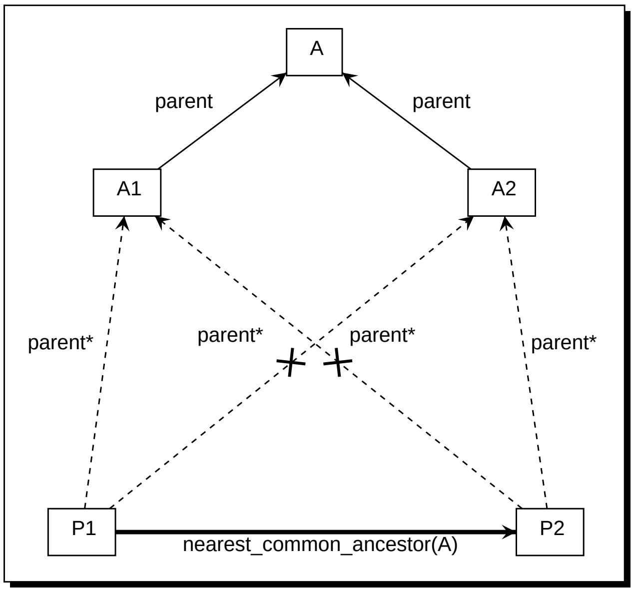

Figure 4: Incorrect GraphLog Definition of Nearest Common Ancestor

```
typedef int boolean;

class Book{int id;};

class BookSec{public: boolean search(Book *book){return 0;}};

class Documents{public: boolean search(Book *book){return 0;}};

class MicroficheFiles{public: boolean search(Book *book){return 0;}};

class Archive{public:
    MicroficheFiles *arch_microfiche;
    Documents *arch_docs;
    boolean search_good_style(Book *book){
        return (arch_microfiche->search(book) ||
            arch_docs->search(book));
    }
};

class ReferenceSec{public:
    Archive *archive;
    BookSec *ref_book_sec;
    boolean search_bad_style(Book *book){
        return (ref_book_sec->search(book) ||
        /**/    archive->arch_microfiche->search(book) ||
        /**/    archive->arch_docs->search(book));
    }
    boolean search_good_style(Book *book){
        return (ref_book_sec->search(book) ||
                archive->search_good_style(book));
    }
};
```

Figure 5: Listing of Example C++ Program

introduces *hygraphs*. An edge in a graph relates one node to another node. A *blob* in a hygraph relates a node to a set of other nodes. Blobs are visualized as rectangular boxes attached to the box for the node that owns the blob. For example, in Figure 6 the node for class `Archive` has two blobs attached to it, $member\_function$ and $member\_variable$. Inside the first blob appear all the nodes connected to `Archive` by this blob, that is, all member functions of the class `Archive`. Inside the second blob appear all member variables of the class `Archive`. Although this picture only contains blobs, in general, edges and blobs can intermix in arbitrary ways. The query, shown in the shaded box at the top, is a filter that only passes the $member\_function$ and $member\_variable$ relations, and displays them as a hygraph.

We will now state the definitions used in the Law of Demeter, and then give the GraphLog translations of some of them. Using GraphLog queries, we can easily change from edge to blob representations of a relationship and back. For convenience, we give the Law of Demeter definitions in terms of edges instead of using blobs as we did in Figure 6.

