Reverse Engineering by Visualizing and Querying

A. Mendelzon
Computer Systems Research Institute, University of Toronto, Toronto M5S 1A1, Canada

J. Sametinger
CD Laboratory for Software Engineering, Institut für Wirtschaftsinformatik, University of Linz, Altenbergerstr. 69, A-4040 Linz, Austria*)

CR Categories and Subject Descriptors: D1.5 [**Software**]: Programming Techniques—*Object-oriented Programming*; D2.8 [**Software Engineering**]: Metrics—*Complexity Measures*; D2.10 [**Software Engineering**]: Design—*Representation.*

General Terms: Software Maintenance, Reverse Engineering, Visualization.

Additional Key Words and Phrases: Metrics, Constraints, Design Patterns, Object-oriented Programming.

Please send correspondence and proofs for correction to

Johannes Sametinger
CD Laboratory for Software Engineering, Institut für Wirtschaftsinformatik, University of Linz, Altenbergerstr. 69, A-4040 Linz, Austria

E-Mail: sametinger@swe.uni-linz.ac.at
Tel.: ++43/732/2468/9435
Fax: ++43/732/2468/9430

---

*) This work has been done during a research visit at the Computer Systems Research Institute of the University of Toronto.

REVERSE ENGINEERING BY VISUALIZING AND QUERYING

Summary

The automatic extraction of high-level structural information from code is important both for software maintenance and reuse. Instead of using special purpose tools, we explore the use of a general purpose data visualization system called Hy+ for querying and visualizing information about object-oriented software systems. Hy+ supports visualization and visual query of arbitrary graph-like databases. We store information about software systems in a database and use the general purpose tool Hy+ for analyzing the source code and visualizing various relationships. In this paper we demonstrate the use of Hy+ for evaluating software metrics, verifying constraints, and identifying design patterns. Software metrics can be used to find components with low reusability or components that are hard to understand. Checking the source code against constraints can help bring design flaws to light, eliminate sources of errors, and guarantee consistent style. Identifying design patterns in a software system can reveal design decisions and help in understanding the code. We conclude that the flexibility achieved by using a general purpose system like Hy+ give this approach advantages over special purpose reverse-engineering tools, although specialized tools will have better performance and more knowledge of specific software engineering tasks. Combining the advantages of the two approaches is an interesting challenge.

Program comprehension plays a major role in software maintenance. The increased reuse of software components facilitated and supported by object-oriented programming makes program comprehension even more important, as existing software must be understood both during development and maintenance.

Very often the only information a programmer can trust is the source code. It is the only accurate, complete and up-to-date representation of a program. However, source code listings are hardly suited to represent design decisions, global system structure, or interactions among components. The extraction of high-level structural information from code, called *reverse engineering* (Chikofsky et al., 1990), is important both for software maintenance and reuse.

Analyzing the source code can help to find components with certain properties, to find possible bottlenecks and weak points of a system, to identify components and their relationships, and to better duplicate the chain of reasoning of the original authors. *Software metrics* can be used to find components with low reusability or components that are hard to understand. Checking the source code against various *constraints* can help to bring design flaws to light, to eliminate sources of errors, and to guarantee consistent style. Identifying *design patterns* in a software system can reveal important facts about the design.

A large amount of information together with complex relationships among various objects characterizes most of today's software systems. Tool support is indispensable for successful analysis. Instead of developing several tools for the various activities mentioned above, we have used a general purpose tool for visualizing and querying software structure. Besides saving the effort of developing separate tools, this has the advantage that a general purpose tool can easily be extended for future applications. We chose the Hy+ and Graphlog system (Mendelzon, 1993) because of its power and high flexibility. In this paper we demonstrate the power of visualizing and querying in the reverse engineering domain by applying the Hy+ visualization system and Graphlog visual query language to the specification of metrics, constraints and design patterns. In Section 2 we introduce Hy+ and Graphlog. In Section 3 we discuss software metrics, in Section 4 constraints, and in Section 5 design patterns. We conclude in Section 6.

1. HY+ AND GRAPHLOG

The Hy+ system is a generic tool for visualizing objects and relationships among them. Hy+ supports a novel visual query language called *GraphLog*.

Hy+ provides a user interface with extensive support for visualizing structural (or relational) data as *hygraphs* (Consens et al., 1994), an extension of graphs inspired by Harel's higraphs (Harel, 1988). For simplicity, in this paper we only use standard directed graphs, a subset of the hygraph formalism.

The Hy+ system supports visualization of the actual *database instances*, not just diagrammatic representations of the database schema. The use of a query language is essential in making this approach scale up to large database instances. This is accomplished in two ways.

The first one is the ability to *define* new relationships by using queries. This is the traditional way of using database queries: the newly defined relationship either gives a direct answer to a user question, or it provides a new view on the existing data. The derived data can later be presented visually by the system. In this way, users can abstract irrelevant details or aggregate information into visualizations of manageable size.

The second capability is a way of using queries to decide what data to *show*. The user can selectively restrict or filter (Consens et al., 1994) the information to be displayed. Selective data visualization can be used to locate relevant data, to restrict visualization to interesting portions of the data (that is, deciding *what* data to present) and to control the level of detail at which the data is presented (that is, to choose *how* to see the data).

To specify queries, Hy+ supports a visual pattern-based notation. The patterns are expressions of the GraphLog query language. We give a short introduction to GraphLog here; a full definition can be found in (Consens, 1989 and Consens et al., 1990). In the remainder of the paper, we will introduce the basic functionality of GraphLog through examples. These examples demonstrate the application of GraphLog to compute software metrics, verify constraints, and find design patterns in a database describing the structure of an object-oriented software system.

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

For example, suppose we have a database containing all classes and methods of some software system we want to explore. This database can be viewed by Hy+ as a graph in which the nodes represent software objects such as classes and methods and the edges represent relationships. A node is labeled by a Prolog-like term, in which functors are used for typing purposes. For example, a method node might be labeled by a two-place functor method(Name, Class) giving the name of the method and the name of the class in which it appears. Methods with the same name appearing in different classes will be represented by different nodes. Edges are labeled by predicates of the form p(F1, ..., FN), where the Fi's are terms. The meaning of a predicate p(F1, ..., FN) between a node labeled F and a node labeled G is the literal p(F, G, F1, ..., FN). The frequent case where relationships are binary and have no extra attributes is represented by labeling the edge with just the predicate name p. For example, each link in the inheritance hierarchy is represented by an edge labeled superclass between two classes. Additionally, a method edge exists to connect classes and their methods. Chapter 6 describes in more detail the contents of the database and how it is automatically constructed from the system's source code. Hy+ provides different ways of visualizing such a graph and manipulating the visualizations; more information on this functionality can be found in (Consens et al., 1994). Figure 1 shows a small portion of such a database.

