Deductive Database Support for Data Visualization

Mariano P. Consens, Alberto O. Mendelzon and Dimitra Vista

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

Abstract. We describe how deductive database technology can 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, considering aggregation as well. We also present an example of using GraphLog and its environment Hy$^+$ in order to support software design understanding.

1 Introduction

The data used in new applications of database technology, such as CAD/CAM, hypertext, software development support, and network management, often have graph-like structures 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 [13].

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 [7] and the database query language GraphLog [1, 6]. As a graphical formalism for visual manipulation of database visualizations, GraphLog is especially suitable for application areas where the data exhibit certain graphical structures, such as system design and software development [9], network monitoring [3], and hypertext [5]. 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. Section 4 describes the translation from GraphLog to LDL and CORAL. Emphasis is placed on the implementation of aggregate functions. We conclude in Section 5.

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 and its programming environment [11] contains a variety of tools. One of them, the Software Landscape, is a tool that displays a complete and concise graphical description (the Landscape) of the architecture of a system at various levels of granularity.

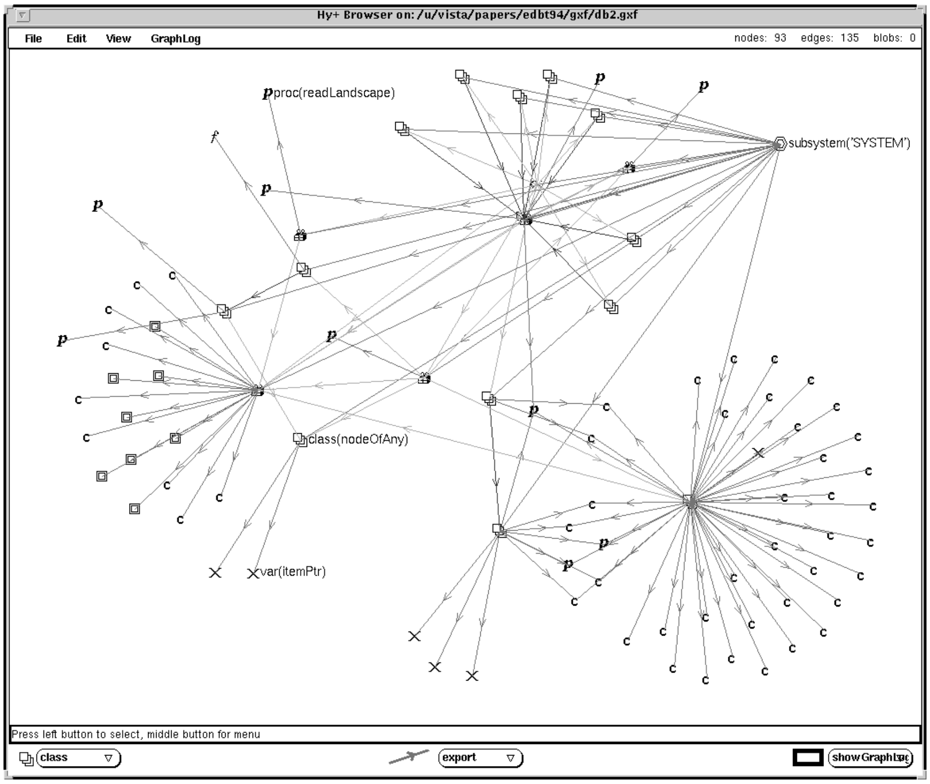

Fig. 1. The database describing the software Landscape

The Software Landscape provides a static visualization of the architecture of a system. GraphLog queries permit a more flexible approach to the visualization process where the user can obtain different visualizations of the same component through different queries.

Landscapes have a set of entities including classes, modules, subsystems, libraries, etc., which are interrelated through relations. Among them are:

- the relation `contain(X,Y)` that states that entity `X` contains sub-entity `Y`,
- the relation `export(X,Y)` that states that entity `X` allows access to its contained entity `Y`,
- 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 edges are color-coded according to the predicate name labelling them. Different icons are displayed according to the functor labelling the the nodes, 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. To avoid clutter, only a few textual labels that were interactively selected have been turned on.