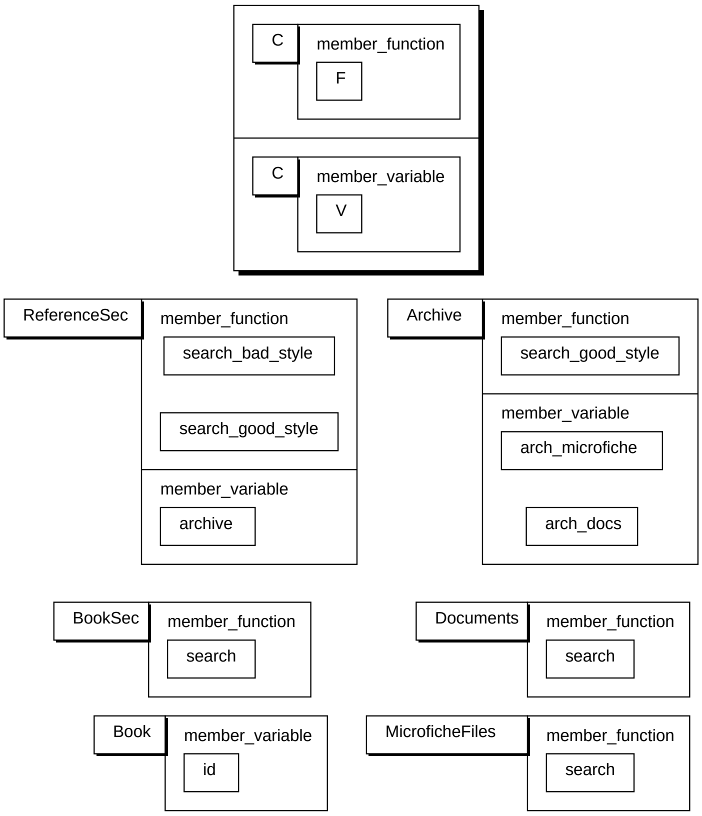

Figure 6: GraphLog Query and Snapshot of Member Functions and Variables

Law of Demeter Definitions

A member function M is a *client* of a class C if M calls a member function F of C or accesses a member variable V of C. A class C is a *supplier* of a member function M if M is a client of C. The GraphLog definition of $client$ and $supplier$ is illustrated in Figure 7.

A class $C1$ is an *argument class* of a member function M if M takes a variable V of type C or M is a member function of C, and $C1$ is C or a direct or indirect base class of C. (Recall that M takes an implicit argument `this` of type C.) The GraphLog definition of $argument\_class$ is illustrated in Figure 8.

A class $C1$ is an *instance variable class* of a class $C2$ if $C2$ has a member variable V of type C, and $C1$ is C or a direct or indirect base class of C. We omit the GraphLog definition of instance variable class to save space.

A class $C1$ is an *acquaintance* of a member function M of class $C2$ if $C1$ is a supplier of

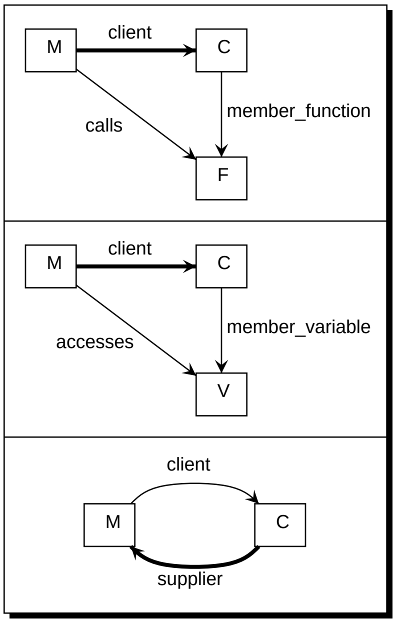

Figure 7: GraphLog Definition of Client and Supplier

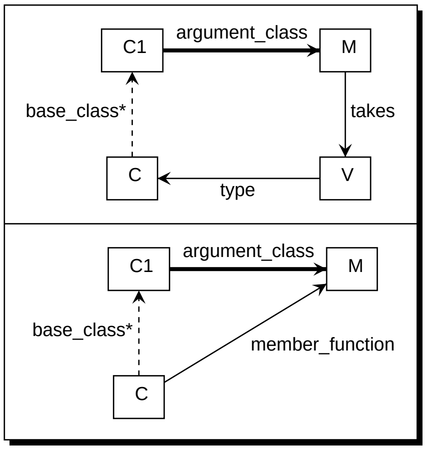

Figure 8: GraphLog Definition of Argument Class

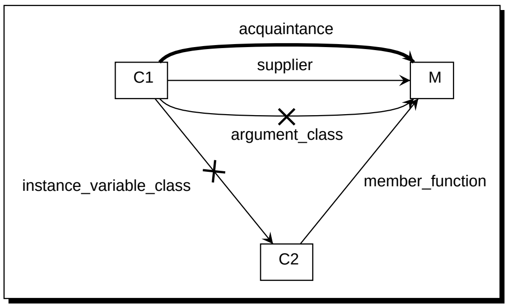

Figure 9: GraphLog Definition of Acquaintance

M and is neither an argument class of M nor an instance variable class of $C2$. The GraphLog definition of $acquaintance$ is illustrated in Figure 9.