GraphLog queries are sets of graphs whose nodes are labeled by terms, and each edge is labeled by an edge label. Node labels can be more general than those in database graphs, because they can include variables; edge labels may not only include variables, but also *regular expressions*. An edge in a query graph may match a path in the database graph, where the labels along the path are constrained by the regular expression.

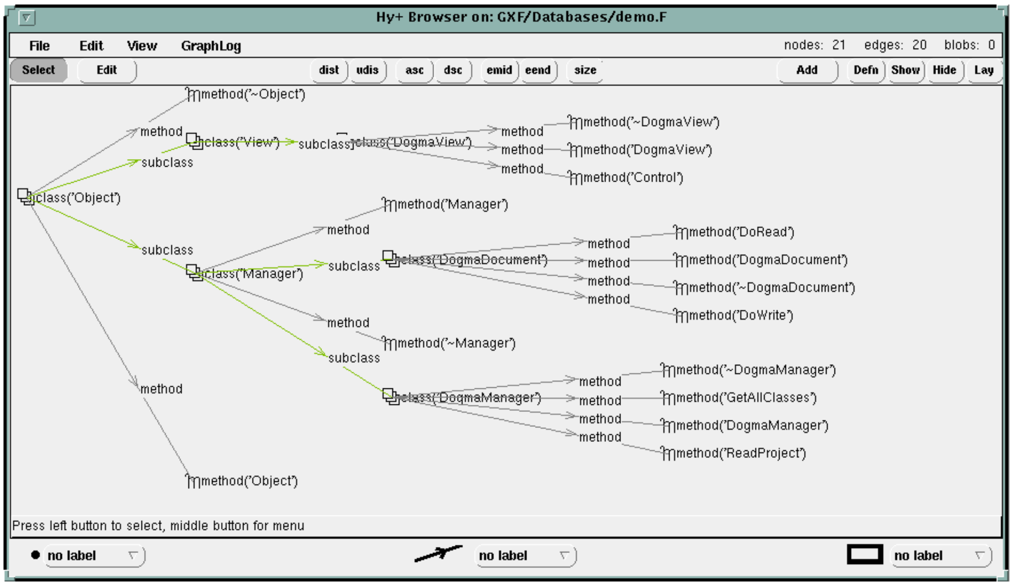

Fig. 1: Visualization of a sample database

There are two types of queries: *define* and *filter*. In both, the query hygraph represents a pattern; the query evaluator searches the hygraph designated as the database for all occurrences of that pattern. The difference between the two types of queries stems from their interpretation of *distinguished elements*, explained below.

A hygraph pattern in a define query (which is enclosed in a `defineGraphLog` box) must have only one *distinguished* edge labeled by a positive literal. The meaning of the [define] query hygraph is to define the predicate in this distinguished literal in terms of the rest of the pattern. The semantics of *define* queries is given by a translation to stratified Datalog (Consens, 1989). Each *define* graph translates to a rule with the label of the distinguished edge in the head, and as many literals in the body as there are non-distinguished edges in the graph. Additional rules may be necessary to define the predicates of non-distinguished edges that are labeled by regular expressions. The generation of these additional rules is based on the structure of the regular expression. Figure 2 shows how the relation subclass can be defined using the relation superclass.

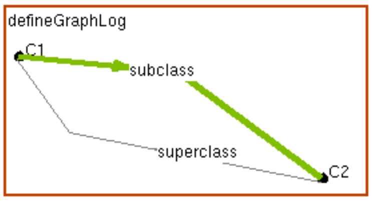

Fig. 2: Define query

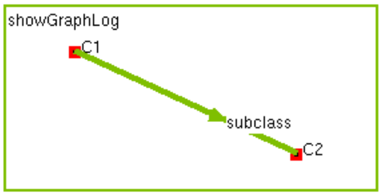

Fig. 3: Filter query

A graph pattern in a filter query (which is enclosed in a `showGraphLog` box) may have several *distinguished* nodes and edges. The meaning of a *filter* query graph is: for each instance of the pattern found in the database, retain the database objects that match the distinguished objects in the query. Given a graph in a `showGraphLog` box, for each distinguished edge, we generate a set of *define* queries that *match* the distinguished object, that is, when they are evaluated they determine all instances of the edge that exist in the portions of the database that match the hygraph pattern. The query evaluator evaluates each of the *define* queries in turn. The results are combined, and the answer to the *filter* query is found. Figure 3 shows a very simple filter query.

GraphLog has the ability to collect multisets of tuples and to compute aggregate functions on them. The aggregate functions supported in GraphLog are the unary operators MAX, MIN, COUNT, SUM, and AVG. They are allowed to appear in the arguments of the distinguished relation of a *define* query as well as in its incident nodes. As an example of the use of aggregation in GraphLog, consider the `defineGraphLog` box of Fig. 4. It defines the relation `coupled` between two classes. The relation `coupled (C1, C2, N))` is defined between `C1` and `C2`, when class `C1` has a total of `N` variables of type `C2`.

