Creating and Filtering Structural Data Visualizations using Hygraph Patterns

by

Mariano P. Consens

A thesis
submitted in conformity with the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
at the University of Toronto

Published as Technical Report CSRI-302

February 1994

Abstract

Creating and Filtering Structural Data Visualizations using Hygraph Patterns

Doctor of Philosophy, 1994

Mariano P. Consens

Department of Computer Science

University of Toronto

Data visualization plays a fundamental role in helping users solve complex, information-intensive problems in scientific, engineering and business applications. This thesis introduces an original framework for the use of queries to create and filter *structural data visualizations* (a term we introduce to refer to the diagrammatic display of the relationships of structured data). *Hygraphs*, a new visual formalism, provides a precise characterization of the diagrammatic visualizations considered. This simple formalism is a convenient abstraction for both graph-based and form-based (or tabular) presentations.

We present theoretical and practical contributions that exploit the synergism between the established field of database query languages and the emerging area of visualization. On the database theory side, we introduce the concept of *filter queries* and provide a formal defini-

tion and expressive power characterization of *filtering programs*. We then extend the visual query language *GraphLog* to make use of *hygraph patterns* for both defining derived data and expressing filter queries. The framework presented here supports the creation of structural data visualizations by using hygraph patterns to both *define* new relationships in the data to be visualized and *filter* the existing data to display only information that is relevant to the user.

On the practical side, we describe the *Hy$^+$ Hygraph Visualization System*. Hy$^+$ embodies a significant amount of the functionality that can be developed within the formal framework described in this thesis. We discuss the application of Hy$^+$ to software engineering and network management to demonstrate the possibilities opened up by creating and filtering structural data visualizations using hygraph patterns.

Acknowledgments

I would like to thank my advisor, Alberto Mendelzon, for his support and encouragement.

I am indebted to Vassos Hadzilacos and David Harel for providing me with extensive and detailed comments on preliminary versions of the thesis. I would like to thank the remainder members of my program committee, Anthony Bonner, Mark Chignell, Eugene Fiume and Dave Wortman, as well as Tova Milo, for their suggestions to improve this document.

I wish to thank the colleagues that participated in the development of the Hy$^+$ system: Frank Eigler, Masum Hasan, Yanni Jew, Brian Sallans, Annie Yeung, and Dimitra Vista. I would like to recognize Arthur Ryman's enthusiastic support of GraphLog and Jacob Slonim's interest in our work, as well as Sergio Fiszman, Shahram Javey, and Darrell Raymond for their collaboration in the Hy$^+$ applications areas.

I am grateful for the financial support received from a University of Toronto Open Fellowship and an IBM Canada Research Fellowship.

My warmest thanks to my wife Verónica for her truly enjoyable support (including thesis proofreading), and my most sincere acknowledgement to Tura and Eugenio Bespali, Ana Maria and Ruben Bianchi, John Farewell, Denis Fox, and Mercedes and Helmuth Fuchs.

Contents

1 Introduction **1**

1.1 Types of Visualization 2

1.2 Related Research 3

1.3 Our Approach 8

1.4 Overview of the Thesis 11

2 A Tour of Hy$^+$ **12**

2.1 Visualizing Data Describing `C++` Code 14

2.2 Simple Queries to Show and Define Data 18

2.3 Defining and Showing Hygraphs 25

2.4 Exploring Large Hygraph Visualizations 29

2.5 Advanced Queries and Selective Layout 36

2.6 Iterative Filtering 42

2.7 Pattern-based Selection in Editors 43

2.8 Discovering Visualizations 46

3 The Hygraph Visual Formalism **49**

3.1 Definition of Hygraphs 50

3.2 Comparing Hygraphs to Higraphs 52
3.3 Visualizing Hygraphs 56
3.4 Hygraph Applications 60

4 Filtering Languages and Hygraph Patterns 65
4.1 Queries and Filters 66
4.2 Expressive Power of Query Languages 70
4.3 Logic Programs 73
4.4 Filtering Programs 80
4.5 GraphLog Patterns in Hy$^+$ 86

5 The Hy$^+$ System 90
5.1 The Architecture of Hy$^+$ 91
5.2 Hy$^+$ Evolution 98

6 Hy$^+$ Applications 102
6.1 Partitioning Code into Overlay Modules 104
6.2 Software Configuration Management 116
6.3 Debugging Traces of Distributed Applications 119
6.4 Network Management Information 121

7 Conclusions 125

List of Figures

1.1 Automatic design of visualizations. 5

1.2 Model based user interface systems. 6

1.3 Overview of the thesis approach. 10

2.1 A Hy^{+} screen for the NIH scenario. 15

2.2 The NIH class hierarchy. 17

2.3 The Hy^{+} query that displays the class hierarchy. 19

2.4 The pattern to show the subclasses of Collection. 20

2.5 Displaying the direct subclasses of Collection. 20

2.6 Defining and showing all_subclasses of Collection. 22

2.7 Displaying all_subclasses of Collection. 22

2.8 Filtering the inheritance subtree for Collection. 23

2.9 Displaying the tree of the subclasses of Collection. 23

2.10 Obtaining the set of subclasses of Collection as an answer. 24

2.11 The methods and variables blobs of class('Bag'). 26

2.12 The query to produce the hygraph for Bag. 27

2.13 The patterns defining the methods and variables blobs. 29

2.14 The patterns that create a large hygraph visualization. 30

2.15 Browsing a large hygraph. 31
2.16 Zooming into a smaller region to appreciate details. 32
2.17 The query to produce a hygraph for the subclasses of Link. 33
2.18 The result with and without hiding the variables blobs. 34
2.19 Recursively hiding nested subclass blobs. 35
2.20 Definitions for usage and coupling relationships. 36
2.21 Color-coding three degrees of coupling. 38
2.22 Superimposing degrees of coupling over the class hierarchy. 38
2.23 Visualizing the degree of coupling among NIH classes. 39
2.24 Selectively hiding usage. 40
2.25 The result of selectively hiding usage. 41
2.26 Filtering function calls to superclasses. 42
2.27 Showing context information for calls to superclasses. 43
2.28 Pattern-based selection of overriding calls. 44
2.29 Highlighting the selection of overriding calls. 45
2.30 Definitions for an inheritance browser. 46
2.31 The pattern to display the inheritance browser. 47
2.32 The inheritance browser for class('SeqCltn'). 48

3.1 The statechart for a stopwatch as a hygraph. 55
3.2 Hygraphs with duplicate node representations. 57
3.3 Seeing a spreadsheet as a hygraph. 59
3.4 Defining blobs for attribute boxes and regions. 60
3.5 References to variables in functions of ArrayOb called from reSize. 61
3.6 A selective diagram with boxes for attributes. 62

3.7 Defining blobs to represent a table with headings. 63
3.8 Show pattern to display a relation with very local calls. 63
3.9 The table for calls less than two lines of code away. 64

4.1 Defining and showing a variation on the same generation query. 81
4.2 A graph showing the same generation edges for Stack. 82
4.3 Defining and showing the derivation trees for sg facts. 84
4.4 The derivation trees for sg(class('Stack',down),X). 85

5.1 Overview of the Hy$^+$ architecture. 92
5.2 Hy$^+$ dialog window for layout boxes 95
5.3 Class hierarchy of the Smalltalk image used for Hy$^+$ development. 96
5.4 Visualizing the classes in Hy$^+$ grouped by categories. 97
5.5 Showing flight connections in context and individually. 99
5.6 Synchronized graphical and textual browsing of source code. 100

6.1 Nested blob display of the memory overlay map. 106
6.2 Tree of the sections and areas. 107
6.3 A portion of the overlay tree with a few functions and calls. 108
6.4 Abstracting call patterns at the section level. 108
6.5 The cluttered reality of section_call. 109
6.6 GraphLog definitions for track using two intermediate predicates. 110
6.7 GraphLog definition for track in one pattern. 111
6.8 Prolog definition of track. 113
6.9 Datalog definition of track. 113
6.10 Focusing on the smashed functions. 114

6.11 The filtered memory map with the smashed functions. 115
6.12 An explanation for the tracking of deskrun. 116
6.13 Relating versions and branches to streams. 117
6.14 The schema of the configuration management data. 118
6.15 Inter-process communication hygraph. 119
6.16 Showing the events, processes and messages. 120
6.17 Buggy vs. correct initialization patterns. 121
6.18 Clustering devices (excluding gateways) in subnets. 122
6.19 Displaying subnets over the physical network topology. 123
6.20 Traffic information superimposed on the topology map. 124

Chapter 1

Introduction

A new approach to computing is emerging in which *data visualization* plays a fundamental role. This trend, supported by advances in computer technology (particularly in graphics and user interfaces), is driven by the need to comprehend the huge amounts of complex data that we are increasingly being exposed to.

There is a variety of factors that contribute to this information overload. A historical factor is the cumulative result of years of storing large quantities of all types of data. With an ever increasing number of computerized activities, the data available electronically keeps growing steadily. Furthermore, the emergence of both local and global connectivity, due to advances in networking technology and its infrastructure, contribute to an explosion in the volume of data available at any single site.

In addition to escape being swamped by the data flood, another basic motivation for data visualization is the more ambitious nature of the tasks themselves for which computer support is sought. A growing number of sophisticated computer users demand software to help them solve complex, information-intensive problems in scientific, engineering and business applications. Nardi and Zarmer [NZ93] make the following statement on how to meet such a demand.

> We want to focus attention on the way that computers can support cognitive activities that are part of the problem-solving associated with the application itself—activities such as perceiving trends, seeing patterns, finding individual data values in large datasets, making comparisons, testing for accuracy and completeness. … It is this kind of cognitive activity for which good visualizations–and good interfaces–are needed.

1.1 Types of Visualization

Research in *scientific visualization* [MDB87, UFK$^+$89] has been very successful in demonstrating how large multi-dimensional datasets can be presented graphically in such a way that information that is too complex to be perceived numerically can be comprehended more intuitively through the use of our visual senses. Scientific visualization techniques deal with the static and animated display of abstract numerical and statistical data (in many cases, things that we could not normally see) in a graphical form so that patterns revealing the underlying structure of the data can be easily comprehended.

In contrast to the attention paid to the display of quantitative data that is characteristic of scientific visualization, the work at Xerox PARC [CRM91, RCM93] is one instance of research that focuses on the display of the structural relationships of abstract data, an activity for which [CRM91] coined the term *information visualization*. A variety of generic three dimensional visualization techniques are presented in [RCM93], and the synergism that can be achieved between information retrieval and information visualization is highlighted.

The work in this thesis deals with the display of the relationships of abstract, structured data (like the datasets stored in databases), an endeavor that we denote by *structural data visualization*. This terminology is introduced to emphasize the main characteristic of the kinds of visualizations that we are concerned with, to differentiate it from the predominantly numerical scientific visualizations, and to reserve the broader designation of information visualization as an all encompassing term. A major issue in scientific data visualization is to determine the most meaningful ways of representing the multiple dimensions of the data. In contrast, structural data visualization deals with the display of abstract relationships of data that in many cases has no dimensionality.

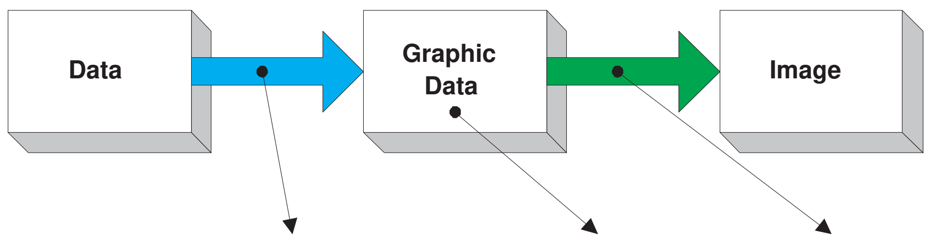
Software engineering constitutes a familiar example of an application area that exhibits plenty of the highly abstract data that is quite amenable to structural data visualization. Nevertheless, there is also quantitative data generated by the software engineering process. Similarly, it is hard to imagine an application domain with the need to visualize complex data, even if predominantly numerical, for which there is not even some form of hierarchical structure imposed on the data. At the very least this hierarchy constitutes an example of structural data. The point we are trying to make is that despite the predominance of data that can be better comprehended by resorting to either structural data visualization or scientific data visualization, there is always a benefit in using the widest possible repertoire of visualization techniques when solving information-intensive problems.

1.2 Related Research

The data visualization process starts with the data and then proceeds to generate a description of the graphics that are finally rendered to obtain an image (see the top portion of Figure 1.1). Systems like APT [Mac88], SAGE [RM90], VISTA [SI91], and BOZ [Cas91] attempt to automate (resorting to AI techniques) the process of obtaining a graphic description of a picture to be rendered. These systems take as input a characterization of the data (e.g., whether attributes represent quantitative, ordinal or nominal data) and use their internal prescriptive theory of graphic design to generate a visualization. Systems like ANDD [Mar91] and TRIP [Kam89] rely on users to provide a declarative specification of the graphical objects together with constraints on how to present them, and then automate the process of heuristically solving an over-constrained specification to produce an image. All of these systems make the assumption that the entire input dataset is to be presented visually. Clearly, this leaves to an earlier stage the selection of the relevant subset of the data. More importantly, once a visualization is created there is no way to keep only those aspects in which the user is interested and discard the rest.

Figure 1.1 summarizes the systems we just mentioned. The first column in the figure describes the information used in going from the data to a graphic description, while the form of this description appears in the second column. The third column describes the processing of the graphic data to obtain an image.

Several systems dealing with the problem of presenting data and their relationships to users have been developed recently in the database and user interface communities (e.g., [BOS91, Dea91, Row92, KN92]). Proteus [AEM86], Humanoid [Sze90, SLN93], ODDS [FM92] and ACE [JNZM93, NZ93] are representative of model-based interface tools. These systems support constructing a declarative model of how the interface should look and behave. This model is constructed by describing how the components (widgets) that represent the objects in the image are assembled using the data and other widgets. One aspect that should be highlighted is that the construction of the widgets requires a traversal of the data that is not specified *globally* (i.e., with *one* expression over the input data that specifies the subset of the data selected for presentation), but instead must be described *locally* at each



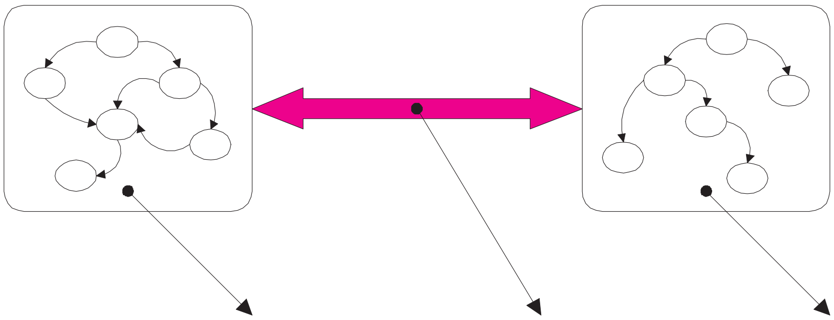
APT	Data Characterization (quantitative, ordinal, nominal)	Formal Graphical Language (Marks)	Rendering
SAGE	Dimensions for Data Characterization (Data, Relationships and Information Seeking Goals)		
VISTA	Scientific Data (quantitative: dimension, rank, scale)		
BOZ	Logical Task Description	Perceptual Operators	
TRIP	Visualization Mapping	Graphical Objects, Drawing Relations and Constraints	COOL
ANDD	Pragmatic Design Directives Expressive Mapping	Network Diagram (Marks and Constraints)	Over-constrained layout

Figure 1.1: Automatic design of visualizations.

widget. This traversal is specified using higher level languages. Care must be taken so that the recursive traversals that are specified to build a hierarchy of widgets do terminate.

An overview of the systems mentioned in the previous paragraph is presented in Figure 1.2. The leftmost column describes the model for the application data, while the rightmost column does so for the objects in the user interface. The column in the center mentions how the mapping between the previous two is described.

The approach taken by ACE [JNZM93, NZ93] is particularly relevant to our work. Users



PROTEUS	Internal Objects	Representation Defining Objects (TEDM)	Layout Objects
HUMANOID	Application Objects	Templates (LOOM)	Widget Tree
ODDS	GemStone Objects	Outlines (Smalltalk)	Layout and Interactor Executors
ACE	Application Data Types	End User Programming (Formula Language)	Visual Formalisms and Selectors
Hy$^+$	Objects and Relations (Logic Facts)	Hygraph Patterns (defineGraphLog, showGraphLog, layoutGraphLog)	Hygraphs (Smalltalk Visual Parts)

Figure 1.2: Model based user interface systems.

of ACE interact with *visual formalisms*, a term introduced by Harel in [Har88] to describe diagrammatic displays with well-defined semantics for expressing relations. The authors of ACE propose using visual formalisms as a basis for user interface design, and argue about the inadequacy of the use of mental models and metaphors in the design of software that supports complex tasks such as design and analysis. They suggest instead using visual formalisms as application frameworks that provide users with a structure into which a model is cast (as opposed to leaving to the users the invention of a new structure). Several reasons are given in support of the use of visual formalisms in ACE, and they are equally applicable to our work. We present them below, slightly reorganized. The first three points are advantageous from the user's perspective, while the last three are more relevant to the software developer.

- *Exploitation of human visual skills.* Visual formalisms are capable of showing a large quantity of data in a small space, and of providing unambiguous semantic information about the relations among the data (see [Ray91] on visual notations).

- *Manipulability.* Computer-based visual formalisms are not static displays. They allow users to view and manipulate the displays and their content in ways appropriate to the application in which they are used.

- *Familiarity.* Because the standard visual diagrams are so useful, they are found everywhere. Not only do they draw on innate perceptual abilities, but through constant exposure we become very familiar with them.

- *Specializability.* Visual formalisms can be specialized to meet the needs of a particular application. They are at the right level of granularity: neither too specific nor too general.

- *Broad applicability.* Visual formalisms are useful because they express a fairly generic set of semantic relations, relevant to a wide range of application domains.

- *Reusability of interaction techniques.* Because a large number of applications can be designed around a given visual formalism, solutions for editing and browsing formalisms can be shared (hence eliminating a great deal of costly low-level programming).

Finally, we should also refer to the research that focuses on those aspects at the end of the visualization process: the actual images presented to the user and the interactions supported. In this regard, the use of three-dimensional graphics for visualizing structural data has been recently investigated at MCC [FPF88] and Xerox PARC [CRM91, RCM93]. This represents a new and exciting trend in user interfaces that support the interactive browsing of large structural visualizations. Similarly, the display of cone trees in palmtop computers [FZC93, Fit93] demonstrates the feasibility of exploring large visualizations using hand held devices.

In short, there is a growing body of research in related areas of computer science dealing with the presentation of very large and complex visualizations. The importance of this style of dense information displays is highlighted by Tufte [Tuf90]:

> High information displays are not only an appropriate and proper complement to human capabilities, but also such designs are frequently optimal. If the visual task is contrast, comparison, and choice—as so often it is—then the *more relevant* information within eyespan, the better.

1.3 Our Approach

One of the novel and fundamental contributions of this work is the simultaneous attention paid to both visualizing and querying the structure of large datasets. The broad objective of this thesis is the presentation of both theoretical and practical contributions that exploit the synergism between the established field of database query languages and the emerging area of visualization. The *Hy*$^+$ *System*, and its application in several areas, demonstrate the capabilities of the original approach to structural data visualization that is presented in the thesis.

Hy$^+$ provides a user interface with extensive support for visualizing structural or relational data (as opposed to quantitative data) as *hygraphs* (of which labelled graphs are a special case). Hygraphs, a new visual formalism defined in this thesis, can be considered as a *hy*brid between Harel's *higraphs* [Har88] and directed *hy*pergraphs (and hence the name). Hygraphs constitute a convenient abstraction that generalizes several diagrammatic notations.

Hy$^+$ supports visualizations of the actual database instances and not just diagrammatic representations of the database schema. Hy$^+$ deals with the presentation of large volumes of data by resorting to two fundamental capabilities: the ability to *define* new relationships (or derived data), and an innovative way of using queries to decide what data to *show*. Using the second capability the user can selectively restrict the amount of information to be displayed. This *filtering* of irrelevant data is fundamental if one is to have any hope of conveying manageable volumes of visual information to the user. Selective data visualization can be used to locate relevant information, to restrict visualization to interesting portions of the data, and to control the level of detail at which the information is presented.

To describe queries, Hy$^+$ relies on a visual pattern-based notation. The patterns are expressions of the *GraphLog* visual query language [Con89, CM90b]. For the actual evaluation of queries Hy$^+$ makes use of deductive database technology, as described in [CMV94]. Overall, the system supports query visualization (i.e., presenting the description of the query using a visual notation), the (optional) visualization of the data that constitutes the input to the query, and the visual presentation of the result. The characteristics of Hy$^+$ are listed at the bottom of Figure 1.2. One fundamental difference with the other systems in the figure is that the patterns in Hy$^+$ describe *globally* which data is selected for the visual presentation.

In this thesis, we provide a characterization of *filter queries* and *filtering languages* in the

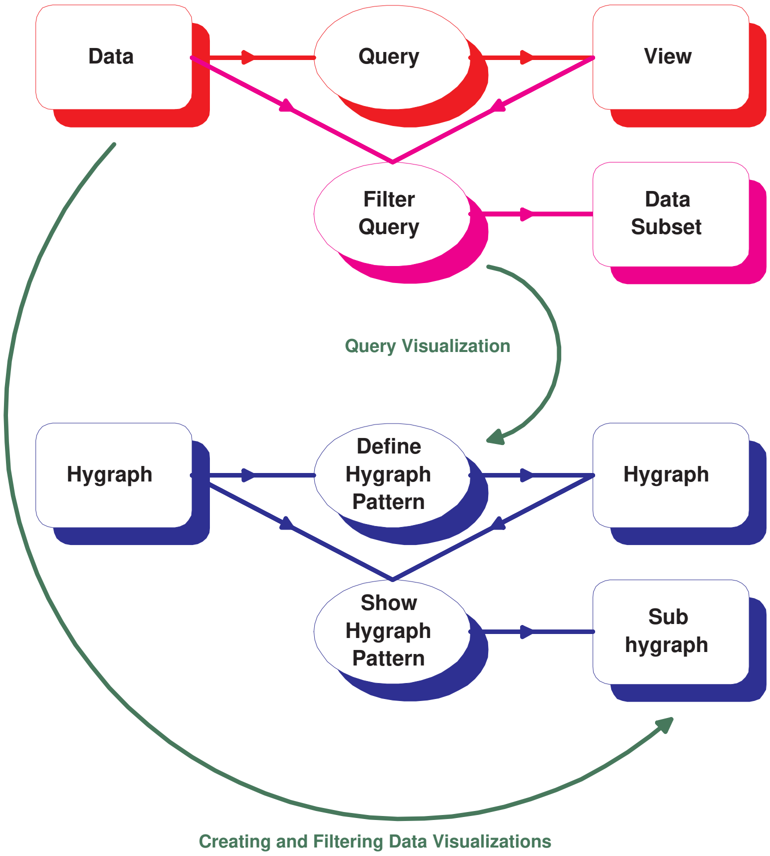


Figure 1.3: Overview of the thesis approach.

context of logic query languages (and therefore, directly applicable to the GraphLog query language used by Hy$^+$). Filter queries always return a subset of the data in the database, as opposed to traditional queries. For instance, in the relational model a traditional query may return a new relation that was not part of the original database, while a filter query will always produce a set of sub-instances of the relations in the input. Filter queries can be applied to a database that may or may not have views defined on top of it. A diagrammatic representation of the framework presented in this thesis appears in Figure 1.3.

Hy$^+$ and GraphLog have been successfully applied in areas where it is helpful to visualize the data using hygraph-based diagrams, such as: exploring `C++` source code [CM93a], formal software design documentation and object code overlay structure [CMR92]; browsing the structure of hypertext documents [CM89]; debugging distributed and parallel programs [CHM93]; and supporting network management [CH93].

1.4 Overview of the Thesis

In Chapter 2 we give an overview of the capabilities of Hy$^+$ from the perspective of a programmer exploring a large code library. The hygraph visual formalism is presented in Chapter 3. The formal definitions for filter queries and the filtering languages, as well as the hygraph pattern language used by Hy$^+$, are described in Chapter 4. The architecture of the system is presented in Chapter 5. Chapter 6 describes some application areas for Hy$^+$. We conclude in Chapter 7.

Chapter 2

A Tour of Hy$^+$

The purpose of this chapter is to provide an overview of the Hy$^+$system from a user's perspective. To do so, we consider a scenario in which a programmer uses the system to get acquainted with a sizeable library of object-oriented code. We have chosen the NIH public domain library [GOP90], a C++ [Str86] library that provides standard data structures by re-implementing portions of the Smalltalk [GR83] class library. Although for concreteness we use a specific programming language and library, some familiarity with object-oriented programming concepts is sufficient to understand the motivations for the examples.

To a large extent, the lure of the object-oriented programming paradigm is based on its promise to promote substantially higher levels of code reuse. Using object-oriented technology, programmers can tap into pre-existing shrink-wrapped class libraries, and specialize their behaviour to suit the specific programming task at hand without actually changing the original code. To deliver on this reuse promise, programmers must be able to *find* and *understand* the code to be reused.

The example library we selected, can be more appropriately considered as a *framework*, rather than just a simple class library. A framework is a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuse at a larger granularity than routines or classes [JF88]. Understanding a framework, therefore, requires knowledge of the overall design of a set of classes and how they cooperate. It is not enough to gain an understanding of each of the classes in isolation.

The programmer in our scenario can have different motivations for familiarizing herself with the library. She may be developing an application that will use library classes. In this case, although she is engaged in forward engineering, she needs to acquaint herself with the classes, their public interfaces and their typical usage patterns. In addition, she may be required to extend the library functionality, by further specializing classes which could eventually get reused in other applications. Any extension of the capabilities of an existing software system (in this case, a framework) involves a substantial initial investment to comprehend it (this is the discovery phase in the reverse engineering of the library). A variation of the latter situation occurs when the programmer's need for understanding the library's code is motivated by the requirement to maintain or even re-engineer several classes in the library. In this case, the programmer must also engage in substantial exploration of the library's code, and she will have to know the answer to several questions concerning the impact that her proposed changes will have on existing users of the library.

The ingredients in our scenario constitute an appropriate setting for benefiting from the use of Hy$^+$. We have large amounts of complex data (tens of thousands of lines of `C++` code) and a user engaged in an information-intensive problem solving activity (programming in the context of a framework). To carry out her task, she needs to visualize (in the literal sense of getting a mental image, and understanding of the information involved) the answers to a multitude of questions related to the complex information associated with the task. She must repeatedly cycle from coming up with the right questions to finding and understanding the answers.

In the rest of this chapter we describe how our hypothetical programmer can take advantage of Hy$^+$ to carry on a variety of tasks involving visualizing and querying large amounts of information.

2.1 Visualizing Data Describing C++ Code

The first step in using Hy$^+$ consists of making the data available to the system. Hy$^+$ is implemented as a front-end, written in Smalltalk, that communicates with other programs to carry on different tasks. In particular, query evaluation is done by one of several database back-ends running as separate processes. To run the queries in our scenario, Hy$^+$ communicates with the CORAL deductive database system [RRS92]. The data can be imported into Hy$^+$ (which takes care of making the data available to the database back-end) in several formats, one of them being text files with Prolog [CM81, MW88] facts which can be directly consulted by CORAL.

The IBM XL C++ compiler [JY92] was used to extract the data describing the NIH code. This particular compiler has an option to produce as part of the output for a compilation, a file with Prolog facts describing all the program entities and relationships present in the source code. The code browsers and other tools which are part of the compiler's programming environment [JMN$^+$92] submit queries to a Prolog server that consults the facts produced by the compiler. For our case study we use a simplified version of a subset of the predicates produced by the compiler. As a result, we obtain 13,000 facts from the 35,000 lines of code in the NIH library.

Figure 2.1 shows a screen shot of a Hy$^+$ session. The top leftmost window (labelled Hy+) is used to control the Hy$^+$ environment and open other windows. One of them is

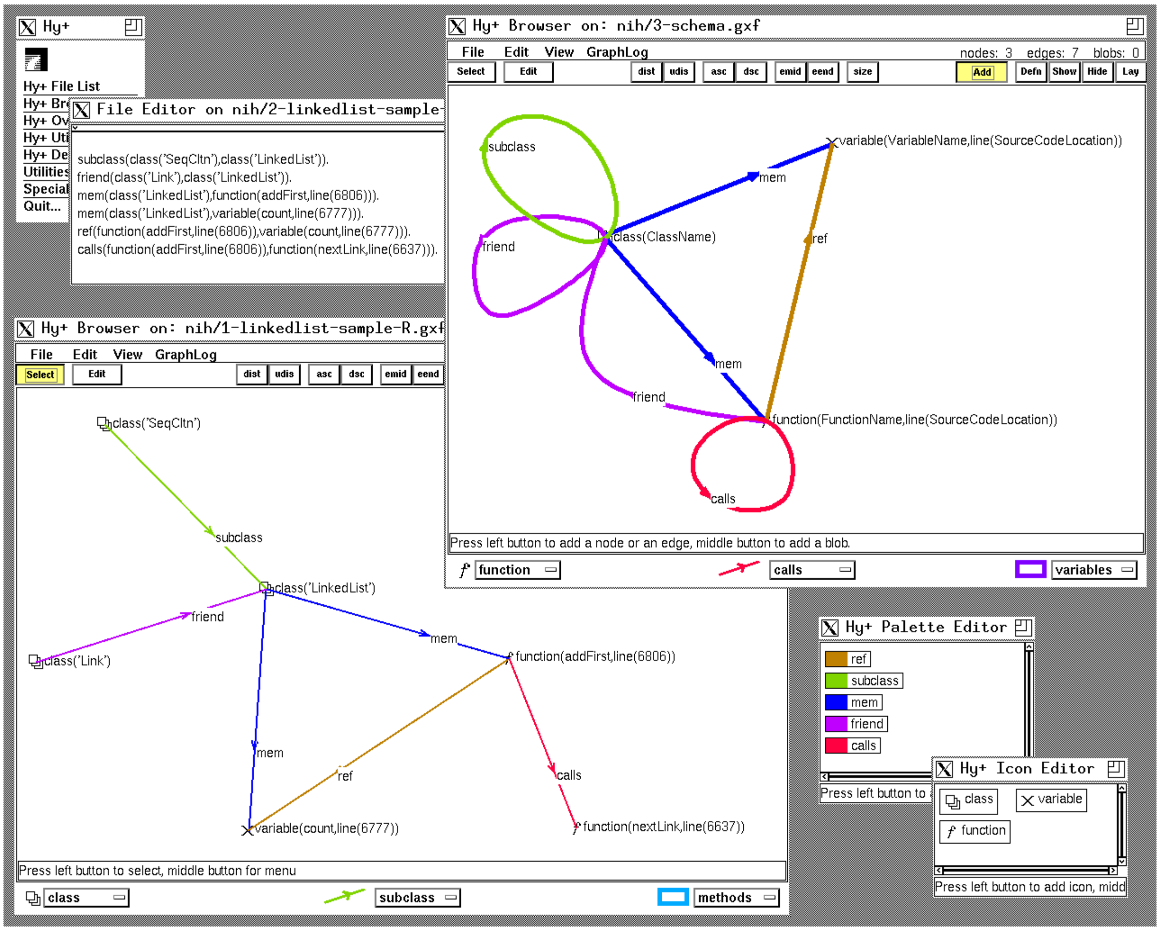


Figure 2.1: A Hy$^+$ screen for the NIH scenario.

a Hy+ File List (not shown in the figure), from which the user can open different editors on the contents of files. The window labelled File Editor shows six facts from the NIH database mentioned above. The same facts are displayed in the bottom left window by a Hy$^+$ browser that has extensive facilities for interactively editing hygraphs. The facts are visualized as a graph where the edges are labelled by predicates and the nodes by terms. We can see a green edge labelled subclass from a node labelled class('SeqCltn') to a node labelled class('LinkedList'), that states that the NIH class for linked lists is subclassed from the class of sequential collections. Similarly, the other edges indicate, depending on the predicate labelling them, whether a class is a friend of another class, or a function is a member of a class, or a variable is a member of a class, or a function references a variable, or a function calls another function. The facts discussed above are a sample of the database representing each one of the predicates in the schema. A visualization of the schema as a graph is shown in the top rightmost window (labelled Hy+ Browser) of Figure 2.1.

The graphs presented by the system make specific use of edge colors and node icons. Hy^+ assigns colors to edges based on the predicates in the edge labels and different icons to nodes based on the functors labelling the nodes. The Hy+ Palette Editor and Hy+ Icon Editor windows in the bottom right corner of the screen let the user select the colors corresponding to the predicates from a palette and pick icons for the functors by grabbing an image from anywhere in the screen. Color-coding relations and assigning different icons to nodes based on the objects they represent, are capabilities that Hy^+ makes directly available to end users for customizing the hygraph visualizations to the semantics of the application.

Visualizing the schema and a few facts, while helpful for the programmer to familiarize herself with the data, is not going to give her much insight into the characteristics of the library. The objective is to use Hy^+ to present hygraph visualizations that are informative. One such visualization, the class hierarchy, is quite familiar to programmers and is extensively used to graphically represent one aspect (the use of inheritance) of the structure of object-oriented programs. The class hierarchy is a directed acyclic graph (due to the occurrence of multiple inheritance) where the edges represent the subclass relationship.

Figure 2.2 displays the NIH class hierarchy, as it appears within a Hy+ Browser. The user can interactively select any object presented in a Hy^+ browser and change their loca-

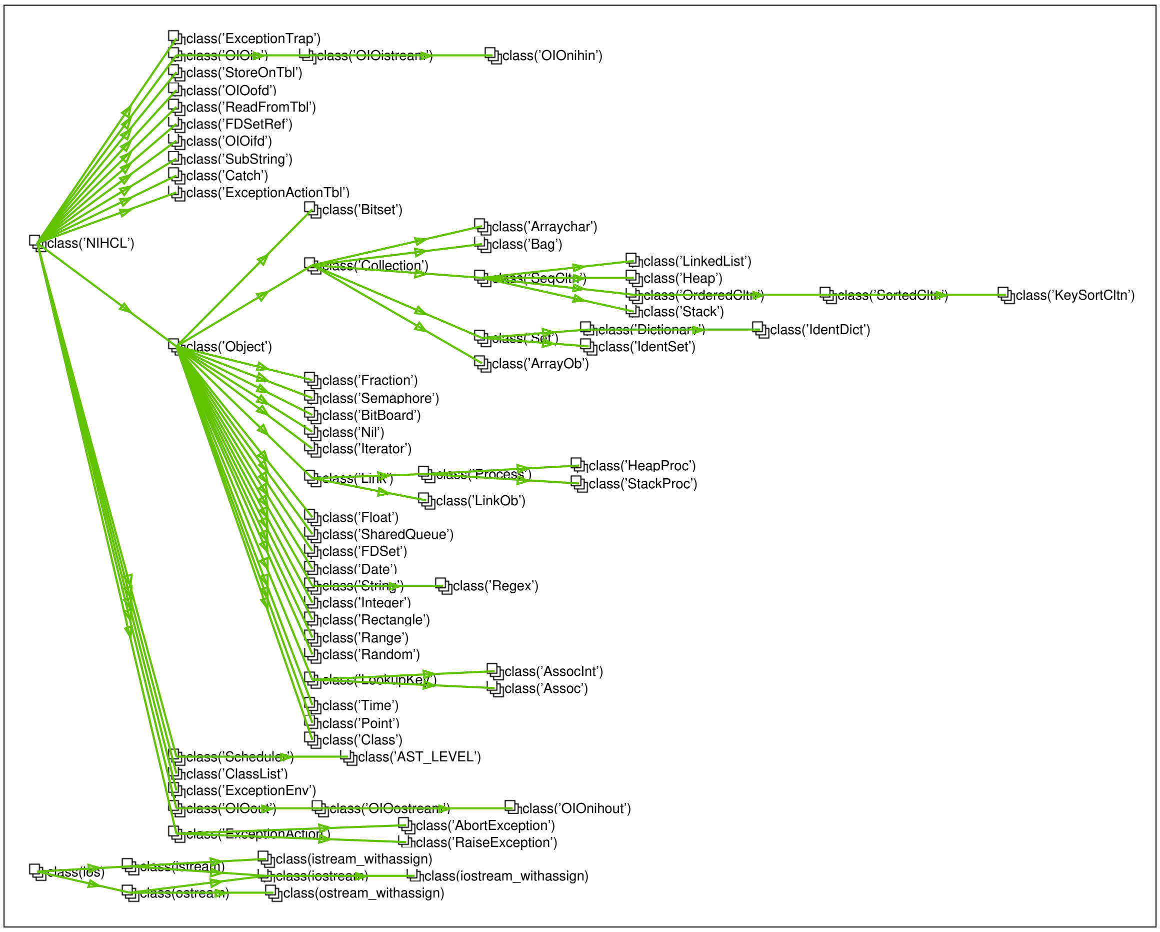


Figure 2.2: The NIH class hierarchy.

tion, visual appearance, or other properties. In this case, the user has chosen to display the textual labels of the nodes but not the edges, to avoid clutter.