A class C is a *preferred acquaintance* of a member function M if C is an acquaintance of M, and M calls a constructor F of C or accesses a global variable V of type C.

A class B is a *preferred supplier* of a member function M of class C if B is a supplier of M, and B is an instance variable class of C or an argument class of M or a preferred acquaintance of M. The GraphLog definitions of $preferred\_acquaintance$ and $preferred\_supplier$, are similar to the above and will be omitted because of lack of space.

The strict version of the class form of the Law of Demeter states that all member functions should have only preferred suppliers.

Figure 10 illustrates a GraphLog query that displays the $acquaintance$ relation in the context of the $calls$ and $member\_function$ relations, and the resulting snapshot for the example program. Note that `search_bad_style` in `ReferenceSec` has two acquaintance classes (`MicroficheFiles` and `Documents`, as a consequence of its directly calling `search` in `MicroficheFiles` and `Documents`), while `search_good_style` has none.

A software engineer could use the GraphLog specification of the Law of Demeter, not just to search for violations, but to display them in the appropriate context. More importantly, a large class of design rules and constraints could be expressed and verified without having to hard-wire an arbitrary set of them into the environment.

As we mentioned in the introduction, the ability to visualize information such as design rules violations in context is of paramount importance. Another important aspect of the approach described here is *scalability*. The same GraphLog patterns that have been described using a small textbook example are equally applicable in a realistic scenario. Figures 11 and 12 taken from the Hy+ System, illustrate this point.

Each rectangle in Figure 11 represents a class in a subtree (laid out horizontally) of the

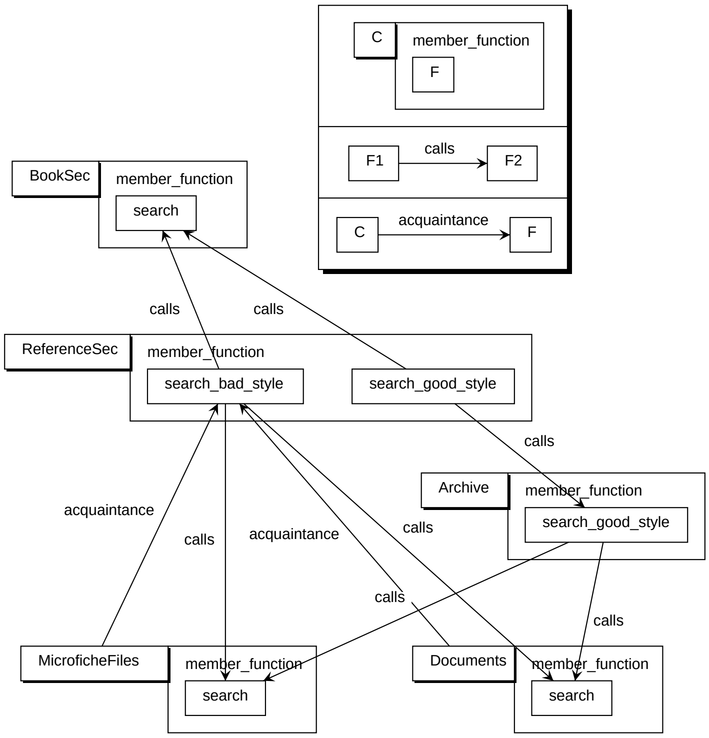

Figure 10: GraphLog Query and Snapshot of Member Functions, Calls, and Acquaintance

class hierarchy of NIH, a public domain C++ library with 35,000 lines of source code [GOP90]. Within each class rectangle there are two blobs, corresponding to the $member\_functions$ and $member\_variables$ of the class. In addition, $calls$ between functions and $references$ from functions to variables are represented as edges. The diagram is a compact representation of the information contained in several thousand lines of source code. Presenting such a visualization to a software engineer provides her with an overview of a significant portion of the library. She can then close into a particular aspect she is interested in, while preserving the context information that we referred to earlier. Figure 12 displays a zoomed-in view of the third class rectangle from the top of the large hygraph shown before (with text labels turned on). The focus of attention are the heavily called functions at the bottom of the $member\_functions$ blob in `class('ArrayOb')`.