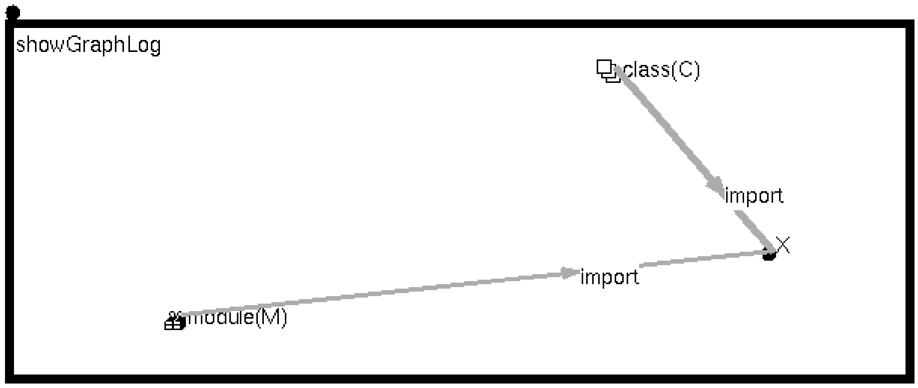

Fig. 2. An example *filter* query

Any visualization within Hy$^+$ can be queried. Figure 2 contains a GraphLog expression inside a box labelled `showGraphLog`, known as a *filter query.* Filter queries (introduced in [2]) are at the core of Hy$^+$'s capabilities for manipulating database visualizations, as opposed to just querying databases. Informally, when executing a query in *filter* mode the pattern does not define any new relation, but it filters out information from the input visualization to produce an output visualization that contains the union of all the sub-hygraphs in the input that match the query pattern. As one can see in the figure, one of the edges in the pattern is thicker. This is a visual way of distinguishing edges that 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.

The GraphLog expression inside the `defineGraphLog` in Figure 3 describes a *define query*. A thick edge here has a different meaning. It represents a 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 3 searches for the same pairs of classes `C` and modules `M` and defines the new relation `common_import` between them.

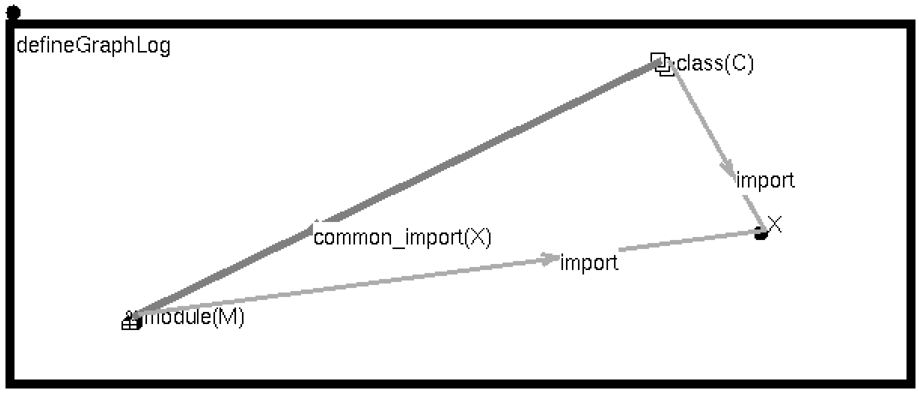

Fig. 3. An example *define* query

Hy$^+$ supports visualizations that are more general than graphs. Hygraphs, defined in [2], are a hybrid between Harel's higraphs [10] and directed hypergraphs. Hygraphs use *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 4 demonstrates how we can change the representation of a relationship from edges to blobs. The blob `contents` clusters together the sub-entities contained in some entity: an entity labeled `E` is enclosed in the blob whose container node is labeled `C`, whenever `contain(C, E)` holds. The filter query in Figure 4 (defined by the three `showGraphLog` boxes) produces a hygraph with `contents` blobs and `import` and `inherit` edges.