Once the user is presented with a visualization, she can extract several pieces of information out of it. From the class hierarchy of Figure 2.2 we can get an idea of the number of classes (over 70), and the characteristics of the use of inheritance (reaches depth 6, with levels 2 and 3 having most of the classes). We can learn that class('Object') provides common protocol for more than half the classes in the library, and that class('Collection') is the superclass of an important subtree of the NIH class hierarchy. We can also observe that there is almost no use of multiple inheritance (only class(iostream) inherits from both class(istream) and class(ostream) at the bottom of the figure), which is not that surprising given the Smalltalk origins of the design of the library.

2.2 Simple Queries to Show and Define Data

What is unique to Hy^+ is the approach used to create visualizations such as the NIH class hierarchy shown before. The user specifies the graph to be displayed by following three steps. First she tells Hy^+ the location of the *database* (in this case the file with 13,000 facts that we discussed before). Second, she interactively edits in a hygraph browser a pattern like the one shown in Figure 2.3. The patterns are expressions of the *Graphlog* visual query language, which we introduce informally below by way of examples. Finally, the user executes the query and Hy^+ displays the answer in a new window.

The pattern in Figure 2.3 consists of a thick edge labelled subclass between two nodes labelled class(C1) and class(C2), enclosed in a box labelled showGraphLog. We will refer to this kind of boxes as *show boxes*, and to the hygraph inside them as *show patterns*. The meaning of the show pattern is as follows: match all facts of the form subclass(C1,C2) and display them as edges. The symbols C1 and C2 starting with capital letters are used (in the logic programming tradition) to denote variables. Consequently, all possible subclass edges are displayed, producing the visualization of the NIH class hierarchy shown in Figure 2.2. Since the database contains no layout information, the answer comes back origi-

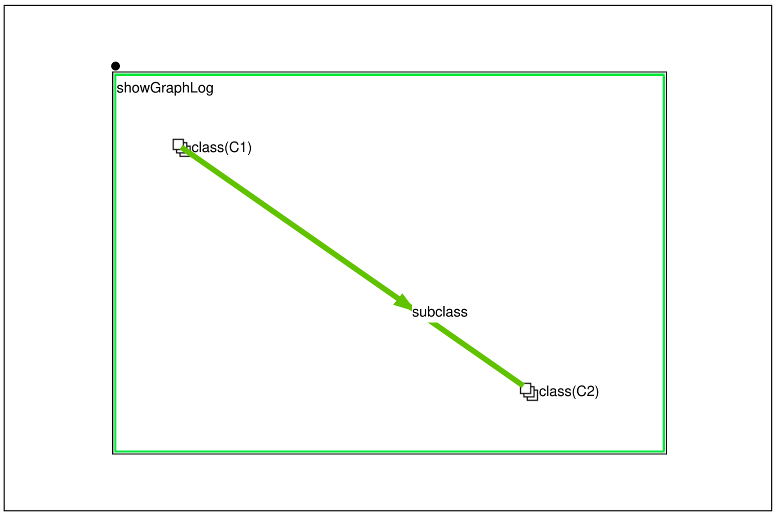


Figure 2.3: The Hy$^+$ query that displays the class hierarchy.

nally with random positions for the nodes of the graph. To obtain a horizontal tree view of the NIH class hierarchy, the user has the option of interactively invoking the execution of an appropriate graph layout algorithm, or specifying the layout using patterns (described later on).

The previous pattern can be easily modified to match a more specific subset of the facts. For instance, the user can edit the label of the node at the origin of the subclass edge, and change it to the constant class('Collection') (producing the show pattern in Figure 2.4). The result of executing this query appears in Figure 2.5. This time, only the subclass edges originating from the node labelled class('Collection') are displayed.

In addition to visualizing existing facts, Hy$^+$ has the ability to define new relationships. To do so, the user must edit a pattern within a box labelled defineGraphLog (we refer to this kind of boxes as *define boxes*, and to the hygraph inside them as *define patterns*). An example define pattern is presented in the leftmost box in Figure 2.6. The pattern con-

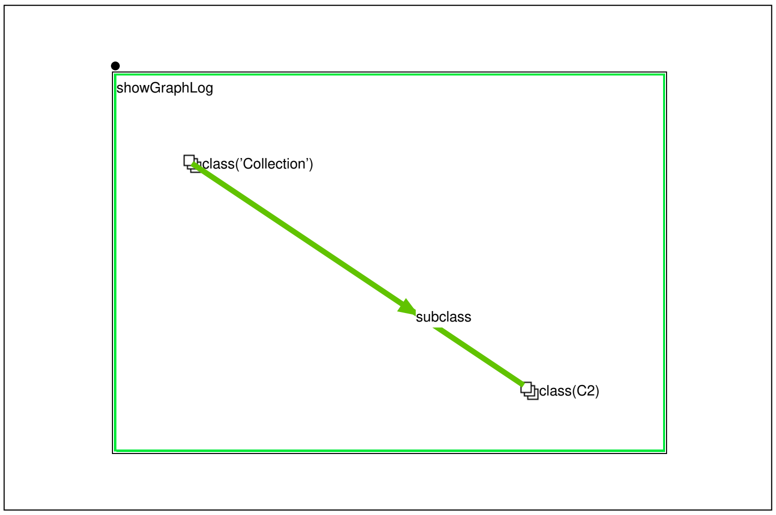


Figure 2.4: The pattern to show the subclasses of Collection.

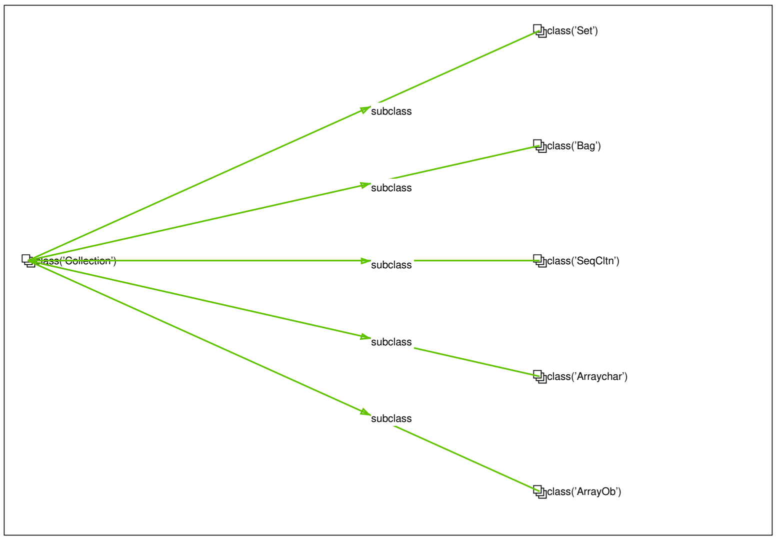


Figure 2.5: Displaying the direct subclasses of Collection.

sists of two nodes labelled class(C1) and class(C2) and two edges connecting them, labelled subclass+ and all_subclasses. Thickness is used to distinguish edges, therefore the edge labelled all_subclasses is known as a *distinguished edge*. The meaning of the define pattern is as follows. First match the transitive closure of the subclass relation, as indicated by the non-distinguished edge labelled subclass+ (where + is the closure operator). To reinforce the intuition that the subclass+ edge in the pattern is matching paths labelled subclass, Hy$^+$ dashes the edge. Second, for each pair of classes in the transitive closure of the subclass relation create a new edge labelled all_subclasses between them. The all_subclasses edges directly connect each class to all of its subclasses (both direct and indirect). These newly defined edges are then considered to be part of the current database.

The show box to the right of Figure 2.6 requests Hy$^+$ to display the all_subclasses edges just defined that originate at class('Collection'). Consequently, the result of the query in Figure 2.6 (containing one define and one show box) is presented in Figure 2.7 (to lay-out the result a non-hierarchical algorithm was interactively invoked).

Figure 2.8 presents a show pattern that matches paths of subclass edges starting at the node labelled class('Collection') and displays them in Figure 2.9. The result shows the inheritance subtree rooted at class Collection. This last example also introduces *layout boxes* (i.e., boxes labelled layoutGraphLog). We will describe them in detail later on; for now it suffices to say that an empty layout box is a convenient way of telling Hy$^+$ how to lay out an answer just before presenting it.

A final variation on the simple patterns presented to introduce GraphLog is shown on the left side of Figure 2.10 The show pattern in the example has a *distinguished node* (labelled class(C2)) and the edge is not thicker, as it was the case in all the previous show patterns. This illustrates how distinguished elements in show patterns are used to identify which objects from the matched pattern should be displayed in the answer, while leaving out

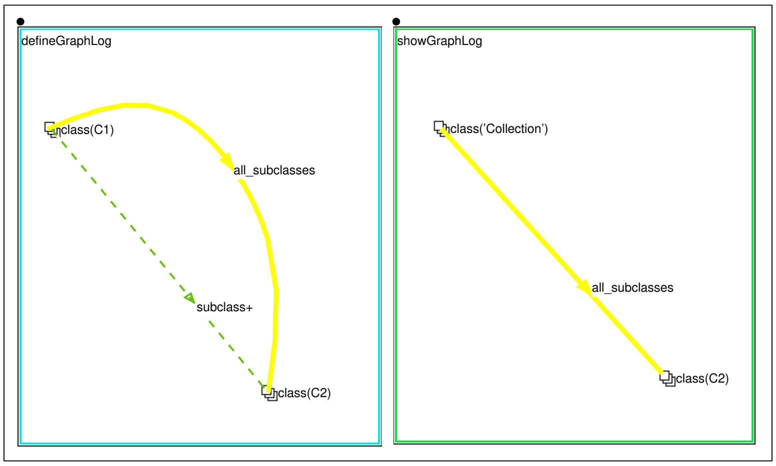


Figure 2.6: Defining and showing all_subclasses of Collection.

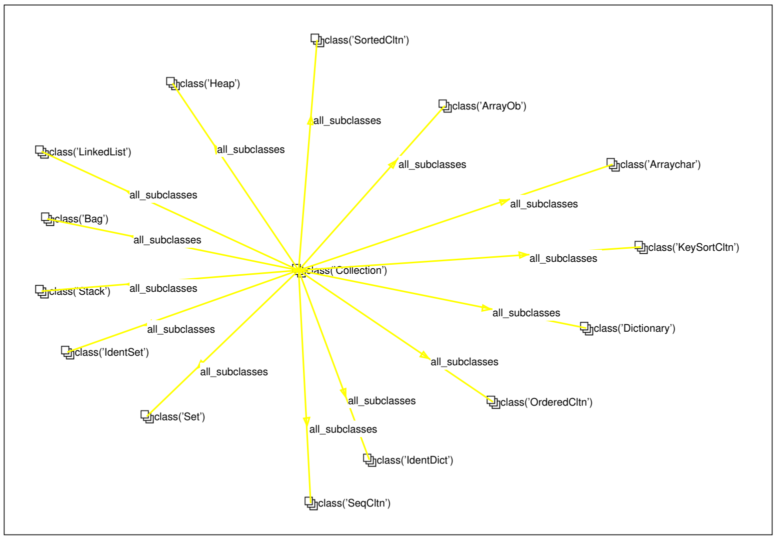


Figure 2.7: Displaying all_subclasses of Collection.

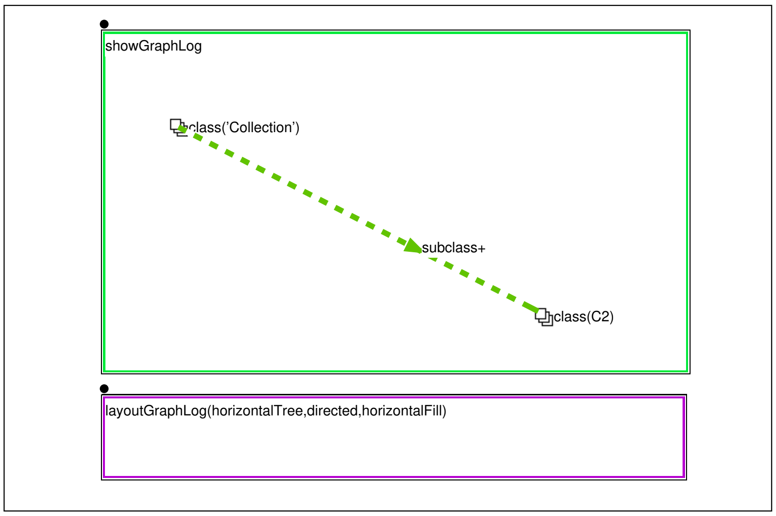


Figure 2.8: Filtering the inheritance subtree for Collection.

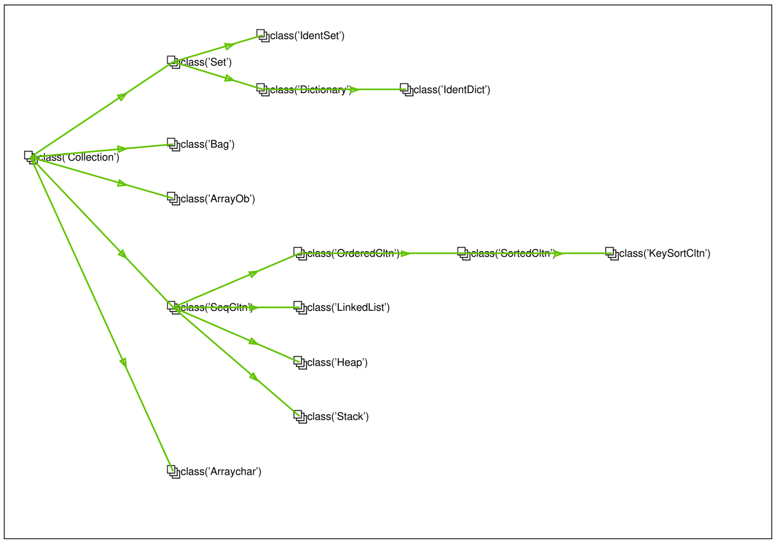


Figure 2.9: Displaying the tree of the subclasses of Collection.

Figure 2.10: Obtaining the set of subclasses of Collection as an answer.

the non-distinguished portions (in this case the remaining node and edge). Consequently, the result is simply a list of classes. While returning a set of objects as an answer produces an uninteresting visualization, it is nevertheless very useful if the user wishes to select those objects to further operate on them (in the example, the programmer can pass the list of selected classes to the `C++` compiler).

At this point, we provide a short description of the language used by the system to express queries. Hy$^+$ queries are sets of hygraph patterns with distinguished objects that are matched against a database. There are two kinds of patterns in a query: show and define (each appearing within the corresponding type of box). For each box in a query, Hy$^+$ evaluates the pattern inside it according to the type of box as follows.

Show: match all the objects in the pattern against the database, then display the distinguished objects.

Define: match the non-distinguished portions of the pattern against the database, then create new relationships for the distinguished edges and add them (as views) to the database.

The patterns are basically expressions of the GraphLog visual query language [Con89, CM90b]. The nodes are labelled by objects (encoded as first order terms) and edges and blobs are labelled by *path regular expressions* on relationships that optionally carry additional arguments (encoded as literals). A formal definition is given in Chapter 4.

2.3 Defining and Showing Hygraphs

It is time now to introduce *hygraphs*, which are formally defined in Section 3.1. Hygraphs are basically graphs augmented with *blobs*. A blob relates a containing node with a set of contained nodes. Blobs can be regarded as an alternative to edges for representing relationships among nodes: a blob replaces all the edges that would otherwise connect the container node of the blob with each of the nodes contained in the blob. Therefore, one advantage of blobs is that they reduce clutter in hygraph diagrams (one blob containing n nodes is required in lieu of n edges). Another desirable property of blobs is that they force the clustering of all the contained nodes within the container node.

In short, a hygraph has nodes that represent objects, and has both edges and blobs to represent relationships among those objects. In terms of the relationships they represent, edges and blobs are completely exchangeable.

Hy$^+$ employs one particular graphic design for displaying hygraphs. We refer to Figure 2.11 for an example hygraph that has two blobs associated with the node labelled class('Bag'). Both blobs are enclosed in a rectangular area associated with the node (which is known as the *blobs region* of the node, and is the area where all blobs associated with the node are

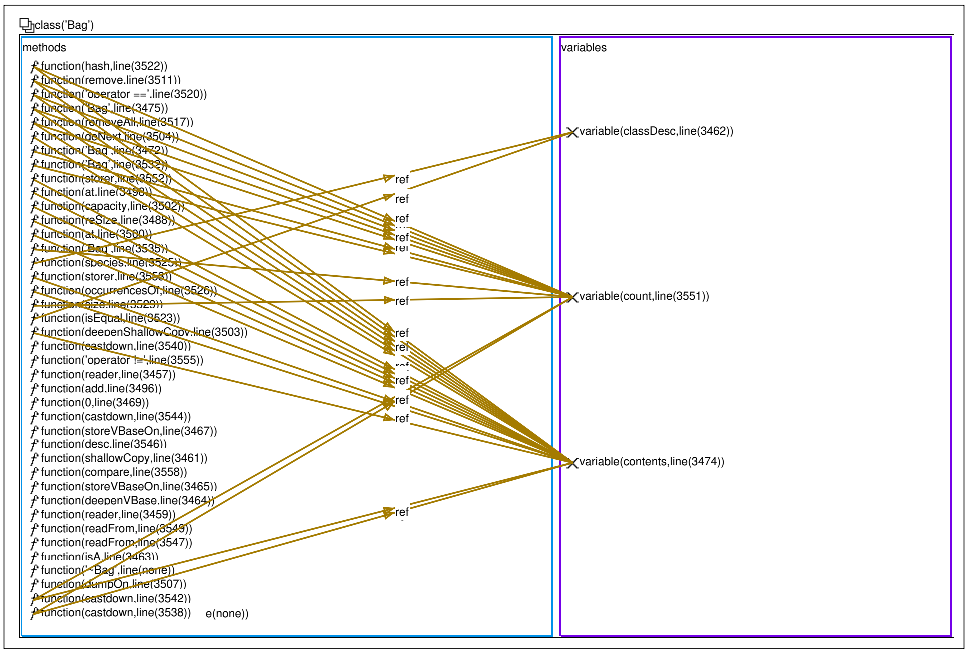


Figure 2.11: The methods and variables blobs of class('Bag').

placed). The blobs themselves are represented by rectangles. Hy^{+} assigns colors to blob boxes according to the relationship labelling them, in the same way colors are assigned to edges. The left blob is labelled methods, and it contains forty nodes inside representing the member functions (or methods) of class Bag. The right blob, labelled variables, contains three nodes representing the member variables of class Bag. In addition, the hygraph has several ref edges pointing from functions to the variables they reference. Notice that edges can connect any pair of nodes in a hygraph regardless of which blobs (if any) contain the nodes.

Blobs, like edges, are labelled by objects that describe the nature of the relationship represented by them. Hence, in the same way that a node n_1 may have multiple outgoing edges,

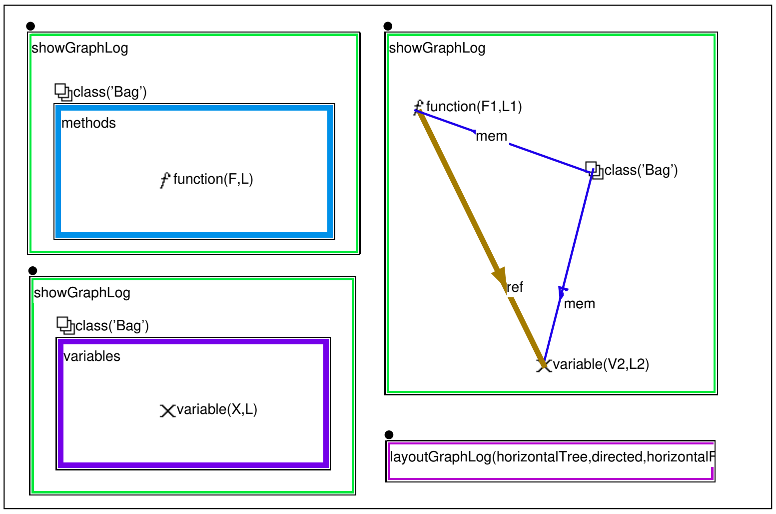


Figure 2.12: The query to produce the hygraph for Bag.

a node n_1 may have multiple blobs (but with different labels) for which n_1 is the container. Notice that it is not allowed to have two blobs b_1 and b_2 with the same label l and the same container node, since one blob labelled l with the union of the nodes contained in b_1 and b_2 represents exactly the same relationship. Since one of the purposes of using blobs is to reduce the number of objects that are required for representing relationships among nodes, hygraphs are required to have only one blob for a given container node and blob label. Otherwise, we could have potentially as many blobs as the number of subsets in which we can partition the set of contained nodes.

Creating a hygraph visualization in Hy$^+$ involves exactly the same steps required to produce a plain graph. The only difference is that the show patterns will have distinguished blobs. To produce the hygraph for class Bag the three show patterns in Figure 2.12 are used. The pattern in the top left corner matches the methods blob for class('Bag') containing any member function function(F,L) and displays them. The bottom left pattern works similarly to display the variables blob. The show pattern on the right matches the member functions of Bag that reference member variables also of Bag, but displays only the ref edges (since this is the only distinguished edge in the pattern). The reader may wonder why two show patterns are needed to display the methods and variables bobs, instead of just one with both blobs (like the hygraph of Figure 2.11). For class Bag the answer is that the combined pattern would do just fine, but that happens because Bag has both methods and variables. For instance, a pattern with both methods and variables blobs will not match (and, consequently, will not display) a class that has no instance variables, as it is the case for an abstract base class.

Since the relationships methods and variables are not part of the original facts we need to add them to the database before the show patterns in Figure 2.12 are executed. The define patterns to accomplish this are shown in Figure 2.13. The top define pattern specifies that whenever the database contains a mem edge representing the fact that function(F,L) is a member of class(C), a new methods blob representing the same fact should be considered part of the database. The bottom pattern defines variables blobs in a similar way.

Actually, Hy^+ is indifferent to whether a blob or an edge is used anywhere in a define pattern, or anywhere in the non-distinguished portion of a show pattern. In both cases, Hy^+ only cares about matching the label of the edge or blob with a suitable fact in the database. Similarly, for the distinguished edges or blobs of a define pattern, Hy^+ adds the appropriate facts as a view to the database. Of course, Hy^+ does take into account whether a fact matches an edge or a blob in the distinguished portion of a show pattern since the system will choose whether to display the fact as an edge or as a blob based on that. In case a show pattern matches one fact as both an edge and a blob, Hy^+ displays the fact in both ways.

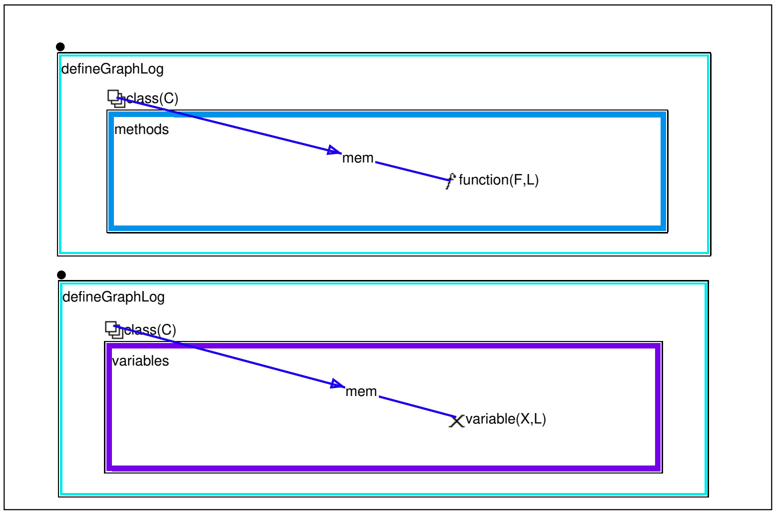


Figure 2.13: The patterns defining the methods and variables blobs.

2.4 Exploring Large Hygraph Visualizations

Now that we have covered the basics of Hy$^+$ queries to create and visualize hygraphs, let us consider creating a diagram that will be useful for the programmer in our NIH scenario.

Since the hygraph diagram of a class with blobs for methods and variables presented in the previous section provides useful information in a readable fashion, it seems natural to generate a diagram that makes use of this representation for specific subsets of the NIH classes. As an example, suppose that the programmer is interested in exploring all the (direct and indirect) subclasses of Collection. This time, in addition to variable references the programmer also wants to see the function calls. Notice that in the hygraph for class Bag we limited the ref edges to references from a method of Bag to an instance variable of Bag. Similarly, we restrict the calls edges displayed to those between classes in the inheritance subtree for class('Collection'). The Hy$^+$ query is shown in Figure 2.14.

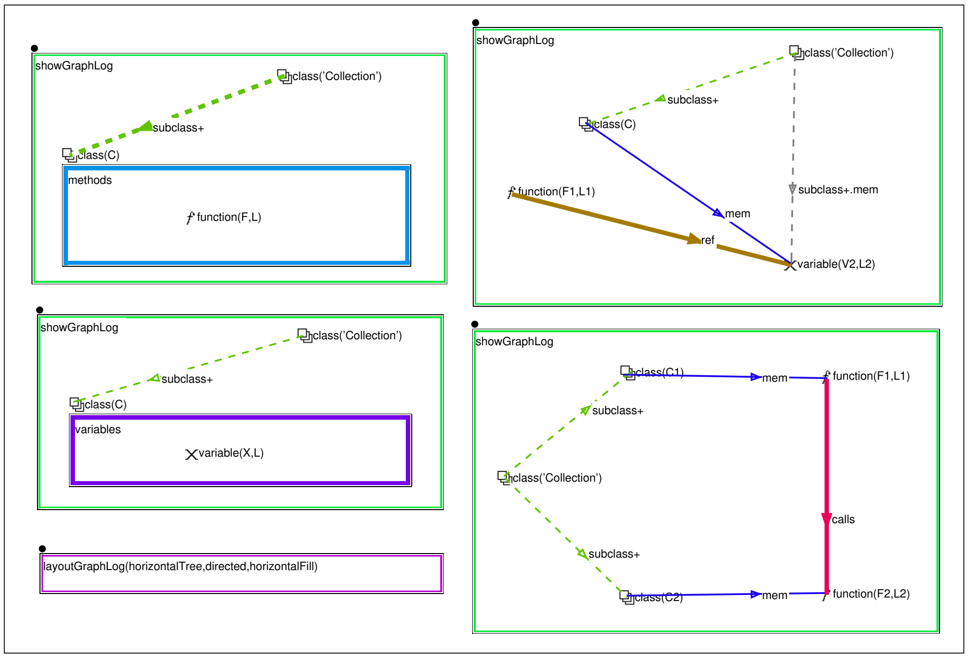


Figure 2.14: The patterns that create a large hygraph visualization.

The top right show pattern contains one redundant edge labelled with the regular expression subclass+.mem (involving a closure and a concatenation). The presence of this edge is equivalent to the presence of both the subclass+ edge from class('Collection') to class(C) and the mem edge from class(C) to variable(V2,L2). The purpose of this redundancy is to bring to the attention of the readers the way in which path regular expressions can be used to reduce the size of a pattern. In fact, GraphLog exhibits the property that regular expressions (with the exception of the transitive closure operator) do not add any expressive power to the language: they are available for succinctness only, and they can always be replaced. A nice consequence of the above, is that the user has control over whether the appearance of her patterns favors the visual aspects (using more nodes, edges

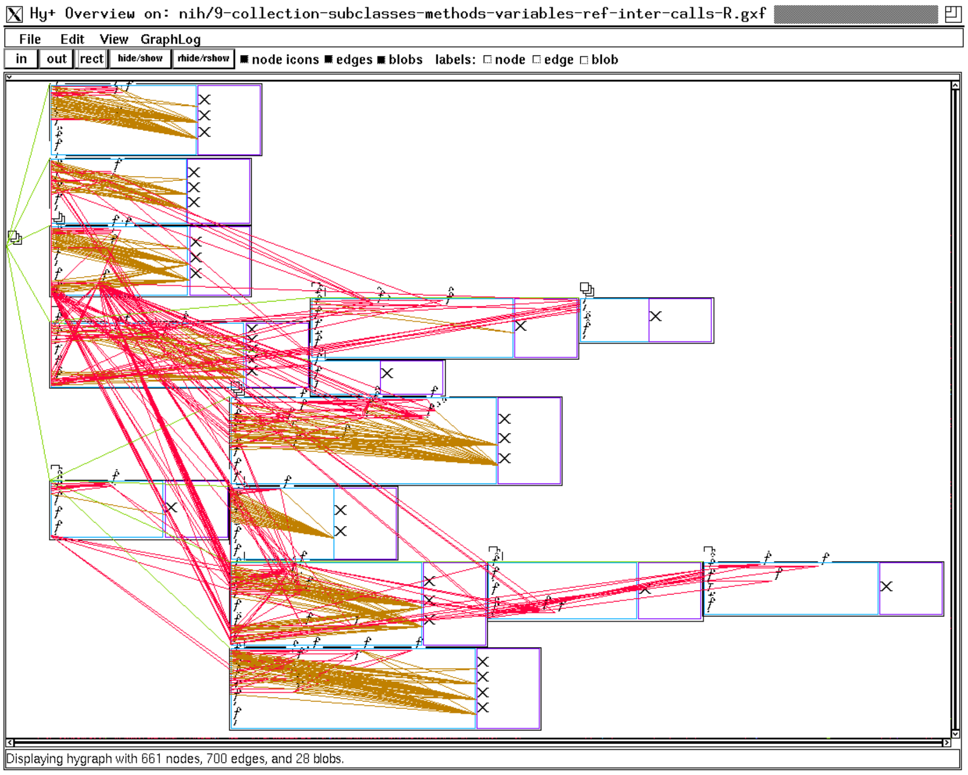


Figure 2.15: Browsing a large hygraph.

and blobs), or the textual ones instead (using longer regular expressions as labels). Notice that the existence of a choice is a property of GraphLog that is in contrast with the use of regular expressions to search for patterns in text.

The answer, shown in Figure 2.15, illustrates how Hy$^+$ supports the browsing of large hygraph visualizations. The system presents the result in a Hy+ Overview window that supports fast display, scrolling and zooming of large hygraphs. This example has over a thousand objects displayed. As an aside, notice how edges cross blob boundaries arbitrarily.

The hygraph in Figure 2.15 conveys a large volume of information coming from several

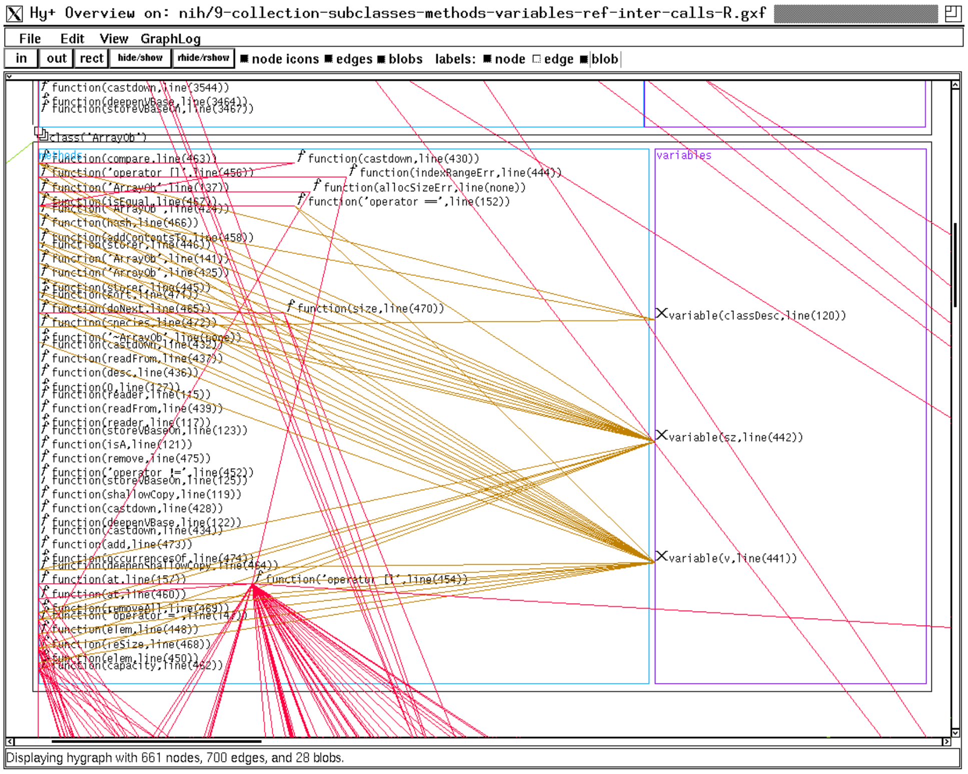


Figure 2.16: Zooming into a smaller region to appreciate details.

thousand lines of the NIH source code in one screen. For the programmer exploring NIH, this kind of overview of the structure of the library provides significant contextual information. The overview can also be quite useful to decide to focus on a specific aspect that may otherwise go unnoticed. For example, the programmer may observe that there is unusually higher call activity around two groups of functions at the bottom of the third box from the top. She then decides to zoom into that area and turn on the text labels (this view appears in Figure 2.16). This action lets her realize that the heavily called functions are members of class ArrayOb (among others, operator[] is called a lot). This is not surprising (for some-

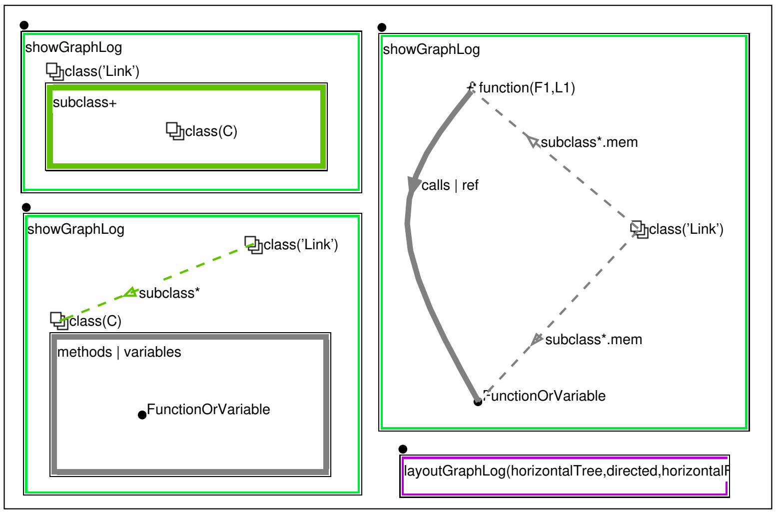


Figure 2.17: The query to produce a hygraph for the subclasses of Link.

body who is already familiar with NIH) since arrays are used in the implementation of several of the container classes in the Collection subtree.

