Deductive Database Support for Data Visualization

Mariano Consens, Alberto O. Mendelzon, Dimitra Vista

{consens, mendel, vista}@db.toronto.edu
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Abstract

We argue that we can use deductive databases to support data visualization. In particular we show how we have used the deductive languages LDL and CORAL for the implementation of the visual query language GraphLog. We discuss in detail the translation function from GraphLog to each of LDL and CORAL, especially when aggregate functions are present. We also present an example of using GraphLog and its environment Hy$^+$ in order to support software design understanding and software design verification.

1. Introduction

The data used in new applications of database technology, such as CAD/CAM, hypertext, software development support, network management, etc., often has a graph-like structure that can be exploited to provide visual presentations of the data and visual query mechanisms. Many of the operations that arise naturally in such visual environments involve pattern matching and finding and traversing paths of various kinds. Such operations are beyond the power of first-order query languages such as relational algebra and SQL; they require additional expressive power, as provided by deductive databases such as LDL [12] or CORAL [14].

In this paper we show how deductive databases can help support the growing demand for visible and visualizable data and for queries and operations on such data. We do so in the context of the Hy$^+$ visualization system [6] and the database specification and query language GraphLog [4]. As a graphical formalism for visual manipulation of database visualizations, GraphLog is especially suitable for application areas where the data explores certain graphical structures, such as system design and software development [7], network monitoring [1], and hypertext [3]. The current version of the Hy$^+$ system, under development at the University of Toronto, implements GraphLog by mapping visual GraphLog queries to Horn clauses which are compiled and executed by one of several deductive back-ends, including Prolog, LDL and CORAL.

The rest of the paper is organized as follows. In Section 2, we show, using an example from the software design field, how Hy$^+$ and GraphLog are used to manipulate visual data in a visual way. Section 3 briefly introduces GraphLog. Sections 4 and 5 describe the translation from GraphLog to LDL and CORAL. Emphasis is placed on the implementation of aggregation functions. Section 6 discusses *filter* queries. Instead of defining new relationships in terms of existing ones, as conventional Horn clause queries do, filter queries request the display of all existing data objects and relationships that match a given pattern. We conclude in Section 7.

2. Data Visualization Example

In this section we present an example of using GraphLog and its environment Hy$^+$ for visualizing the structure of an OOT program. OOT is an object oriented programming language [10] and its programming environment contains a variety of tools. One of them, the Software Landscape [9] is a tool that displays a complete and concise graphical description (the Landscape) of the architecture of a system at various levels of granularity.

As an experiment, we imported into Hy$^+$ one Landscape describing the design of an OOT environment tool. In Hy$^+$ we were able not only to visualize the Landscape but also to formulate a series of GraphLog queries to further explore as well as verify its structure. The Software Landscape provides a static visualization of the architecture of a system. GraphLog queries permit a more dynamic approach to the visualization process where the user can obtain different visualizations of the same component through different queries.

Landscapes have a set of constructs including classes, modules, subsystems, libraries, etc., which are interrelated through relations. Among them are a) the relation `contain(X, Y)` that states that entity `X` contains sub-entity `Y`, b) the relation `export(X, Y)` that states that entity `X` allows access to its contained entity `Y`, b) the relation `import(X,Y)` that states that an entity `X` accesses external entity `Y`. The database of Figure 1 depicts the organization of these relations for the specific Landscape of the OOT program that we imported into Hy$^+$. The layout of this hygraph was generated by a *spring* layout algorithm[1]. For simplicity of presentation most of the labels are turned off. However, the user can still see the overall structure of the relations. Colors on the edges allow the user to differentiate among relations. Icons on the nodes let the user know the type of each node, e.g., **P** refers to procedures, **C** refers to constants (i.e., the tokens of the code), **X** to variables, **f** to functions, the icon with the three overlapping pages is for classes, and so on.

The visualization in Figure 1 is a *hygraph* (an extension of the notion of a graph inspired by Harel's higraphs [8]) and can be queried. Figure 2 contains two examples of queries. The first query is what in GraphLog is called a *filter query*. It describes a pattern to be matched in the hygraph of the database. As one can see, one of the edges in this pattern is thick, i.e., distinguished. This is a visual way of distinguishing edges which the user actually wants to see after the match is found. This particular *filter* query searches the database for all pairs of classes `C` and modules `M` that both import the same entity `X`, and for each such pair it displays only the `import` relation for the classes that are found. More details on *filter* queries are given in Section 6.

The other query is called a *define query*. A thick edge here has a different meaning. It represents a

[1] In this algorithm [13] nodes are treated as similarly charged point sources that repel one another; arcs, on the other hand, act as springs or elastics that pull connected nodes together, thus the name.