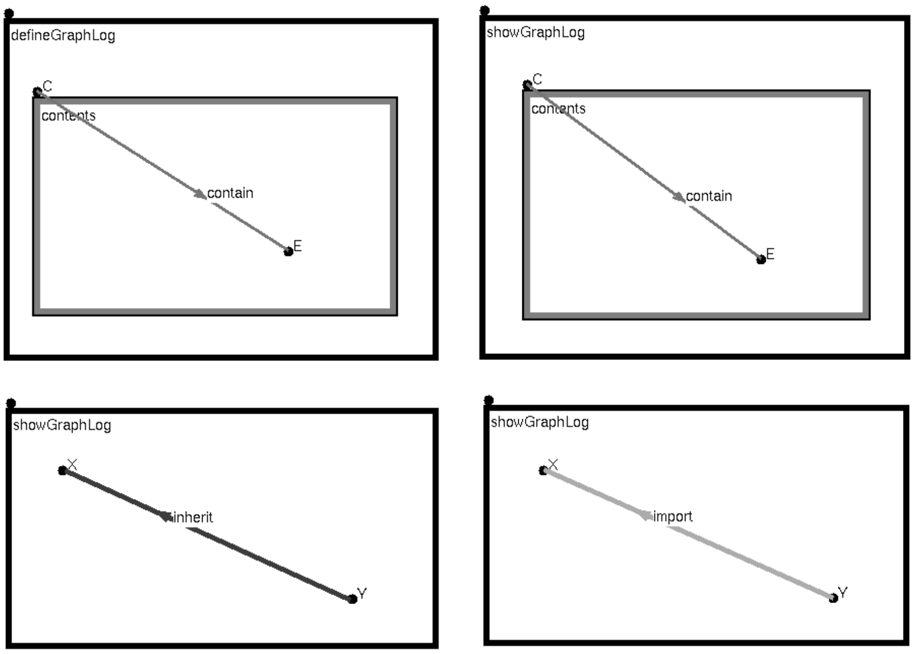

Fig. 4. Define and show queries to produce a hygraph

When Hy$^+$ executes the queries in Figure 4 against the database of Figure 1, the result is the hygraph visualized in Figure 5.

3 GraphLog in Hy$^+$

In this section we give a more precise definition of GraphLog, and we describe how it is used in the Hy$^+$ system. GraphLog database visualizations and query expressions are based on the notion of hygraphs.

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 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:

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

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.

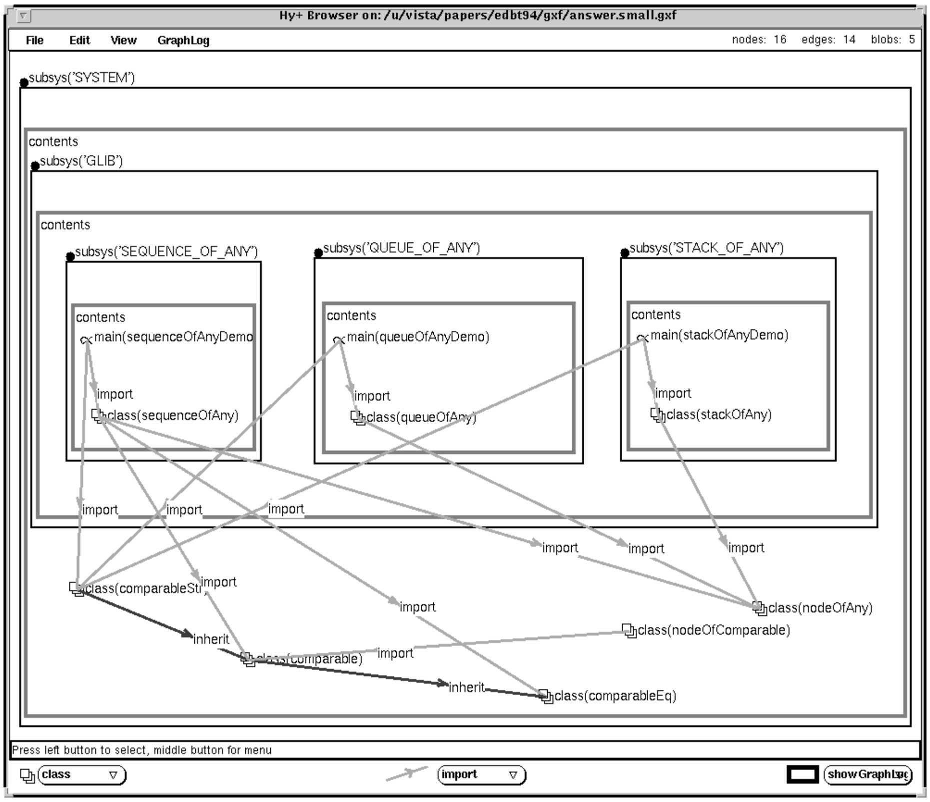

Fig. 5. The result of filtering `contents`

For example, an edge (blob) labelled $p(\bar{X})$ from a node labelled T_1 to a node (containing a node) labelled T_2 corresponds to tuple $(T_1, T_2, \bar{X})$ of relation p in the relational model.