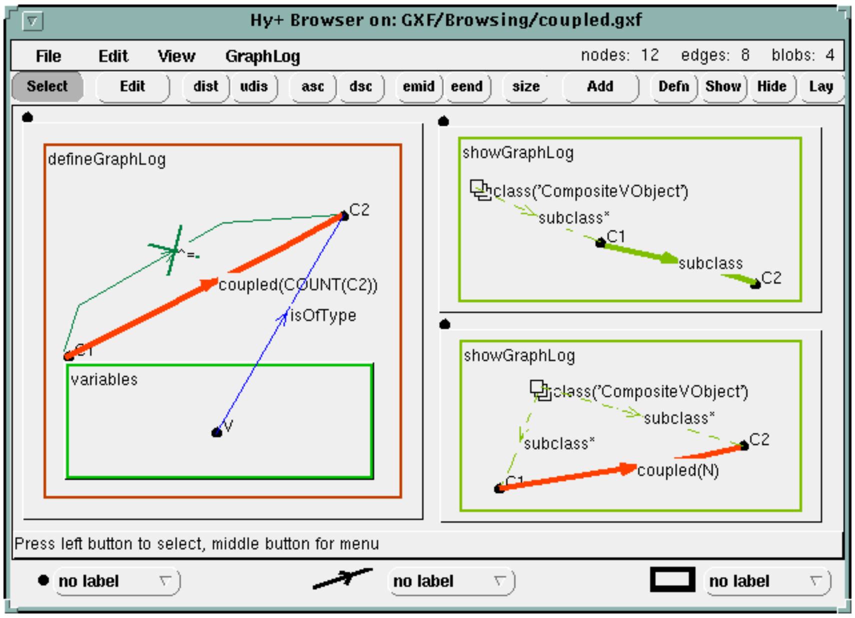

Fig. 4: Aggregate function in a query

2. METRICS

Measures about the size and complexity of software systems are helpful for estimating costs and schedules, improving software quality, and anticipating and reducing future maintenance requirements (Chidamber et al., 1991). Software metrics are desired by software managers for project planning and evaluation, and can be used to enforce quality-assurance.

In the reverse engineering process extracting measures from an existing software system can help in finding candidates for

- components with low reusability
  Classes with a large number of methods are more likely to be application specific and thus less reusable than others. If this is not the case they might be candidates for a redesign.
- components that require extensive testing
  Errors in leaf classes are more local than in classes with many children (subclasses). The greater the number of code lines is in a method, the greater is the probability of an error in that method.
- components that are hard to understand
  Encapsulation is important for hiding details and prohibiting unallowed access to the details of certain components. Inter-object coupling should be kept to a minimum. Understanding and testing of highly coupled classes tends to be more difficult.

Metrics oriented towards object-oriented systems can also aid in evaluating the degree of object-orientation. This becomes important considering the need for many conventional programmers to switch to object-oriented thinking.

Environments for the evaluation of metrics do exist. For example, ATHENA allows easy implementation of design and code metrics with a specially designed specification language (Tsalidis et al., 1992). We did not try to support the measure of well-known metrics like McCabe or Halstead. Their benefit in object-oriented systems is less important than in conventional ones, because the complexity of these systems shifts from statements and functions to components and their interactions.

With Graphlog's ability to specify arithmetic expressions in queries it is a rather straight-forward process to evaluate various metrics. We will demonstrate on a simple example how easy it is to extract measures from (both object-oriented and conventional) software systems.

In object-oriented systems dynamic binding complicates the comprehension process, because sending a message can cause the invocation of many methods. Suppose we want to know all those methods (and their corresponding classes) of which a call can result in more than, let's say, 20 different method calls; i.e., we want to know whether a certain method is overridden in more than 20 subclasses, because tracing the control flow in that case is understandably difficult. Let us see how we could find such methods using Hy+ and Graphlog.

To find methods that are overridden in more than 20 subclasses, suppose first that our database contains an edge labeled *ovCount*, that connects each method M to a node labeled with the number of methods that override M. Then all we need to do is filter out from the graph those methods connected by an *ovCount* edge to an integer greater than 20. Fig. 5 shows a filter query that does this.

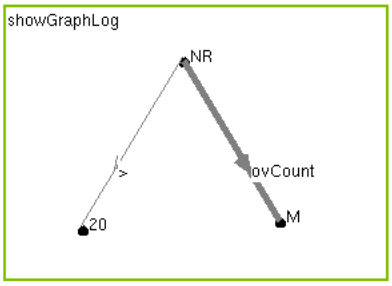

Fig. 5: Filtering methods with *ovCount*>20

Recall a filter query is interpreted as a graph pattern; for each instance of the pattern found in the database, the distinguished (bold) objects are retained and displayed as part of the answer. In this example, the pattern requires methods M that are overridden by a number NR of methods, where NR>20; the distinguished object is just the *ovCount* edge. The answer, shown in Fig. 6, will be a set of edges connecting all methods of interest to the count of methods that override them.

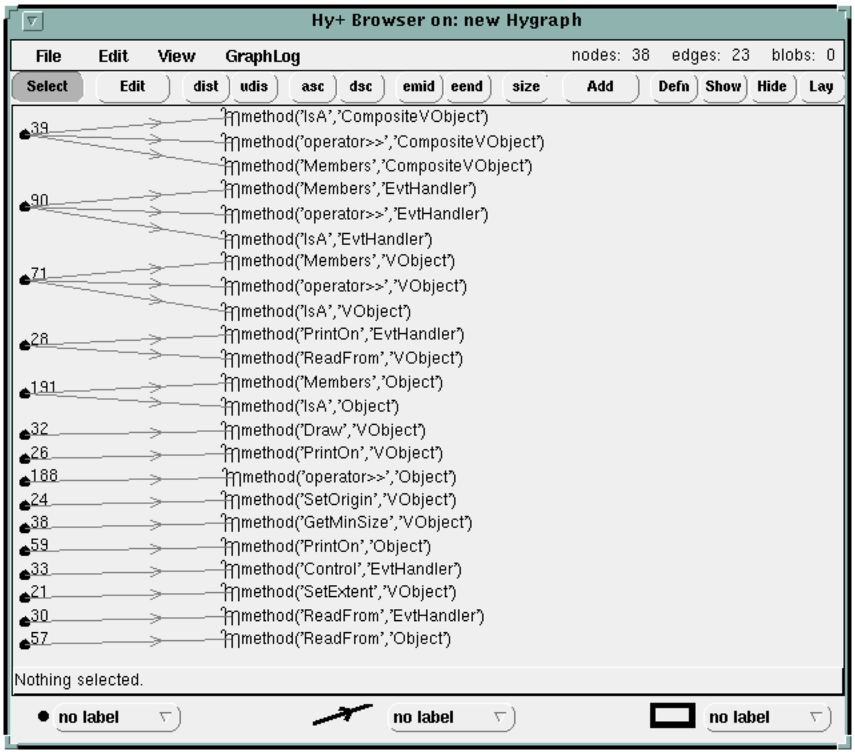

Fig. 6: Result of the *ovCount* query

Instead of just showing how many times each method is overridden, we would probably be interested in knowing in which classes the overrides occur. To see this, we simply add to the filter query another method node M2 and a distinguished *overrides* edge from M2 to M. To avoid a too complex answer for this paper we additionally specify the method name Draw of class VObject instead of M in Fig. 7, thus restricting the answer to overriding methods of this particular method. The answer, shown in Fig. 8, will then show all the classes where each of the methods of interest is defined or redefined. In this case, we can see that method Draw of class VObject is overridden 32 times (*ovCount*). On the left side all the overriding methods are listed.