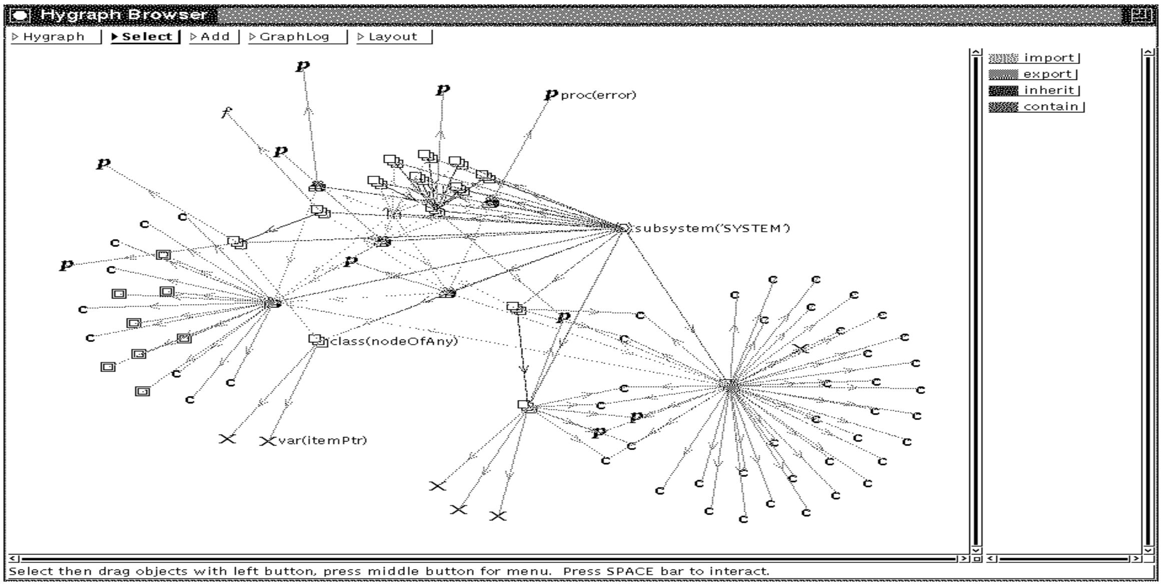
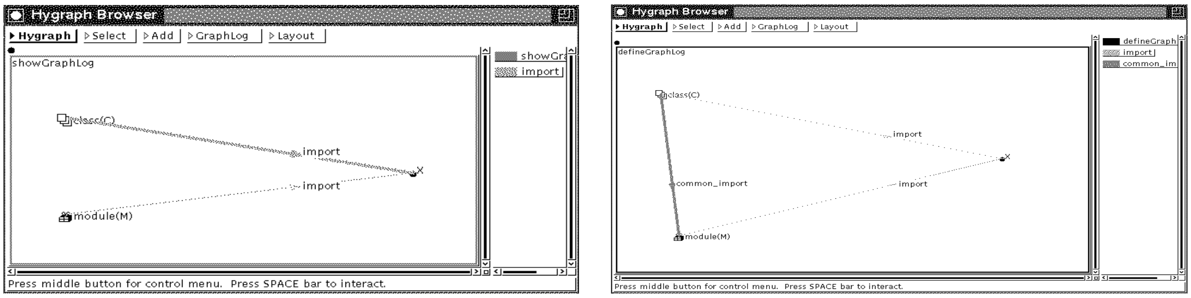


Figure 1: The database describing the software Landscape

relation that is defined every time the pattern consisting of all the other edges in the query is found in the database. The *define* query in Figure 2 searches for the same pairs of classes `C` and modules `M` and defines the new relation `common_import` between them.



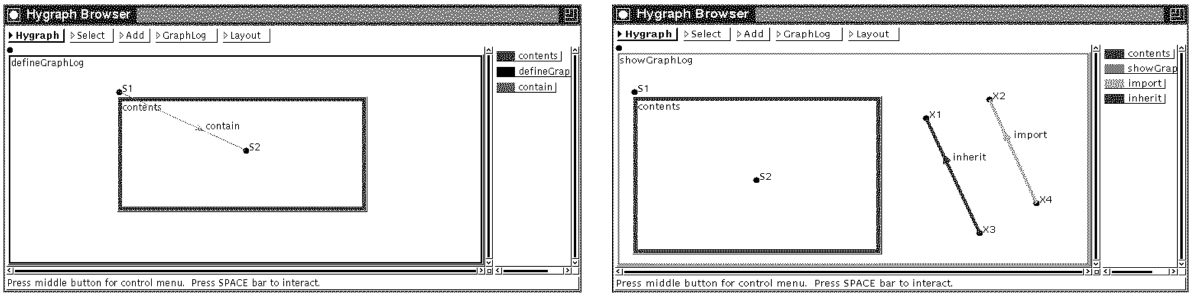
An example *filter* query

An example *define* query

Figure 2: Examples

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 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 3 demonstrates how we can change the representation of a relationship from edges to blobs. The blob `contents` clusters the sub-entities of an entity together: a node labeled `S2` is enclosed in the blob whose container node is labeled `S1`, whenever `contain(S1, S2)`. The second hygraph in Figure 3 is a *filter* query that produces a hygraph with `contents` blobs and `import` and `inherit` edges.

When Hy$^+$ executes the above *define* query against the database of Figure 1 and then the above



Definition of `contents` Filter of `contents`

Figure 3: Examples with blobs

filter query is executed against the result of the *define* query, the answer is what appears in Figure 4.