Queries are sets of 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 elements*, explained below.

A hygraph pattern in a `defineGraphLog` blob must have 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. 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 query of Figure 3 is the following: `common_import(class(C), module(M))` $\leftarrow$ `import(class(C), X), import(module(M), X).`

It has been shown elsewhere [1, 6] 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.

A hygraph pattern in a `showGraphLog` blob 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. As an example, consider again the 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*.

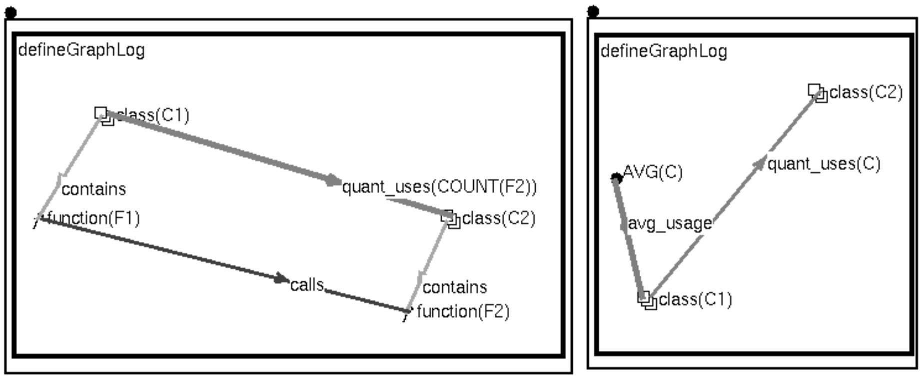

Fig. 6. An example with aggregate functions

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. 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 6. The first one defines the relation `quant_uses` as follows: the relation `quant_uses(C1, C2, C)` is defined between two classes `C1` and `C2`, whenever `C1` contains a function `F1` such that `F1` calls `C` functions `F2` that are contained in class `C2`. The second pattern defines the relation `avg_usage` as follows: the relation `avg_usage` is defined between a class `C1` and a number `C`, if `C` is the average of all numbers `C` 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 and its endpoints.

In this paper we are only concerned with non-recursive aggregation. For a review of recursive aggregation in GraphLog see [8].

4 Translating GraphLog

This section describes the translation from GraphLog to LDL [12] and CORAL [13]. The translation of a program resembles the logical function used to define the semantics of GraphLog in terms of Datalog programs [1, 6]. An alternative optimized translation is described in [14].

Since Hy$^+$ is implemented in the Smalltalk object oriented programming language, we were able to encapsulate within classes the functionality that deals with parsing and translating `defineGraphLog` blobs. In addition, we used inheritance to specialize classes to deal with each of the target back-end systems, while re-using common code inherited from superclasses. This approach greatly reduces the effort required to support additional deductive database systems. The details of the translation are given in the next two subsections.

4.1 The Basic Translation

We consider first *define* queries in which there is no aggregation. For every hygraph in a `defineGraphLog` blob with a distinguished edge or blob there are corresponding LDL and CORAL programs that, when evaluated under LDL or CORAL semantics, compute the meaning of the GraphLog expression. Each `defineGraphLog` blob corresponds to a set of logical rules that expresses the relation labelling 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.

We describe the translation to LDL below and provide a sample of the output code produced by Hy$^+$. The basic translation (no aggregation) to CORAL [13] is almost identical, except for some trivial syntactic differences between the two languages. The main difference is that CORAL requires the information to be organized into modules. Every GraphLog *define* query 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.