The next example is a variation of the previous one. The programmer is interested now in the subclasses of Link. In addition she decides to look at the subclass relationship as blobs instead of edges. The expression she uses appears in Figure 2.17. This query also illustrates how regular expressions with disjunctions reduce the number of patterns the user must draw.

Representing the subclass relationship as blobs produces a visualization with nested blobs, as shown in the top portion of Figure 2.18. The hygraph at the bottom shows the result of interactively *hiding contents* of blobs, in this case all the variables blobs. As a side effect of hiding blob contents, the incoming and outgoing edges to and from the nodes contained in the hidden blobs are not shown. This may produce a drastic reduction of the information displayed. If desired, the user has several choices for displaying edges that summarize the information not shown because of hiding. And, of course, more flexibility is available by

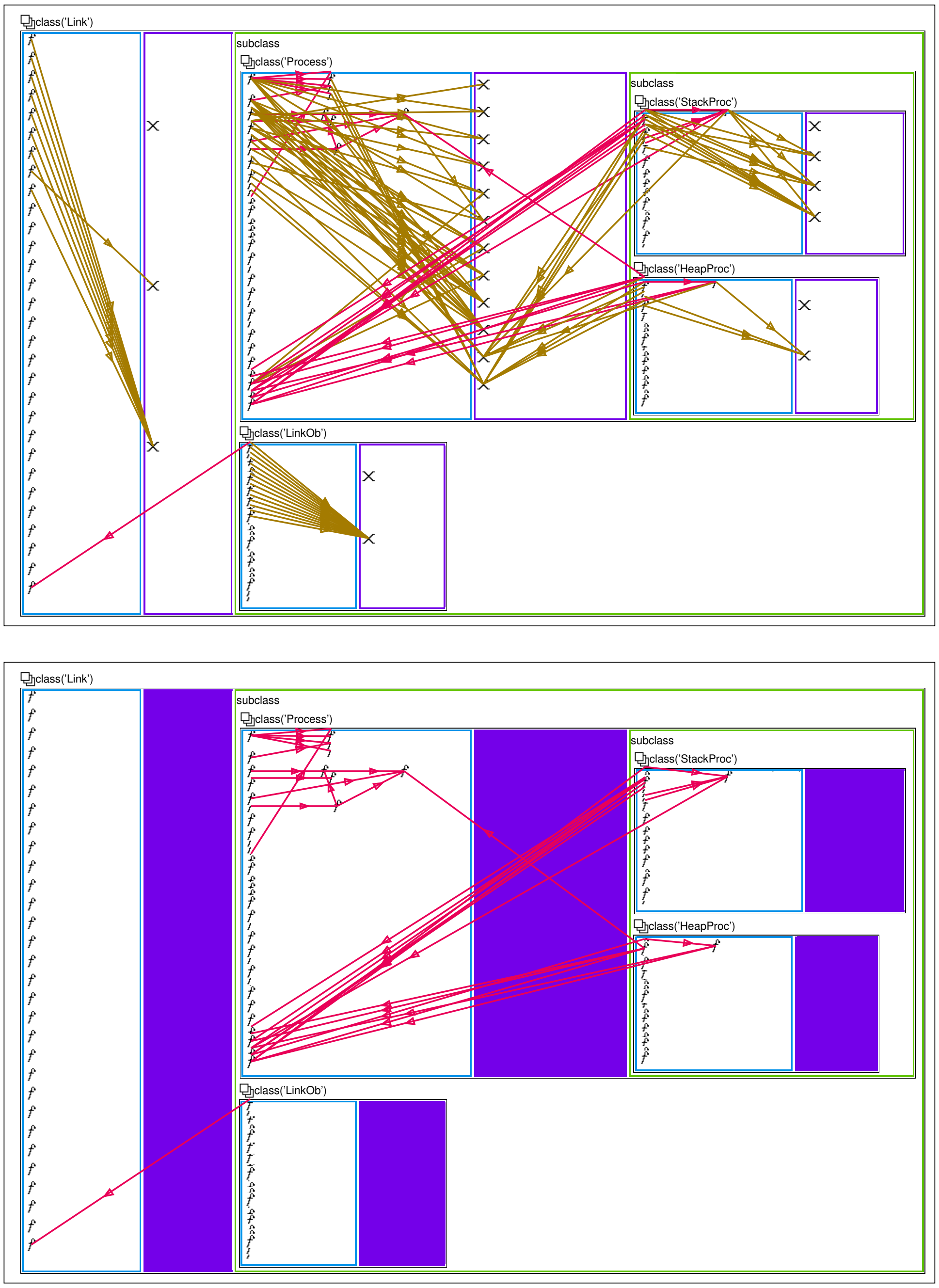


Figure 2.18: The result with and without hiding the variables blobs.

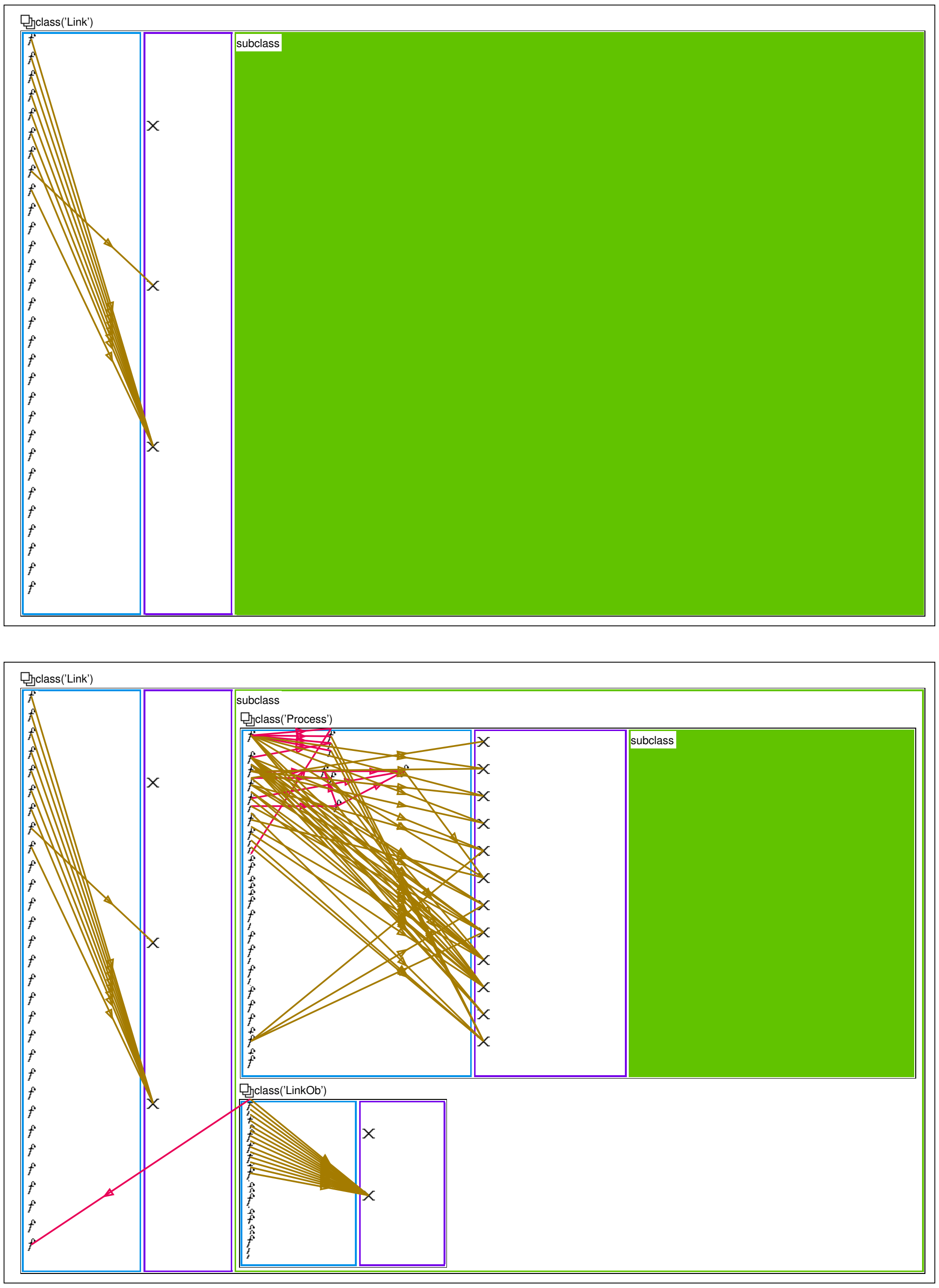


Figure 2.19: Recursively hiding nested subclass blobs.

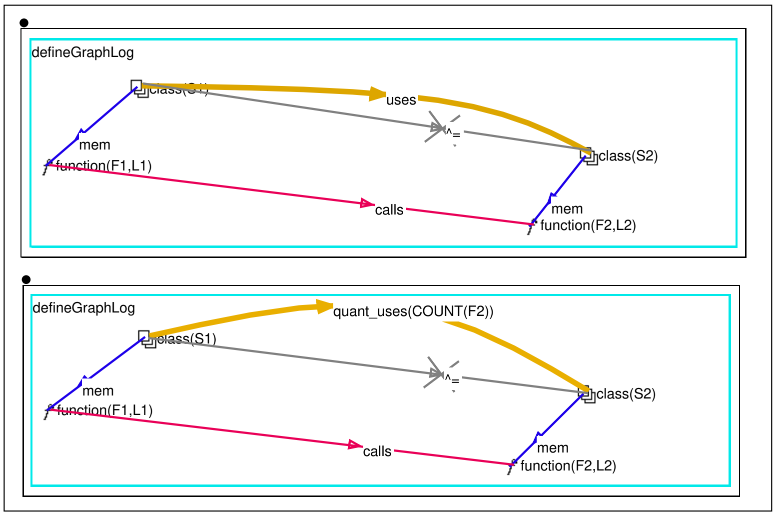


Figure 2.20: Definitions for usage and coupling relationships.

defining ad-hoc ways of summarizing the information of hidden objects through the use of queries. Hy^{+} also supports the recursive hiding of the contents of nested blobs, which provides one way of suppressing details at different levels of a hierarchy (this is illustrated in Figure 2.19).

We have not encountered yet a situation in which a node is contained within two or more blobs. We will see later on that when this happens Hy^{+} still displays a strict hierarchy of non-overlapping regions to represent blobs. This is achieved by creating as many occurrences of a node (all with the same label) as there are blobs containing the original node.

2.5 Advanced Queries and Selective Layout

An important relationship among the classes of an object-oriented program is *usage* (also known as a *client-server* relation). One form of usage among pairs of classes occurs when one of the classes (called the client) has a function that calls another function in the second class (called the server or supplier). Another form of usage occurs when a client class has an instance variable of the supplier class type. The top define pattern in Figure 2.20 creates a new edge labelled uses that points from the client to the server when the first kind of usage occurs. Note that there is a crossed-over edge labelled ˆ =, a graphical notation that reinforces the intuition that an edge labelled = must not be present in the matched pattern (negation is denoted by ˆ). In this case the negated relationship is equality, but it could have been any other kind of edge.

Once the usage relation among two classes is established, it is of interest to attempt a quantification of how much the classes know about each other. The concept we are referring to is known as a measure of *coupling*. The pattern at the bottom of Figure 2.20 defines such a measure by creating an edge labelled quant_uses(C) that carries an additional argument C that represents the count of the number of functions called in the supplier class. The query uses the *aggregate* operator COUNT with an argument F2 to define the value of the argument C of the edge quant_uses(C) as the number of values of F2 that match the pattern for a given pair of classes S1 and S2. GraphLog supports other aggregate operators in addition to COUNT, such as MAX, MIN and SUM.

The measure we have just defined provides an indication of how closely coupled two classes are[1]. The programmer would like to get a general impression of the degree of coupling among the classes in the library. While coupling in NIH reflects the level of code reuse that the library designers were able to achieve, from the point of view of a programmer who must learn about a bunch of new classes it seems less daunting if the classes are not tightly coupled (and therefore can be understood fairly independently). To obtain an appropriate

[1]This particular measure may also indicate that the design of the supplier class forces the clients to call several different functions in the public interface of the supplier.

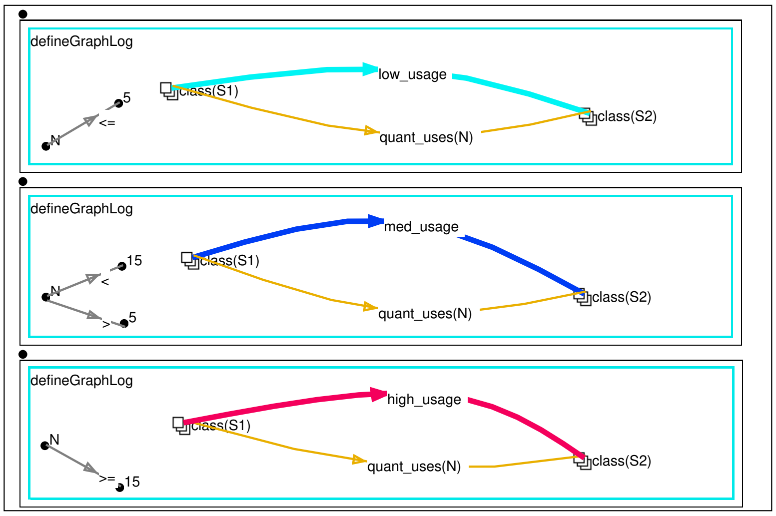


Figure 2.21: Color-coding three degrees of coupling.

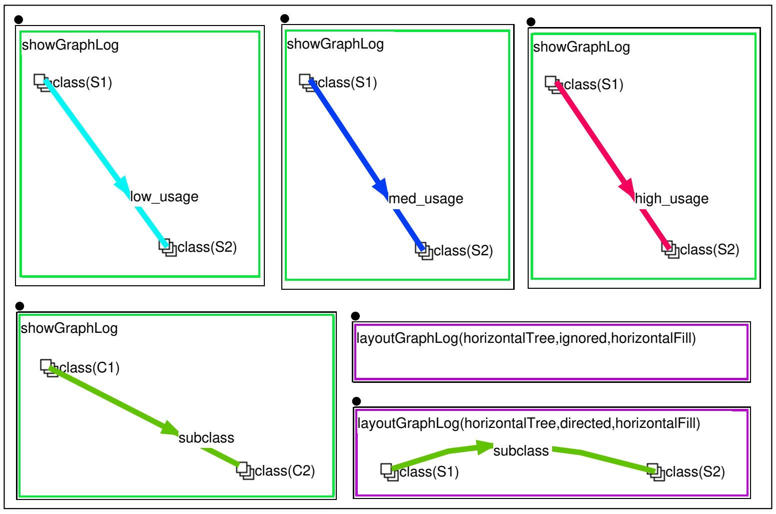


Figure 2.22: Superimposing degrees of coupling over the class hierarchy.

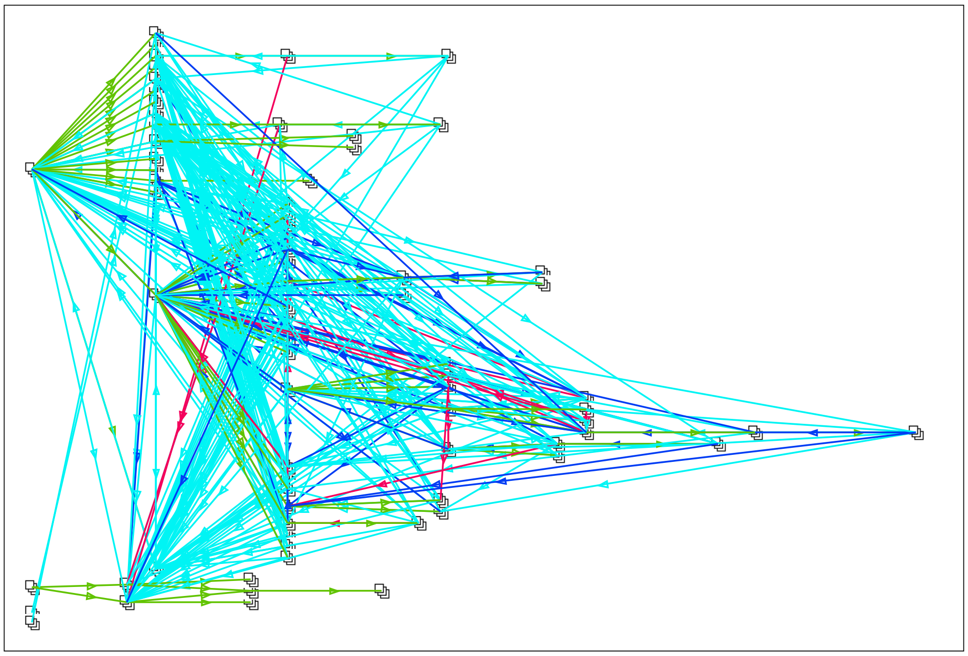


Figure 2.23: Visualizing the degree of coupling among NIH classes.

visualization, the programmer first defines three edges to color-code different degrees of coupling. Figure 2.21 defines low_usage as coupling with a count less than or equal to 5, med_usage when the count is between 5 and 15, and high_usage when the count is greater than or equal to 15.

The programmer wants this coupling information superimposed on top of the class hierarchy. This is achieved by the show and layout boxes in Figure 2.22. The bottom right layout box contains a pattern with a distinguished subclass edge. The meaning of this layout pattern is to associate the layout information in the label of the box with the matched objects. In this case, the subclass edges are labelled directed. The empty layout box sets the default layout for all other edges as ignored. The combined effect of the two layout boxes

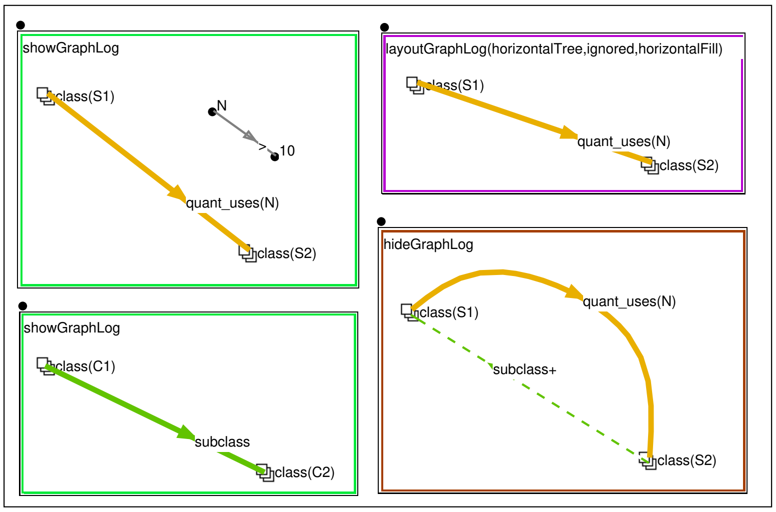


Figure 2.24: Selectively hiding usage.

is that the subclass edges (considered in their normal direction) are laid out as a horizontal tree (therefore positioning the classes according to the class hierarchy), while all the other edges are completely ignored during layout (hence, they are simply superimposed on top of the inheritance tree). The result can be appreciated in Figure 2.23. A discussion of layout boxes is presented in Chapter 5.

The previous visualization of the degrees of coupling among NIH classes may seem discouraging for the programmer since it indicates a significant amount of class usage in the library. However, most of the edges correspond to low_usage. It would seem reasonable to try to produce a diagram where only the relationships among closely related classes are shown. Figure 2.24 does precisely this by displaying quant_uses edges with a count higher than 10, once more on top of the class hierarchy. But, in addition to restricting the usage edges displayed based on the degree of coupling, a *hide box* (the one labelled hide-

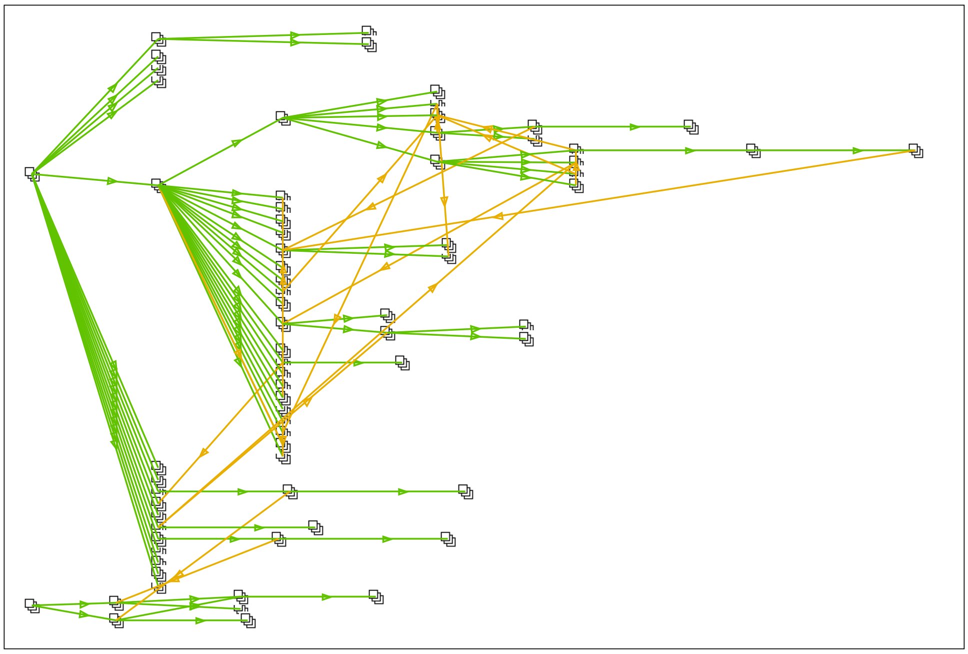


Figure 2.25: The result of selectively hiding usage.

GraphLog) is used to eliminate[2] from the final diagram usage relationships going upward in the class hierarchy. The reasoning behind not paying attention to tight coupling along the inheritance tree is that usage is actually expected to occur, so we avoid cluttering the picture by not displaying the obvious. The result, shown in Figure 2.25, attests to how effectively we filtered the information in the original cluttered view (the one in Figure 2.23 with all the usage relationships in NIH) to concentrate on the most interesting ones.

[2]The meaning of a hide box is to match the pattern against the visualization produced by the accompanying show boxes, and then delete the matches from the final result.

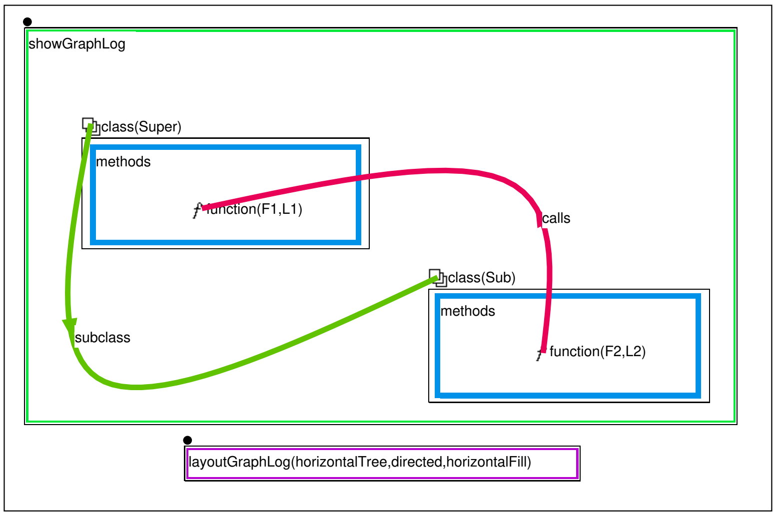


Figure 2.26: Filtering function calls to superclasses.

2.6 Iterative Filtering

We have been showing how the programmer exploring the NIH code can create several diagrams to help her visualize the structure of the code in the library. But, so far we have always used the entire dataset produced by the compiler to create the views. Clearly the task of finding adequate ways of looking at the data does not have to start from scratch every time; it can be approached as an iterative process. We may decide to look for something in a previously created visualization. As an example, consider locating all the calls to functions in the superclasses among the subset of the information visualized in the large hygraph of Figure 2.15. To do so, the user indicates to Hy^{+} that she wants to use the hygraph just mentioned as the current database. The show pattern displaying calls to superclasses is shown in Figure 2.26. Because all the relevant objects are distinguished in the query, the result displayed in Figure 2.27 provides the appropriate context for the answer (as opposed

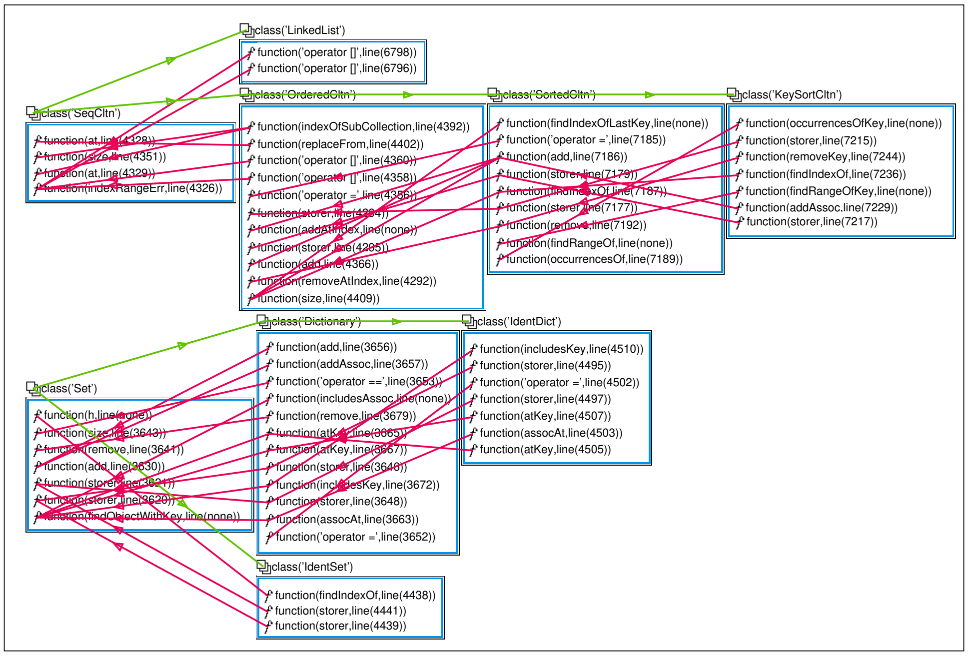


Figure 2.27: Showing context information for calls to superclasses.

to just a list of function calls). We have just seen another instance in which specific filtering criteria expressed as a Hy$^+$ query are quite successful in drastically reducing the amount of information displayed (compare the input visualization in Figure 2.15 with the filtered output shown in Figure 2.27).

2.7 Pattern-based Selection in Editors

The answer to the previous query finding the function calls to superclasses returns many instances of a more specific kind of function call: calls from a member function being redefined to the member function in the superclass that is being overridden. This situation is not

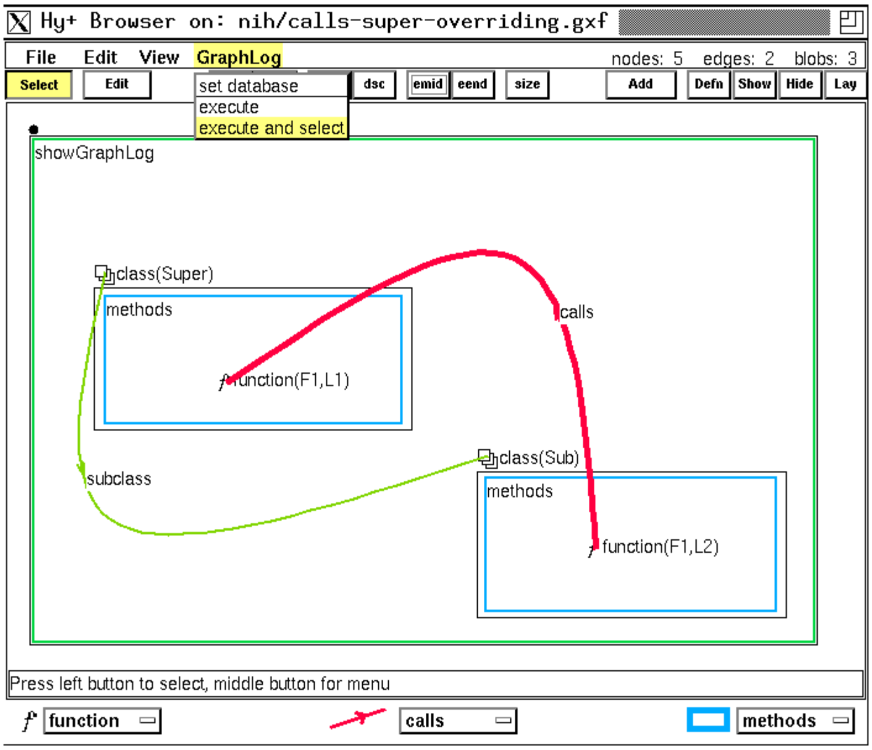


Figure 2.28: Pattern-based selection of overriding calls.

unusual in object-oriented code. It arises when a method in a class that overrides the method with the same name in the superclass invokes the code in the superclass (and usually adds some more processing of its own afterwards). Modifying the query in Figure 2.26 so that the variable F1 is the same for the two function names in the pattern achieves the desired result of finding the overriding calls. The new pattern is shown in Figure 2.28, and this time the only object distinguished is the calls edge.

If the user executes the query in Figure 2.28 as usual, the answer will consist of a graph of calls edges. Instead of doing so, the user decides to set the hygraph in Figure 2.27 as the current database and then executes the query in the execute and select mode. In this

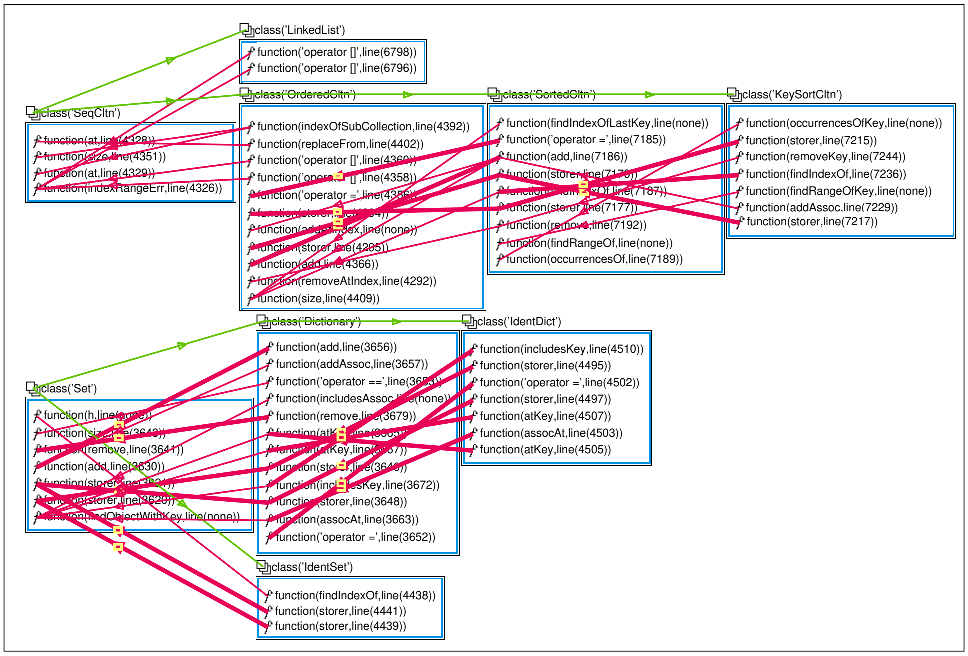


Figure 2.29: Highlighting the selection of overriding calls.

mode, Hy$^+$ does not return the answer in a new window. Instead, the system *selects* in the current database all the objects that are returned in the answer to the query. Once the objects are selected, any one of the operations available in a Hy+ Browser can be applied to them. In the example shown in Figure 2.29, the user chooses to distinguish the objects selected as a result of the query. Hence, a subset of the calls edges in the diagram are displayed as thick edges (in addition to have the selection boxes around the arrowheads, to show that they are still the current selection of the editor). It is important to stress the flexibility that the execute and select mode gives to hygraph editors since most of their operations apply to selected objects (like removing, coping and pasting, hiding contents, changing the way in which labels are displayed, and so on).

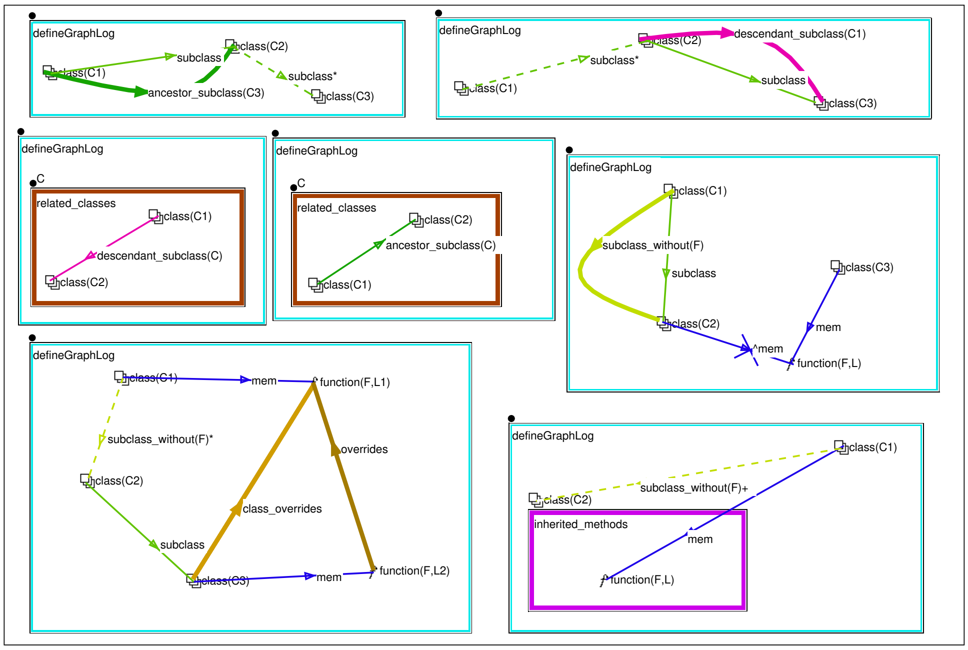


Figure 2.30: Definitions for an inheritance browser.