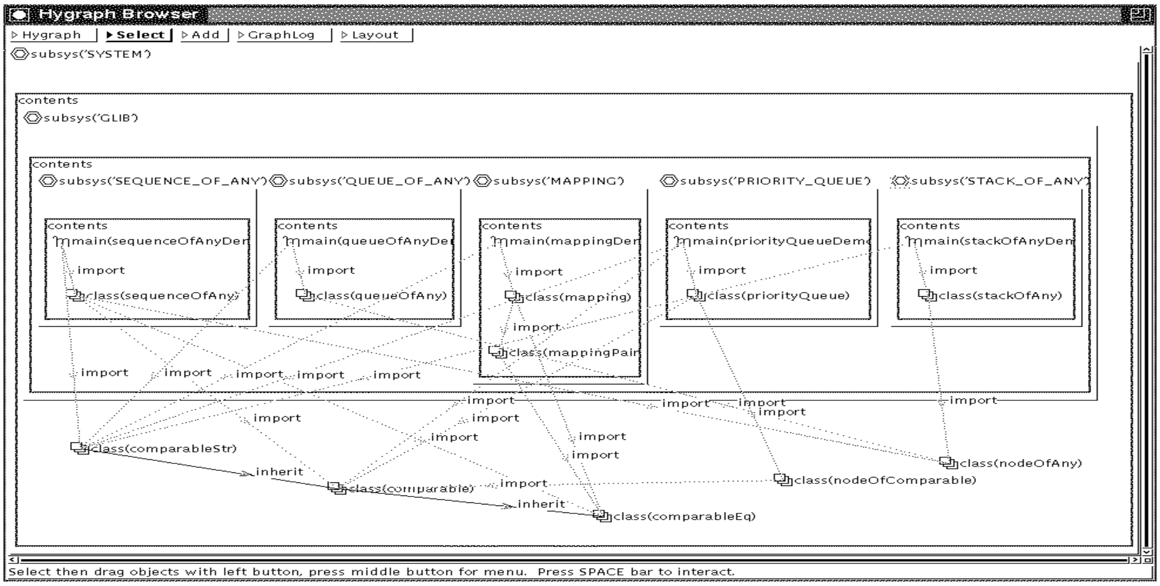


Figure 4: The result of filtering `contents`

Apart from design understanding, GraphLog queries can be used in design verification. The Software Landscape defines a set of rules that determines what constitutes a syntactically correct design. Without verification rules, designs are simply an arbitrary set of entities with an arbitrary set of relations (or in CASE environments nothing but pretty pictures).

We present two such rules. The first rule states that an entity `E1` can export another entity `E2`, only if `E1` contains `E2`. The hygraph in Figure 5 verifies this rule by requesting all instances of entities that export entities which they do not contain. The database satisfies the constraint, if, and only if, the answer to this query is empty.

Before presenting the second rule we need a definition. We say that an entity `X` can access another entity `Y`, if there exists an entity `E1`, such that a) `E1` contains `X`, and either b.1) `E1` contains `Y`, or b.2) `E1` imports `Y`, or b.3) there exists entity `E2` that is imported by entity `E1` and `E2` recursively exports

Y. The hygraph in Figure 6 can be used to define the `canAccess` relation. The well formedness rule then says that entity `E1` imports entity `E2`, only if `E1` can access `E2`. To verify this rule, we can request all entities that import entities which they cannot access using the last hygraph. See the hygraph in Figure 7. Again, the database satisfies this constraint, if, and only if, the answer to this query is empty.

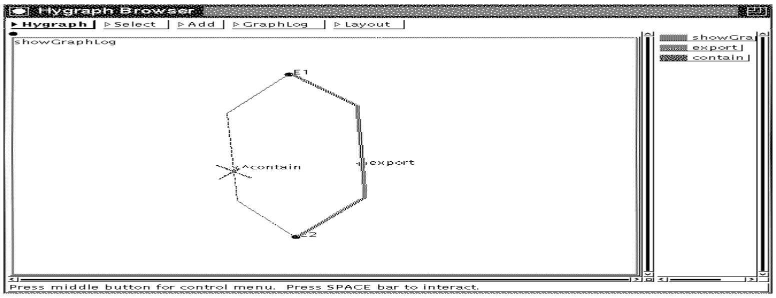


Figure 5: Design verification rule 1

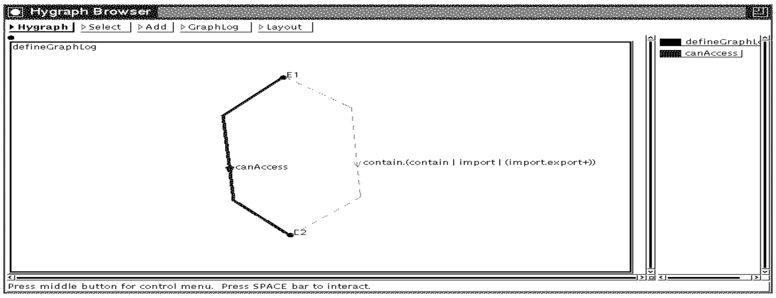


Figure 6: Definition of `canAccess`

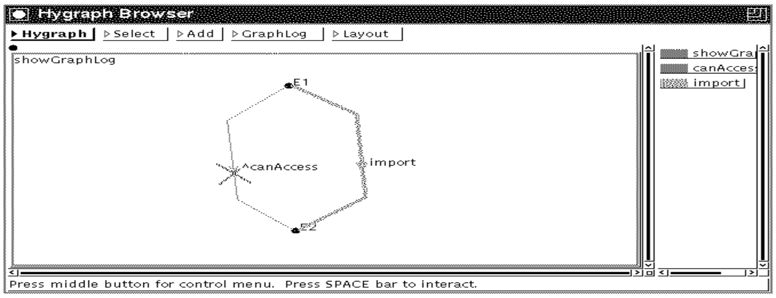


Figure 7: Design verification rule 2

3. Syntax and Semantics

GraphLog visualizations are based on the notion of hygraphs. Formally, a *hygraph* is a tuple

$$(N, L_N, v, L_E, E, L_B, B)$$

where N is a finite set of nodes; L_N is a set of node labels; v is the node labelling function from N to L_N; L_E is a set of edge labels; $E \subseteq N \times N \times L_E$ is a finite set of labelled edges; L_B is a set of blob labels; and, $B \subseteq N \times 2^N \times L_B$ is a finite set of labelled blobs.