Let $P(\bar{T})$ 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

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

where p_{e_i} is the predicate of the relation of e_i. In particular, if edge e_i is incident to nodes with labels X_1 and X_2 and $\bar{X} \equiv X_1, X_2, \bar{X_3}$, then

1. if e_i is $p(\bar{X_3})$, then $p_{e_i}(\bar{X_i})$ is $p(X_1, X_2, \bar{X_3})$,
2. if e_i is $\neg p(\bar{X_3})$, then $p_{e_i}(\bar{X})$ is $\neg p(X_1, X_2, \bar{X_3})$,
3. otherwise $p_{e_i}(\bar{X})$ is defined by the following rules (also in P):
   - $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, X_2, \bar{X_3})$, if $e_i \equiv (e_1)$,
   - $p_{e_i}(X_2, X_1, \bar{X_3}) \leftarrow p_{e_1}(X_1, X_2, \bar{X_3})$, if $e_i \equiv -e_1$,
   - $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, X_2, \bar{X_3})$, and
     $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, Y, \bar{X_3}), p_{e_i}(Y, X_2, \bar{X_3})$, if $e_i \equiv e_1+$,
   - $p_{e_i}(X_1, X_1, \bar{X_3}) \leftarrow p_{e_1}(X_1, X_2, \bar{X_3})$, and
     $p_{e_i}(X_2, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, X_2, \bar{X_3})$, and
     $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, Y, \bar{X_3}), p_{e_i}(Y, X_2, \bar{X_3})$, if $e_i \equiv e_1*$.
   - $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, X_2, \bar{X_3})$, and
     $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_2}(X_1, X_2, \bar{X_3})$, if $e_i \equiv e_1 \mid e_2$,
   - $p_{e_i}(X_1, X_2, \bar{X_3}) \leftarrow p_{e_1}(X_1, Y, \bar{X_3}), p_{e_2}(Y, X_2, \bar{X_3})$, if $e_i \equiv e_1 \,.\, e_2$.

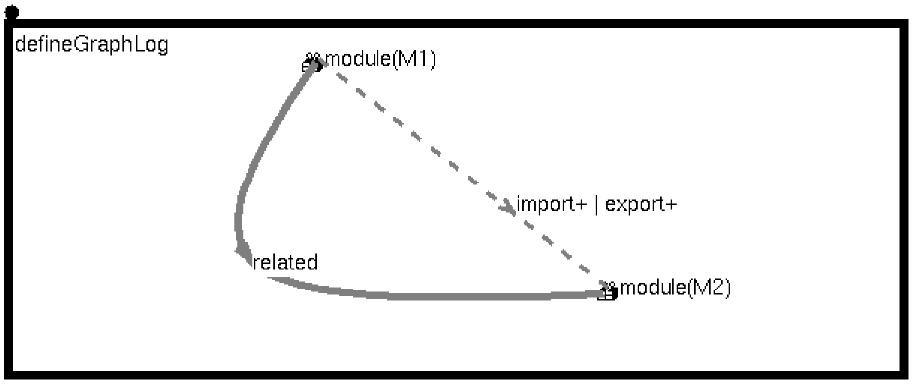

Fig. 7. The relation `related`

The query hygraph in Figure 7 defines the relation `related`: a module `M1` is related to module `M2`, if `M1` either imports or exports `M2`, directly or indirectly. The translation produced by Hy$^+$ is:

```
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).
```

We now turn our attention to describing the translation of *filter* queries. Originally, hygraphs in `showGraphLog` blobs were implemented by translating them to sets of define queries (in a process outlined below). The motivation for this approach was to speed up development by directly reusing existing code. We later modified the translation to generate a single CORAL module per filter query.

Given a hygraph G in a `showGraphLog` blob, 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.

4.2 Handling Aggregation

The translation process presented in the previous section does not consider aggregate functions. If aggregation is present in a GraphLog expression, additional rules to compute the aggregate functions are necessary. On top of this, since LDL does not directly support multisets (over which GraphLog defines aggregate functions), the translation must add rules to simulate multisets using lists. To illustrate this, Figure 8 shows the output of the Hy$^+$ translation for the rightmost `defineGraphLog` blob of Figure 6.