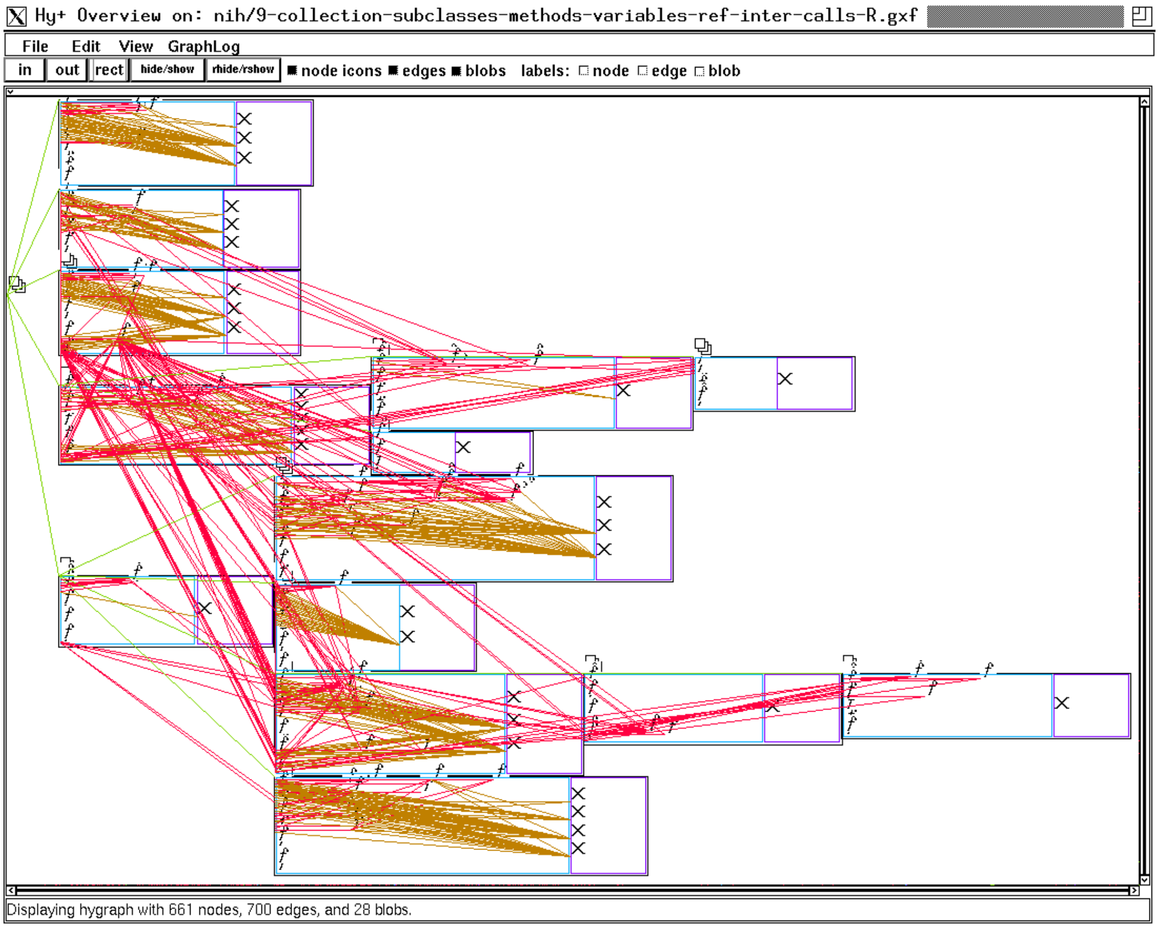

Figure 11: Browsing a large hygraph.

4 Conclusions and Open Problems

We have shown two examples, on toy databases, of how GraphLog can be used to express very different kinds of visual patterns about relationships among software objects, and to filter out some of the resulting objects and relations for visualization. We have experimented with much more substantial databases. For example, we have used a 13,000 fact database describing the NIH library used for the illustrations of the last section and a database of similar size describing the ET++ class library [WGM88], as well as a database describing the *Imagedit* IBM software product [CMR92]. On industrial-scale databases, techniques for producing helpful visualizations of the results of queries are also important; we do not have space to elaborate that aspect here, but see some of the papers in [Men93].