2.8 Discovering Visualizations

We conclude the tour of Hy$^+$ presenting one visualization that attempts to give an overall impression of the use of inheritance in any `C++` program. The queries in Figures 2.30 and 2.31 produce the visualization shown in Figure 2.32 presenting a picture of the use of inheritance among the subclasses and superclasses of the NIH class SeqCltn (the class of sequential collections). The diagram presents all the inherited methods in the leftmost blob, and all the methods defined in the class itself in the rightmost blob. The blob in the center has a tree with all the superclasses connected by edges of one color, and all the subclasses connected by edges of another color. In addition, there are two more kinds of edges in the diagram. The first connects the inherited methods to the superclass where they are defined.

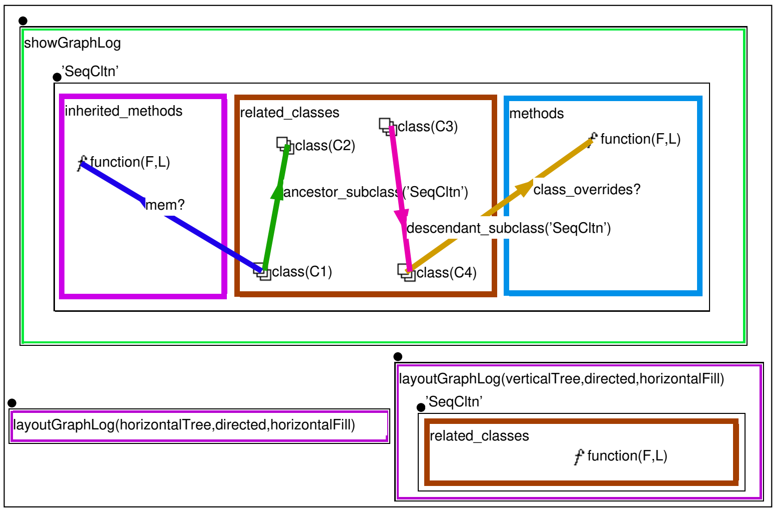


Figure 2.31: The pattern to display the inheritance browser.

The second connects the methods of class('SeqCltn') to the subclasses that override them. The result conveys an image of how much inherited code is used in a given class, as well as how much of the code in the class is later redefined in the subclasses.

Clearly, the queries that produce this visualization (as well as some of the others we have seen in this chapter) can be applied to any C++ program. One can readily see the advantage of providing Hy$^+$ with libraries of parametric queries that are applicable to a given domain. It is interesting to look at the process of creating such a collection of queries itself. Discovering visualizations such as the view of inheritance described above can be greatly simplified by the support provided by Hy$^+$ for experimenting with different alternatives.

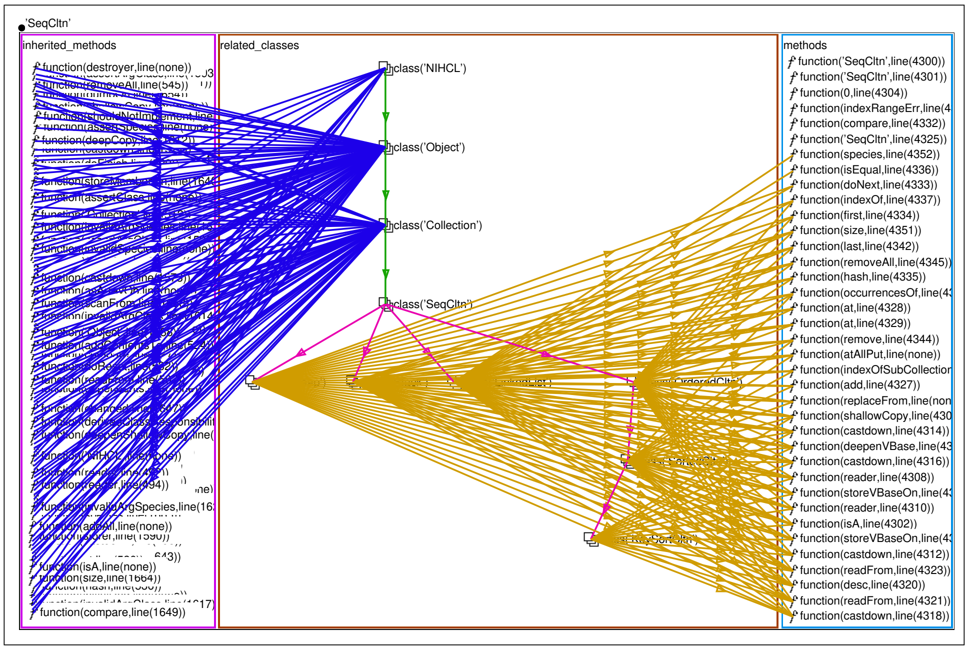


Figure 2.32: The inheritance browser for class('SeqCltn').

Chapter 3

The Hygraph Visual Formalism

In this chapter we present the definition of the hygraph visual formalism. We argue that hygraphs are a simple abstraction for a wide variety of diagrammatic notations. In particular, they share with labelled graphs the property that the actual semantics given to the relationships represented by nodes and edges (and blobs as well, in the case of hygraphs) is based on an interpretation of the labels which is dependent on the application. In this way, hygraphs are suitable for a broad spectrum of situations that range from representing completely abstract concepts (e.g., a derivation tree) to the most concrete devices (e.g., the topology of a computer network).

The inspiration for introducing the notion of hygraphs comes from the work of Harel [Har88], where the case for visual formalisms is presented as follows:

> The intricate nature of a variety of situations can, and in our opinion should, be represented by *visual formalisms*: visual, because they are to be generated, comprehended, and communicated by humans; and formal, because they are to be manipulated, maintained, and analyzed by computers. …

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

Our work is not centered around the introduction of a new diagrammatic notation (whether formal or not). Instead, we emphasize its manipulation, maintenance and analysis by computers. Hygraphs are an appropriate visual formalism for representing structural visualizations that are manipulated extensively by computations (queries and filters), which in turn are described by hygraph patterns.

The definition of hygraphs is presented in Section 3.1. A comparison of hygraphs to Harel's higraphs is given in Section 3.2. Section 3.3 discusses some aspects of the problem of creating pictorial visualizations of hygraphs, while some additional applications are presented in Section 3.4.

3.1 Definition of Hygraphs

A hygraph extends the notion of a graph by incorporating *blobs* in addition to edges. A blob relates a containing node with a set of contained nodes, instead of relating a node to another node as edges do. Blobs are diagrammatically represented by a closed curve that is associated with the container node and that encloses the contained nodes.

The diagrams representing hygraphs make use of the topological notions of *connected-*

ness and *enclosure*, hence hygraphs constitute another example of a *topovisual formalism* in the sense of [Har88]. Edges make use of connectedness, while blobs resort to both connectedness (for the container node) and enclosure (for the contained nodes).

Definition 3.1.1: A *hygraph* H is a septuple

$$(N, L_N, \nu, L_E, E, L_B, B)$$

where: N is a finite set of *nodes*; L_N is a set of *node labels*; ν, the *node labelling function*, is a function from N to L_N that associates with each node in N a label from 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*.

A restriction is placed in the labelled blobs relationship B to ensure that there is only one tuple (n, N, l) in B with the same values for the *container node* n and the blob label l (i.e., the container node and blob label values functionally determine the value of the set of *contained nodes* N, so B can be considered as a function $B : N \times L_B \to 2^N$). □

Hygraphs are purely syntactic objects. The actual semantics given to the relationships represented by the nodes, edges and blobs is based on an interpretation of the labels which is left to the application that makes use of hygraphs as a formal visual notation. Moreover, edges and blobs are completely interchangeable in terms of the semantic relationships they represent. However, since the topological representation of edges and blobs is quite different, it is expected that many applications will have a definite choice for only one of edges or blobs to represent specific relationships. The following definition highlights the importance of the interpretation given to the labels of hygraphs.

Definition 3.1.2: The *information content* of a hygraph H given by

$$(N, L_N, \nu, L_E, E, L_B, B)$$

consists of a unary relation $\{\nu(n) : n \in N\}$ and a ternary relation

$$\{(\nu(n_1), \nu(n_2), l) : (n_1, n_2, l) \in E\} \cup \{(\nu(n_1), \nu(n_2), l) : (n_1, N, l) \in B \text{ and } n_2 \in N\}$$

□

In the above definition, the unary relation is the set of node labels in the graph, while the ternary relation describes the relationship among two node labels and the label of the edge or blob that relates them. In some applications (Hy$^+$ being one of them), two hygraphs are considered as equivalent representations whenever they have the same information content.

The name *hygraph* comes from regarding them as *hy*brids between Harel's *higraphs* [Har88] and directed *hy*pergraphs [Ber73]. Despite the similarities with both, hygraphs are syntactically different from both higraphs and directed hypergraphs. Defining a very special kind of directed hyperedge in which the direction is from one node to all of the remainder nodes in the hyperedge is basically equivalent to defining a blob. We are not aware, though, of any proposal for assigning such a particular direction to hyperedges of (a variation of) directed hypergraphs. In the following section we take a closer look at the similarities and differences between hygraphs and Harel's higraphs.

3.2 Comparing Hygraphs to Higraphs

The major difference between hygraphs and Harel's higraphs is that the former are merely syntactic objects that do not provide any semantics to the relationships among nodes, edges and blobs (i.e., they are as uninterpreted as graphs are), while the latter impose a fixed semantic interpretation for some aspects of a higraph (edges in higraphs continue to be uninterpreted): blobs represent sets, the containment relationship is interpreted as set inclusion, and blobs can be partitioned to represent components of set products. Hence, hygraphs constitute a more "abstract" (in the sense of being considered apart from application to a particular instance) visual formalism, while higraphs represent a more "concrete" one.

Being more "abstract", hygraphs can be "particularized" to have the same semantics of Harel's higraphs, as we discuss below. A more subtle difference between Harel's higraphs and hygraphs arises when one assigns to hygraphs the set-theoretic semantics of higraphs. To discuss this issue, we present below the formal definition of higraphs. It does make use of the *unordered cartesian product* operation on sets, defined as $S \otimes T = \{\{s, t\} : s \in S, t \in T\}$.

Definition 3.2.1: (from [Har88])

A *higraph* H is a quadruple (B, σ, π, E) where B is a finite set of elements, called *blobs*, and $E \subseteq B \times B$, the set of *edges*, is a binary relation on B. The *subblob function* $\sigma : B \rightarrow 2^B$ assigns to each blob $x \in B$ its set $\sigma(x)$ of subblobs, and is restricted so that $x \notin \sigma^+(x)$, where σ^+ is the transitive closure of σ. The *partitioning function* $\pi : B \rightarrow 2^{B \times B}$ associates with each blob $x \in B$ some equivalence relation $\pi(x)$ on the set of subblobs, $\sigma(x)$. The equivalence classes $\pi_1(x), \ldots, \pi_{k_x}(x))$ induced by the relation $\pi(x)$ describe the breakup of x into its orthogonal components (and there is the additional requirement that blobs in different orthogonal components of x must be disjoint, i.e., for any $y, z \in \sigma(x)$ if they are not in the same orthogonal component, then $\sigma^+(y) \cap \sigma^+(z) = \emptyset$).

Given a higraph H, a *model* for H is a pair $M = (D, \mu)$, where D is a set of unstructured elements called the *domain* of M and $\mu : A \rightarrow 2^D$ assigns disjoint subsets of D to the set A of *atomic blobs* of H (defined as $A = \{x \in B : \sigma(x) = \emptyset\}$). The function μ is extended inductively to all blobs $x \in B$ as follows:

$$\mu(x) = \otimes_{i=1}^{k_x}(\cup_{y \in \pi_i(x)} \mu(y))$$

which, in case $k_x = 1$ and hence no unordered cartesian product is taken, becomes:

$$\mu(x) = \cup_{y \in \sigma(x)} \mu(y)$$

Finally, the edge set E induces a semantic relation E_M defined by $(\mu(x), \mu(y)) \in E_M$ iff $(x, y) \in E$. $\Box$

We describe below how given a higraph $H' = (B', \sigma, \pi, E')$ we can represent the same structure with a hygraph H that has a specific interpretation for some of its elements. Let $H = (N, L_N, \nu, \emptyset, E, L_B \cup \{s\}, B)$. The node labels L_N are the blobs of H' (i.e., $L_N = B'$). There are no edge labels in H (as it is the case in H') and $(n_1, n_2) \in E$ iff $\nu(n_1) = x, \nu(n_2) = y$ and $(x, y) \in E'$. The blob labels L_B are names for the orthogonal components of the unordered cartesian products, and there is one distinguished label s that is used to represent the subblob function. That is, $(n, \{n_1, \ldots, n_m\}, s) \in B$ iff $\nu(n) = x, \nu(n_1) = x_1, \ldots, \nu(n_m) = x_m$ and $\sigma(x) = \{x_1, \ldots, x_m\}$, and furthermore $(n, \{n_1, \ldots, n_i\}, \pi_i) \in B$ iff $\nu(n) = x, \nu(n_1) = x_1, \ldots, \nu(n_m) = x_m$ and $\pi_i(x) = \{x_1, \ldots, x_m\}$.

The definition of higraph makes a distinction between syntax and semantics (the concrete model assigned to a higraph). As it has been mentioned before, hygraphs are only syntactic objects that can be assigned semantics through an appropriate interpretation of the labels. In particular, a hygraph H that (syntactically) represents a higraph H' as described above, can be assigned identical semantics (i.e., bot H and H' can have the same model $M = (D, \mu)$).

To illustrate one specific aspect of the representation of Harel's higraphs by hygraphs let's look at Figure 3.1 which reproduces as a hygraph the statechart (a specific kind of higraph used to represent extended state-transition diagrams) for a stopwatch (an example taken from [Har88]). While the statechart semantics can be assigned to both the stop-

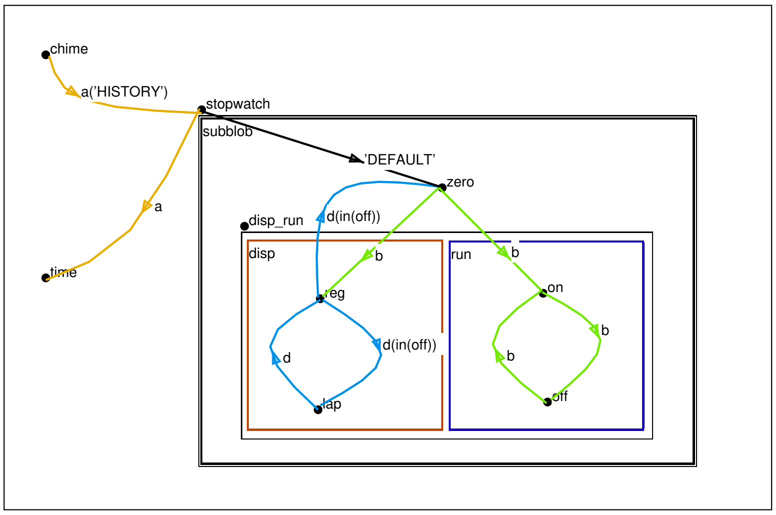


Figure 3.1: The statechart for a stopwatch as a hygraph.

watch diagram as a hygraph or higraph (as discussed in the previous paragraph), we believe the syntax for unordered cartesian product in the hygraph version to be considerable simpler to deal with. In particular, the unordered cartesian product of disp and run that defines the state labelled disp_run) is described at the syntactic level quite differently in hygraphs and higraphs. In the former, a blob labelled disp contains (denoting the semantic relationship of being an orthogonal component) the states reg and lap (and similarly for run). The latter associates with the node labelled disp_run an equivalence relation R on the set {reg,lap,on,off} and the equivalence classes of R are the ones determining the contents of the components disp and run.

In summary, the attractiveness of hygraphs over higraphs comes from their simpler syntax and the flexibility to assign any semantics to the relationship represented by the blobs (the meaning will be based on some interpretation of the blob labels). Also, there can be more than one blob associated with a given node (in the same way that there can be multiple edges from a node). Thus, one could simultaneously represent both set inclusion and set membership as different blobs, a situation that cannot be represented in higraphs (an issue mentioned as a weakness of higraphs in [Har88]).

3.3 Visualizing Hygraphs

So far, we have provided the formal definition of hygraphs as a mathematical structure and stated that it constitutes an example of a topovisual formalism. As such, we described how the topological notions of connectedness and enclosure are employed in by hygraph elements: edges are represented making use of connectedness between the two endpoint nodes, while blobs resort to both connectedness(for the container node) and enclosure (for the contained nodes).

When producing a diagrammatic representation of a hygraph, the above topological relationships must be preserved. But clearly, there is plenty of freedom to choose different pictorial representations for hygraphs. This is completely analogous to the rendering of graphs, where infinitely many drawings can be produced in two or three dimensional spaces, with different glyphs used to represent nodes and edges, as long as the topological relations implied by the graph structure are preserved. In this context, the kind of hygraph renderings produced by Hy$^+$ should be regarded as illustrating one possible way of drawing hygraph diagrams. We discuss below some aspects related to the presence of blobs when producing hygraph visualizations.

Within Hy$^+$, all blobs associated with one container node are represented by rectangles contained within a rectangular region that has the container node in the top left corner. Blob labels are drawn in the interior of the top left corner of the rectangle representing the blob. In addition, the system enforces a strict hierarchy of nested blob rectangles by creating, if

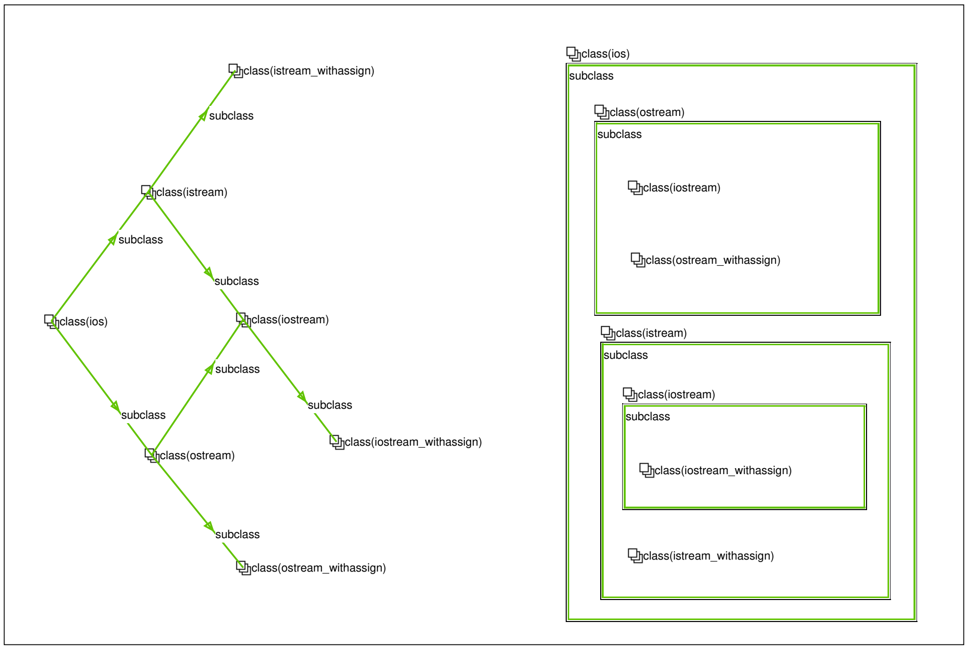


Figure 3.2: Hygraphs with duplicate node representations.

necessary, multiple occurrences of a node with the same label (i.e., it ensures that the containment relationship among blobs is a forest). Used in this way, blobs constitute a flexible mechanism for clustering information and they support views at varying levels of abstraction in the display of both hierarchical and non-hierarchical data. Additional control over the level of detail displayed is achieved by interactively hiding and showing blob contents. Examples of hygraphs (and hiding of blob contents as well) have been presented in Chapter 2.

We should take a closer look at the generation of multiple occurrences of a node with the same label to enforce that the containment relationship among blobs is a forest. Consider the graph of Figure 2.2 containing the NIH class hierarchy. The bottom portion of the graph appears on the left of Figure 3.2, while on the right the same information is represented by blobs. Using blobs to represent the subclass relationship will require the two subclass blobs of the two classes istream and ostream to contain the node labelled class(iostream). In this situation, Hy^{+} still displays a strict hierarchy of non-overlapping regions to represent blobs by creating as many occurrences of nodes with the same label as there are blobs containing the original node. So in the example discussed above, two nodes labelled class(iostream) are created and placed inside the two subclass blobs of classes istream and ostream, as required. The creation of duplicate nodes with identical labels is required whenever the union of the relations represented by blobs is not a forest (i.e., it could be an acyclic graph as in the subclass example, or even a cyclic one). Only one of the multiple representations of a given node is selected to have the blobs contained in the original node (if any).

The enforcement of a strict hierarchy for the blob containment relationship when drawing hygraphs is quite justifiable. Even if the information content relation for the blobs is acyclic, representing such a hygraph without duplicating nodes constitutes a challenge. To begin with, although there exits a drawing using convex polygons to represent the blobs, it is not always the case that exists a drawing that uses regular n-gons (or, more relevantly, rectangles) to represent the blobs [Urr89]. Furthermore, if one has a drawing and wants to add a new blob to it (representing it by any kind of curve), the problem is intractable (e.g., extending a Venn diagram of order n to order $n + 1$ is NP-complete [JP87]). On top of the technical reasons mentioned above, an argument can be made about the readability of the resulting diagrams. We use Venn's words (taken from his 1894 *Symbolic Logic* book [Ven94]) for this purpose:

> It will be found that when we adhere to continuous figures there is a tendency

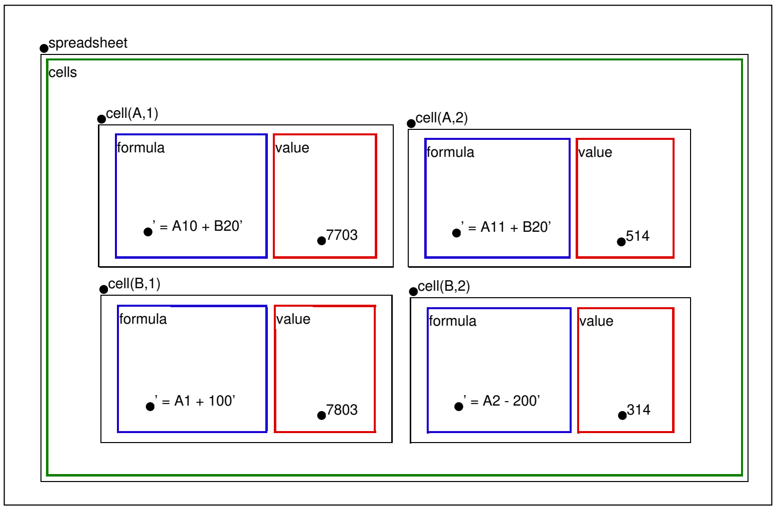


Figure 3.3: Seeing a spreadsheet as a hygraph.

> for the resultant outlines thus successively drawn to assume, after the first four or five, a comb-like shape. …
>
> There is no trouble in drawing such a diagram for any number of terms which our paper will find room for. But, as has already been repeatedly remarked, the visual aid for which mainly such diagrams exist is soon lost on such a path.

In addition to the convenience of enforcing a strict hierarchy for blob containment by creating multiple node occurrences, the semantics of the application modeled may require such a course of action (e.g., for arguments on the suitability of extending the statechart notation to allow overlaps see [HK92]).

As an example in which intersecting blobs can be dealt with easily, consider the representation of the information in a spreadsheet as a hygraph (see Figure 3.3). We could easily add blobs contained in nodes representing the columns and rows, and these blobs can be

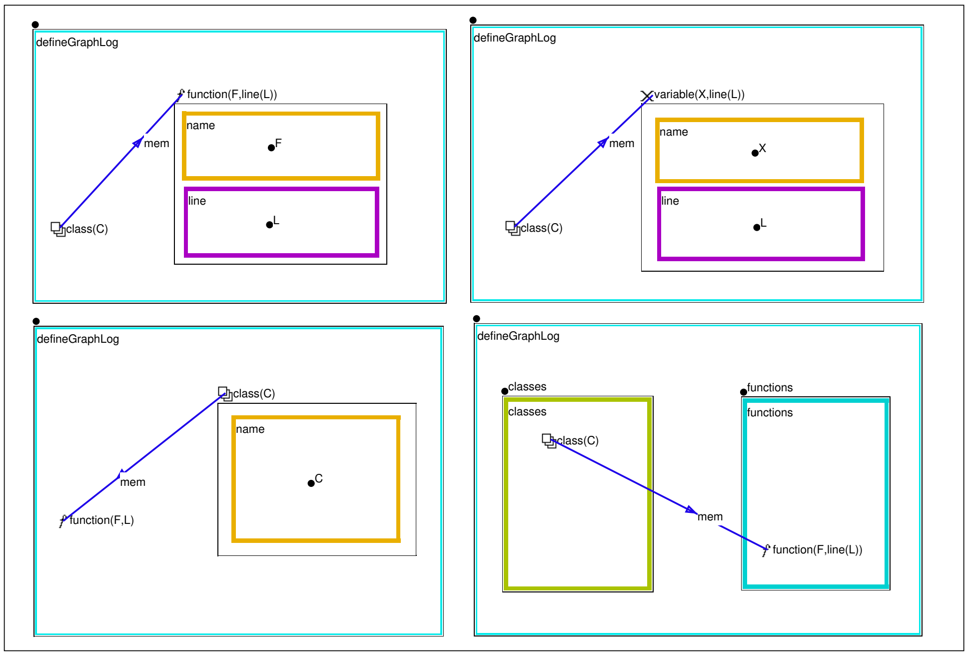


Figure 3.4: Defining blobs for attribute boxes and regions.

drawn to intersect at the appropriate cells.

3.4 Hygraph Applications

We have discussed above the issue of creating a drawing for a hygraph, that is, visualizing an instance of a structure. We can also take the opposite view, and given a diagram, consider a hygraph as an abstraction (retaining only the topology of the diagram) of the information represented by the diagram. As such, hygraphs are a convenient formal abstraction for several styles of diagrammatic data presentations, not just graphs and extensions. By ignoring edges, and making use of multiple blobs associated with the same node as well as recursive

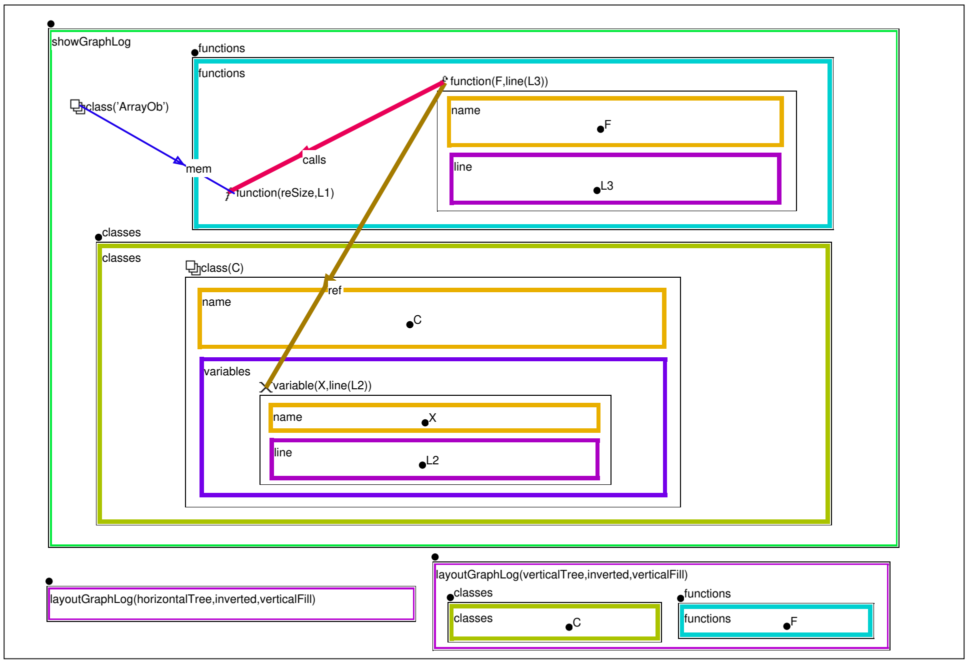


Figure 3.5: References to variables in functions of ArrayOb called from reSize.

node containment, one can easily model traditional (nested) form-based presentations. The motivation for our interest in looking at hygraphs as an abstract representation of diagrams comes from our use of hygraph patterns to specify queries and filters: in this context, hygraph patterns can be seen as generating diagrams as well as describing mappings between diagrams.

An illustration of a hygraph representing information in a mixture of a form-based and graph-based notation appears in Figure 3.6. The hygraph patterns used to obtain it (using the NIH data described in Chapter 2) are presented in Figure 3.4 and Figure 3.5.

Tables are a very common special case of form-based visual presentation of data. An example of a table representing a relation of arity four is given in Figure 3.9. We stress that the

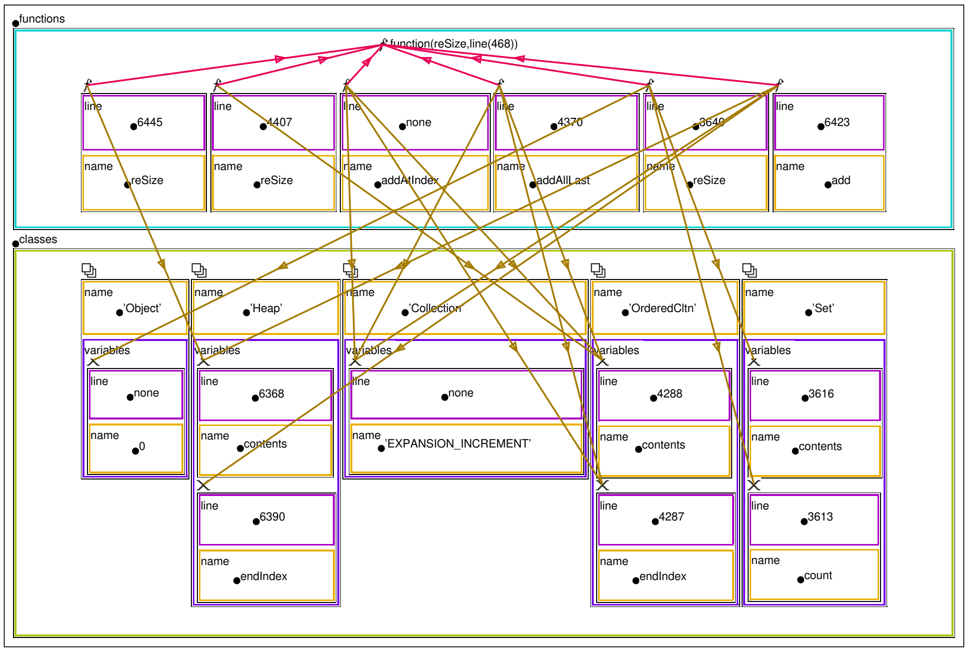


Figure 3.6: A selective diagram with boxes for attributes.

point of this diagram is not to convince the reader that the particular rendering of hygraphs used by Hy$^+$ is appropriate to produce a table drawing (although the layout specification succeeds in simulating a tabular appearance), but rather that a hygraph is a simple abstraction of the relationships represented in a table. The define and filter patterns to produce the table appear in Figure 3.7 and Figure 3.8. The reader will notice that the last two figures contain patterns that have a strong resemblance to expressions of the QBE query language [ZBM76].

Finally, a third view of hygraphs consists of looking at them as the description of a hypermedia web. In the context of hypermedia presentations, the edges in the hygraph can be interpreted as *links* while the blobs can be regarded as *fat links* [Hal88]. This view of

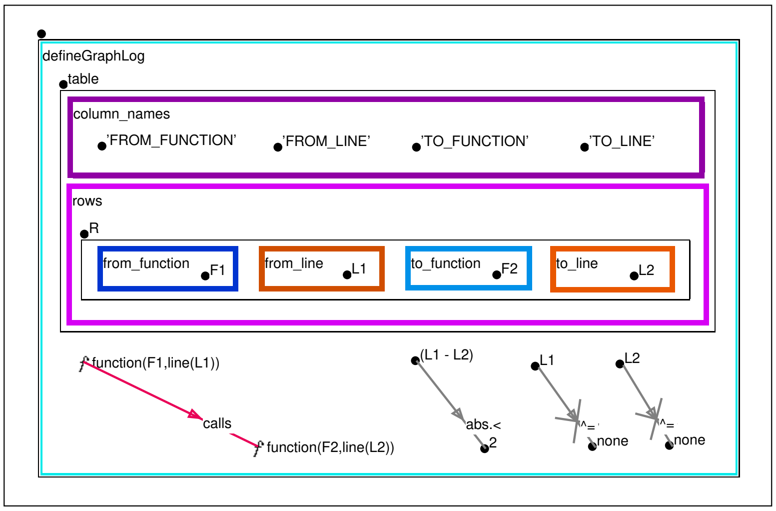


Figure 3.7: Defining blobs to represent a table with headings.

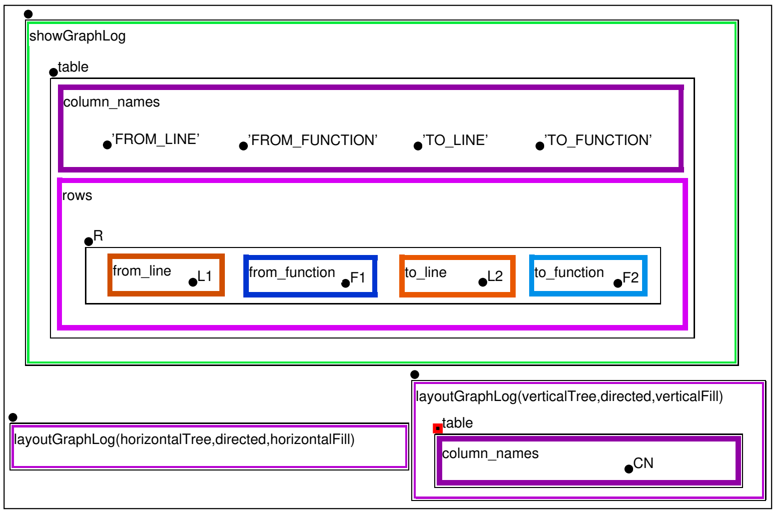


Figure 3.8: Show pattern to display a relation with very local calls.

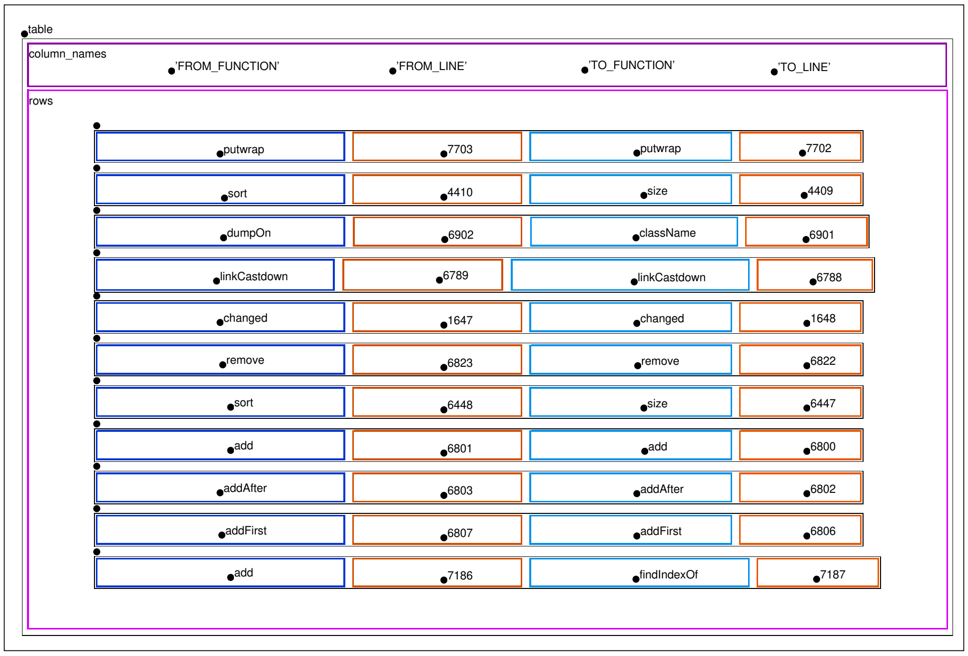


Figure 3.9: The table for calls less than two lines of code away.

hygraphs, similar to Tompa's directed hypergraphs[1] [Tom89], makes them a suitable abstraction of the navigational choices presented to the users that browse hypermedia-like information structures.