In GraphLog, a *term* is a 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 function f applied to a number of terms. The function f could be anonymous, i.e., (X, SUM(X), (Y)) is a valid term. An *edge* or *blob label* is an expression generated by the following grammar, where $\tilde{T}$ is a sequence of terms and p is a predicate:

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

Database instances are hygraphs whose nodes are labelled with ground terms and whose edges and blobs are labelled with predicates. Database instances of the object oriented or relational model can easily be visualized as hygraphs. For example, an edge (blob) labelled $p(\tilde{X})$ from a node labelled T_1 to a node (containing a node) labelled T_2, corresponds to tuple $(T_1, T_2, \tilde{X})$ of relation p in the relational model.

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

A *define* query must have one *distinguished* edge or blob, labeled by a positive literal. The meaning of a *define query* 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. Each *define* hygraph G 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 G. For example, the meaning of the second query of Figure 2 is the following:

```
common_import(class(C), module(M)) ← import(class(C), X), import(module(M), X).
```

A *filter* query can have several distinguished edges and blobs; they represent those objects that the user wishes to see. The query evaluator searches the database for all occurrences of the pattern, and retains the objects that match the distinguished edges and blobs from each occurrence. Edges and blobs defined by *define* queries can of course be used in *filter* queries. *Filter* queries are discussed in more detail in Section 6.

It has been shown that the expressive power of GraphLog is the same as stratified linear Datalog, the same as first order logic with transitive closure, and the same as non-deterministic logarithmic space, when an ordering in the domain is assumed [4].

3.1. Aggregation in GraphLog

GraphLog has the ability to collect multisets of tuples and to compute aggregate functions on them. These functions allow the specification of a large class of queries not expressible in relational algebra or relational calculus. This class includes, however, useful queries which is justified by the fact that many commercial query languages, including SQL, support aggregate functions. The aggregate functions supported in GraphLog are the unary operators MAX, MIN, COUNT, SUM, and AVG. They are allowed to appear in the terms of the distinguished relation of a *define* query as well as in its incident nodes. For example, consider the query in Figure 8.

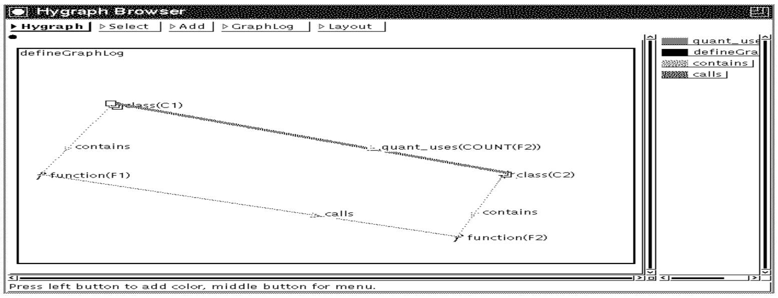


Figure 8: An example with aggregate functions

The meaning of this query is as follows: define the relation `quant_uses(C1, C2, N)` between two classes `C1` and `C2`, whenever `C1` contains a function `F1` such that `F1` calls `N` functions `F2` that are contained in class `C2`. That is, for each tuple of `C1, C2, F1, F2` values, it *counts* all the `F2` values associated with each pair of `C1, C2` values. Note there is no explicit "group-by" list; grouping is done over all variables appearing in the distinguished edge and its endpoints.

In this paper we are only concerned with non-recursive aggregation. For a review of recursive aggregation in GraphLog see [5]. GraphLog's non-recursive aggregation is equivalent in expressive power to what Mumick *et. al.* call *stratified aggregation*[11].

4. From GraphLog to LDL

This section describes the translation from GraphLog to LDL [12]. The translation of a program essentially resembles the logical function used to define the semantics of GraphLog in terms of Datalog programs [4].

4.1. Classifying Terms

A database schema is a description of the stored data and its organization in facts. The schema specifies among other things the format in which the data is stored and the type of each constant. LDL requires that a database schema be provided by the application program together with the facts that constitute the extensional part of the database. Thus, from a given database graph we induce a database schema. The schema of a constant is `integer`, or `real`, or `string`, depending on the type of the constant, the schema of an arithmetic expression is `integer` or `real` depending on the type of the operands, the schema of a function or an aggregate function is classified as `any`, etc. The schema of a relation $p(X_1, X_2, \ldots, X_n)$ consists of the strongest schema found in the graph for each of the arguments. For example, if `import(class(ootScanner), globalModule)`, and `import(ootParser, errorModule)` are found, we generate `p(any, string)` as the schema of `import`. A database schema consists of the set of schemata for the relations stored in it. For example, a database schema may be:

$$\texttt{database}(\{\texttt{import}(\texttt{any}, \texttt{string}), \texttt{export}(\texttt{any}, \texttt{string})\}).$$

4.2. Translating *Define* Query Graphs

For every *define* query with distinguished edges and blobs there is a corresponding LDL program that, when evaluated under LDL semantics, computes the meaning of the *define* hygraph under GraphLog semantics. In other words, every *define* hygraph corresponds to a set of logical rules, one for each distinguished edge or blob. Each logical rule expresses the distinguished edge or blob in terms of the literals that correspond to the non-distinguished edges and blobs in the hygraph. Additional rules may be necessary to define these literals. For example, a transitive closure relation requires two rules to be defined.