Our experience using GraphLog within the Hy+ system at the University of Toronto and

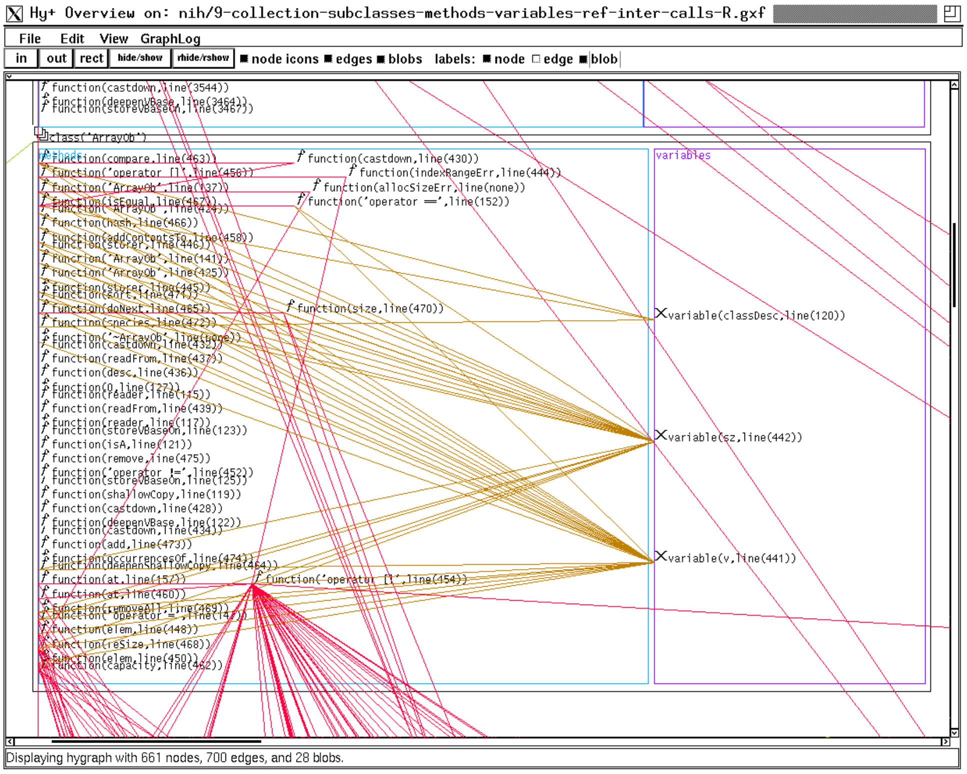

Figure 12: Taking a closer look at the relations displayed.

the 4Thought system at IBM Canada has convinced us of the importance of the following open problems, which we divide into database problems and software engineering problems, although several of them require the interaction of both disciplines.

Database problems

Query optimization. It is hardly news to the database community that this is an essential problem. However, visual query languages pose some new optimization problems. For example, GraphLog queries behave as *filters* on the database, mapping databases to (smaller) databases; conventional queries map databases to single relations. Little work has been done on optimizing queries that produce multiple relations as answers.

Transactions and updates. A language such as GraphLog can form the basis for expressing not only queries but also updates. For example, Lieberherr shows that any program can be transformed into one that satisfies the Law of Demeter, and defines two transformations, *lifting* and *pushing*, for accomplishing this. These transformations could also be expressed as GraphLog transactions, and would allow the designer to graphically edit the design database by direct manipulation of the snapshots. In this mode of working, changes to the base relations would automatically cause corresponding changes to the derived relations, giving the designer immediate feedback and allowing the assessment of different alternatives.

Incremental query answering. To efficiently keep up to date visualizations in the presence of changing data, such as the example of program transformations mentioned above, we must avoid recomputing queries from scratch. The problem has been examined in databases in the context of view materialization [BLT90, GMS93], but little is known for the case where views are defined by languages more powerful than relational algebra.

Dynamic data. Suppose, instead of querying the structure of a software system, we are visualizing run-time behaviour in real time. In addition to the need for incremental query evaluation mentioned above, we need to extend our query formalism to allow easy expression of temporal patterns, for example, "highlight the procedures that account for over 50% of execution time over the past 2 minutes."