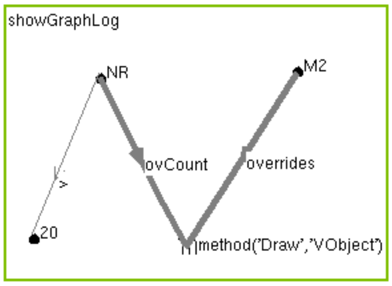

Fig. 7: Extended filter query

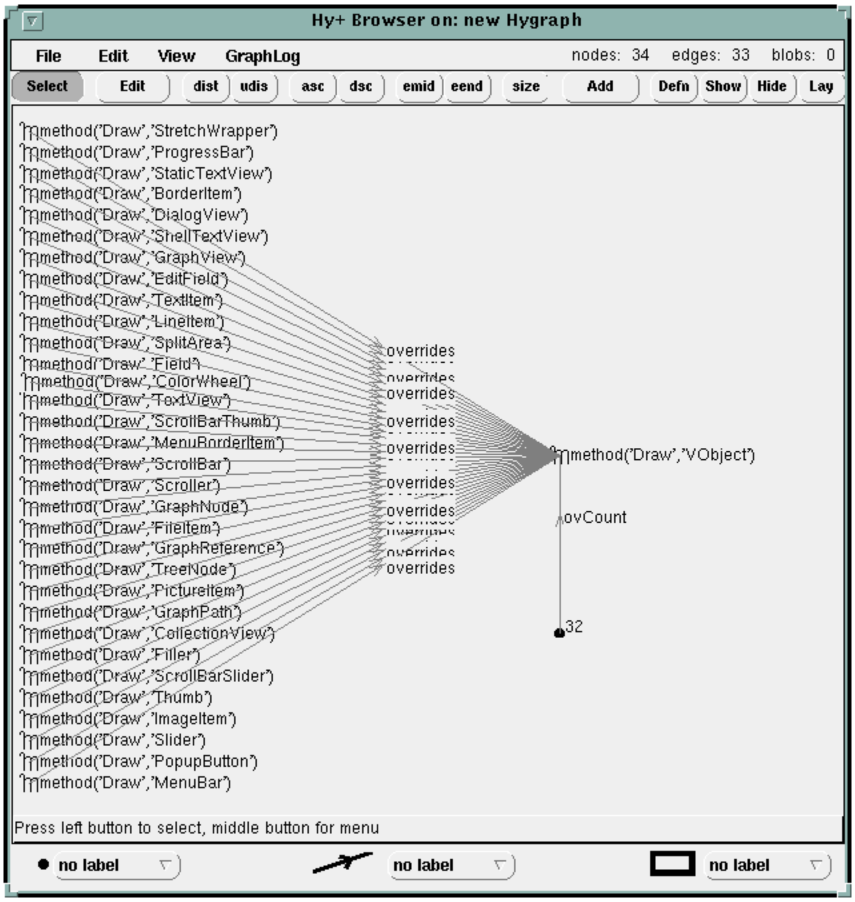

Fig. 8: Result of filter query in Fig. 7

We could add even more information by adding a separate `showGraphLog` box requesting that the subclass hierarchy be also displayed, as shown in the left box of Fig. 9. C1 is a direct or indirect subclass of class CompositeVObject, which is indicated by the use of transitive closure in the regular expression subclass+. The transitive closure is drawn dashed to indicate that it may match a path of arbitrary length in the database, not just an edge. For the resulting nodes the subclass relationship is shown in the answer (indicated by the distinguished *subclass* relation to C2). And again we restrict the query to a certain subclass hierarchy (class CompositeVObject) in order to get an answer that is not too complex to be clearly shown in this paper. The answer is in Fig. 10.