The translation provides a rule that groups together all variables that appear in the rule body (cf. line 1). Values that are bound to these variables form a tuple.

```
1:  body_avg_usage(class(C1), <(C1, C2, C)>) ←
        quant_uses(class(C1), class(C2), C).
2:  avg_usage(class(C1), Result_C) ←
3:      body_avg_usage(class(C1), TheSet),
4:      aggr(setToList, TheSet, TheList),
5:      laggr(sum, 3, TheList, Sum),
6:      laggr(count, 3, TheList, Count),
7:      Result_C = Count / Sum.
8:  empty(setToList, []).
9:  single(setToList, L, [L]).
10:  multi(setToList, L1, L2, L) ← append(L1, L2, L).
11: laggr(Op, Pos, [], V) ← lempty(Op, V).
12: laggr(Op, Pos, [T], V) ← component(T, Pos, Comp),
                             lsingle(Op, Comp, V).
13: laggr(Op, Pos, [H|T], V) ← component(H, Pos, V1),
                             laggr(Op, Pos, T, V2),
                             lmulti(Op, V1, V2, V).
14: lempty(sum, 0).
15: lsingle(sum, X, X).
16: lmulti(sum, N1, N2, N1+N2).
17: lempty(count, 0).
18: lsingle(count, _, 1).
19: lmulti(count, _, N2, 1+N2).
```

Fig. 8. The LDL program for a GraphLog query with aggregation.

The set TheSet in line 3 contains all such tuples. The LDL built-in predicate aggr is used to generate the list TheList from all elements in TheSet (cf. line 4).

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.

To provide the definition of some aggregation function, all one must 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` (cf. lines 8–10).

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` 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 set of rules that describe the parameter predicates for the definitions of the aggregate functions `count` and `sum` appear in lines 14–19 of Figure 8..

In contrast with LDL, CORAL provides grouping by multiset construction. Furthermore, in CORAL certain common aggregation functions, such as `count`, and `max`, are built 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 include 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 may be 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, as can be seen from comparing the program below with its equivalent LDL program in Figure 8.

```
module bodyModule.
export avg_usage(ff).

avg_usage(class(C1), avg(<C>)) :-
        quant_uses(class(C1), class(C2), C).

quant_uses(class(C1), class(C2), count(<F2>)) :-
        contains(class(C1), function(F1)),
        calls(function(F1), function(F2)),
        contains(class(C2), function(F2)).

end_module.
```

5 Conclusions

We described how deductive database engines can be used as back-ends 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.

We have showed how the GraphLog visual language can provide an intuitive and consistent user interface for deductive database systems with different underlying semantics (in particular, concerning aggregation). In addition, we mentioned how the use of an object oriented programming language in the implementation of Hy^+ reduced the effort required to support multiple deductive databases.

We have applied the Hy+/GraphLog prototype to several domains, including software design [9], network management [3], and debugging of parallel programs [4]; 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. P. Consens. Graphlog: "Real Life" Recursive Queries Using Graphs. Master's thesis, Department of Computer Science, University of Toronto, 1989.
2. M. P. Consens. Visual Manipulation of Database Visualizations. PhD Thesis Research Proposal, 1992.
3. M. P. Consens and M. Hasan. Supporting Network Management through Declaratively Specified Data Visualizations. In *Proceedings of the IFIP/IEEE 3rd International Symposium on Integrated Network Management, III*, pages 725–738. Elsevier North Holland, 1993.
4. M. P. Consens, M. Hasan, and A. O. Mendelzon. Debugging Distributed Programs by Visualizing and Querying Event Traces. In *Declarative Database Visualization: Recent Papers from the Hy^+/GraphLog Project, University of Toronto Technical Report CSRI-285*, pages 11–34, 1993. Available by ftp to db.toronto.edu, in pub/papers.

5. M. P. Consens and A. O. Mendelzon. Expressing Structural Hypertext Queries in GraphLog. In *Proceedings of 2nd ACM Hypertext Conference*, pages 269–292, 1989.
6. M. P. Consens and A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *Proceedings of 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.
7. 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, 1993. Video Presentation Summary.
8. M. P. Consens and A. O. Mendelzon. Low Complexity Aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):379–394, 1993.
9. M. P. Consens, A. O. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *14th. Intl. Conference on Software Engineering*, pages 138–156, 1992.
10. D. Harel. On Visual Formalisms. *Communication of ACM*, 31(5):514–530, 1988.
11. 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.
12. S. A. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases*. Computer Science Press, 1988.
13. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of International Conference on Very Large Databases*, 1992.
14. D. Vista and P. Wood. Efficient Visual Queries for Deductive Databases. In *Proceedings of the Workshop on Programming with Logic Databases*, 1993.