Tool integration. Query and visualization facilities such as we have described should be implemented as components within a framework such as SoftBench [Cag90] or OMG's CORBA [Gro91], providing an architecture that allows them to interoperate with other software tools. For example, a debugger could request visualization services to display and animate the call graph of a program, or a reverse engineering tool could issue queries to compute software quality metrics. In the other direction, the visualization component could ask a software configuration manager to produce a current release, pass it to the query component for filtering, and to the layout component for layout before actually displaying it on the screen.

Software Engineering Problems

Formal visual notations. Research in formal methods for software engineering has concentrated on textual notations. Many commercial software development environments provide tools for manipulating diagrams, but these are informally defined. We believe the use of *formal* diagrams as a complement to other notations can retain the advantages of clear semantics while improving readability.

Expressive power balance. Software engineering research could borrow a page from database theory by devising formalisms whose expressive power is balanced with their computational complexity, leading to specifications that can be viewed as query languages and executed efficiently on industrial-scale software bases.

User studies. The evidence we have collected so far of the advantages of a notation like GraphLog, while persuasive, is anecdotal. Controlled studies would be useful to verify and quantify these advantages.

References

[BLT90] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Sys.*, 14(3):22–31, 1990.

[Cag90] M.R. Cagan. The HP Softbench environment: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, pages 36–47, June 1990.

[CM90] M.P. Consens and A.O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proc. 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CM93a] M.P. Consens and A.O. Mendelzon. Hy+: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 511–516, Washington, D.C., 1993.

[CM93b] M.P. Consens and A.O. Mendelzon. Low complexity aggregation in Graphlog and Datalog. *Theoretical Computer Science*, 116(1):95–116, 1993.

[CMR92] M.P. Consens, A.O. Mendelzon, and A.G. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, Melbourne, Australia, May 1992.

[Con89] M.P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, University of Toronto, Toronto, Ontario, January 1989.

[Con94] Mariano P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns*. PhD thesis, Department of Computer Science, University of Toronto, February 1994.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 157–166, Washington, D.C., 1993.

[GOP90] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, Chichester, 1990.

[Gro91] Object Management Group. The common object request broker: Architecture and specification, December 1991. OMG Document Number 91.12.1.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[IBM92] IBM, North York, Ontario. *AIX XL C++ Compiler/6000 User's Guide Version 1*, first edition, April 1992.

[JMN+92] Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the XL C++ Browser. In John Botsford, Arthur Ryman, Jacob Slonim, and David Taylor, editors, *Proceedings of the 1992 CAS Conference, Volume I*, pages 369–379, Toronto, Ontario, November 1992.

[Lam88] David A. Lamb. *Software Engineering: Planning for Change*. Prentice-Hall, Englewood CLiffs, New Jersey, 1988.

[LH89] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[LH91] Lung-Chun Liu and Ellis Horowitz. Object database support for CASE. In Rajiv Gupta and Ellis Horowitz, editors, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, pages 261–282. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Men93] A.O. Mendelzon, editor. *Declarative Database Visualization: Recent Papers from the Hy+/GraphLog Project*. University of Toronto, June 1993. Technical Report CSRI-285.

[Rum87] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA '87 Proceedings*, pages 466–481. ACM Press, 1987.

[Rym92] Arthur Ryman. Foundations of 4Thought. In John Botsford, Arthur Ryman, Jacob Slonim, and David Taylor, editors, *Proceedings of the 1992 CAS Conference, Volume I*, pages 133–155, Toronto, Ontario, November 1992.

[Rym93a] Arthur Ryman. Constructing software design theories and models. In *Proc. of the Workshop on Studies of Software Design*, pages 124–138, Baltimore, Maryland, April 1993.

[Rym93b] Arthur Ryman. Illuminating software specifications. In *Proc. CASCON '93 Volume I Software Engineering*, pages 412–428, Toronto, Ontario, October 1993.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International (UK) Ltd., London, England, second edition, 1992.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ - an object oriented application framework in C++. In *OOPSLA '88 Proceedings*, pages 168–182, San Diego, November 1988.