Let $P(\tilde{T})$ be the label of the distinguished edge (blob) e in the *define* hygraph with non-distinguished edges e_i. Also suppose that the nodes incident to e (the container node and a containing node) have labels T_1 and T_2. Program P produced by the translation contains the rule

$$p_e(T_1, T_2, \tilde{T}) \leftarrow p_{e_1}(\tilde{X_1}), \ldots, p_{e_n}(\tilde{X_n}).$$

where p_{e_i} is the predicate of the relation of e_i[2]. In particular, if edge e_i is incident to nodes with labels X_{i1} and X_{i2} and $\tilde{X_i} \equiv X_{i1}, X_{i2}, \tilde{X_{i3}}$, then

1. if e_i is $p(\tilde{X_{i3}})$, then $p_{e_i}(\tilde{X_i})$ is $p(X_{i1}, X_{i2}, \tilde{X_{i3}})$,
2. if e_i is $\neg p(\tilde{X_{i3}})$, then $p_{e_i}(\tilde{X_i})$ is $\neg p(X_{i1}, X_{i2}, \tilde{X_{i3}})$,
3. otherwise $p_{e_i}(\tilde{X_i})$ is defined by the following rules (also in P):
   - $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., if $e_i \equiv (e_1)$,
   - $p_{e_i}(X_{i2}, X_{i1}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., if $e_i \equiv -e_1$,
   - $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., and
     $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, Y, \tilde{X_{i3}}), p_{e_i}(Y, X_{i2}, \tilde{X_{i3}})$., if $e_i \equiv e_1+$,
   - $p_{e_i}(X_{i1}, X_{i1}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., and
     $p_{e_i}(X_{i2}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., and
     $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, Y, \tilde{X_{i3}}), p_{e_i}(Y, X_{i2}, \tilde{X_{i3}})$., if $e_i \equiv e_1*$.
   - $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., and
     $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_2}(X_{i1}, X_{i2}, \tilde{X_{i3}})$., if $e_i \equiv e_1 \mid e_2$,
   - $p_{e_i}(X_{i1}, X_{i2}, \tilde{X_{i3}}) \leftarrow p_{e_1}(X_{i1}, Y, \tilde{X_{i3}}), p_{e_2}(Y, X_{i2}, \tilde{X_{i3}})$., if $e_i \equiv e_1 \ . \ e_2$.

The query hygraph in Figure 9 defines the relation `related`: a module `M1` is related to module `M2`, if `M1` either imports or exports `M2`, directly or indirectly. The translation is:

[2]It is to be understood here that in order for the above rule to be safe all variables in $\{T_1, T_2, \tilde{T}\}$ appear in some predicate in the body.

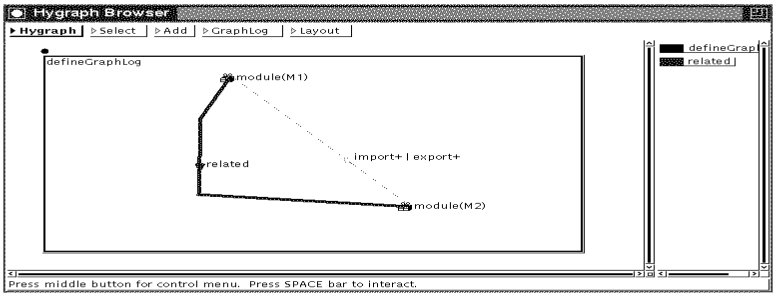


Figure 9: The relation `related`

```
related(module(M1), module(M2)) ← alter(module(M1), module(M2)).

alter(module(M1), module(M2)) ← tc_import(module(M1), module(M2)).
alter(module(M1), module(M2)) ← tc_export(module(M1), module(M2)).

tc_import(Xtemp1, Ytemp1) ← import(Xtemp1, Ytemp1).
tc_import(Xtemp1, Ytemp1) ← import(Xtemp1, Ztemp1), tc_import(Ztemp1, Ytemp1).

tc_export(Xtemp1, Ytemp1) ← export(Xtemp1, Ytemp1).
tc_export(Xtemp1, Ytemp1) ← export(Xtemp1, Ztemp1), tc_export(Ztemp1, Ytemp1).
```

4.3. Aggregation in LDL

The translation process presented in the previous section does not handle aggregate functions. If aggregate functions are present, additional rules to compute them are necessary. Since in LDL there is no support for multisets, over which GraphLog defines aggregate functions, the translation must simulate multisets using lists.