[1]These are hypergraphs with hyperedges directed from sets of nodes to sets of nodes.

Chapter 4

Filtering Languages and Hygraph Patterns

In this chapter we develop the formal basis for assigning semantics to Hy$^+$ patterns. The material presented here contributes to the theory of database languages. The following statement from Chandra [Cha88] provides an excellent description of our objectives.

> One goal of the theory of database queries is to provide an understanding of query language constructs so that query languages could be designed that are natural to use, expressive, and efficient in practice.
>
> The other, as always, is elegance.

The notion of *computable query*, defined by Chandra and Harel [CH80], is central to the theory of database queries. A computable query, presented in the context of Codd's relational model [Cod70], is a partial recursive function which, given a relational database as input, produces as output a relation on the domain of the database, and satisfies a consistency criterion (if two databases are isomorphic, then their outputs are also isomorphic under the same isomorphism).

The concept of computable query (originally presented having queries typed by their input database but not the output relation), was later extended to consider all relational database transformations (i.e., queries and updates) both deterministic and non-deterministic by Abiteboul and Vianu [AV87]. A further generalization by Abiteboul and Kanellakis [AK89] extends the notion to queries in object-oriented data models supporting the creation of new object identifiers.

In this work we present our definition of what we have identified as a very important subclass of database mappings: *filter queries*. We also introduce an appropriate notation for expressing the particular mappings defined by filter queries. A *computable filter query*, in the context of the relational model, is a partial recursive function which, given a relational database as input returns as output subsets of some of the relations in the input database, while satisfying the consistency criterion for computable queries. This notion, and the associated notation, are the formal basis for giving semantics to Hy$^+$ show patterns.

In Section 4.1, after a brief survey of computable queries, we provide a formal definition of computable filter queries. The survey continues in Section 4.2 (with the notions of expressive power and data complexity) and Section 4.3 (where datalog and logic programs are presented). The definition of filtering logic programs and their expressive power characterization appears in Section 4.4. Finally, Section 4.5 describes the meaning of the hygraph patterns used to specify queries and filters in Hy$^+$.

4.1 Queries and Filters

We begin by very briefly recalling the definitions of Codd's relational model [Cod70].

Definition 4.1.1: A *database scheme* D is a set of *relation schemes* R, each one being a name with an associated positive integer denoted $|R|$ and called the *arity* of R. A *database* (over D) is a tuple $d = (\delta, r_1, \ldots, r_m)$, where δ, the *domain of* d, is a finite nonempty set of values and each *relation* r_i (also known as an *instance* of R_i) is a finite subset of $\delta^{|R_i|}$. $\square$

Before presenting the definition of computable query we formalize the consistency criterion. The intuition for this criterion is that the mappings defined by queries should not depend on implementation details, should not violate the abstraction of relations as unordered sets, and should be invariant under possible reorganizations of the database.

Definition 4.1.2: Two databases over D, $d = (\delta, r_1, \ldots, r_m)$ and $d' = (\delta', r'_1, \ldots, r'_m)$, are *isomorphic* (denoted $d \overset{h}{\leftrightarrow} d'$) iff there is a bijection $h : \delta \leftrightarrow \delta'$ that extends componentwise to tuples and is such that for all $1 \leq i \leq m$, $h(t) \in r'_i$ if $t \in r_i$ and $h^{-1}(t') \in r_i$ if $t' \in r'_i$. $\square$

The condition above (in the form of an automorphism) was stated in [Ban78, Par78, AU79]. When constants are considered, the output to the query should be invariant only under isomorphisms that map the constants to themselves [AU79, AV87]. We will not consider this issue (nor similar ones, like the presence of an order relation in the domain, or the use of aggregate functions), since the extensions are straightforward.

The definition of query below (from [CH80]) captures three intuitions: queries should be computable, values in the answer of a query should be taken from the domain of the database, and the outputs of queries should be invariant under isomorphisms.

Definition 4.1.3: Let D be a database scheme and R a relation scheme such that $|R| = n$. A *computable query* (or *query* for short) of *type* $D \to \{R\}$ is a partial function q, which on input database $d = (\delta, r_1, \ldots, r_m)$ over D has output database $q(d)$ over $\{R\}$ such that:

1. q is partial recursive,

2. $q(d) = (\delta, r)$, where $r \subseteq \delta^n$,

3. if $d \overset{h}{\leftrightarrow} d'$ then $q(d) \overset{h}{\leftrightarrow} q(d')$.

The set of computable queries is denoted by CQ. □

In the above definition of query the answer to a query consists of one relation, or more precisely a database with one relation. While the fact that the output database is a single relation does have an impact in practical terms (as we discuss at the end of this chapter), from a theoretical point of view it suffices to consider tuples of queries to map from databases to arbitrary databases. We will denote by TCQ the set of mappings between arbitrary databases that are expressed by a tuple of computable queries. Hence, queries in TCQ are functions from databases to databases. Closure under queries is an important property of the relational model.

Below we give our definition of computable filter queries. The intuition is simply to capture those queries that return as a result subsets of some of the relations in the input database.

Definition 4.1.4: Let D be a database scheme. A *computable filter query* (or *filter* for short) of type $D \rightarrow D'$ is a partial function f, which on input database $d = (\delta, r_1, \ldots, r_m)$ over D has output database $f(d)$ over $D' \subseteq D$ such that:

1. f is partial recursive,

2. $f(d) = (\delta, r'_1, \ldots, r'_n)$ where for each $i, 1 \leq i \leq n$, there exists $j, 1 \leq j \leq m$, such that $r'_i \subseteq r_j$,

3. if $d \overset{h}{\leftrightarrow} d'$ then $f(d) \overset{h}{\leftrightarrow} f(d')$.

The set of computable filter queries is denoted by FQ. □

From the definitions it follows immediately that FQ $\subset$ TCQ.

The above result simply states that filter queries are a special case of queries. We would like to emphasize though, that filter queries have interesting properties of their own (listed below).

- *Preservation of the application interface.* Since the relations in the result are all part of the input database, any application program accessing the input database can run unchanged accessing the result of a filter query.

- *Immunity from the view update problem.* The views defined by a filter query are trivially updatable, hence there is no *view update problem* [BS81, DB82] for this special kind of view. This property holds even if the input relations for the filter query are not really base relations, but are defined as views: as long as the input views are updatable[1] the resulting views defined by filter queries preserve updatability.

- *Conceptual simplicity.* Again, since the relations in the result are all part of the input database, no new relations have to be understood in order to comprehend the answer to a filter query.

- *Natural visual interpretation.* If there is a visualization of an instance of the input database, the process of *eliding* portions of it, or *filtering-out* irrelevant information from the visualization, can be formally described as a filter query (e.g., consider Figure 2.15 and the corresponding filtered visualization in Figure 2.27).

[1] View updatability can be guaranteed in a formalism like Transaction Logic [BK93] in which view updates can be disambiguated explicitly.

4.2 Expressive Power of Query Languages

We now present the definition of another basic notion in the theory of database queries; *query languages*, the languages for specifying queries. Since we introduced the notion of filter queries, we will also consider languages suitable for describing this particular kind of queries.

Definition 4.2.1: A *query language* (resp., *filter query language*) is a pair $L = (E, \mu)$, where E is a set of expressions and μ is a *meaning* function such that for every expression e in E, $\mu(e)$ is a query (resp., a filter query). The set $L_Q = \{q : q = \mu(e), e \in E\}$ is the set of queries (resp. filter queries) *expressed* by L. $\square$

The syntax and semantics of a query language are given by its expressions and meaning function, respectively.

Once we have defined a query language, it is natural to ask what is the set of queries that can be expressed in the language. The answer to this question characterizes the *expressive power* of the language, a notion which is a fundamental object of study in the theory of database queries. A relevant observation is that query languages, unlike commonly used programming languages, do not all have the same computational power.

Definition 4.2.2: Let $L = (E, \mu)$ and $L' = (E', \mu')$ be query languages (or filter languages). Then L is *less expressive* than L' if $L_Q \subset L'_Q$, and L and L' have *equivalent expressive power* iff $L_Q = L'_Q$. $\square$

The earliest result on expressive power of query languages was due to Codd [Cod72], where he demonstrated the equivalence of the relational algebra and calculus. The class of queries defined by these languages is also known as *first order queries*, and we denote it by FO.

From now on we will not make a notational distinction between a query language L and the set of queries L_Q expressed by it; both will be denoted by L. The following definition is from [Var82].

Definition 4.2.3: Let $L = (E, \mu)$ by a query language. The *data complexity* of L is the complexity of the membership problem for the set $\{(t, d) : \exists e \in E, q = \mu(e), t \in q(d)\}$. $\square$

Data complexity describes how "expensive" it is to answer a query in the language as a function of the size of the database. In particular, first order queries are computationally very inexpensive, since their data complexity is well below LOGSPACE [Imm88b].

We have considered the complexity of recognizing that a tuple is in the output, instead of considering the complexity of computing the whole output. The two measures are related as follows: if tuples in the output of a query can be recognized in time $T(n), T(n) \geq n$ (resp. space $S(n), S(n) \geq \log(n)$), then the output can be computed in time $n^k T(n)$ for some k (resp. in space $S(n)$).

While the computable queries span all the complexity classes, it is interesting to consider the sets of queries QC that have data complexity C, for any complexity class C. The relationships among sets of queries mirror the ones among complexity classes [CH82]: $\mathrm{QC}_1 \subset \mathrm{QC}_2$ iff $\mathrm{C}_1 \subset \mathrm{C}_2$, provided that C_1 and C_2 are closed under logspace reducibility (a condition satisfied by all the complexity classes that we consider).

Given the potential advantages of implementations with a high degree of parallelism (already exploited in existing systems that implement FO by profiting from concurrent storage access), QNC (the set of queries recognizable in poly-logarithmic time using polynomially many processors) appears as a desirable upper bound for the expressive power of query languages.

Complexity and expressibility are closely related to each other. This was first noticed in [Fag74], were it was shown that QNPTIME coincides with the set of queries expressible by existential second order formulas. In addition, there are several other complexity classes that coincide with the set of queries expressible in some language. In particular, Immerman [Imm88b] showed that $\text{FO}^{<}$ (the set of first order queries extended with an order relation plus another predefined relation to test bits) coincides with QAC^{0} (the set of queries recognizable by Alternating Turing Machines requiring logarithmic space and constant time, or equivalently, a uniform sequence of polynomial size, unbounded fan-in boolean circuits of constant depth). The need for an order relation is a technicality to capture classes below NPTIME, which can be substituted by the use of non-determinism [AV87, AV88].

The property of a query language capturing precisely the set of queries expressible in a complexity class is central to the area of descriptive complexity [Imm88a]. However, the notion of a language capturing a complexity class has not only theoretical relevance. In the author's opinion, capturing a complexity class is a highly desirable objective in the design of practical query languages. This property shows that, once a language is capable of expressing queries in a given complexity class, it does in fact express all of them. Informally, given that one must "pay" for a language that "costs" C (where C is the complexity class of the computational problem resulting from implementing the language), one gets back the maximum possible "value" of being able to express all possible queries with the same "price tag" C. In particular, we are certainly interested in finding query languages that capture complexity classes below NC.

4.3 Logic Programs

We now introduce Horn-clause based query languages, also known as logic query languages [Ull88]. These languages are an adaptation to the database field of the approach pioneered in logic programming (see [Llo84]). We will first focus on a restricted class of logic programs, called *datalog*, which can be seen as a direct extension of the first order languages for the relational model. We start with some preliminary definitions.

Definition 4.3.1: Let $p, q, p_1, p_2, \ldots$ denote *predicate* symbols. An *atom* is an atomic formula which is either of the form $p(X_1, \ldots, X_n)$ (abbreviated p, when variables are not relevant) or $X = Y$. A *literal* is either a *positive* (non-negated) atom or a *negative* (negated) atom.

A *valuation* θ is a function $\theta : \mathcal{V} \rightarrow \delta$ from variables to values. If s is a literal, then $s\theta$ (called a *ground literal*) is the result of replacing in s each variable X by $\theta(X)$.

A *clause* is a disjunction of literals. A *Horn-clause* is a clause with at most one positive literal. A *rule* r is a Horn-clause with one positive literal denoted by

$$r : p \leftarrow s_1, s_2, \ldots, s_k.$$

where p is called the *head* of the rule and the *body* of the rule is a list $s_1, s_2, \ldots, s_k$ of positive literals referred to as *subgoals*. □

Definitions for programs and some aspects of their structure are given below.

Definition 4.3.2: A *(datalog) program* $\mathcal{P}$ is a finite set of rules containing two classes of predicate symbols: *IDB (intentional database) predicates*, denoted $p, p_1, p_2, \ldots$, are the ones that appear in some rule head; and *EDB (extensional database) predicates*, denoted $q, q_1, q_2, \ldots$, are the ones that do not appear in any rule head.

The *dependence graph* of a logic program $\mathcal{P}$ is a directed graph whose nodes are the IDB and EDB predicates of $\mathcal{P}$ and such that there is an edge from p_j (resp. q_j) to p_i iff there is a rule in $\mathcal{P}$ whose head is p_i and which has p_j (resp. q_j) in the body.

A rule is *recursive* if it has one or more subgoals, called *recursive*, in the same strongly connected component of the dependence graph as the head; the remaining subgoals are called *non-recursive*. A *recursive predicate* is one that appears as a recursive subgoal in some rule. $\square$

The syntax of the datalog query language is given by datalog programs. The semantics can be given in several different ways. The first corresponds to the *proof-theoretic* interpretation of the program: for each rule we derive all the ground atoms that are derivable from it. Alternative semantics can be given by resorting to the *minimal model* (under set inclusion) of the program, where a *(Herbrand) model M* of a program $\mathcal{P}$ is a set of ground atoms such that $M \models r$ for each rule $r \in \mathcal{P}$. A third possibility is to define a fixpoint operator for a program and then consider the *least fixpoint* of such an operator [vEK76]. An important result from the logic programming field [vEK76, AvE82] is that the minimal-model semantics, the proof-theoretic semantics and the least fixpoint semantics of logic programs, all coincide.

For our purposes, the most appropriate alternative is to give proof-theoretic semantics to datalog programs. With this objective we introduce a preliminary concept.

Definition 4.3.3: The set of *derivation trees* $DT(\mathcal{P}, d)$ for a program $\mathcal{P}$ and a database $d = (\delta, r_1, \ldots, r_m)$ over $D = \{R_1, \ldots, R_m\}$, with each of the EDB predicates q_i corresponding to the relation scheme $R_i, 1 \leq i \leq m$, is defined recursively as follows.

- For each tuple $t \in r_i, 1 \leq i \leq m$, there is a derivation tree consisting of a single node labelled with the corresponding ground literal of q_i.

- For each rule $r : p \leftarrow s_1, s_2, \ldots, s_k$ in $\mathcal{P}$ and valuation θ such that for $1 \leq j \leq k$, $\exists t_j \in DT(\mathcal{P}, d)$ with the root labelled by $s_j\theta$, there is a derivation tree with the root labelled by $p\theta$ and with $t_j, 1 \leq j \leq k$, as subtrees of the root.

□

We can now present the definition of the datalog query language.

Definition 4.3.4: *Datalog* is the query language DATALOG $= (E, \mu)$. The set of expressions is $E = \bigcup_D E_D$, where E_D is the set of pairs $(\mathcal{P}, p)$ such that

- $\mathcal{P}$ is a datalog program over the database scheme $D = \{R_1, \ldots, R_m\}$
- p is a distinguished IDB predicate (called the *carrier* of $(\mathcal{P}, p)$)
- each of the EDB predicates q_i corresponds to the relation scheme $R_i, 1 \leq i \leq m$.

The meaning function μ is given by $\mu((\mathcal{P}, p)) = q$, such that p has arity n and

$$q(d) = (\delta, \{(a_1, \ldots, a_n) : \exists t \in DT(\mathcal{P}, d), p(a_1, \ldots, a_n) \text{ labels the root of } t\})$$

□

The equivalent fixpoint semantics can be interpreted as giving *operational semantics* to datalog. The resulting algorithm is known as the *naive evaluation* of the datalog program. The naive evaluation has a polynomial bound on the number of iterations (tuples are monotonically added to the result, whose size is bounded by a polynomial in the size of the database). Hence we have that DATALOG $\subset$ QPTIME.

Given that DATALOG queries are monotonic, we would like to extend the definitions to be able to express negation within the logic programming framework. Negation in logic programming is a research area by itself (see [She88] for a survey). We will present first a proposal in [CH85], studied by [ABW88, vG88] and others.

Definition 4.3.5: A *general rule* is a Horn-clause with one or more positive literals. A *general datalog program* is a datalog program that admits general rules.

A general datalog program $\mathcal{P}$ is called *stratified* if there is a partition in *strata* $S_1, \ldots, S_l$ of its predicates (called a *stratification*) such that for $1 \leq i \leq l$ it is the case that: *(i)* if a predicate p occurs positively in $r \in S_i$ then all the rules with head p are in $\bigcup_{1 \leq j \leq i} S_j$ (i.e., in lower or equal strata), and *(ii)* if a predicate p occurs negatively in $r \in S_i$ then all the rules with head p are in $\bigcup_{1 \leq j < i} S_j$ (i.e., in lower strata). $\Box$

We can assign meaning to a stratified program $\mathcal{P}$ by using a fixpoint computation at each stratum that takes the complement of negative literals wherever they appear (note that it is always the case that they have been already computed at some previous stage). Although the above approach for computing the result might seem arbitrary, it turns out that it yields a minimal fixpoint of the stratified program (not necessarily the least because there can be several minimal fixpoints), called the *perfect fixpoint* of $\mathcal{P}$.

The query language *stratified datalog*, denoted S-DATALOG, has stratified logic programs as expressions whose meaning is given by the perfect fixpoint discussed above. If we consider the set of queries expressed by stratified Datalog programs that have no recursive IDB predicates (a set that we will denote by SNR-DATALOG), we have that SNR-DATALOG $=$ FO (see [Ull88]).

We will be particularly interested in Datalog programs whose only recursive predicates define (slightly extended) transitive closures. The next definition characterizes precisely these programs (introduced in [Con89]), together with the well known subclass of linear logic programs.

Definition 4.3.6: A *linear* logic program is one in which each rule has at most one recursive subgoal.

A *TC logic program* is a linear program in which each recursive IDB predicate p (called a *TC predicate*) is the head of exactly two rules of the form

$$\begin{array}{lcl} p(\overline{X},\overline{Y},\overline{W}) & \leftarrow & p_0(\overline{X},\overline{Y},\overline{W}). \\ p(\overline{X},\overline{Y},\overline{W}) & \leftarrow & p_0(\overline{X},\overline{Z},\overline{W}), p(\overline{Z},\overline{Y},\overline{W}). \end{array}$$

where $\overline{X},\overline{Y},\overline{Z}$ are sequences of $n, n \geq 1$, distinct variables and $\overline{W}$ is a sequence of $m, m \geq 0$, distinct variables. $\Box$

The set of queries expressed by stratified linear datalog programs (resp., stratified TC datalog programs) is denoted by SL-DATALOG (resp., STC-DATALOG).

The *GraphLog* visual query language, defined in [Con89], is given semantics by a translation to STC-DATALOG. The following result characterizes the expressive power of Graphlog (where TC denotes the first order queries extended with a transitive closure operator, as in [AU79], and we make use of an order relation).

Theorem 4.3.1: (from [Con89])

$\text{TC}^< = \text{STC-DATALOG}^< = \text{GRAPHLOG}^< = \text{SL-DATALOG}^< = \text{QNLOGSPACE}$

The previous expressive power characterization is based on Immerman's result showing that $\text{QNLOGSPACE} = \text{TC}^<$ [Imm88a, Imm88c]. This a significant (and somewhat surprising) result since it shows that nondeterministic space (logspace, in particular) is closed under complement[2], a formerly long-standing open problem in complexity theory, that, for instance, answers whether the context sensitive languages are closed under complement [HU79].

[2] A result independently proved by Immerman and Szelepcsenyi.

The fact that GraphLog captures precisely QNLOGSPACE is a very attractive property of the language. As it was discussed before, this provides with the maximum expressive power at the lowest possible computational complexity. In the case of GraphLog, the TC predicates express transitive closures, which is a logspace complete problem for nondeterministic logspace, therefore NLOGSPACE is the lowest possible complexity class.

The class of queries expressed by Graphlog represent a good compromise in terms of maximizing expressive power while remaining in a complexity class below NC. In fact, it is the best that can be done with the current state of knowledge, since there is no language known to capture QNC, and furthermore, there are no known "natural" NC-complete problems [Coo85].

The expressive power results for GraphLog were extended in [CM90c, CM93b] to consider the addition of aggregate operators (in particular, recursive aggregation). An extended class of GraphLog queries with aggregates was defined and its data complexity was characterized within NC.

We should finally mention that datalog programs can be generalized to full logic programs including the presence of function symbols. This is done by extending the definition of atoms as follows.

Definition 4.3.7: A *general atom* has the form $p(t_1, \ldots, t_n)$, where each $t_i, 1 \leq i \leq n$ is a *term* built recursively as follows: *(i)* a variable is a term, *(ii)* a constant is a term, and *(iii)* if f is a function symbol and $t'_1, \ldots, t'_k$ are terms, then $f(t'_1, \ldots, t'_k)$ is a term. $\Box$

Once function symbols are considered in logic programs (let's denote this class of programs by LP), not only do we go outside the realm of the relational model (where values must be atomic), but the models of the programs may no longer be finite.

There are subclasses of LP for which finiteness can be guaranteed, and consequently we can regard the mapping as a well-defined (extended) query. The extension is due to the fact that the presence of function symbols allows the creation of new values in the output outside the domain of the input database (as we mentioned before, extending the definition of queries to account for object invention has been done in [AK89]).

One of the subclasses for which finiteness can be easily shown is defined by the stratified non-recursive logic programs (with function symbols), which we will denote by SNR-LP. The observation is based on the use of the ATOV operation defined in Chapter 12 of [Ull89].

Since the only recursive rules in TC programs define transitive closures, TC programs (with function symbols) define another well-behaved subclass of LP, denoted STC-LP. Furthermore, we can conclude that STC-LP, and hence also GraphLog extended with function symbols (which we denote below by GRAPHLOG-LP), are still in NLOGSPACE.

Proposition 4.3.1: $\text{STC-LP}^{<} = \text{GRAPHLOG-LP}^{<} = \text{QNLOGSPACE}$

The proposition above follows by a slight modification of Theorem 4.3.1 that takes into account the fact that the complexity of the only new operation introduced (the ATOV operation mentioned early) is within NLOGSPACE, together with the fact that the only recursive predicates are TC predicates which do not introduce new elements of the Herbrand universe. The result, however, cannot be extended to linear logic programs with function symbols, since for this class of programs it is not the case that recursive predicates do not introduce an infinite number of new elements of the Herbrand universe.

4.4 Filtering Programs

We now turn our attention to a novel way of expressing filter queries by means of logic programs: *filtering datalog programs*. It turns out that a filter query defined by a filtering program has almost the same syntax as a datalog program, with one small addition: a subset of the EDB predicates (in addition to the carrier IDB predicate) is singled out (and we refer to these predicates as *filtering predicates*). Furthermore, the underlying semantics assigned to the logic program is the same in filtering datalog and in (plain) datalog. The fundamental difference lies in what is defined to be the answer to the filter query: for a datalog program the answer is the set of tuples derived for the carrier, while for a filtering datalog program the answer is a tuple of relations corresponding to the filtering predicates, where each output relation contains the set of tuples that "contribute" (we formalize this notion below) to some tuple in the carrier.

To illustrate the concept of filtering datalog programs, we will resort to a variation of a favorite program of the deductive database literature: that expressing the *same-generation* query. Our version of the $\mathcal{SG}$ program is:

$$\begin{array}{rcl} \mathsf{sg(X,Y)} & \leftarrow & \mathsf{flat(X,Y).} \\ \mathsf{sg(Y,X)} & \leftarrow & \mathsf{up(X,X1), sg(Y1,X1), down(Y1,Y).} \\ \mathsf{s(X)} & \leftarrow & \mathsf{sg(class('Stack',down),X),} \end{array}$$

Our presentation has a twist, though. We use Hy$^+$ to synthesize an example dataset from real NIH data, and then use the system as a didactic tool to visualize the answer to the same-generation problem. Figure 4.1 creates two isomorphic trees of up and down edges, connected by flat edges, with data from a portion of the NIH inheritance class hierarchy presented in Figure 2.2. Then we use two define patterns and a show pattern (in the lower part of

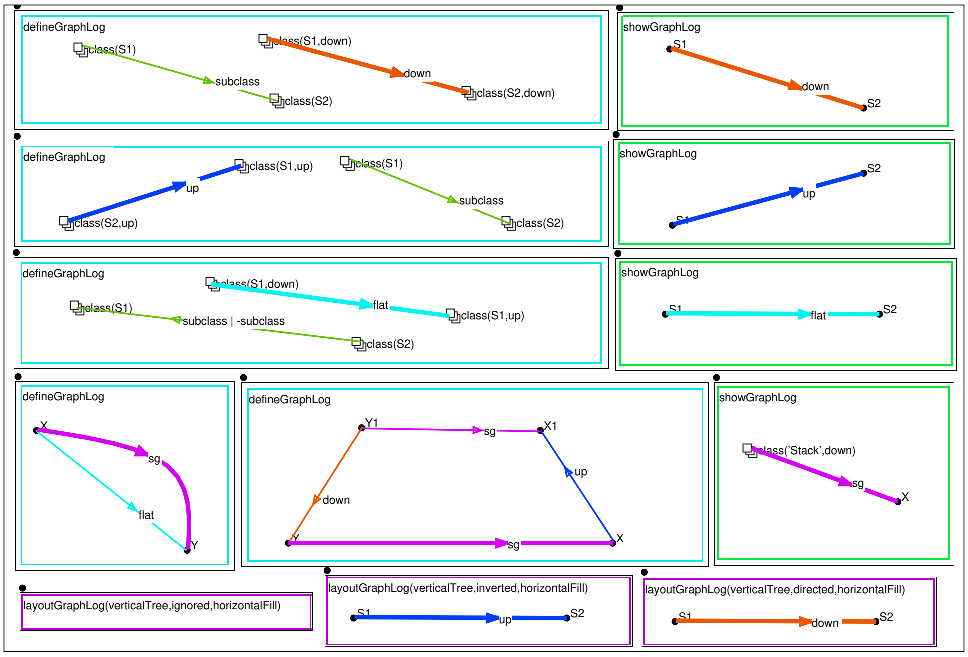


Figure 4.1: Defining and showing a variation on the same generation query.

the figure) to express the sg predicate and visualize the sg(class('Stack',down),X) edges. The result appears in Figure 4.2.

If we consider the query defined by the program $(\mathcal{SG}, \mathsf{s})$ (where s is the carrier), the answer is the set of the five tuples of the form s(class(X,up)) with X taking the values IdentSet, Dictionary, Heap, OrderedCltn and Stack.

If instead we are interested in the tuples from the base predicates up, down and flat that contribute to a tuple in s, then we can interpret the same-generation program as a filtering program $(\mathcal{SG}, \mathsf{s,up,down,flat})$, with carrier s and filtering predicates up, down and flat. The result in Figure 4.2 has the edges of the filtering predicates up, down and flat that are returned as answers for the filtering program distinguished (i.e., drawn thicker).

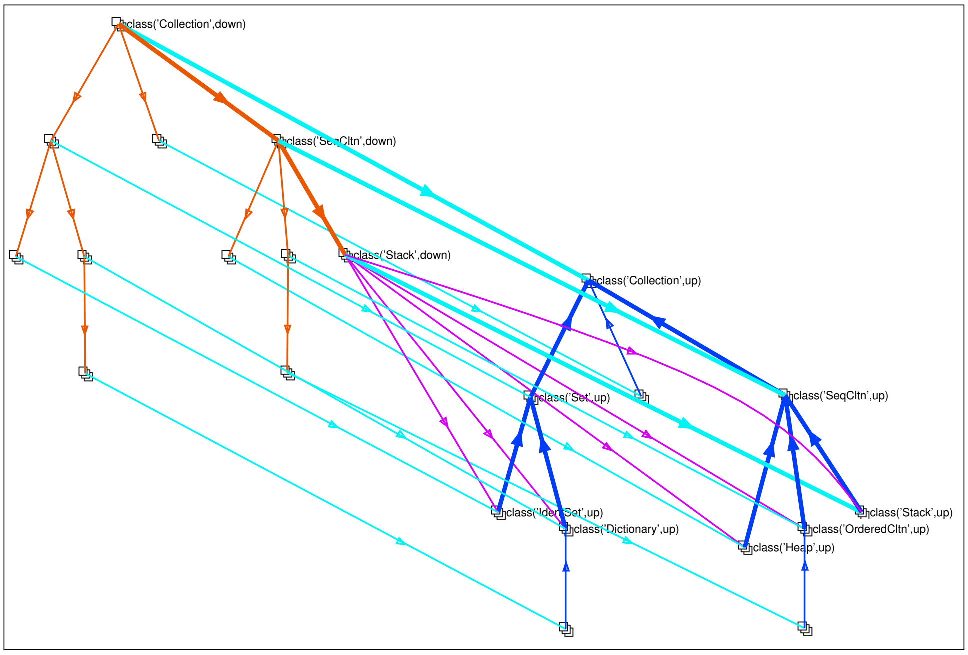


Figure 4.2: A graph showing the same generation edges for Stack.

We give below the definition of the filtering datalog language.

Definition 4.4.1: *Filtering Datalog* is the query language F-DATALOG $= (E, \mu)$. The set of expressions is $E = \bigcup_D E_D$, where E_D is the set of tuples $(\mathcal{P}, p, q_{i_1}, \ldots, q_{i_n})$ such that

- $\mathcal{P}$ is a datalog program over the database scheme $D = \{R_1, \ldots, R_m\}$
- p is a distinguished IDB predicate (called the *carrier* of $(\mathcal{P}, p, q_{i_1}, \ldots, q_{i_n})$)
- each of the EDB predicates q_i corresponds to the relation scheme $R_i, 1 \leq i \leq m$
- each EDB $q_{i_j}, 1 \leq j \leq n$, has $1 \leq i_j \leq m$ and is called a *filtering predicate* of $(\mathcal{P}, p, q_{i_1}, \ldots, q_{i_n})$

The meaning function μ is given by $\mu((\mathcal{P}, p, q_{i_1}, \ldots, q_{i_n})) = q$, such that $q(d) = (\delta, r'_1, \ldots, r'_n)$ where for $1 \leq j \leq n$,

$$r'_j = \{q_{i_j} : \exists t \in DT(\mathcal{P}, d), p \text{ labels the root of } t \text{ and } q_{i_j} \text{ labels a node of } t\}$$

□

The above definition can be easily adapted to the classes that are of interest to us, and define corresponding classes of filtering programs denoted F-SNR-DATALOG, F-STC-DATALOG, F-SNR-LP, and F-STC-LP.

We have required filtering predicates to be EDB predicates, so that the mappings defined will be filter queries on the original database. Clearly, we can easily consider IDB predicates as filtering predicates as well (it suffices to regard the IDB relation as part of the input database for the filter mapping).

Another straightforward extension of filtering programs consists of distinguishing each one of the (syntactic) occurrences of a predicate q in a filtering program $\mathcal{P}$ and then selecting which of those occurrences contribute tuples to the output of the filter and which do not. This is equivalent to: *(i)* renaming each occurrence of q with distinct predicates q^i, all with the same extension as q, *(ii)* using each q^i as a filtering predicate, and *(iii)* returning their union as the value of the filter.

We now have two ways, which can be regarded as dual, of defining filter queries using logic programs. The first one uses a tuple of programs where the carriers define subinstances of the EDB predicates (let's denote the filters denoted by this approach TF-DATALOG), while the second one (defined above as F-DATALOG) uses only one program in which the EDB predicates are designated as filtering predicates. In what follows, we prove that the two approaches for defining filter queries are equivalent, thus characterizing their expressive power.

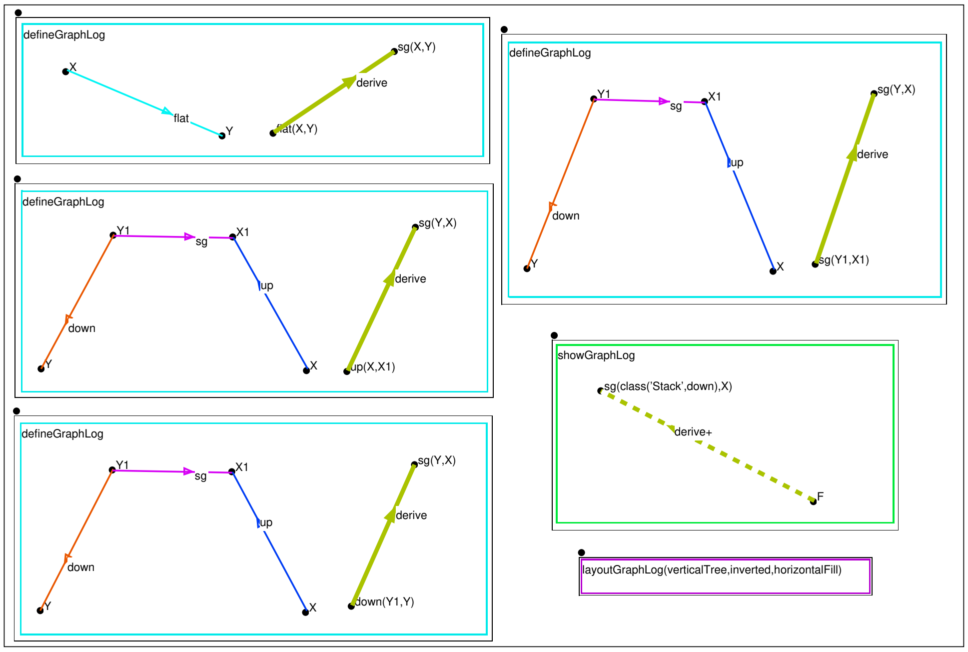


Figure 4.3: Defining and showing the derivation trees for sg facts.

Theorem 4.4.1: TF-DATALOG = F-DATALOG

To show that there exists an F-DATALOG program that can express the same filter defined by an arbitrary TF-DATALOG tuple of programs, we take the union of the tuple of programs, resulting in a new program $\mathcal{P}$ with a new carrier s. The predicate s is defined as the union of the intersections of each one of the carriers from the programs in TF-DATALOG with the predicates q_i corresponding to the relations being filtered. Clearly, $\mathcal{P}$ with carrier s and filtering predicates q_i is a filtering program in F-DATALOG expressing the same filter that the original tuple of programs in TF-DATALOG.