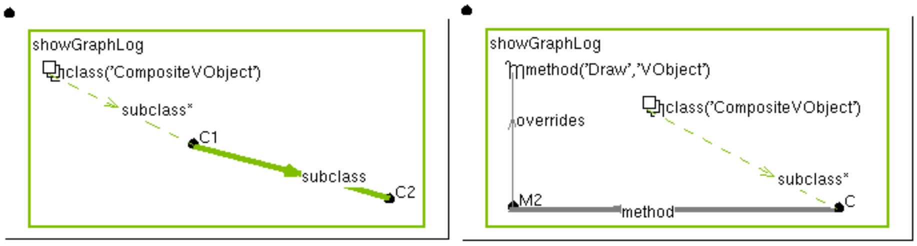

Fig. 9: Filter query for additionally displaying the subclass hierarchy

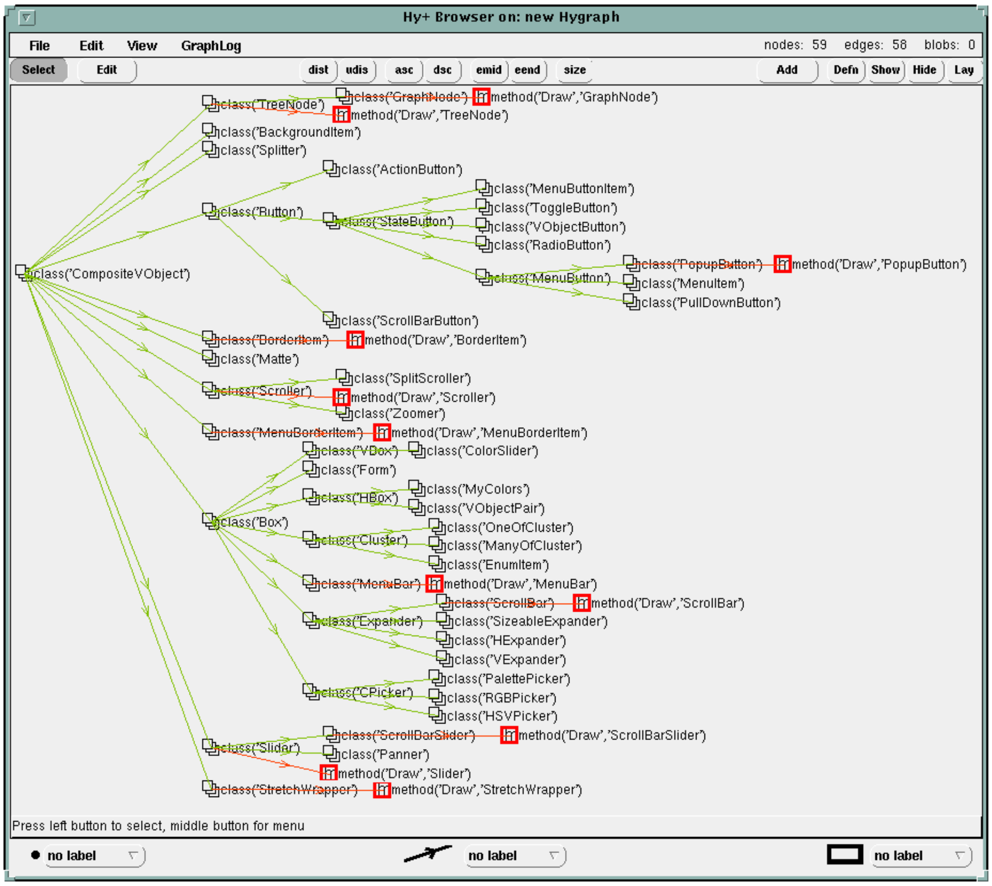

Fig. 10: Result of the query specified in Fig. 9

Unfortunately, the relations *overrides* and *ovCount* do not exist in our database. But with Graphlog we can easily define new relations based on existing ones. (This gives us the flexibility to define arbitrary metrics.) Thus, in Fig. 11 we define *ovCount* as being the number of methods M1 that override M2. Note that this is a define query. The distinguished object, i.e., the one being defined, is the *ovCount* relationship. This query also shows the use of aggregation, in this case to count, for each method M2, how many methods M1 override it.

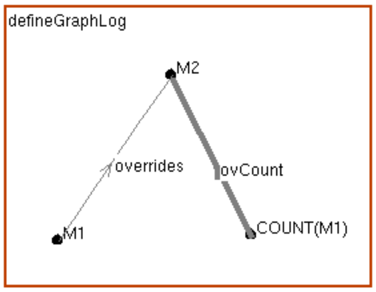

Fig. 11: Definition of the relation *ovCount*

Finally, we define the relation *overrides*. A method in class C1 overrides another method in class C2 if both have the same name (without taking parameters into account), and C1 is a superclass of C2. This fact is expressed by drawing two classes (see Fig. 12) each with a method named M. C2 is a direct or indirect superclass of C1, which is indicated by the use of transitive closure in the regular expression superclass+. And, of course, C1 and C2 have to be different. This is ensured by drawing a non-equals edge between them. The crossing-out of an edge is GraphLog's general mechanism to indicate negation.

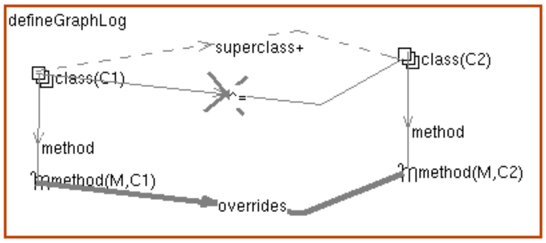

Fig. 12: Definition of the relation *overrides*

As result of the query described above we obtain the graph shown in Fig. 6 shown above. The methods are specified by name of method and name of class. Thus, the methods PrintOn of class EvtHandler and ReadFrom of class VObject are overridden by 28 methods, and so on.

Examples for metrics that can be specified in a very similar way include:

- methods per class
- depth of inheritance
- number of children (subclasses)
- coupling between objects
- response for a class
- lines of code (for classes, methods, functions, files)
- number of methods (for clients/heirs/friends)
- number of (instance) variables
- number of strongly coupled classes

These metrics are targeted at the source code only. It is often argued that monitoring metrics after coding is far too late in a software project. It is possible with Hy+ to shift the focus to other software artifacts, e.g., requirement specification. However, it has not been our goal to provide a general software metrics tool with Hy+, but rather to use its flexibility for program comprehension and reverse engineering purposes.

We have used Hy+ to extract measures from a large C++ class library in order to easily find out things listed above. The library (application framework) ET++ (Weinand et al., 1989) has been made by rather experienced programmers. Therefore, it was not our goal to detect flaws but rather to support the comprehension process by visualizing the system structure and to give prominence to classes and methods with certain characteristics. On the other hand, querying student projects helped in locating sources of trouble, which were then discussed in order to improve the design and/or eliminate existing flaws.

3. CONSTRAINTS

With most programming languages we are not able to express many important constraints about our software systems. Such constraints contain design, implementation and stylistic conventions and are necessary to improve software quality like reliability, maintainability, and readability. Automatic detection of violations can help to ensure project or company wide styles for programs, and to eliminate a variety of sources of troubles. In the software maintenance and also the reverse engineering process this is important as well (Chowdhury et al., 1993).

There are different ways to detect constraint violations. Assertions or annotations are available for various languages like Ada, Eiffel, and C++. CCEL, the C++ Constraint Expression Language, is a very powerful language, that allows users to specify and automatically detect violations of C++ constraints (Meyers et al., 1993).

The following constraints are taken from (Chowdhury et al., 1993), (Meyers, 1992) and (Meyers et al., 1993), where they are explained in more detail. We will list some examples for design, implementation, and stylistic constraints.

DESIGN CONSTRAINTS

Design constraints are not specific to a certain programming language. They can, however, be specific to certain libraries and/or applications. Typical examples are:

- Subclasses must not redefine inherited non-virtual method.
- A method in a certain class must be overridden in all subclasses of that class.
- A class should not have any public data members (encapsulation).
- Structures in C++ should not be used like classes, i.e., they should have only public members, no methods and should not have a base class.

IMPLEMENTATION CONSTRAINTS

Implementation constraints are design-independent. Violations against them can lead to incorrect program behavior. Thus, it makes sense to have a closer look at locations where implementation constraints are violated. Typical examples are:

- An assignment operator and a copy constructor must be defined for classes that declare a pointer member.
- Every base class should declare a virtual destructor.
- Every class with a constructor should declare a destructor and vice-versa.
- If there exist multiple public inheritance paths from a derived class to one of its base classes, then all these paths should be declared virtual.