As an example of a *define* query containing aggregates consider the query hygraph of Figure 10. It defines the relation `new` for each `X, Y` that is connected through a `p` edge. The semantics of this hygraph is as follows: for each different pair of values of `X, Y` connected through a `p` edge, define the relation `new` with arguments `X, Y`, the sum of all `X`'s for the same `X-Y` pair, and the count of all `Y`'s for the same `X-Y` pair. The translation for this hygraph is given below.

```
new(X, Y, Result_X, Result_Y) ←
    body_new(X, Y, TheSet),
    aggr(setToList, TheSet, TheList),
    laggr(sum, 1, TheList, Result_X),
    laggr(count, 2, TheList, Result_Y).

empty(setToList, []).
single(setToList, L, [L]).
multi(setToList, L1, L2, L) ← append(L1, L2, L).
```

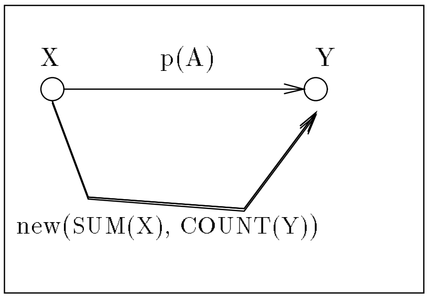


Figure 10: Define query with aggregate functions

```
9:   laggr(Op, Pos, [], V) ← lempty(Op, V).
10:  laggr(Op, Pos, [T], V) ← component(T, Pos, Comp), lsingle(Op, Comp, V).
11:  laggr(Op, Pos, [H|T], V) ← component(H, Pos, V1), laggr(Op, Pos, T, V2), lmulti(Op,
V1, V2, V).

12:  lempty(sum, 0).
13:  lsingle(sum, X, X).
14:  lmulti(sum, N1, N2, N1+N2).

15:  lempty(count, 0).
16:  lsingle(count, _, 1).
17:  lmulti(count, _, N2, 1+N2).

18:  body_new(X, Y, <(X, Y, A)>) ← p(X, Y, A).
```

The translation provides a rule that groups together all variables that appear in the rule body (cf. line 18). Values that are bound to these variables form a tuple. The set `TheSet` in line 2 contains all such tuples. The LDL built-in predicate `aggr` is used to generate the list `TheList` from all elements in `TheSet` (cf. line 3).

In LDL the built-in `aggr` predicate is used to define relationships involving the manipulation of sets. It is defined by the following:

```
aggr(Op, S, V) ← if empty(Op, V) then true
            else if S = {X} then single(Op, X, V)
            else (partition_once(S1, S2, S),
                  aggr(Op, S1, V1),
                  aggr(Op, S2, V2),
                  multi(Op, V1, V2, V)).
```

The rule uses variable `Op` that should be thought of as the parameter that describes the aggregate function to be computed using the rule. If `Op` is bound to `max` for example, the instantiation of the rule will be a description of how the function `max` is performed on sets. The body of the rule uses three built-in predicates: `empty` which describes how the function is performed on the empty set, `single` which says how to perform the function on a singleton, and `multi` which says how to perform the function on a set of cardinality greater than one. For example, these parameters for the operation `max` when computed over a set say that a) the maximum of an empty set is zero, i.e., `empty(max, 0)`, b) the maximum of a singleton set containing `X` is `X`, i.e., `single(max, X, X)`, and c) if the set `S` has cardinality larger than 1, and is partitioned into two sets with maximum `N1` and `N2`, then the maximum of `S` is the greater of `N1` and `N2`, i.e., `multi(max, N1, N2, N1)` $\leftarrow$ `N1` $>$ `N2)` and `multi(max, N1, N2, N2)` $\leftarrow$ `N1` $\leq$ `N2)`.

In order to provide the definition of some aggregation function, all one has to do is to specify the values for the predicates `empty`, `single`, and `multi`. The translation uses this technique to specify how we can transform a set to a list. The operation is called `setToList`. Its parameters are: a) from an empty set we get the empty list, i.e., `empty(setToList, [])`, b) from a singleton containing element L we get the list with that element, i.e., `single(setToList, L, [L])`, and c) the list L corresponding to a set S that is partitioned into two sets with corresponding lists L1 and L2 is the list that results from appending L2 to L1, i.e., `multi(setToList, L1, L2, L)` $\leftarrow$ `append(L1, L2, L)` (cf. lines 6-8).

By imitating the definition LDL provides for sets we can obtain a definition of how to perform aggregation on positions of list elements. Note that every position of an element of the list corresponds to a distinct variable in the rule body. If we consider, for example, the position `Pos` of list elements, all values that appear in the position `Pos` form a multiset because the same value may appear several times. Aggregation is performed on positions of list elements, or positions of the list for short. Performing aggregation over position `Pos` is semantically equivalent to performing aggregation over the multiset that contains all different values of the variable that corresponds to position `Pos`.

The next program describes how an aggregate operation `Op` can be computed on the position `Pos` of a list. It is a general definition in the sense that only the parameter predicates `lempty, lsingle`, and `lmulti`[3] need to be specified in order for `Op` to be performed on position `Pos` resulting `V`. The definition assumes a predicate `component(T, Pos, Comp)` specifying the component `Comp` in the position `Pos` in a list element `T`.

```
laggr(Op, Pos,[], V) ← lempty(Op, V).
laggr(Op, Pos, [T], V) ← component(T, Pos, Comp), lsingle(Op, Comp, V).
laggr(Op, Pos, [H|T], V) ← component(H, Pos, V1), laggr(Op, Pos, T, V2), lmulti(Op, V1,
V2, V).
```

The following set of rules describe the parameters for the definitions of the aggregate functions currently supported by GraphLog.

AVG

```
lempty(sum, 0).
lsingle(sum, X, X).
lmulti(sum, N1, N2, N1+N2).
```

COUNT

```
lempty(count, 0).
lsingle(count, _, 1).
lmulti(count, _, N2, 1+N2).
```

[3]`lempty` is the parameter for the empty list, etc.