To demonstrate the converse, i.e, that there exists a TF-DATALOG tuple of programs that can express the same filter defined by an arbitrary F-DATALOG program, we first encode the

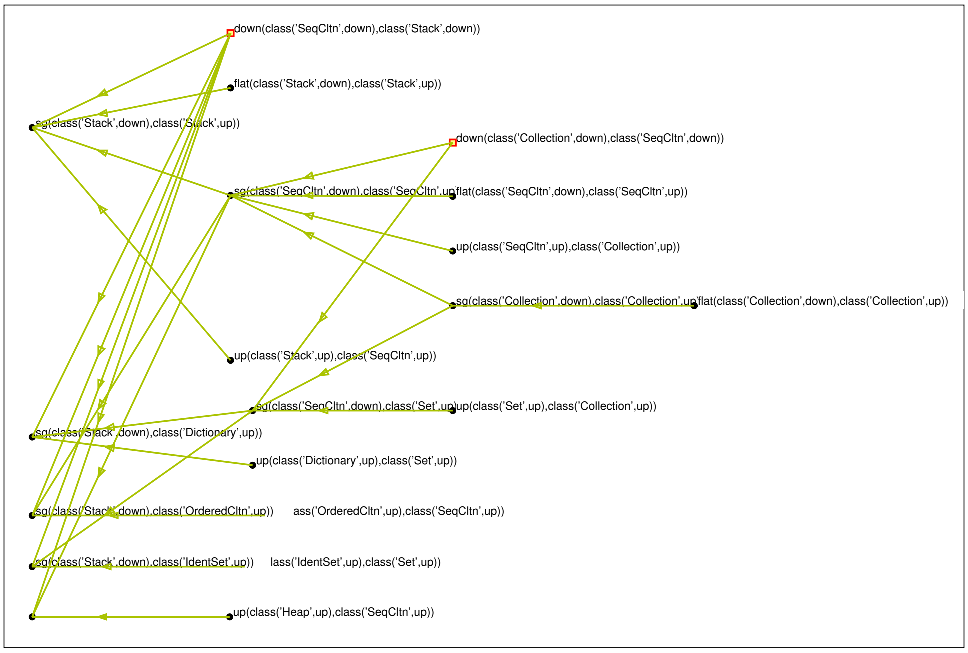


Figure 4.4: The derivation trees for sg(class('Stack',down),X).

derivation trees of the F-DATALOG by adding rules producing a new program $\mathcal{P}$. This encoding is illustrated on the same-generation example by the patterns in Figure 4.3 (with the resulting visualization of the derivation trees presented in Figure 4.4). While, for simplicity, the encoding of the example uses function symbols, they can be simulated in datalog by widening the predicates of the derivation tree. To complete the proof, it suffices to use a tuple of simple projection queries composed with the query defined by program $\mathcal{P}$; this composition yields the desired TF-DATALOG tuple of programs that can express the same filter defined by the original F-DATALOG program.

We make the final remark that the above construction can be easily adapted to demonstrate the equivalence result for other classes of logic programs (i.e., TF-SNR-DATALOG

= F-SNR-DATALOG, TF-STC-DATALOG = F-STC-DATALOG, TF-SNR-LP = F-SNR-LP, and TF-STC-LP = F-STC-LP).

4.5 GraphLog Patterns in Hy$^+$

In this section we make use of the results presented earlier in the chapter to provide a formal definition for the visual language used by Hy$^+$ to express queries and filters.

It is interesting to note that the two examples given in the well known survey [Mye90] of visual languages that have succeeded in the real world are spreadsheets and QBE [ZBM76]. The first one is characterized by [NZ93] as a visual formalism, while the second is a pattern-based query language. It is not coincidental that Hy$^+$ combines the use of the hygraph visual formalism with visual queries expressed as hygraph patterns. The notion of resorting to patterns as a visual notation to describe queries has a popular predecesor in QBE, while the more closely related idea of resorting to graph patterns originates in G$^+$ [CMW87, CMW88].

The visual queries supported by Hy$^+$ are expressions of the GraphLog query language [Con89, CM90b], suitably extended to hygraphs. GraphLog queries are hygraphs whose nodes are labeled by sequences of terms, and whose edges and blobs are labeled by *path regular expressions* on relations. GraphLog has higher expressive power than SQL; in particular, it can express, with no need for recursion, queries that involve computing transitive closures or similar graph traversal operations. The language is also capable of expressing first order aggregate queries as well as aggregation along path traversals (e.g., shortest path queries)[CM90c]. Precise theoretical characterizations of the expressive power of GraphLog and of its computational complexity can be found in the references cited above (and were also summarized in the previous chapter).

Formally, GraphLog *define patterns* (or *query hygraphs*) are hygraphs with no isolated nodes having the following properties: *(i)* the nodes are labeled by terms, *(ii)* each edge and blob is labeled by a literal (either an atom or a negated atom) or by a *closure literal*, which is simply a literal s followed by the positive closure operator, denoted s^+, that can only appear between nodes labeled by sequences of the same length, and *(iii)* there are one or more *distinguished* edges and blobs (drawn thicker), which can only be labeled by positive non-closure literals.

A *GraphLog query* is a finite set of query hygraphs. The semantics of GraphLog queries are given by a translation to stratified linear Datalog (suitably extended, as discussed in the previous chapter, if function symbols are present). Each query hygraph H in a GraphLog query corresponds to a rule r for each distinguished edge and blob, 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 H. An edge or blob of the query hygraph labeled with a closure literal s^+ introduces a predicate defined by additional rules expressing the transitive closure of the predicate in s. The body of r contains the predicates introduced by the closure literals and the remaining edge and blob labels of H.

We allow as expressions of the GraphLog query language only those GraphLog queries whose distinguished edges and blobs define non-recursive predicates. Note that, although we disallow explicit recursion, recursion is nevertheless implicit in the use of closure literals.

We should mention though, that within the Hy$^+$ system explicit recursion in query hygraphs is allowed. This is motivated by the convenience of extending the GraphLog visual notation to be able to express all the queries accepted by the underlying deductive database systems (which do not limit themselves to the evaluation of GraphLog queries). To express the same generation query in Figure 4.1 we did make use of Hy$^+$ ability to express nonlinear queries using the GraphLog visual notation.

The language can be made considerably more concise by generalizing literals and closure literals to arbitrary regular expressions. Each operator introduced is definable in terms of the basic language and is added only for convenience. In addition to the usual operators for positive and Kleene closure, optional (i.e., the operator ? denoting zero or one occurrence), alternation, and concatenation, two new ones are defined: inversion reverses the direction of the edge or blob labeled by the regular expression, and negation negates the predicate defined by its argument.

To summarize the syntax of GraphLog as it is used in Hy$^+$, a *term* is either a constant, a variable, an anonymous variable (an underscore), or a function f applied to a number of terms. Nodes are labelled by terms. Edges or blob labels are expressions generated by the following grammar

$$E \rightarrow E|E; E.E; -E; \neg E; (E); E+; E*; E?; S$$

where S is any literal of the form $p(t_1, \ldots, t_n)$ and $t_i, 1 \leq i \leq n$ are terms.

The syntax of GraphLog *show patterns* is analogous to define patterns, except that: *(i)* nodes can also be distinguished (and they have a special unary predicate associated with them, hence isolated nodes are allowed), and *(ii)* non-negated path regular expressions can label distinguished edges and blobs.

It only remains to formally describe the semantics of GraphLog show patterns. The same translation that produces a stratified linear Datalog program for a define pattern applies to a show pattern, except that the head of the corresponding rule has a new predicate in which all the variables present in the body of the rule appear (we call this new predicate a *match predicate*). The semantics of a set of show patterns is given by the filtering program obtained as the union of the programs for each show pattern, and has: *(i)* the union of the match predicates for each show pattern as the carrier, and *(ii)* each one of the predicates in a distinguished object as a filtering predicate (in particular, if a distinguished edge or blob is labelled by a path regular expression, all of the predicates occurring in it are filtering predicates).

We can see now how distinguishing objects in a show pattern constitutes a natural way of denoting the filtering predicates, and that filtering programs provide a very general way of assigning semantics to the notion of filtering by pattern matching.

Chapter 5

The Hy$^+$ System

In Chapter 2 we introduced the Hy$^+$ system from a user's perspective discussing in detail one of its applications. In addition to the significance of the system as a visual database front-end built around the framework described in the previous chapters, Hy$^+$ can be regarded as a graph visualization system (like [NT90, KS90, BSMW90, DGGR90, HH91]), or even more specifically as a software visualization system (due to the emphasis given to this application domain).

The reader should bear in mind that Hy$^+$ is a general system supporting the creation and filtering of hygraph visualizations. The combination of visualization and querying constitutes one of the most original contributions of the system. The system can be successfully applied to visualize data from any domain as hygraphs, as long as this is a suitable kind of diagram for the intended application. As an example of a potential application area not considered here, we could mention supporting experiment data exploration (and schema exploration) within scientific databases [ILH92].

A comprehensive taxonomy for software visualization systems is presented in [PBS93].

According to this taxonomy, the query capabilities of Hy$^+$ give the system a high rating on scalability, support for varying granularity and elision, and tailorability.

Since the queries in Hy$^+$ are expressed visually, the areas of visual languages and visual programming are closely related to the work presented in the thesis. Concerning visual query languages in particular, a survey of visual query systems can be found in [BCCL91]. Work on graph based approaches to database querying appears in [CMW87, CMW88, GPG90, GG93]. A proposal for a language for querying user-defined visualizations of data can be found in [Cru93].

The two sections in this chapter describe the architecture of the Hy$^+$ system and provide a brief retrospective on its evolution. Within the architecture section we discuss a specific contribution of Hy$^+$ as a graph visualization system: the declarative specification of hygraph layout using hygraph patterns.

5.1 The Architecture of Hy$^+$

The Hy$^+$ system is implemented as a front-end, written in Smalltalk, that communicates with other programs to carry out tasks such as data acquisition, query evaluation, hygraph layout and invoking external programs to browse the objects represented by the visualizations (i.e., editing source code in a software engineering application). The front-end provides browsers that let users interact with the hygraph-based visualizations, as well as supporting parsing, query translation, back-end communication and answer management.

An overview of the Hy$^+$ system architecture is given in the diagram in Figure 5.1. The following discussion is organized according to the main components in the diagram.

The Hy$^+$ system relies on other programs (which are part of the Data Acquisition module) to supply the raw data to be visually manipulated within the system. The File Man-

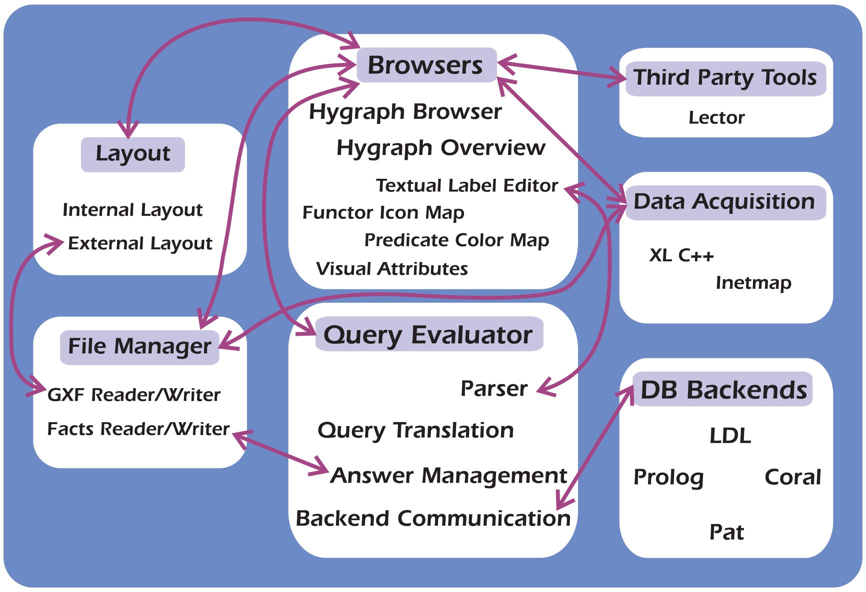


Figure 5.1: Overview of the Hy$^+$ architecture.

ager module can directly import files containing logical facts (like the ones produced by the IBM XL C++ compiler). These files can also be obtained from relational and deductive databases. In addition, the system supports the GXF file format [Eig93], developed internally, which describes not just the logical facts, but also all the positioning and visualization-related information that completely defines the appearance of the data on the screen. Using GXF, a data acquisition tool can provide Hy$^+$ with a specific visualization as a starting point for the querying process supported by the system.

Hy$^+$ executes queries by translating the patterns into back-end programs that are evaluated against the current database hygraph. This translation, as well as the communication with the back-end and the processing of the answers, are carried by the Query Evaluator component. There are three back-end query processors used by the system: LDL [NT89], CORAL [RRS92], and a previously developed GraphLog interpreter implemented in Prolog [Fuk91].

To illustrate the query execution process carried out by Hy$^+$, we discuss below the workings of the Query Evaluator module when CORAL is used as the back-end query processor. The reader is referred to [CMV94] for a more detailed description of the use of deductive database technology in the Hy$^+$ system.

We consider first define patterns: for every hygraph contained in a defineGraphLog blob the translation produces a set of CORAL rules that expresses the relation labelling each of the distinguished edges or blobs 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. If aggregation is present in a GraphLog expression, further rules to compute the aggregate functions must be added. The definition of the new predicates are grouped in CORAL *modules* that export the *query form* that is generated as part of the translation. The resulting CORAL program is kept by Hy$^+$ which sends the program to CORAL only when a Hy$^+$ user invokes the execution of a show pattern.

We now turn our attention to describing the execution of filter queries. Originally, hygraphs in showGraphLog blobs were implemented by translating them to sets of define queries (this is basically executing a filter query as a tuple of traditional queries). Given a hygraph H in a showGraphLog blob, for each distinguished edge (blob) e, a set of define queries are generated that return the filtering predicates labelling e (i.e., when evaluated they determine all e that exist in the portions of the database that match the pattern H). These define queries are constructed as follows: *(i)* if a distinguished edge (blob) e in H is labelled with a predicate, then add to a new hygraph H' (that is identical to H but where all edges and blobs are non-distinguished) a distinguished edge (blob) between the two nodes e connects (between the container and the contained nodes of e), and generate the define query for H'; *(ii)* if e is labelled by a path regular expression (i.e., not simply a predicate), then recursively expand the path regular expression until all generated edges (blobs) are labelled with predicates at which point the process described in case *(i)* is applied. The query evaluator evaluates each of the define queries constructed by sending the programs to CORAL. The results are combined to create a hygraph representing the answer to the filter query. The motivation for the approach described above was to speed up development by directly reusing existing code. The translation was later modified to generate a single CORAL module per filter query in order to reduce the amount of repeated computation.

Hy$^+$ browsers and overviews (which are part of the Browsers module) have extensive facilities for interactively editing hygraphs, including copy, cut and paste; panning and zooming; and textual editing of node and edge labels. In particular, the parser communicates with the label editor, to support immediate checking of the syntax of labels in query patterns.

The Layout component of the system includes both internal layout algorithms, and the ability to communicate with an external suite of layout programs [Noi93b, Noi93a].

An original contribution of the Hy$^+$ system is that the execution of layout algorithms (see [DETT93] for an annotated bibliography on the subject) is integrated with the query evaluation component. In particular, within the Internal Layout component, it is possible to specify three different kinds of layout strategies depending on whether the hygraph element is a blob, edge or node. The meanings of these specifications are as follows.

Blobs Describes the algorithm used to lay out the nodes inside each blob.

Edges Describes in which way (one of directed, undirected, inverted or ignored) the edges within a blob should be treated by the algorithm that positions the nodes in the

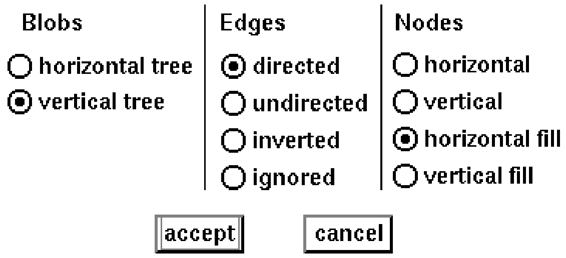


Figure 5.2: Hy$^+$ dialog window for layout boxes

blob.

Nodes Describes how the blobs that are contained in each node (i.e., that are within the blobs region of the node) should be positioned.

The selection of a specific algorithm (of the appropriate kind) for each of the blobs, edges and nodes in a hygraph, can be done interactively or it can be specified using a layout box (as is done throughout the examples in the thesis). In the latter case, when a user creates a layout box in a query, Hy$^+$ presents her with the dialog in Figure 5.2. Execution of the layout box assigns the appropriate kind of layout specification to the corresponding kind (blob, edge or node) of distinguished objects present in the pattern. The declarative layout specification mechanism used by Hy$^+$ can be seen as an extension of the work in [HH91] to include hygraphs, and more importantly, to incorporate the use of pattern-based queries for the specification of subhygraphs.

The system has the ability to invoke external programs that, for instance, browse an object being represented by a node in one of the graphs displayed by the system. The Hy$^+$ visualizations can be used as overviews to locate information and then invoke third-party browsers to display the contents associated with the relevant objects. An obvious advan-

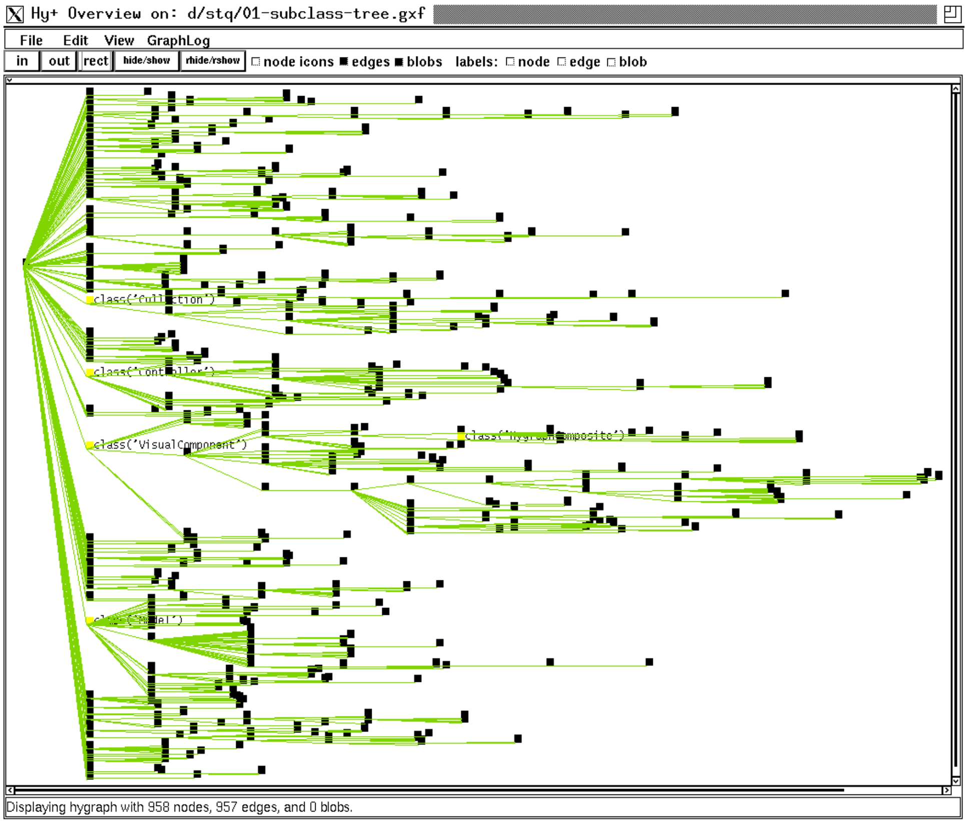


Figure 5.3: Class hierarchy of the Smalltalk image used for Hy$^+$ development.

tage of this approach over a purely navigational one, is the ability to use the convenience and expressive power of GraphLog patterns to retrieve the objects of interest, instead of attempting an often impractical brute force search. This is in addition to the use of hygraph patterns to generate as many specifically tailored overviews as needed.

Hy$^+$ support for the integration of third party tools is part of the overall goal of providing an open architecture. This integration can be used, for example, to support network management stations that require the co-existence of several browsers provided by different

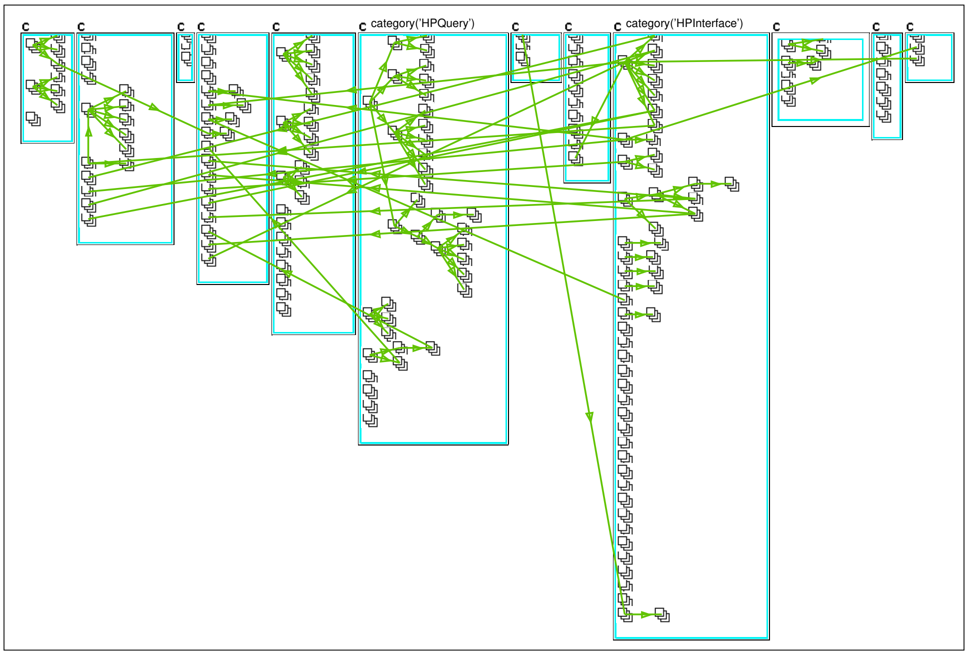


Figure 5.4: Visualizing the classes in Hy$^+$ grouped by categories.

monitoring tools. In this way, the user can resort to Hy$^+$ to create and filter visualizations of network-related information, and in addition he can navigate, in hypermedia style, through the information contained in management information bases. Another example of combining Hy$^+$ with other tools is presented in the next section.

The design of Hy$^+$ takes advantage of the object-oriented development framework provided by Smalltalk. An overview of the inheritance tree of the classes in a development image of Hy$^+$ is presented in Figure 5.3. There are almost one thousand classes available to the Hy$^+$ developer. In particular, an interesting subtree of the hierarchy is rooted at class Visual Component. This class is the abstract class for the Smalltalk widget hierarchy. As such, it is the parent of the class HygraphComposite, the ancestor class for several of the widgets used to display hygraphs in Hy$^+$. For the actual rendering of hygraphs in Hy$^+$ the system uses the classes which are part of ParcPlace's Smalltalk Portable Imaging Model.

Smalltalk classes are grouped in categories. The diagram in Figure 5.4 shows the twelve categories containing the Hy$^+$ code and the almost 200 classes in them. The picture also shows the inheritance relationship among the Hy$^+$ classes. Not shown in the diagram are the parent classes which are part of the Smalltalk development environment. As expected, two of the major categories in Hy$^+$ are HPQuery and HPInterface, which correspond to the Query Evaluation and Browsers components of the architecture diagram in Figure 5.1.

The implementation of Hy$^+$ benefits from the object-oriented approach not only for the user interface aspects of the system. In particular, the Query Evaluation component achieves a significant amount of code reuse in supporting a variety of different database back-ends.

5.2 Hy$^+$ Evolution

The architecture of Hy$^+$ reflects the experience acquired during the development of earlier systems. The first one, originally described in [Con89], evolved into the G$^+$/GraphLog Visual Query System [CM90a]. A retrospective on the system can be found in [CCM92], while a description of its architecture is given in [CKM91]. A subset of the GraphLog queries were supported by the system (and they were evaluated within the Smalltalk image, using graph traversal algorithms).

A G$^+$/Graphlog visualization of a flights database instance appears in the large window of Figure 5.5. The flights database contains information about a few airlines that connect several cities; its instance visualization has city names labelling nodes, and airline codes labelling edges. This graph overview can be used as a starting point for navigating and inspecting the contents of the database. For instance, a postcard of Montevideo is displayed

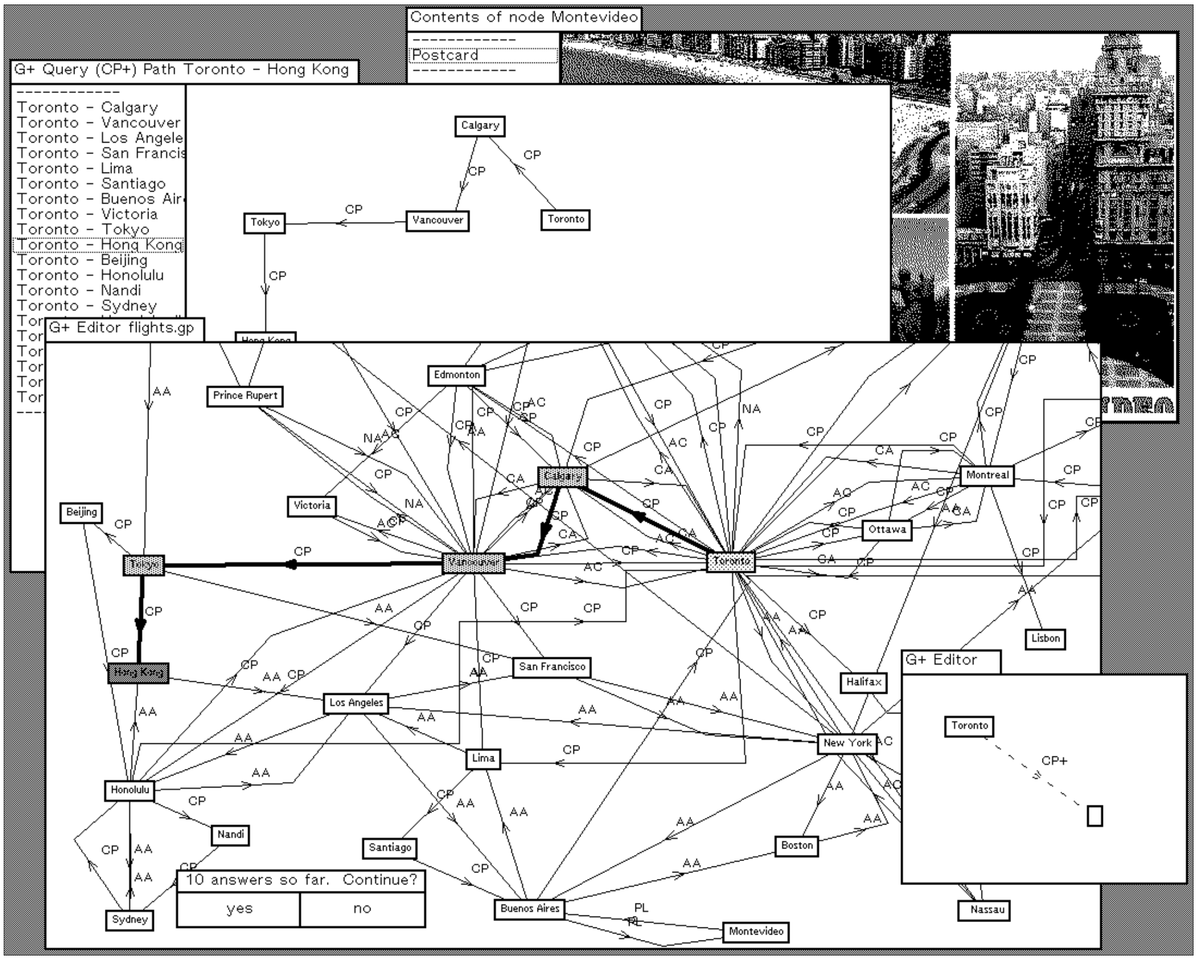


Figure 5.5: Showing flight connections in context and individually.

in the top left corner.

The pattern in the small window on the bottom right corner of the figure matches paths of CP (Canadian Pacific) flights originating in Toronto. As a result of evaluating the visual query one possible trip with CP from Toronto to Hong Kong is shown highlighted in the large window. This is the tenth answer being shown to the user, after being prompted for them one by one. The window in the top left corner shows an alternative way of displaying

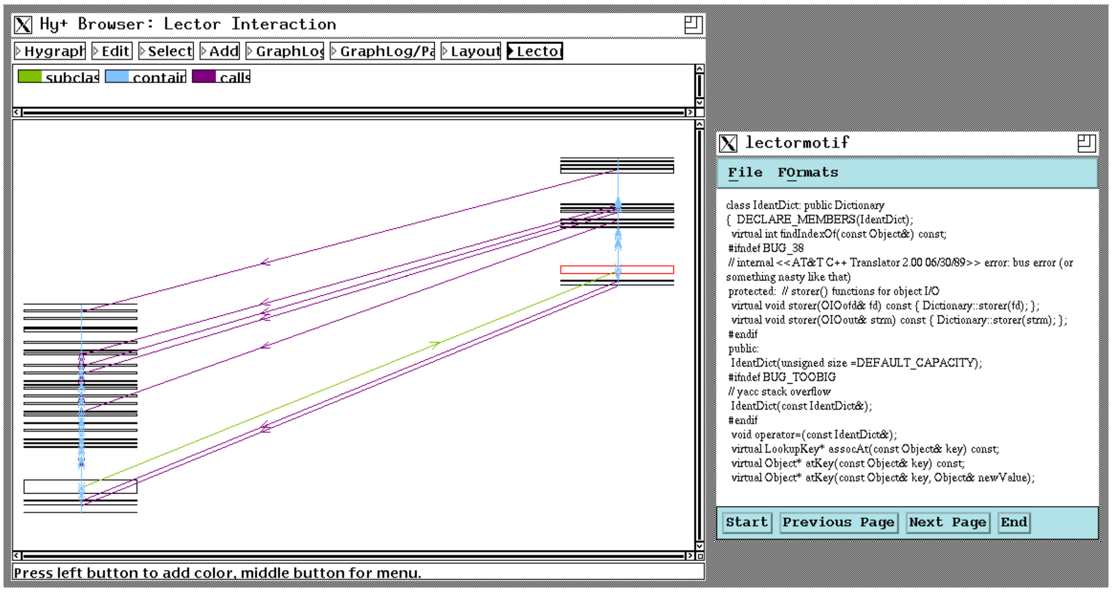


Figure 5.6: Synchronized graphical and textual browsing of source code.

the answer to the same query by collecting in a separate window all paths that answer the query. In this case, the same trip from Toronto to Hong Kong is being visualized in isolation, and we can see a list with 20 other possible trips out of Toronto. The one-by-one mode for returning the answers to a query supported by G^+/Graphlog is particularly useful to debug a query.

The current Hy^+ System (which shares no code with the G^+/GraphLog system) had an immediate ancestor that relied only in the Prolog to GraphLog translator [Fuk91] for query evaluation. Figure 5.6 shows an example of the integration in that earlier version of Hy^+ (presented in [CM93a]) of the Lector[1] [Ray92] text browsing tool. To the right of a specialized hygraph browser there is a Lector window that displays the source code associated with

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

the object selected in the browser (the code for the NIH class IdentDict). The display synchronization works both ways: when the user changes the page of source code displayed by Lector the object selected in the Hy$^+$ browser adjusts accordingly. Furthermore, the query evaluation component was extended to handle a mixture of traditional and textual queries [Yeu93]. The latter kind of queries are handled by the PAT Text Searching Engine [Gon87], which was incorporated as an additional query processor back-end.

Chapter 6

Hy$^+$ Applications

This chapter presents several applications of the Hy$^+$ System. In Chapter 2 we discussed extensively the use of Hy$^+$ to explore a `C++` library. Clearly, the same ideas are applicable regardless of the programming language. In this thesis, Chapter 5 shows diagrams created by Hy$^+$ from Smalltalk code and the figures in Chapter 4 illustrate the use of Hy$^+$ in the context of a logic programming environment. In addition, [CMV94] presents an application of Hy$^+$ using data obtained from Object Oriented Turing source. The Object Oriented Turing environment [MHP93] employs a visual formalism suitable for programming-in-the-large (called the Software Landscape [Pen93]) which, once reproduced within Hy$^+$ can be analyzed and explored using GraphLog patterns. We will discuss below how the use of a visualization tool like Hy$^+$ can be equally beneficial at other stages in the development cycle.

Software systems may be viewed as consisting of a set of components that use or depend on each other. The components of a system are usually grouped into modules that are units of conceptual organization or work assignment. A module is said to use, or depend on, another one if it contains a component that uses a component contained in that other module (this concept is analogous to the class usage relationship defined in Figure 2.20). While designing a software system it may be advantageous to ensure that this induced relation is acyclic, since later on this may simplify code maintenance (ripples caused by changes to a module then propagate strictly up the dependency hierarchy). The large number of components in typical systems makes the task of reducing or eliminating cyclic dependencies difficult to perform manually. In [CMR92] the application of GraphLog to the above design task is discussed. This is one possible scenario for the application of Hy$^+$ during the design step.

The applicability of a hygraph visualization tool to software engineering is hardly surprising. The following two paragraphs from Brooks [Bro87] deserve a comment at this point.

> As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another.
>
> In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools.

The admission that a multiplicity of graphs is appropriate for diagramming the structure of software systems seems in contradiction with the statement that software is unvisualizable. Obviously, we do not believe that software cannot be visualized. However, we do agree that software structure is inherently difficult to display. It is precisely because of this difficulty that powerful tools with the capabilities of Hy$^+$ are sorely needed.

The following three sections discuss applications of Hy$^+$ at different stages of the software development process. The final section refers to the applicability of the approach implemented in Hy$^+$ for the support of network management tasks. The first application is described in detail, while for the others a more concise presentation is given (the interested reader is referred to the appropriate publications for further examples and motivations).

6.1 Partitioning Code into Overlay Modules

In this section we explore one very specific problem that can be characterized as part of the application deployment stage, and particularly as a form of performance tuning. This example is also presented in [CMR92], but there the emphasis is given to the suitability of GraphLog as a visual notation to express queries in the context of a software engineering environment. Below, we highlight the advantages of our approach for supporting the declarative creation and filtering of visualizations in the context of the same example. The very particular characteristics of the application are taken as illustrative of many of the "one of a kind" situations that software engineers face during the lifetime of many of their projects. A generic visualization tool, such as Hy$^+$, provides powerful capabilities to analyze the information related to the problem, while remaining a cost-effective approach to deal with these ad-hoc situations.

Computer programs often have memory limits and performance requirements imposed on them. A common method for reducing the former is to partition the code into overlay modules that can be independently loaded as the program executes. In this way, the code fits into a fixed amount of main memory and the memory occupied by code that is no longer needed may be reused by other modules. However, if this partitioning is not done carefully, performance may be severely degraded by the overhead of loading the modules, a condition known as *disk thrashing*.