STYLISTIC CONSTRAINTS

Some set of naming conventions are adopted in almost all software projects. They are intended to increase the readability of software systems. Typical examples are:

- Class and method names must begin with an upper case letter.
- Constants and global variables must begin with the lower case letters 'c' and 'g', respectively.
- Public, protected and private members of a class must be specified in this order.
- Variable (or any other) names may not contain any underscore characters ("_"), or two consecutive underscore characters ("__").

The power of Hy+ and Graphlog depends on the database being queried. Basically, we are able to check against all constraints that can, for example, be specified with CCEL. However, we have to admit, that for efficiency reasons we do not store as much information as would be necessary to compete with CCEL. But in contrast to CCEL we graphically define queries in order to find violations against various constraints. And additionally, we do not need a special language and a separate, constraint specific tool. We are currently working on investigating also the dynamic behaviour of object-oriented software systems. Whereas CCEL is limited to check against static information we will be able to process dynamic information as well.

Finding stylistic constraints is rather trivial, because in Graphlog we can use regular expressions. Fig. 13 shows how to simply find classes which start with a lower case letter and therefore violate one of the stylistic constraints mentioned above.

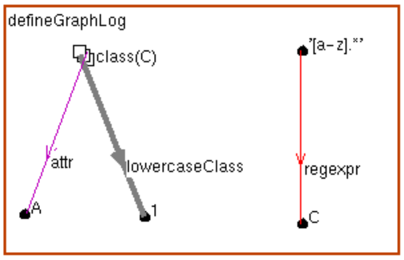

Fig. 13: Definition of classes with lower case letters

A more complex, but still simple example is how to find classes that redefine non-virtual methods (design constraint). For that purpose we need the relation *overrides* defined in Fig. 12. In C++ it is sufficient (and necessary) that the first method in the inheritance path is declared as virtual. We define the relation *overridesVirtual* between a method M3 and a method M2 if M3 overrides M2 and M2 overrides another method M1 which is virtual (see Fig. 14). The attributes node contains information about a method; virtual(yes) indicates that the method has been defined as virtual.

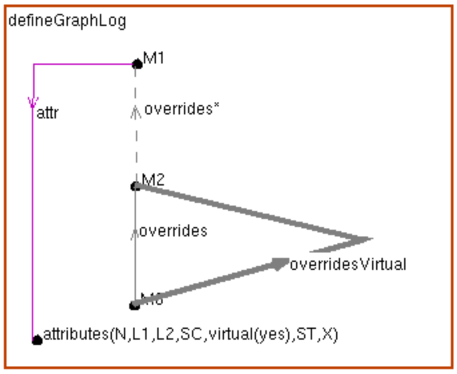

Fig. 14: Definition of the relation *overridesVirtual*

Next, we can simply define the relation overridesNonVirtual, see Fig. 15. Method N overrides a non-virtual method M when N overrides M and the relation overridesVirtual does not hold between M and N.

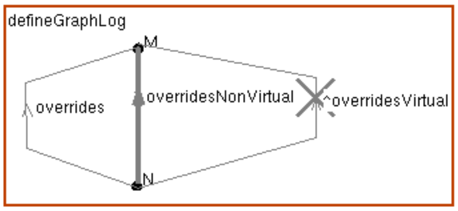

Fig. 15: Definition of methods that override non-virtual methods

We have been rather successful in checking student projects against constraint violations. This is not surprising. For example, our students liked to have public data members, hardly ever implemented a copy constructor or an assignment operator, and some of them did not clean up their systems with destructors. This has become better now, because we explicitly mention these points in the lectures.

However, one of the amazing facts was that there were constraint violations in the framework ET++ that could lead to incorrect program behavior, although the framework had been in use by many programmers for years. We found two classes with non-virtual methods that were overridden in subclasses. Additionally, in ET++ classes assignment operators are not defined. However, this is not a problem as long as all objects are declared as being pointers. But users of the framework could run into troubles.

4. DESIGN PATTERNS

Reverse engineering involves the identification of groups of building blocks like modules and subsystems. For example, the Rigi system gives support for subsystem structure identification (Müller, 1993). In object-oriented systems the identification of modules, i.e., classes, becomes obsolete. Thus, we tried to go one step further by identifying design patterns in a subject system. With patterns piecework is standardized to larger units. For example, a symphony consists of single notes, but various patterns (which are well-known to composers and other musicians) express the design of a piece of music. Software design patterns capture the intent behind the design of a software system. For example, many times there exists a special arrangement of classes and/or objects in order to avoid reuse errors. A subsystem is a set of classes with high cohesion within themselves and low coupling to classes outside the subsystem. Design patterns can correspond to subsystems, but often they have a finer level of granularity.

Design patterns have been identified to avoid dependence on (Gamma et al., 1993):

- classes when creating objects,
- particular operations,
- specific representation or implementation,
- particular algorithms, and
- subclassing as extension mechanism.

Examples of such patterns are:

- **Chains of responsibility**
  create hierarchies of responsibilities, typically arranged from more specific to more general, amongst objects for handling a request.
- **Factory methods**
  allow base classes to create instances of subclass-dependent objects.
- **Flyweights**
  provide stateless objects that can be shared.
- **Glue objects**
  provide a higher level of encapsulation for subsystems by providing an interface for clients to access services of and the objects in a subsystem.
- **Solitaires**
  are classes that have only one instance (e.g., to access unique global services and variables).

For more patterns and more details see (Gamma, 1993) and also (Coad, 1992). Research work on design patterns is still going on and it is not yet clear, which patterns will become widely accepted. But they promise to be one further step in increasing the abstraction level in software development. They can help both in improving the development process and in recapturing design decisions behind the structure of certain parts in a system.

Recapturing design decisions is the crucial point in reverse engineering. This can be done a lot easier if existing design patterns can be identified in a given architecture. Design patterns could be made visible in software systems by special language constructs or special conventions for comments, i.e., annotations. But such constructs or conventions won't be available as long as the fundamentals of the patterns are not clear. Besides, this would help only for identifying patterns in future software systems.

Many of the patterns published so far do not have a clear, identifiable structure. Most of them are defined by various inheritance and use relations. We tried to find some heuristics in order to find possible candidates for various patterns. Sometimes naming conventions are the only way for successful identification. The following heuristics can be used to identify the patterns presented above:

- **Chains of responsibility**
  Classes with "Handler" (e.g., EvtHandler) or "Delegate" in their name or classes that maintain a reference to objects of the same class might be candidates for responsibility chains.