MAX

```
lempty(max, 0).
lsingle(max, X, X).
lmulti(max, N1, N2, X) ← if (N1 > N2 then X = N1 else X = N2).
```

MIN

```
lempty(min, 0).
lsingle(min, X, X).
lmulti(min, N1, N2, X) ← if (N1 < N2 then X = N1 else X = N2).
```

SUM

```
lempty(sum, 0).
lsingle(sum, X, X).
lmulti(sum, N1, N2, N1+N2).
```

4.4. Translating Database Graphs

Graphs that are meant to represent databases have a rather straightforward translation too. These hygraphs have ground terms labelling nodes and predicates labelling edges. Every edge $p(\tilde{T_3})$ between nodes with labels T_1 and T_2 simply generates the fact $p(T_1, T_2, \tilde{T_3})$. *Filter* queries are discussed in Section 6.

5. From GraphLog to CORAL

The translation to CORAL [14] is almost identical with that to LDL except for some trivial syntactic differences between the two languages. The main difference is that CORAL requires the information to be organized into modules. A module in CORAL is to be understood as a unit defining one or more predicates with one or more exported facts or relations.

Every CORAL program is a collection of facts and a collection of modules. The translation of a database hygraph provides the necessary collection of facts. Every GraphLog *define* query on this database is translated into a module that defines the predicates indicated by the distinguished edges of the query. This module exports the query form that is generated by the query. An example module is:

```
aModuleName.
   export related(ff).

   related(module(M1), module(M2)) :- alter(module(M1), module(M2)).

   alter(module(M1), module(M2)) :- tc_import(module(M1), module(M2)).
   alter(module(M1), module(M2)) :- tc_export(module(M1), module(M2)).

   tc_import(Xtemp1, Ytemp1) :- import(Xtemp1, Ytemp1).
   tc_import(Xtemp1, Ytemp1) :- import(Xtemp1, Ztemp1), tc_import(Ztemp1, Ytemp1).
```

```
    tc_export(Xtemp1, Ytemp1) :- export(Xtemp1, Ytemp1).
    tc_export(Xtemp1, Ytemp1) :- export(Xtemp1, Ztemp1), tc_export(Ztemp1, Ytemp1).
end_module.
```

This module is generated by the query in Figure 9 that defines the relation `related(module(M1), module(M2))`. The export statement contains the query form that is used to compile the query and a description of the allowable queries. The string "ff" states that both arguments of these queries may be free variables.

5.1. Aggregation in CORAL

In contrast with LDL, CORAL provides grouping by multiset construction. Furthermore, in CORAL certain common aggregation functions such as `count`, `max`, etc., are built-in into the language. These functions are computed on the fly, thus avoiding the generation of multisets that was necessary for LDL. CORAL does not support user defined aggregate functions but the functions that it already supplies includes the set of aggregate functions currently supported by GraphLog. Due to some implementation problems of the CORAL release currently used by the system (and not restrictions of the CORAL language per se) only one aggregate function maybe specified in each rule head.

Since aggregate functions are part of the CORAL language the translation of GraphLog constructs containing aggregate functions into CORAL code is much simpler. For example, the translation of the query in Example 3 is the following:

```
module aModuleName.
export new(ffff).

new(X, Y, sum(<X>), count(<Y>)) :-
    p(X, Y, A).

end_module.
```

6. Filtering in GraphLog

In addition to the *define* queries whose translation we considered so far, Hy^+ allows the user to execute *filter* queries. A filter query is a hygraph with several distinguished objects (nodes, edges, or blobs). The meaning of a filter query is: for each instance of the pattern found in the database, retain the database objects that match the distinguished objects in the query. As an example consider again the first hygraph of Figure 2. The pattern requested consists of all pairs of classes `C` and modules `M` that import the same entity `X`. From the portions of the database that match this pattern only the `import` relation is displayed to the user.

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*.

It is not hard to see that filter queries, while very convenient, do not alter the theoretical expressive power of the language; in fact, we implement filter queries by translating them to sets of define queries. Given a *filter* query G, for each distinguished edge (blob) e, we generate a set of *define* queries that *match* e, i.e., when evaluated they determine all e that exist in the portions of the database that match the pattern of G. These *define* queries are constructed as follows. We consider the hygraph which is identical to G but where all edges and blobs are non-distinguished. If distinguished e of G is labelled with a predicate, then in this hygraph we add a distinguished edge (blob) between the two nodes e connects (between the container and the contained nodes). This distinguished edge (blob) essentially matches the predicate of e. If e is not labelled with a predicate, then we recursively expand e until all generated edges (blobs) are labelled with predicates. Then, we create the corresponding *define* queries for each of them as described above. The query evaluator evaluates each of the *define* queries in turn. The results are combined and the answer to the *filter* query is found.