Designing a good code overlay structure for programs that have thousands of functions is a very difficult task. During the development of ImagEdit V1.0, a PC-DOS application, Prolog was used as a design aid [Rym91] for this problem. In what follows we elaborate on the hypothetical use of Hy$^+$ to support the design of an overlay structure for ImagEdit (using the original data). Partitioning is critical under the PC-DOS operating system, which has a strict limit of 640KB for the size of the executables. The issue of partitioning code is not just an oddity of one specific OS, since it continues to be a problem under Microsoft Windows and OS/2 (and there are commercial tools to address this specific issue [Yao91]).

We regard the code as a set of functions that call each other. The code is partitioned into *sections*, which are load modules. Each section consists of a set of functions and a set of overlay *areas* that are arranged in *series* in memory. The memory requirement of a section is therefore the sum of the memory requirements of its functions and areas. An area consists of a set of sections that are arranged in *parallel* in memory. The memory requirement of an area is therefore the maximum of the memory requirements of its sections. Clearly, the way to reduce overall memory requirements is to place as much code in parallel sections as possible, without causing disk thrashing.

A series-parallel map of the sections, areas and functions is given in Figure 6.1. Layout patterns were used to arrange areas horizontally (they are in series) and sections vertically (they are in parallel). The information used to produce the visualization comes from three kinds of facts reported by the linker: section_function(S,F) denotes that section S contains function F, section_area(S,A) that section S contains area A, and area_section(A,S) that area A contains section S. Although not shown in Figure 6.1, facts of the form calls(F1,F2), stating that function F1 calls function F2, were also available (obtained from a cross reference utility).

The sections, areas, and functions form the nodes of a tree whose leaves are the functions. Figure 6.2 shows the tree defined by the concatenation of the relations section_area

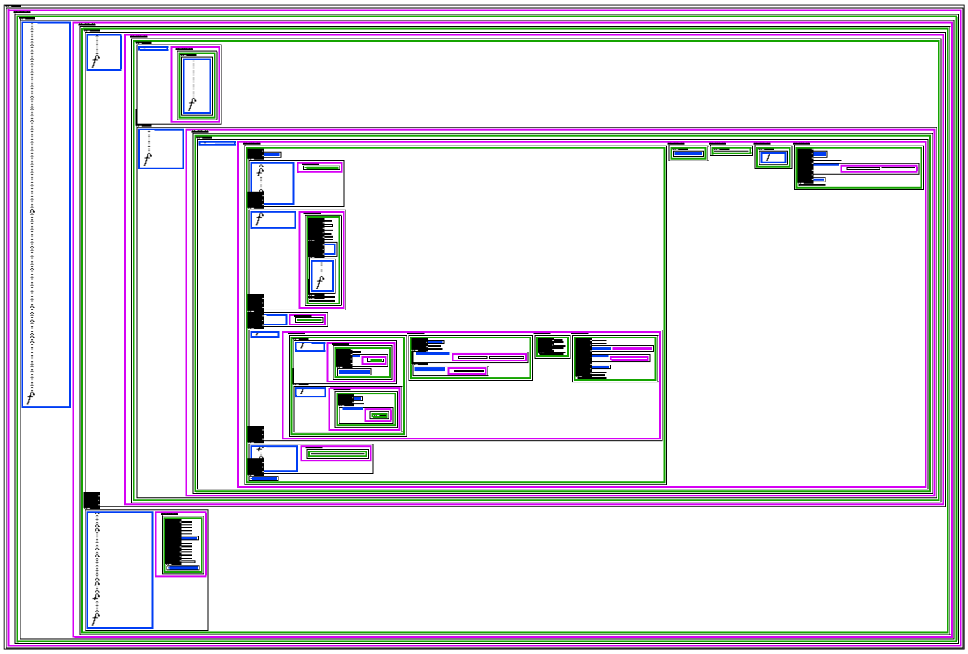


Figure 6.1: Nested blob display of the memory overlay map.

and area_section. Figure 6.3 has a nice colorful view of a small subset of the sections, areas, functions and function calls in ImagEdit. However, an attempt to display all of the functions and function calls in one graph produces a messy image with a red blur (due to the red calls edges, of which there are over five thousand).

In situations like this we have to resort to Hy$^+$ abilities to *abstract* information and then produce a filtered view that presents less data at a higher level of abstraction (or at a coarser granularity). The patterns in Figure 6.4 define a new relation section_call to abstract call patterns among functions at the level of sections, and then display the new relationship on top of the section area tree. The result, in Figure 6.5, while readable, is still quite cluttered. This is a statement on how complicated the reality of the call patterns are for the ImagEdit application. This kind of complexity may very well be justified by the application, and soft-

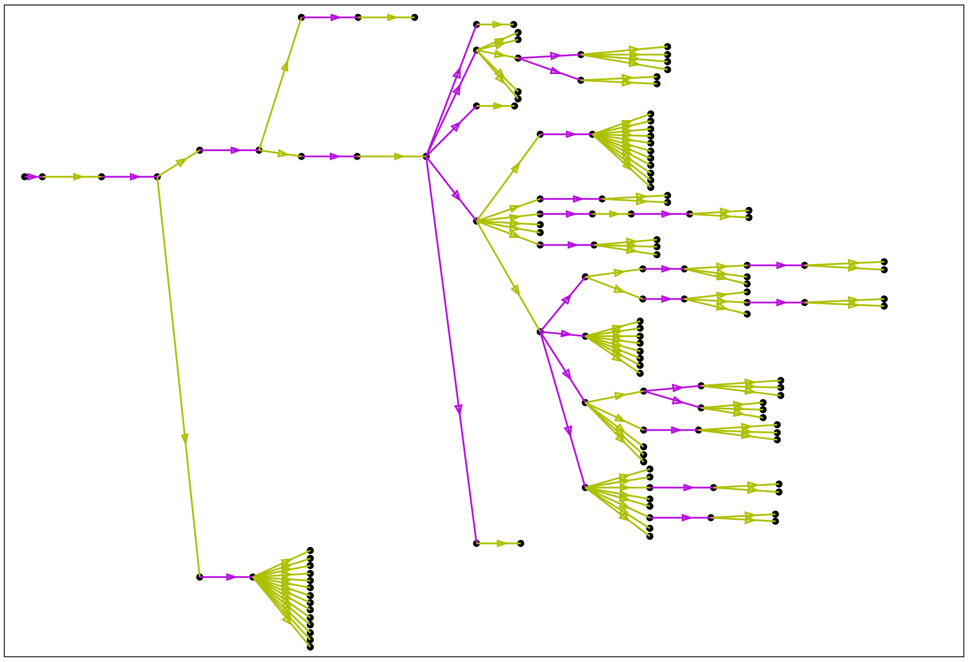


Figure 6.2: Tree of the sections and areas.

ware engineers are forced to deal with it.

The dynamics of the loader can now be described. When the program is executed, a root section is loaded and remains resident in memory until termination. All other sections are loaded on demand, and the loader guarantees that all ancestors of a section in the section area tree are in memory whenever the section is. When a section is loaded it may *overwrite* other sections (this is necessary since otherwise the memory requirements of the code would not be reduced). For example, parallel sections in the same overlay area overwrite each other. Sections overwrite each other when they are either directly or indirectly contained in distinct parallel sections.

The GraphLog definition of the overwrites predicate is given in the leftmost blob of Figure 6.6. Section S1 overwrites section S2 if there exist two distinct sections SP1 and

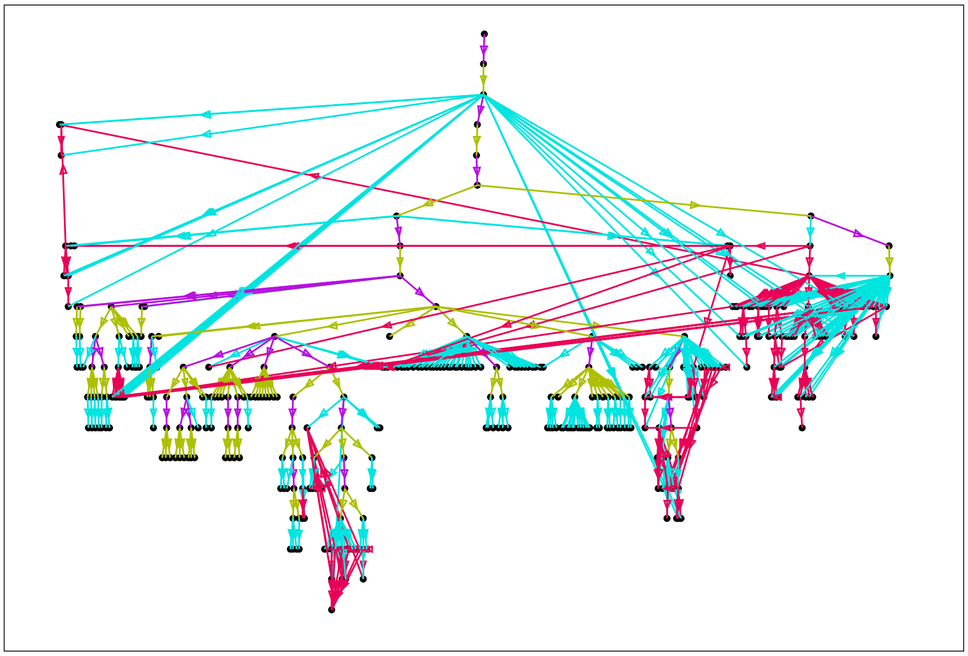


Figure 6.3: A portion of the overlay tree with a few functions and calls.

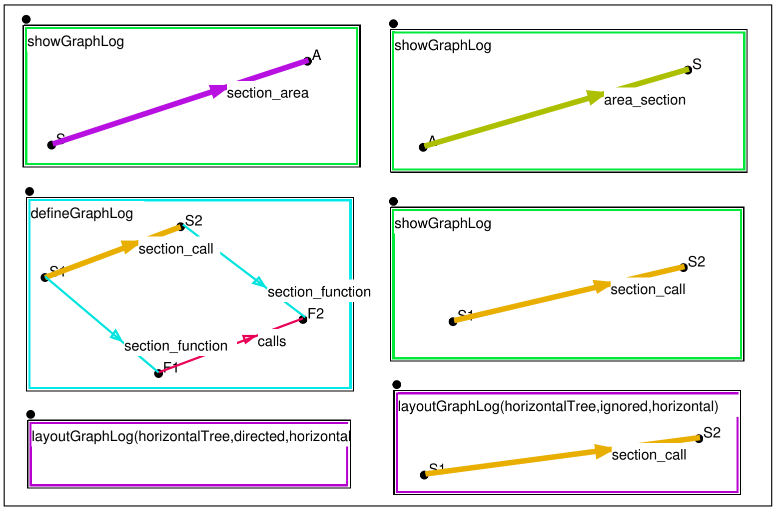


Figure 6.4: Abstracting call patterns at the section level.

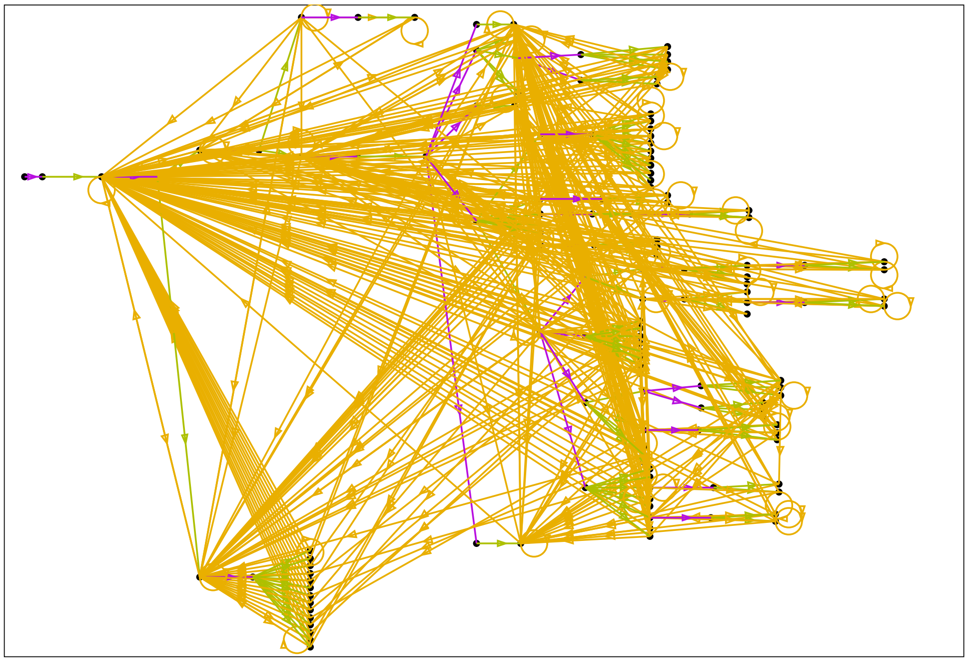


Figure 6.5: The cluttered reality of section_call.

SP2 at the top level of the same area such that SP1 (resp. SP2) is an ancestor of S1 (resp. S2), or it is S1 (resp. S2) itself (notice the use of Kleene closure in the regular expression label).

The loader's normal mode of operation, which is adequate for programs whose call topology is essentially tree-like, is very restrictive in general. If a call is made to a parallel section, the loader will overwrite (smash) the caller with the called section, thereby invalidating the code stored at the return address and resulting in a system crash when control is returned to the caller. To address this problem, the loader possesses another mode of operation, called *reload*. In this mode, all calls between sections are vectored, and a return stack is maintained by the loader. When a function call returns, the loader checks that the section being returned to is still in memory, and if not, reloads it. This feature has the advantage that no

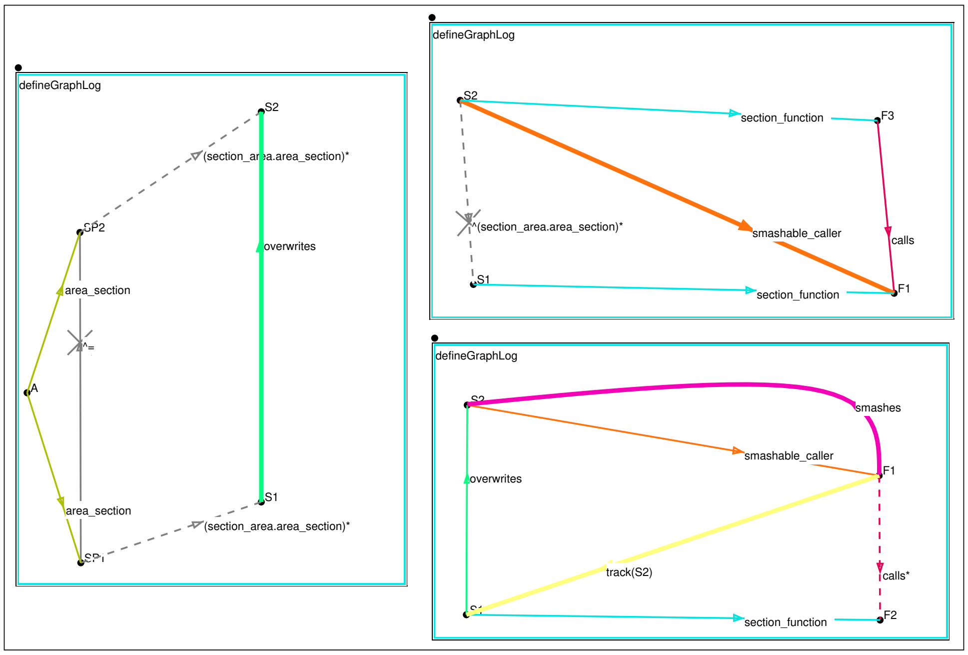


Figure 6.6: GraphLog definitions for track using two intermediate predicates.

errors can occur from returning to overwritten sections. Unfortunately, the reload feature introduces a severe performance penalty.

In order to keep the code space reduction afforded by the reload feature but retain acceptable performance, a programmer can use a selective form of reload, called *track*. Unlike reload, which applies to all calls, track can be used for selected calls. Any call to a tracked function is placed on a return stack by the loader, and its caller is reloaded if necessary prior to return. However, specifying the functions to track is very difficult without a careful analysis.

Recall that while any function is executing, its section and all of its ancestors in the section area tree will be in memory (the loader handles this automatically); thus only calls to the

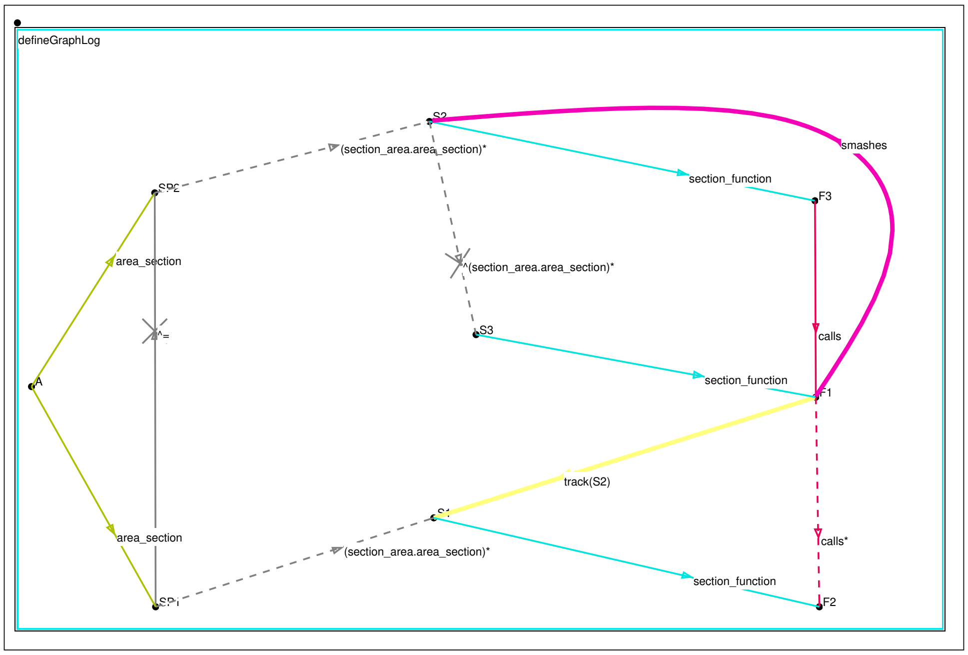


Figure 6.7: GraphLog definition for track in one pattern.

function from sections that are not ancestors or at the same level need to be considered. We say that section S2 is a *smashable caller* of function F1 (contained in section S1) if: *(i)* S2 contains a function F3 that calls F1; and *(ii)* S2 is not an ancestor in the section area tree of the section S1. The GraphLog definition of the smashable_caller predicate is given in the top right blob of Figure 6.6. Section S2 is potentially smashable because it is not guaranteed to be in memory while F1 is executing.

Finally, the predicate track(F1,S1,S2) means that the function F1 needs to be tracked because it causes the section S1 to smash the section S2. The calling section S2 is smashed if a function F2, called directly or indirectly by F1, belongs to a section S1 that overwrites S2. In this situation it is also useful to define the predicate smashes(F1,S2) to indicate that S2 gets effectively smashed by F1. The GraphLog definitions are presented in the bottom right blob of Figure 6.6. Alternatively, the definitions of track and smashes can be provided in one query graph, without the need for defining the intermediate predicates overwrites and smashable_caller (see Figure 6.7).

It is illustrative to compare the GraphLog notation with the Prolog code that was actually used in the ImagEdit project, and with a simplified version of the Datalog program produced by Hy^+ from the definition of track. The Prolog definition of track (fully described in [Rym91]) is listed in Figure 6.8 to show the amount of effort required to produce an efficient Prolog implementation of track. This program was difficult to write and is difficult to understand. The Datalog definition of track is listed in Figure 6.9. While there can be subjective arguments about the comparative benefits (e.g., understandability, brevity) of the three notations, one objective comparison is the number of variable occurrences in each definition. The GraphLog definition of track has 10, while the Prolog program has 154 and the Datalog definition has 67. This is an objective argument about the conciseness of the GraphLog visual notation. Conciseness has been singled out as a problematic area for visual languages in general [Mye90] (and for "boxes and arrows" ones in particular).

The track predicate was used in two ways. First, the designer used it as an aid to detect and eliminate undesirable code smashes. Second, it was used nightly by the build process to automatically generate correct linker control files. This was necessary since code changes made during the day could have changed the function call graph thereby invalidating the calculation of which functions to track. Having a tool that automatically generated correct linker control files allowed the development team to significantly reduce their product's memory requirement while maintaining acceptable performance.

There was little support however for the first task of the designer, namely eliminating undesirable code smashes. This kind of redesign task could have been definitely better sup-

```
track(G,R,E) :-
  smashable(G,C),
  smash(C,[G],[],R,E).

smashable(G,C) :-
  section_function(B,G),
  findall(X,smashable_caller(X,G,B),L),
  set(L,C).

smashable_caller(X,G,B) :-
  calls(F,G),
  section_function(X,F),
  not(le(X,B)).

smash([],_,_,_,_) :-
  !,
  fail.

smash(_,[],_,_,_) :-
  !,
  fail.

smash(S,[F|_],_,R,E) :-
  section_function(R,F),
  member(E,S),
  overwrites(R,E).

smash(S,[F|T],H,R,E) :-
  findall(G,calls(F,G),L),
  update_pending(T,L,[F|H],P),
  smash(S,P,[F|H],R,E).

member(E,[E|_]).

member(E,[_|L]) :-
  member(E,L).

overwrites(X,Y) :-
  area_section(_,X),
  area_section(_,Y),
  not(le(X,Y)),
  not(le(Y,X)),
  le(A,X),
  area_section(Area,A),
  le(B,Y),
  A<>B,
  area_section(Area,B).
le(root,root).

le(X,X) :-
  area_section(_,X).

le(X,Y) if
  ancestor(X,Y).

ancestor(X,Y) :-
  immediate_ancestor(X,Y).

ancestor(X,Z) :-
  immediate_ancestor(Y,Z),
  ancestor(X,Y).

immediate_ancestor(X,Y) :-
  area_section(A,Y),
  section_area(X,A).

update_pending(O,[],_,O).

update_pending(O,[G|T],H,N) :-
  member(G,H),
  !,
  update_pending(O,T,H,N).

update_pending(O,[G|T],H,N) :-
  update_pending(O,T,H,M),
  adjoin(G,M,N).

adjoin(X,A,A) :-
  member(X,A),
  !.

adjoin(X,A,[X|A]).

set([],[]).

set([H|T],S) :-
  member(H,T),
  !,
  set(T,S).

set([H|T],[H|S]) :-
  set(T,S).
```

Figure 6.8: Prolog definition of track.

```
track(F1, S1, S2) :-
  smashable_caller(S2, F1),
  kl_closure_calls(F1, F2),
  section_function(S1, F2).
  overwrites(S1, S2).

kl_closure_calls(X, X) :- calls(X, Y).
kl_closure_calls(Y, Y) :- calls(X, Y).
kl_closure_calls(X, Y) :- calls(X, Z),
  kl_closure_calls(Z, Y).

smashable_caller(S1, F2) :-
  section_function(S1, F1),
  calls(F1, F2),
  section_function(S2, F2),
  not kl_closure_contains(S1, S2).

overwrites(S1, S2) :-
  area_section(A, SP1),
  area_section(A, SP2),
  not SP1 = SP2.
  kl_closure_contains(SP1, S1).
  kl_closure_contains(SP2, S2).

kl_closure_contains(X, X) :- contains(X, Y).
kl_closure_contains(Y, Y) :- contains(X, Y).
kl_closure_contains(X, Y) :- contains(X, Z),
  kl_closure_contains(Z, Y).

contains(X, Y) <- section_area(X, Z),
  area_section(Z, Y).
```

Figure 6.9: Datalog definition of track.

ported by a visualization tool that would have allowed the programmer to analyze the smash information in context. In what follows, we present two examples that illustrate how Hy$^+$ could have been used for this smash analysis task.

A good starting point for the smash analysis task would be a memory map with the actual smashes shown. Notice that the only functions that are of interest are the ones that must be

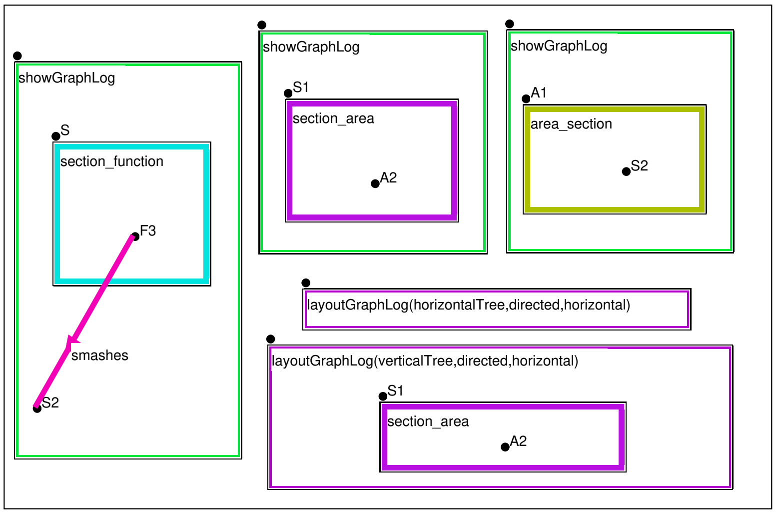


Figure 6.10: Focusing on the smashed functions.

tracked, and for them we would like to know which section they are smashing. The query in Figure 6.10 produces the desired display, which is shown in Figure 6.11 (where the contents of several blobs as well as the node at the bottom left have been hidden).

The programmer must then try to reduce the number of functions that need to be tracked, by relocating the appropriate functions in different sections. In this application there were in the order of one hundred functions to be tracked. Obviously, not all of the functions tracked will have the same impact on the performance of the program. Therefore, the programmer must combine profiling data with the overlay data to decide for which tracked functions it is worth spending the time and effort trying to relocate them to avoid a smash.

Suppose that the programmer has established that function deskrun is critical for the performance of the application, and that every effort should be made to avoid tracking it. This requires obtaining a diagram that shows the reasons for which deskrun is currently being tracked, and then try to eliminate them by shifting functions around sections (or even

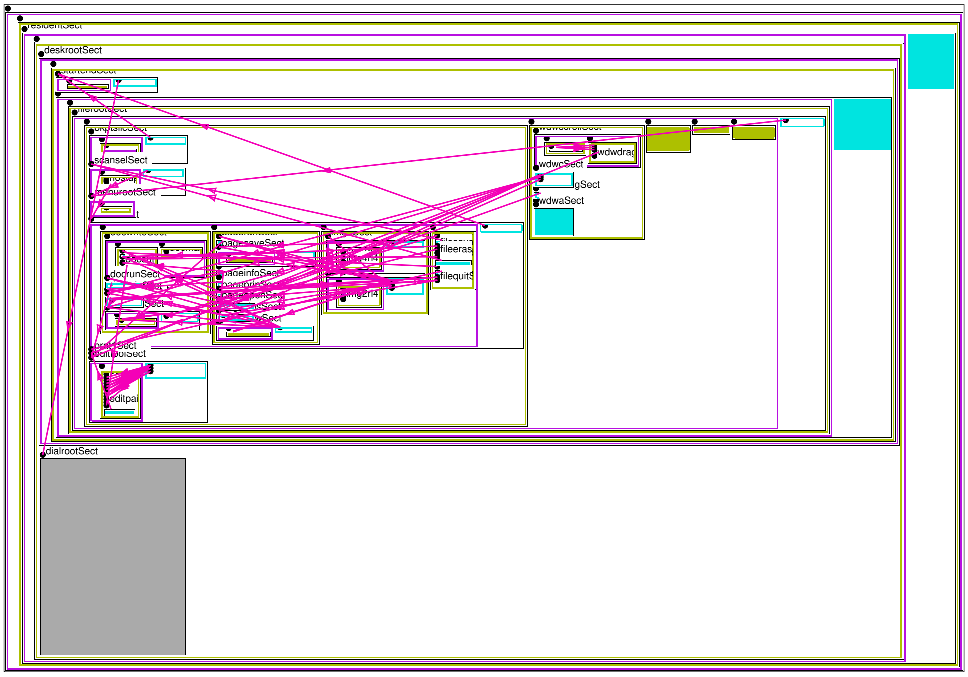


Figure 6.11: The filtered memory map with the smashed functions.

re-shaping the area section tree). The two show patterns on the left of Figure 6.12 provide such an explanation. The top one checks whether it is the case that deskrun itself is in a section that overwrites the smashed section. The answer (on the right of the show pattern) is affirmative: main (contained in the smashed section startendSect)) calls deskrun which is in the section deskrunSect that overwrites startendSect. So the programmer will have to do something about deskrun itself. This may not be enough, though. The show pattern at the bottom finds the functions called by deskrun that are in sections that overwrite startendSect. The answer (once more on the right) shows that five functions are called and that they are included in three different sections that overwrite startendSect.

It is important to stress that the kind of filtering presented in the last example is typical of

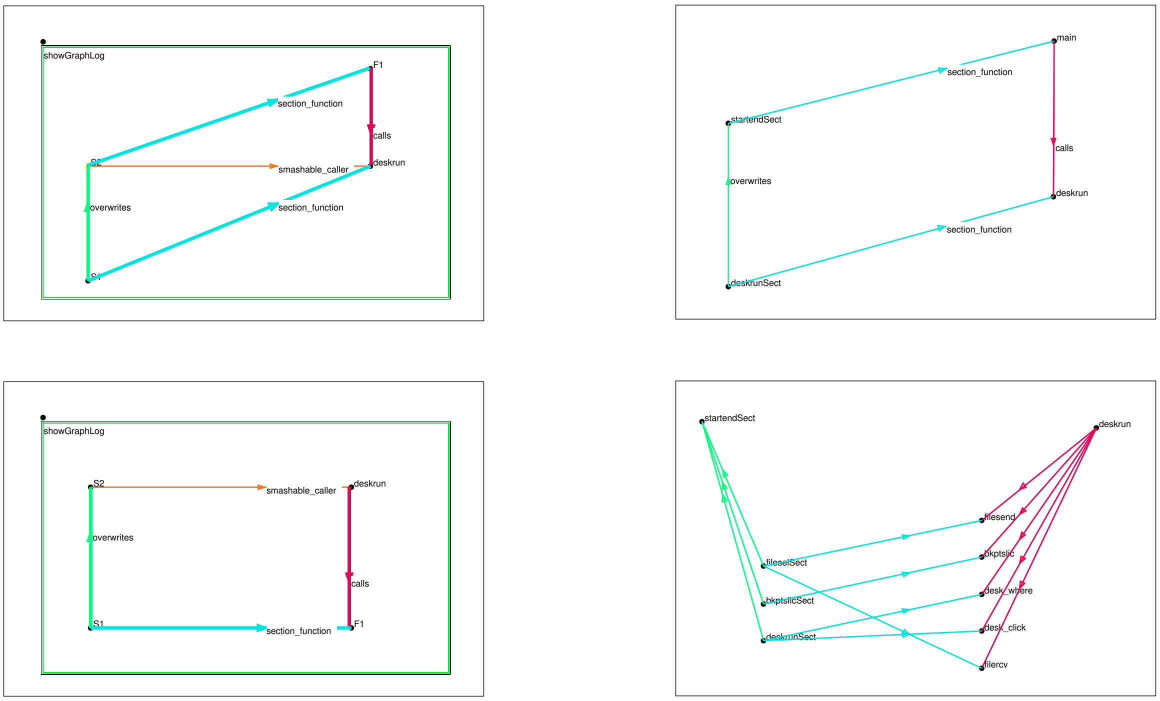


Figure 6.12: An explanation for the tracking of deskrun.

situations in which somebody must not just find the objects that satisfy complex conditions (in the overlay example, finding functions that must be tracked); the person must also analyze the situation to find an *explanation* of why the condition holds, and then try to change it.

6.2 Software Configuration Management

The way software products are designed and released typically requires that several versions of every software component built be maintained. There are several reasons for the existence of multiple versions: the previous deployment in the field of older software components that need to be fixed or upgraded, the necessity to create several variations of the

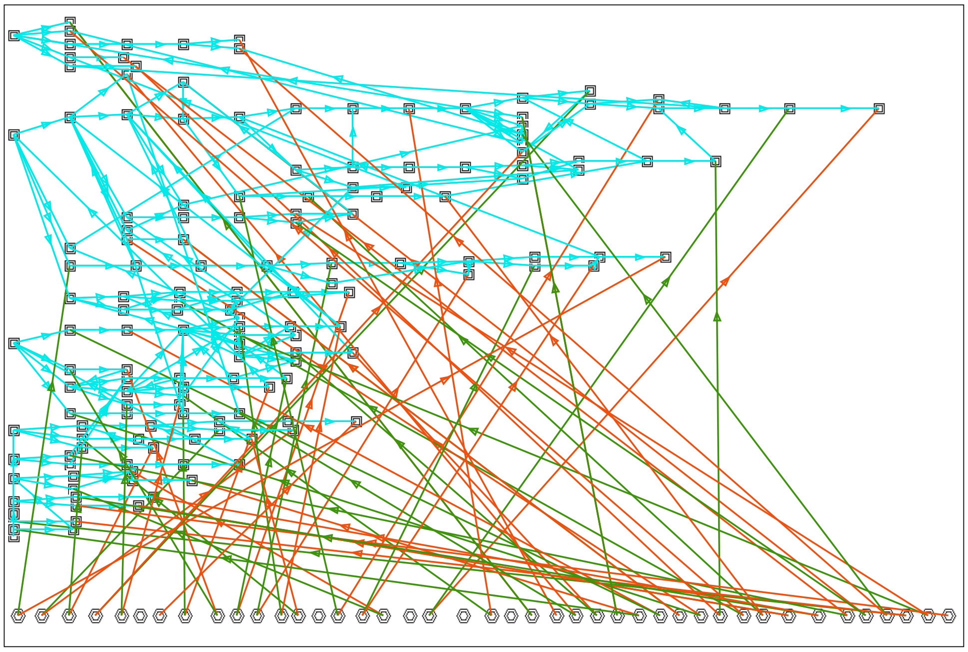


Figure 6.13: Relating versions and branches to streams.

same components for distribution in a heterogeneous environment, and so on. Software engineers working in large teams need access to the different versions on a frequent and ongoing basis. Software configuration management systems are used to support the task of maintaining and accessing the versions of objects created in the software process.

Figure 6.13 displays configuration information taken from data stored within the IDE [Ide93] configuration management environment. The sample data was extracted from a commercial OODBMS that serves as the repository for the IDE entities. The versioning model adopted by IDE is a named branch-versioning scheme wherein the versioned objects are called *nodes*. These nodes represent, or contain, pieces of information, such as files or other nodes, for which history must be tracked and access rights checked. The actual con-

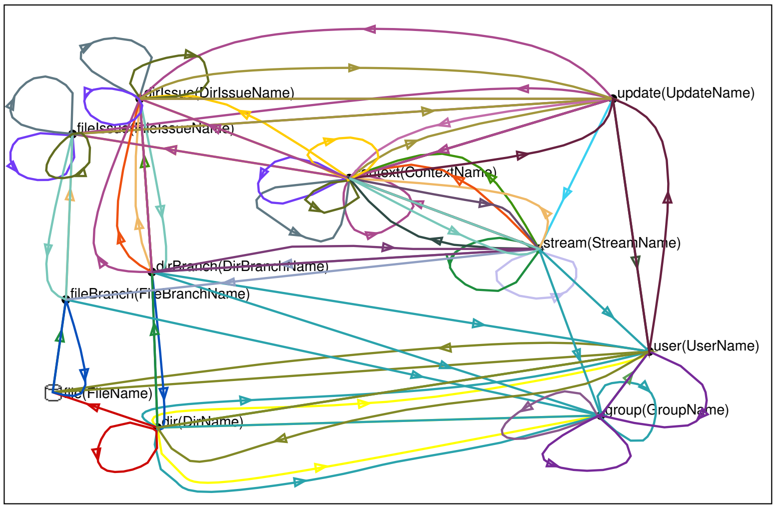


Figure 6.14: The schema of the configuration management data.

tent of the files is not limited to source code, but it includes documentation at all levels, test cases, and a variety of other information related to the application development process. The top portion of Figure 6.13 shows a tree representing the versions (and branches) of a few IDE nodes owned by a specific programmer. At the bottom of the figure there is a row of *streams* (threads that tie together items that are all targeted towards the delivery of a common end product), each one pointing to the first (with a green edge) and last (with a red edge) version within the stream. The layout of such a picture required the creation of two blobs, one grouping the nodes and the other the streams, that were removed from the diagram once the appropriate layout algorithms were executed for each one of the blobs.