- **Factory methods**
  Possible factory methods are virtual methods containing the words "Make" or "Create", e.g., DoMakeWindow, MakeMenus, CreateScrollbar.

- **Flyweights**
  Any classes without state or only minimal state (<3 instance variables) are possible flyweights.

- **Glue objects**
  We can look for subsystems that have no references to the outer world and vice versa, i.e., a class' subclasses do not have references to classes other than themselves, and classes outside this subtree do not have any references to any of these subclasses.

- **Solitaires**
  We regard all classes as possible solitaires that have at least one static method and/or variable.

With Graphlog it is very simple to define queries in order to find possible design patterns with the heuristics described above. For example, the query for finding factory methods is rather trivial, all we have to do is to look for virtual methods containing the word "Make" or "Create". This is easy with Graphlog, because it not only provides arithmetic functions but also lets the user specify regular expressions.

We will demonstrate how we can easily specify the more complex query to find subsystems that have no references outside and vice versa. Figure 16 defines the relation *pGlue*, which holds for classes that do not have subclasses with a reference outside the subtree of the class (relation *subclassHasRefOutside*) and that do not have classes outside the subtree with a reference inside the subtree (relation *notSubclassHasRefIn-*

side). The superclass relations to a superclass and from a subclass indicate that we do not want to check root and leaf classes.

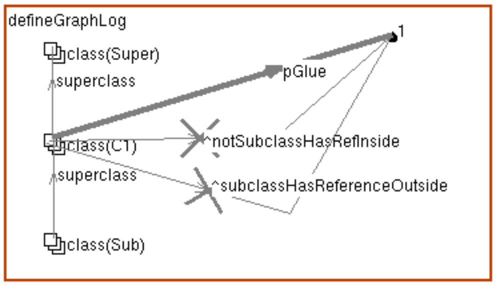

Fig. 16: Definition of pGlue

Next, we define the relation *subclassHasRefOutside*, which is true if anyone of the subclasses has a reference outside the class. For this purpose we use another relation hasRefOutside, see Fig. 17.

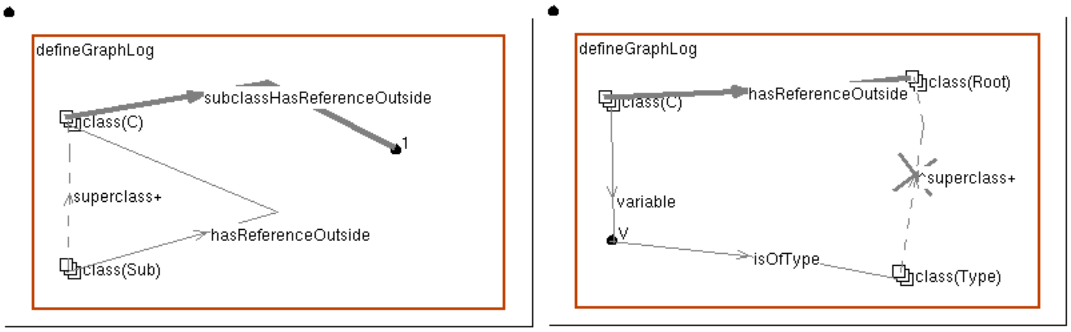

Fig. 17: Definition of subclassHasRefOutside and hasRefOutside

The relation *isOfType* has been defined in the database already. It simply indicates that the type of a variable is of a certain class. The relation *notSubclassHasRefInside* is very similar to *subclassHasRefOutside*, see Fig. 18.

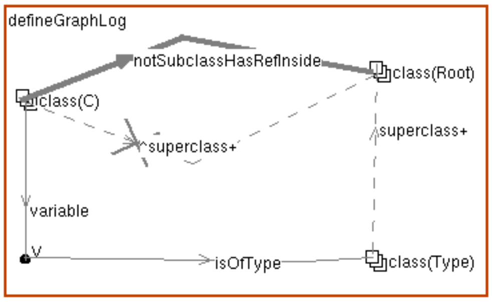

Fig. 18: Definition of notSubclassHasRefInside

The output of a Graphlog query can be modeled very flexible as well. Besides simply showing the resulting methods (and corresponding classes) the result can, for example, be embedded in the inheritance tree with all classes. Figure 19 shows part of the inheritance tree with factory methods. On the screen different colors make the distinction between the relations superclass and factoryMethod easier.

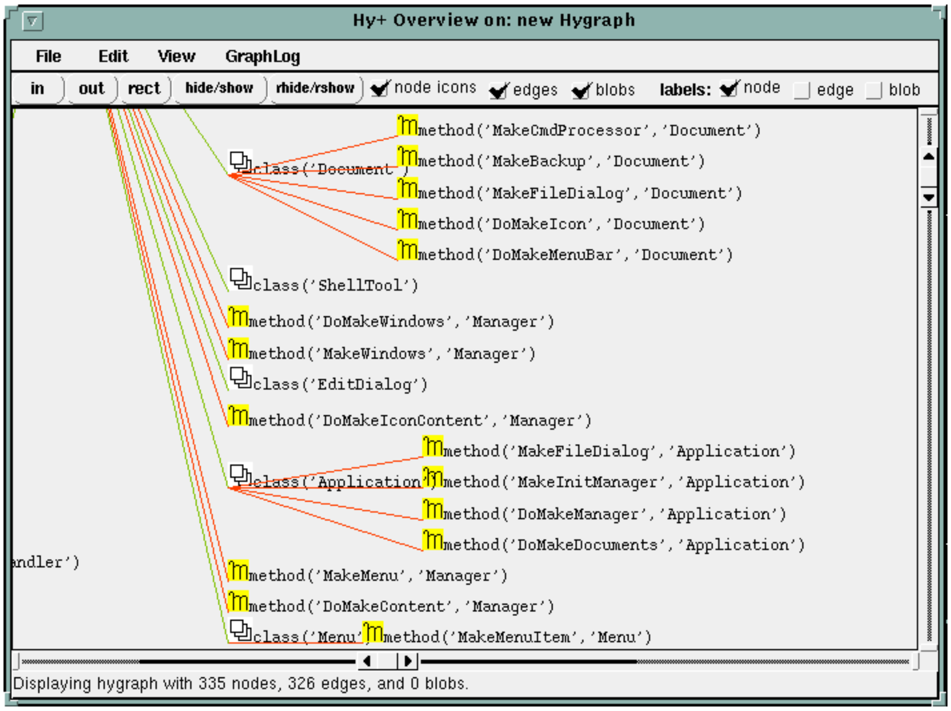

Fig. 19: Factory methods and the inheritance tree

We have to stress that the results yielded by the design pattern queries neither contain all patterns in a software system nor contain only patterns. The heuristics used are helpful in finding possible candidates for certain patterns, i.e., the final decision can be made only by looking at the source code.