Thus, for each distinguished relation e in the *filter* query G, we define the set of *define* queries that are answer-equivalent to G; e is recursively expanded and replaced by one or more edges so that the distinguished edge that is sent to the evaluator is always labelled with a predicate. $f_e(G)$, where $e \equiv (u, v, expr)$, defines the set of *define* queries that determine e as follows:

- $f_e(G) = \{G\}$, if $expr \equiv some\ predicate$. In G all edges are non-distinguished expect from a new edge that is added from u to v.
- $f_e(G) = f_{e_1}(G - e + e_1)$, where e_1 is $(u, v, expr_1)$, if $expr \equiv (expr_1)$.
- $f_e(G) = f_{e_1}(G - e + e_1)$, where e_1 is $(v, u, expr_1)$, if $expr \equiv -expr_1$.
- $f_e(G) = f_{e_2}(G - e + e_1 + e_2 + e_3)$, where e_1 is $(u, w, expr_1*)$, e_2 is $(w, z, expr_1)$, and e_3 is $(z, v, expr_1*)$ for two new nodes w and z, if $expr \equiv expr_1+$.
- $f_e(G) = \{G - e + e_2\} \cup f_{e_1}(G - e + e_1)$, where e_2 is $(u, v, =)$, and e_1 is $(u, v, expr_1+)$, if $expr \equiv expr_1*$.
- $f_e(G) = \emptyset$, if $expr \equiv \neg expr_1$.[4]
- $f_e(G) = f_{e_1}(G - e + e_1) \cup f_{e_2}(G - e + e_2)$, where e_1 is $(u, v, expr_1)$ and e_2 is $(u, v, expr_2)$, if $expr \equiv expr_1 \mid expr_2$.
- $f_e(G) = f_{e_1}(G - e + e_1 + e_2) \cup f_{e_2}(G - e + e_1 + e_2)$, where e_1 is $(u, w, expr_1)$ and e_2 is $(w, v, expr_2)$, for some new node w, if $expr \equiv expr_1\ .\ expr_2$.

To demonstrate these steps consider the *filter* query in the first hygraph of Figure 11. The pattern that this query asks for is all `p` edges followed by a `q` edge followed by an `s` edge. From all instances of this pattern that occur in the database, only the `p` and `q` edges will be displayed, since only those appear on the distinguished edge. In order to determine the *define* queries that are sent to the query evaluator, the above algorithm:

- removes the distinguished edge `p.q`,

[4] In fact for safety negated edges cannot be distinguished.

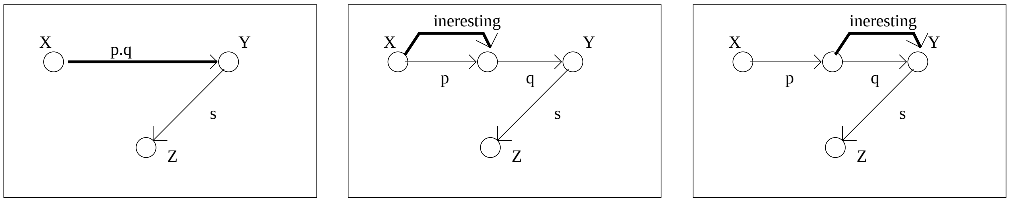


Figure 11: An example

- adds a new node between the two endpoints of the removed edge,
- adds an edge from the start of the removed edge to the new node labelled `p`,
- adds a new edge from the new node to the end of the removed edge labelled `q`, and
- adds a distinguished edge between the end points of each of these edges in turn.

The two *define* queries that are generated also appear in Figure 11.

This approach is rather naive: the generated code does not take advantage of the fact that in all these queries there is computation that repeats over and over again. We are currently investigating the possibility of generating a single query instead of many. This will speed the execution of *filter* queries and allow the generated code to better use the optimizations done by LDL or CORAL.

7. Conclusions

We showed how a deductive database engine can provide an efficient back-end for a data visualization and visual query system. We discussed the technical details of the translation from visual GraphLog queries to LDL and CORAL rules, concentrating on aggregate operators. We have applied the Hy+/GraphLog prototype to several domains, including software design [7], network management [1], and debugging of parallel programs [2]; preliminary experience indicates that the combination of a visual interface with the power of deductive database technology provides unique advantages over more conventional tools.

References

[1] M. Consens and M. Hasan. Supporting Network Management through Declaratively Specified Data Visualizations. In *Proceedings of the Third IFIP/IEEE International Symposium on Integrated Network Management*. Elsevier North Holland, 1993. To be published.

[2] M. Consens, M. Hasan, and A. O. Mendelzon. Debugging Distributed Programs by Visualizing and Querying Event Traces. Abstract to be presented at the ACM/ONR Workshop on Parallel and Distributed Debugging, 1993.

[3] M. Consens and A. O. Mendelzon. Expressing Structural Hypertext Queries in GraphLog. In *Proceedings of Second ACM Hypertext Conference*, 1989.

[4] M. Consens and A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *Proceedings of Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1990.

[5] M. Consens and A. O. Mendelzon. Low Complexity Aggregation in GraphLog and Datalog. In *Proceedings of Third International Conference on Database Theory*, 1992.

[6] M. Consens and A. O. Mendelzon. Hy$^+$: A Hygraph-based Query and Visualization System. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

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

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

[9] S. Mancoridis, R. C. Holt, and D. A. Penny. A Conceptual Framework for Software Development. In *Proceedings of the Twenty-First ACM Computer Science Conference*, 1993.

[10] S. Mancoridis, R. C. Holt, and D. A. Penny. A "Curriculum-Cycle" Environment for Teaching Programming. In *Proceedings of the Twenty-Fourth ACM SIGCSE Technical Symposium on Computer Science Education*, 1993.

[11] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. Duplicates and Aggregates in Deductive Databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, 1990.

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

[13] E.G. Noik. Unix Filters to Draw and Manipulate GXF Graphs. In A.O. Mendelzon, editor, *Declarative database visualization: recent papers from the Hy+/GraphLog project.* Forthcoming technical report, CSRI, Univ. of Toronto, 1993.

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