The schema of the information extracted from IDE (shown in Figure 6.14) is certainly non-trivial, and reflects not just the object-oriented design of the data stored in the IDE repository, but also the inherent complexity of an industrial strength configuration management system. In the scenario in which IDE is used, hundreds of programmers need to access

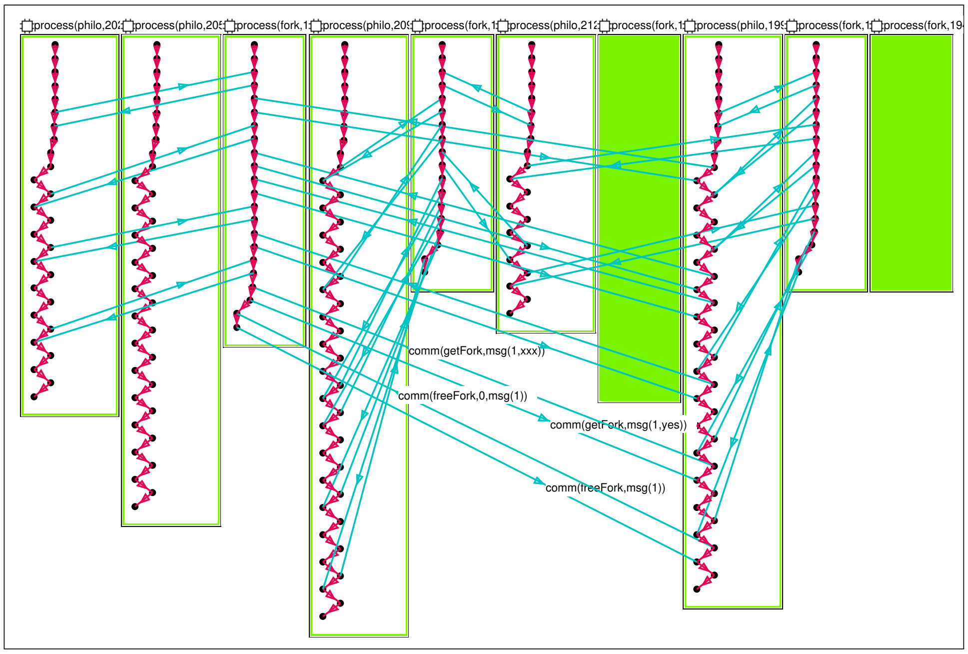


Figure 6.15: Inter-process communication hygraph.

the complex data in the repository on an everyday basis to carry on their development tasks. In this context, it is clearly important to provide programmers and especially their managers, with powerful tools to visualize and query the configuration database.

6.3 Debugging Traces of Distributed Applications

Hy$^+$ can be used to support the analysis of the behaviour of parallel and distributed programs [CHM93, CHM94]. Debugging distributed applications is usually performed by looking at execution traces produced by one or more executions. It is quite common that huge volumes of trace data are produced as a result. Hence, we are once more confronted with a situation

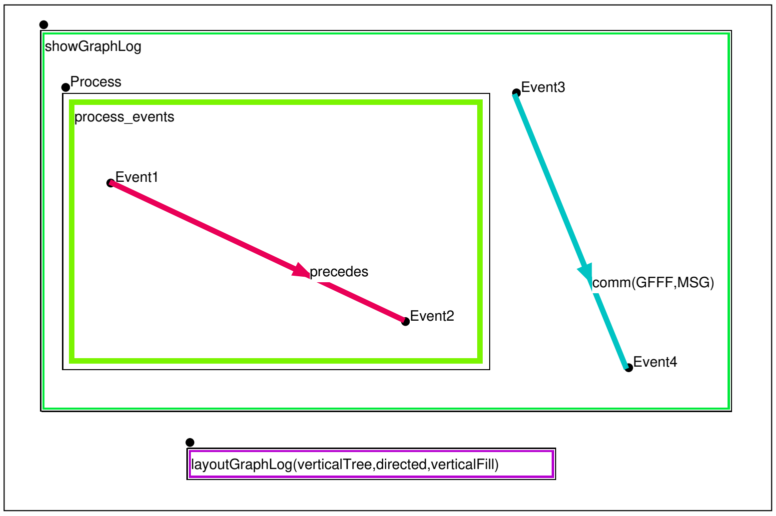


Figure 6.16: Showing the events, processes and messages.

where large amounts of complex abstract information must be analyzed. As expected, the programmer wants to look at the traces at a level of *abstraction* that matches his model of the application. In this application, Graphlog patterns can be used to define appropriate abstractions from the traces as well as to specify normal and abnormal patterns of behaviour for the particular application.

The hygraph diagram in Figure 6.15 represents the exchange of messages among processes (representing philosophers and forks) obtained from the traces of a distributed version of the dining philosophers problem. The query used to create the diagram (shown in Figure 6.16) is equally applicable to the traces of any other distributed application (i.e., it can be considered part of a library of generic queries for the analysis of event traces).

The graphs in Figure 6.17 illustrate one possible kind of abstraction that can be expressed in GraphLog. From the raw trace the programmer formulates GraphLog queries (specific to the dining philosophers application) showing the left and right fork initialization patterns.

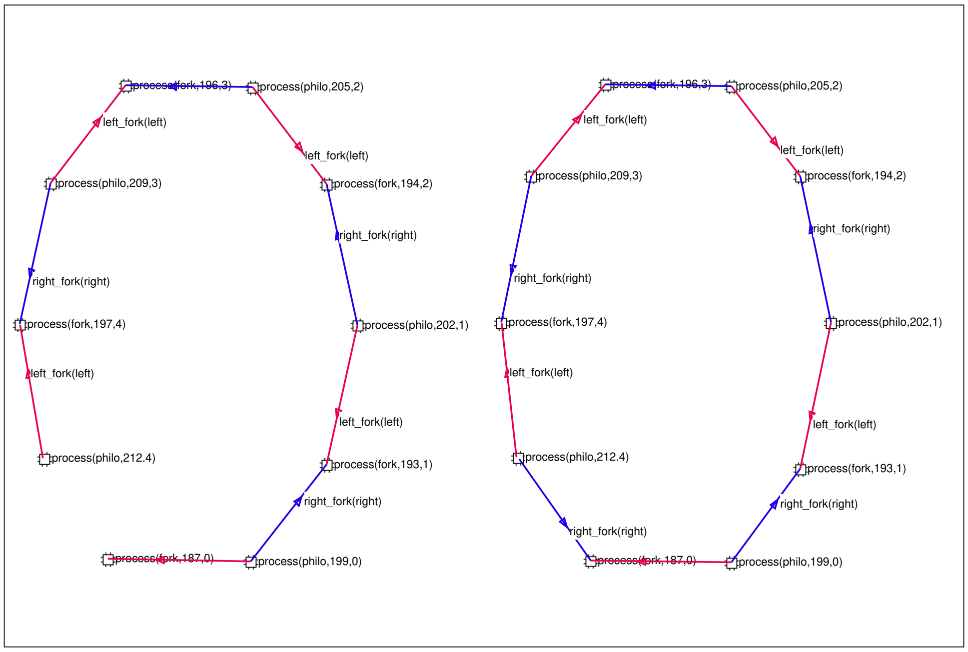


Figure 6.17: Buggy vs. correct initialization patterns.

The graph in the left (with a hole) reveals a bug as a hole in what should have been a circular pattern (as shown in the rightmost graph corresponding to a correct version of the program).

6.4 Network Management Information

The complexity of managing and controlling large heterogeneous networks requires the availability of advanced management stations capable of presenting to human managers a complete picture of the relevant scenarios. The overwhelming volume and complexity of the information involved in network management scenarios poses a major challenge. Examples of data visualizations that are relevant for network management are the network topol-

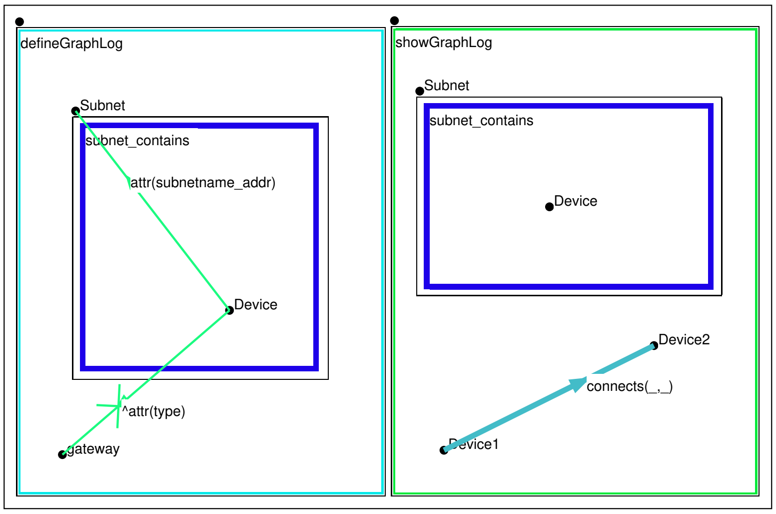


Figure 6.18: Clustering devices (excluding gateways) in subnets.

ogy at different levels of abstraction, the presentation of network configuration information, and the display of management information bases and their history traces. Employing a fault management scenario as motivation, [CH93] describes how Hy$^+$ can produce all of the above visualizations and several of their combinations, as well as entirely new ones generated through ad-hoc visual queries. The following example hygraphs, showing network management information such as connectivity and traffic patterns, are taken from [CH93].

Figure 6.18 contains the patterns that are necessary to cluster devices in logical subnets (leaving gateways outside of all the subnets they connect). The resulting diagram (in Figure 6.19) displays the subnets superimposed on top of the physical network topology.

The final hygraph in Figure 6.20 shows traffic information (obtained by monitoring the network) between two specific subnets superimposed over the previously shown topology map. This example illustrates how a network manager can selectively choose both: *(i)* the specific portion of the network she is interested in looking at, and *(ii)* which particular in-

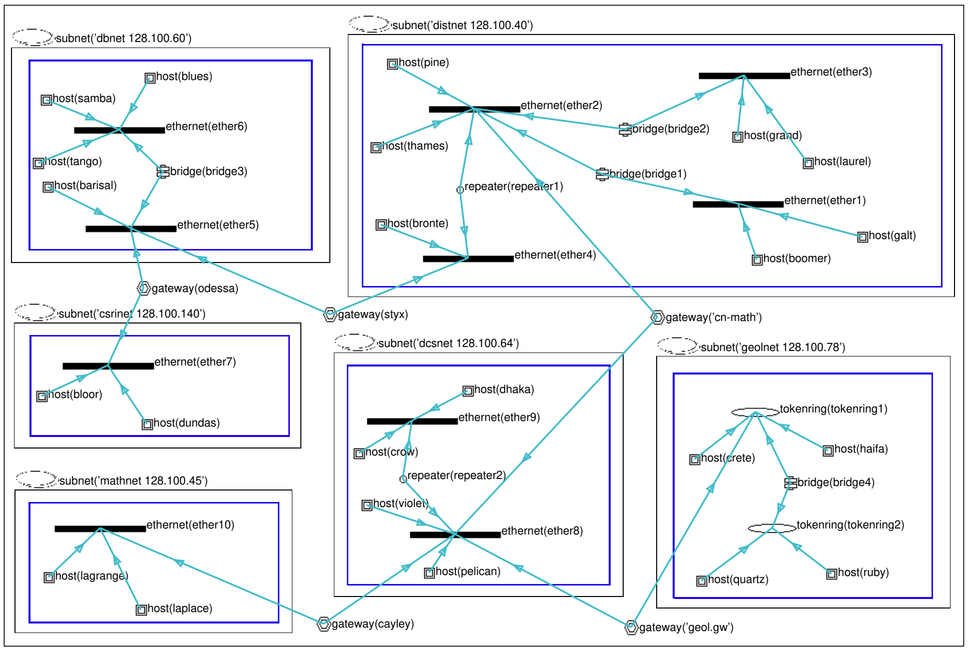


Figure 6.19: Displaying subnets over the physical network topology.

formation to superimpose on it.

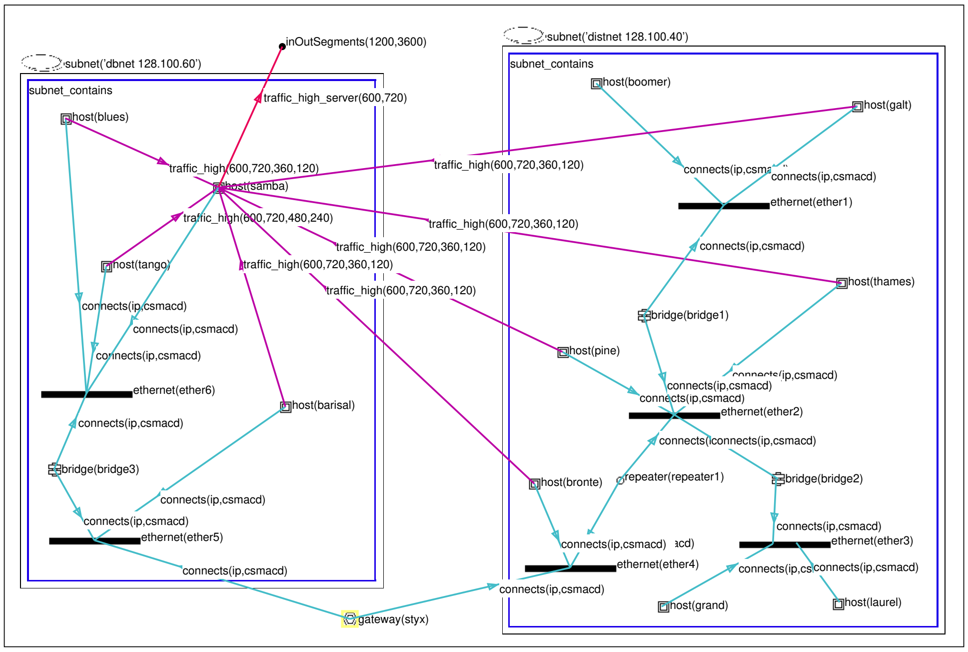


Figure 6.20: Traffic information superimposed on the topology map.

Chapter 7

Conclusions

This thesis introduces an original framework for the use of queries to create and filter *structural data visualizations* (a term we introduce to refer to the diagrammatic display of the relationships of abstract, structured data). The framework presented here exploits the synergism between the established field of database query languages and the emerging area of visualization. The contributions of this work are summarized below.

1. The definition of a new visual formalism: the *hygraph*. Hygraphs provide a precise characterization of the diagrammatic visualizations considered in the thesis. We believe that this simple formalism is a convenient abstraction for both graph-based and form-based (including tabular) presentations and as such represents a contribution on its own.

2. On the database theory side, we bring to the attention of the research community an important class of queries that, until now, did not receive special treatment: *filter queries*. We introduce the associated notion of *filtering language* and we also define and characterize the expressive power of *filtering logic programs*. We then extend the

visual query language *Graphlog* [Con89, CM90b, CM93b] to make use of *hygraph patterns* for both defining derived data and supporting filtering.

3. The simultaneous treatment (within the framework described above) of creating structural data visualizations and using database queries to:

   - *Define* new relationships in the data that is to be visualized.
   - *Filter* the existing data to come up with a display that contains only the information that is relevant to the user.

4. On the practical side, the design and direction of the collaborative implementation effort of the Hy^+ hygraph visualization system. Hy^+ embodies a significant amount of the functionality that can be developed within the formal framework described in this thesis, and as such, it constitutes proof of the applicability of the ideas presented in this work. More specific contributions of the Hy^+ system are:

   - A modular architecture to visualize and query data as hygraphs that separates the following concerns: data acquisition, hygraph editing, very large hygraph browsing, query evaluation, and hygraph layout.
   - A demonstration of the suitability of deductive database back-ends as query engines for the support of selective visualizations.
   - A novel approach to use pattern based queries to support a very high-level and declarative specification of hygraph layout.
   - Supporting pattern based selection in diagram editors (which can be regarded as analogous to the use of regular expressions in text editors).

- Providing an environment for interactively designing and discovering new kinds of structural visualizations (described by the patterns that create them) that can be applied to arbitrary datasets in a given application domain.

5. The application of Hy^+ to several different scenarios (using real data from the areas of software engineering and network management), which provides a demonstration of the possibilities opened up by creating and filtering structural data visualizations using hygraph patterns.

There are several areas arising from the work in this thesis that merit future research. There is still plenty of room for improvement in the functionality offered by a hygraph-based query and visualization system like Hy^+. The suitability of several aspects of the current Hy^+ architecture must also be tested in a wider variety of scenarios. In addition, applications such as network management and dynamic debugging motivate the support for active database features and incremental query evaluation. In general, the challenge is to exploit the framework described here in other application areas, and to consider what are suitable extensions once the need for them is established.

An important area for further research involves finding efficient techniques for the evaluation of filter queries taking advantage of their special properties. The following comments attempt to bring to the attention of the reader the fact that there is no work done on evaluation techniques specifically tailored for filters (although, potentially, work in related areas could be of relevance).

As a concrete example, current relational database management systems only contemplate returning one relation as an answer. As a result, filter queries must be evaluated as sets of queries. While there has been research done to discover *common subexpressions* [Hal76] that can be evaluated once for the entire set [Fin82, PS88, Sel88], optimizing a set of queries does not take into account the fact that filters are a very specific kind of queries. An illustrative situation is discussed below.

Consider a filtering program that joins n relations, that is, the program $r_1 \bowtie \ldots \bowtie r_n$, and which has each $r_i, 1 \leq i \leq n$, as filtering predicates (i.e., the filtering program returns r'_i with the tuples from r_i that join with all the other relations). What is an appropriate set of (traditional) queries to issue against a relational system to obtain the set of filtering predicates?

A naive approach would be to evaluate each $r'_i, 1 \leq i \leq n$ as $r'_i = \pi_{R'_i}(r_1 \bowtie \ldots \bowtie r_n)$. Re-evaluating the join expression for each r'_i is clearly repeating a lot of computation. Evaluating the join only once may yield an intermediate result that is orders of magnitude larger than the size of all the resulting r'_i combined (e.g., if the join expression degenerates to a cartesian product).

Another strategy consists of making use of techniques developed for the evaluation of queries in distributed systems. A textbook treatment of the following issues can be found in [Mai83]. The strategy is based on semijoins [BC81]: the semijoin of r with s is defined as the relation $\pi_R(r \bowtie s)$. Distributed systems try to minimize the transmission costs of evaluating the join $r_1 \bowtie \ldots \bowtie r_n$ (when the r_i's are at different sites) by computing a sequence of semijoins (known as a semijoin program) to obtain the r'_i's. When such a program exists (and existence can be determined based on several equivalent conditions, like acyclicity of the schema, existence of a Graham reduction, etc.) the system can reduce transmission costs by shipping around the r'_i's instead of the full r_i's. Clearly this strategy can be used within one system with the only objective of obtaining the filtering predicates r'_i. While the use of semijoins can improve the evaluation of the $r_1 \bowtie \ldots \bowtie r_n$ filtering program when compared with the naive approach, the execution of a semijoin program, when all of the r_i are available at one site, still involves a significant amount of redundant computation.

A final area where further research is required involves taking into account additional visible aspects of the data visualizations. The work presented here emphasizes queries and filters applied to hygraphs, which as a topovisual formalism do not capture enough graphical information as it would be desirable.

As an example, the patterns do not take into account the spatial positioning of the objects visualized in a hygraph, and as such they do not query directly the spatial information contained in the actual rendering of hygraphs. This particular situation can be alleviated by including in the data to be visualized not just the data itself, but also the graphic data (referring to the description given in Figure 1.1). This does indeed provide access to the graphic data, and as such the class of transformations expressed by the patterns act on the graphic data as well.

The interesting research issues in this direction involve not just looking at applying the kind of relational mappings that we have considered to the graphic data, but instead defining new kinds of transformations (with associated natural notations) that are more appropriate to the graphical nature of the data manipulated.

Bibliography

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., 1988.

[AEM86] T. Lougenia Anderson, Earl F. Ecklund,Jr., and David Maier. PROTEUS: Objectifying the DBMS User Interface. In *Intl. Workshop on Object-Oriented Database Systems*, 1986.

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 159–173, 1989.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[AV87] Serge Abiteboul and Victor Vianu. A transaction language complete for database update and specification. In *Proceedings of the Sixth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 260–268. Assoc. for Comp. Machinery, 1987.

[AV88] Serge Abiteboul and Victor Vianu. Procedural and declarative database update languages. In *Proceedings of the Seventh ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 240–250. Assoc. for Comp. Machinery, 1988.

[AvE82] K. Apt and M. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[Ban78] F. Bancilhon. On the completeness of query languages for relational databases. *Proc. 7th Symp. on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 64*, pages 112–123, 1978.

[BC81] P. A. Bernstein and D. M. Chiu. Using Semi-joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, January 1981.

[BCCL91] C. Batini, T. Catarci, M. F. Costabile, and S. Levialdi. Visual query systems. Report 04.91, Universita degli Studi di Roma La Sapienza, March 1991.

[Ber73] C. Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[BK93] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming (ICLP)*, Budapest, Hungary, June 1993.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The gemstone object database management system. *Communications of the ACM*, 34(10):64–77, 1991.

[Bro87] F. Brooks. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[BS81] F. Bancilhon and N. Spyratos. Update Semantics in Relational Views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[BSMW90] R. Becker, Eick S., E. Miller, and A. Wilks. Dynamic graphics for network visualization. In *Proceeedings of the IEEE Conference on Visualization*, pages 93–96, 1990.

[Cas91] S. M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, 1991.

[CCM92] Mariano Consens, Isabel Cruz, and Alberto Mendelzon. Visualizing queries and querying visualizations. In *ACM SIGMOD Record*, pages 39–46, 1992.

[CH80] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[CH82] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[CH85] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, 1985.

[CH93] Mariano Consens and Masum Hasan. Supporting network management through declaratively specified data visualizations. In *Proceedings of the Third IFIP/IEEE International Symposium on Integrated Network Management*, pages 725–738. IFIP Transactions C-12, Elsevier North-Holland, 1993.

[Cha88] Ashok K. Chandra. Theory of database queries. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–9, 1988.

[CHM93] Mariano Consens, Masum Hasan, and Alberto Mendelzon. Debugging distributed programs by visualizing and querying event traces. Technical Report CSRI-285, University of Toronto, 1993. (In Declarative Database Visualization: recent papers from the Hy+/GraphLog project, pages 51-66).

[CHM94] Mariano Consens, Masum Hasan, and Alberto Mendelzon. Visualizing and querying distributed event traces with Hy^+. To be published in *Proceedings of the ADB'94 Conference*, 1994.

[CKM91] Mariano Consens, Christine Knight, and Alberto Mendelzon. The architecture of the G^+/GraphLog visual query system. Technical Report TR 74.054, IBM Canada Laboratory, 1991. (Also available as an internal report from Computer Systems Research Institute, University of Toronto, 1990).

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[CM89] Mariano Consens and Alberto Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proceedings of the Second ACM Hypertext Conference*, pages 269–292, 1989.

[CM90a] Mariano Consens and Alberto Mendelzon. The G^+/GraphLog visual query system. In *Proceedings of the ACM-SIGMOD 1990 Annual Conference on Management of Data*, page 388, 1990. Video presentation summary.

[CM90b] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CM90c] Mariano Consens and Alberto Mendelzon. Low complexity aggregation in GraphLog and Datalog. In *Proceedings of the Third International Conference on Database Theory, Lecture Notes in Computer Science Nr. 470*, pages 379–394. Springer-Verlag, 1990. A revised version has been published in *Theoretical Computer Science*, 116(1), 1993, pages 95–116.

[CM93a] Mariano Consens and Alberto Mendelzon. Hy$^+$: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD 1993 Annual Conference on Management of Data*, pages 511–516, 1993. Video presentation summary.

[CM93b] Mariano Consens and Alberto Mendelzon. Low complexity aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):379–394, 1993. An earlier version has been published in Proceedings of ICDT'90, Springer-Verlag LNCS 470, 379-394.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *14th. Intl. Conference on Software Engineering*, pages 138–156, 1992.

[CMV94] Mariano Consens, Alberto Mendelzon, and Dimitra Vista. Deductive database support for data visualization. In *Proceedings of the EDBT'94 Conference*, 1994. (An earlier version appears in Declarative Database Visualization: recent papers from the Hy$^+$/GraphLog project, Technical Report CSRI-285, University of Toronto, pages 51–66, 1993).

[CMW87] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. A graphical query language supporting recursion. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 323–330, 1987.

[CMW88] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. G^+: Recursive queries without recursion. In *Proceedings of the Second International Conference on Expert Database Systems*, pages 355–368, 1988.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, Englewood Cliffs, N. J, 1972.

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

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–22, 1985.

[CRM91] Stuart Card, George Robertson, and Jock Mackinlay. The information visualizer, an information workspace. In *Proceedings of the Conference on Computer Human Interaction*, pages 181–188, 1991.

[Cru93] I. Cruz. *Querying object-oriented databases with user-defined visualizations*. PhD thesis, University of Toronto, Department of Computer Science, 1993.

[DB82] U. Dayal and P. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[Dea91] O. Deux and et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, 1991.

[DETT93] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. Available via anonymous ftp from wilma.cs.brown.edu, file /pub/gdbiblio.ps.Z, June 1993.

[DGGR90] G. Di Battista, A. Giammarco, Santucci G., and Tamassia R. The architecture of diagram server. In *Proceedings of Visual Languages*, pages 60–65, 1990.

[Eig93] Frank Ch. Eigler. GXF: A Graph Exchange Format. Technical Report CSRI-285, University of Toronto, 1993. (In Declarative Database Visualization: recent papers from the Hy+/GraphLog project, pages 91-107).

[Fag74] R Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, volume 7, pages 43–73. SIAM–AMS, 1974.

[Fin82] S. Finkelstein. Common Expression Analysis in Database Applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 235–245, 1982.

[Fit93] G. W. Fitzmaurice. Situated information spaces and spatially aware palmtop computers. *Communications of the ACM*, 36(7):38–49, July 1993.

[FM92] B. B. Flynn and D. Maier. Supporting display generation for complex database objects. *SIGMOD Record*, 21(1):18–24, March 1992.

[FPF88] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas. SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases. In *Cognitive Science and its Applications for Human-Computer Interaction*, pages 201–233, 1988.

[Fuk91] Milan Fukar. Translating GraphLog into Prolog. Technical report, Center for Advanced Studies IBM Canada Limited, October 1991.

[FZC93] G. W. Fitzmaurice, S. Zhai, and M. H. Chignel. Virtual reality for palmtop computers. *ACM Transactions on Office Information Systems*, 11(3):197–218, July 1993.

[GG93] J-L Guerin and P. Y. Gloeass. GrafOLog: a Visual Language for a Logic with Objects. *Journal of Visual Languages and Computing*, 4:301–324, 1993.

[Gon87] Gaston Gonnet. PAT 3.1: An efficient text searching system. Technical report, UW Centre for the New OED, University of Waterloo, 1987.

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

[GPG90] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A graph-oriented object database model. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 417–424, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Hal76] P. A. Hall. Optimization of a Single Relational Expression in a Relational Database System. *IBM Journal of Research and Development*, 20(3):244–257, 1976.

[Hal88] Frank G. Halasz. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, 1988.

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

[HH91] T. R. Henry and S. E. Hudson. Interactive graph layout. In *Proceedings of UIST'91*, pages 55–64, 1991.

[HK92] D. Harel and C. Kahana. On statecharts with overlapping. To appear in ACM TOSEM, 1992.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[Ide93] *Unix IDE User's Guide, Bell Northern Research*, 1993.

[ILH92] Y. E. Ioannidis, M. Livny, and E. M. Haber. Graphical user interfaces for the management of scientific experiments and data. *SIGMOD Record*, 21(1):47–53, March 1992.

[Imm88a] Neil Immerman. Descriptive and computational complexity. Technical report, Department of Computer Science, Yale University, New Haven, 1988.

[Imm88b] Neil Immerman. Expressibility and parallel complexity. Technical report, Department of Computer Science, Yale University, 1988.

[Imm88c] Neil Immerman. Nondeterministic space is closed under complementation. In *Third Structure in Complexity Theory Conference*, 1988.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JMN+92] S. Javey, K. Mitsui, H. Nakamura, T. Ohira, K. Yasuda, K. Kuse, T. Kamimura, and R. Helm. Architecture of the XL C++ Browser. In *Proceedings of the IBM CAS Conference (CASCON)*, pages 369–379, 1992.

[JNZM93] J. A. Johnson, B. A. Nardi, C. L. Zarmer, and J. R. Miller. ACE: Building Interactive Graphical Applications. *Communications of the ACM*, 36(4):41–55, 1993.

[JP87] D. S. Johnson and H. O. Pollak. Hypergraph planarity and the complexity of drawing Venn diagrams. *Journal of Graph Theory*, 11(3):309–325, 1987.

[JY92] S. Javey and K. Yasuda. The Conceptual Model for the C++ Program Database. Technical Report TR.74.093, IBM, May 1992.

[Kam89] Tomihisa Kamada. *Visualizing Abstract Objects and Relations*. World Scientific, 1989.

[KN92] R. King and M. Novak. Building Reusable Data Representations with FaceKit. *SIGMOD Record*, 21(1):11–17, March 1992.

[KS90] A. Karrer and W. Scacchi. Requirements for an extensible object-oriented tree/graph editor. In *Proceedings of UIST'90*, pages 84–91, 1990.

[Llo84] J. W. Lloyd. *Foudations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[Mac88] Jock Mackinlay. Applying a theory of graphical presentation to graphic design of user interfaces. In *Proceedings of the Conference on Computer Human Interaction*, 1988.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[Mar91] J. Marks. A formal specification scheme for network diagrams that facilitates automated design. *Journal of Visual Languages and Computing*, 2:395–414, 1991.

[MDB87] B. H. McCormick, T. A. DeFanti, and M. D. Brown (editors). Visualization in scientific computing. *SIGGRAPH Computer Graphics*, 21(6):30–42, November 1987. (entire issue devoted to the topic).

[MHP93] 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.

[MW88] D. Maier and D.S. Warren. *Commputing with Logic: Logic Programming with Prolog*. Benjamin-Cummings, Menlo Park, CA, 1988.

[Mye90] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[Noi93a] E. G. Noik. Exploring large hyperdocuments: fisheye views of nested networks. In *ACM Conference on Hypertext*, 1993.

[Noi93b] E.G. Noik. Graphite: A suite of hygraph visualization utilities. Technical Report CSRI-285, University of Toronto, 1993. (In Declarative Database Visualization: recent papers from the Hy+/GraphLog project, pages 108-126).

[NT89] Shamim Naqvi and Shalom Tsur. *A logical language for data and knowledge bases*. Computer Science Press, New York, 1989.

[NT90] F. Newbery and W. Tichy. EDGE: An extendible graph editor. *Software–Practice and Experience*, 20:63–88, June 1990.

[NZ93] B. A. Nardi and C. L. Zarmer. Beyond Models and Metaphors: Visual Formalisms in User Interface Design. *Journal of Visual Languages and Computing*, 4:5–33, 1993.

[Par78] J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.

[PBS93] B. A. Price, R. M. Baecker, and I. S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.

[Pen93] D. A. Penny. *The Software Landscape: A Visual Formalism for Programming-in-the-large*. PhD thesis, University of Toronto, Department of Computer Science, 1993.

[PS88] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the IEEE International Conference on Data Engineering*, 1988.

[Ray91] D. Raymond. Characterizing visual languages. In *Proceedings of the IEEE Workshop on Visual Languages*, pages 176–182, 1991.

[Ray92] Darrell Raymond. Flexible text display with lector. *IEEE Computer*, 28(8):49–60, 1992.

[RCM93] G. G. Robertson, S. K. Card, and J. D. Mackinlay. Information Visualization using 3D Interactive Animation. *Communications of the ACM*, 36(4):57–71, 1993.

[RM90] Steven Roth and Joe Matis. Data characterization for intelligent graphics presentation. In *Proceedings of the Conference on Computer Human Interaction*, 1990.

[Row92] L. Rowe. A retrospective on database application development frameworks. *SIGMOD Record*, 21(1):5–10, March 1992.

[RRS92] D. Srivastava R. Ramakrishnan and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[Rym91] A. Ryman. Code Overlay Design and Analysis Using Prolog. Technical Report TR.74.052, IBM, April 1991.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[She88] J.C. Shepherdson. Negation in logic programming. In *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann Publishers, Inc., 1988.

[SI91] I. Senay and E. Ignatius. Compositional analysis and synthesis of scientific data visualization techniques. In *Porceedings of Computer Graphics International*, pages 262–282, 1991.

[SLN93] P. Szekely, P. Luo, and R. Neches. Beyond interface builders: model-based interface tools. In *INTERCHI'93*, pages 383–390, 1993.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Sze90] Pedro Szekely. Template-based mapping of application data to interactive displays. In *Proceedings of the Conference on User Interface Software and Technology*, 1990.

[Tom89] Frank Tompa. A data model for flexible hypertext database systems. *ACM Transactions on Office Information Systems*, 7(1):85–100, 1989.

[Tuf90] Edward R. Tufte. *Envisioning Information*. Graphic Press, Cheshire, Connecticut, PO Box 430, 06410, 1990.

[UFK+89] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, Gurwitz R., and A. Van Dam. The application visualization system: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Potomac, Md., 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, Potomac, Md., 1989.

[Urr89] J. Urrutia. Partial orders and euclidean geometry. In I. Rival, editor, *Algorithms and Order*, pages 387–434. Kluwer Academic Publishers, 1989.

[Var82] M.Y. Vardi. The complexity of relational query languages. *Proc. 14th Ann. ACM Symp. on Theory of Computing*, pages 137–146, 1982.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Ven94] J. Venn. *Symbolic Logic*. Chelsea, New York, second edition, 1894.

[vG88] A. van Gelder. Negation as failure using tight derivations for general logic programs. In *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann Publishers, Inc., 1988.

[Yao91] P. Yao. Tuning the Performance of Windows and OS/2 Programs with Micro-Quill's Segmentor. *Microsoft Systems Journal*, 6(2):49–55, March 1991.

[Yeu93] A. Yeung. Text Searching in the Hy^+ Visualization System. Master's thesis, Department of Computer Science, University of Toronto, October 1993.

[ZBM76] M. M. Zloof, J.A. Bondy, and U.S.R. Murty. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the International Conference on Very Large Databases*, pages 1–24, 1976.