We tried to identify design patterns in the application framework ET++. Although we have not been able to define heuristics for all patterns we were able to extract useful information for the comprehension process. For example, a lot of factory methods could be identified that even programmers who were familiar with the inspected class library were not aware of. Factory methods are available not only for application, document and dialog classes, which are used by application programmers, but also for system, window and printer classes, which have not to be extended in order to build applications. In student projects we have never found any of these design patterns. This stems from the fact that they are not experienced enough to make good (excellent?) designs, their projects are rather small, and they know nothing about these patterns because they have to learn more fundamental things.

The Graphlog queries we used so far are custom tailored to the application framework ET++. More research work has to be done on the design patterns and, subsequently, on the heuristics and queries in order provide users with a powerful tool for extracting design decisions from a given software system.

5. DEFINING THE DATABASE FOR REVERSE ENGINEERING PURPOSES

Information about a software system can be obtained by doing syntax and semantics analysis and storing everything a compiler has to know about a system, i.e., structural information (classes, methods) as well as detailed information about the definition and the use of any identifiers. One possible way to get this information is to extend any tool that has information about a software system, e.g., a compiler or a programming environment, in order to output the desired information in the desired format. We extended the programming environment DOgMA (Sametinger, 1992) to generate structural information in the GXF format (Eigler, 1993), which can be read by Hy+. Hy+ stores both databases and queries in the GXF format, see Fig. 20.

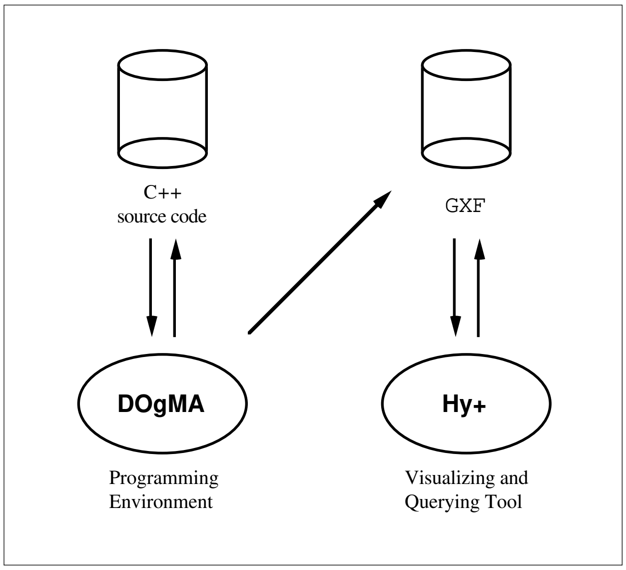

Fig. 20: Architectural overview

We tried to keep the database very simple, reducing the amount of information wherever possible, trying to find a good compromise between less information, that can easily and efficiently be retrieved from software systems, and still useful queries, that allow software engineers to support their comprehension process.

For the queries described in this paper we used the following information in our database:

- classes: class name, line numbers, super classes
- methods method name, class name, line numbers, virtual, scope
- variables: variable name, class name, line number, reference, scope, type
- files file name, number of lines

The number of metrics, constraints and design patterns that can be evaluated, checked and identified, respectively, depends on the amount of information in the database. With the information shown above we cannot, for example, determine the metric *lack of cohesion in methods*, see (Chidamber et al., 1991). Also, some constraints listed in

(Meyers, 1992) require information about parameter types and single statements, e.g., constraints on the assignment operator:

- do not return a reference to *this
- assign a value to all data members
- do not assign a value to self
- return type must be a reference to the class

It is not possible to check for these constraints without the necessary information. With little effort, however, this and other information can be added to the database as well. In contrast to metrics and constraints, more information in the database would not help much in recognizing more design patterns. To achieve better results we need more research work being done on the design patterns themselves.

6. CONCLUSION

The importance of program comprehension, the necessity of getting familiar with unknown source code, and extracting information from low level data increases with the object-oriented programming paradigm. The comprehension process could additionally benefit from extending the database and finding more valuable queries. Especially, taking dynamic behavior into consideration could bring new insights.

Reverse engineering activities become necessary not only for maintenance but also for development purposes. With Hy+ and Graphlog we have a very flexible tool that can be used for those purposes. Special purpose tools—like ATHENA, CCEL and Rigi—are custom tailored and can hardly be surpassed by general purpose tools in their application domains. However, the power of Hy+ and Graphlog lies in its universality, which makes it more easily applicable also for future formulations of questions. Additionally, the visualization and navigation features can help in mastering the complexity of large-scale software systems. In this papers we mainly focused on querying features, but Hy+ also offers a lot of functionality for adjusting the visualization of results (e.g., using fisheye views), applying filters, etc. (Consens et al., 1991) and (Mendelzon, 1993). And, even if Hy+ and Graphlog are not applicable in certain domains for performance reasons (which might become true when the database becomes very extensive), they are excellent tools for evaluating needed functionality before special purpose tools are being built.

7. REFERENCES

Chikofsky E.J. and Cross II J.H. (1990). Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, Vol. 7, No. 1, pp. 13-17.

Chidamber S.R. and Kemerer C.F. (1991). Towards a Metrics Suite for Object Oriented Design, OOPSLA '91 Proceedings, pp. 197-211.

Chowdhury A. and Meyers S. (1993). Facilitating Software Maintenance by Annotated Detection of Constraint Violations, IEEE Conference on Software Maintenance, Montréal, Canada, September 1993, and Brown University, Technical Report No. CS-93-37.

Coad P. (1992). Object-Oriented Patterns, Communications of the ACM, Vol. 35, No. 9, pp. 152-159.

Consens M.P. (1989). Graphlog: "Real Life" Recursive Queries Using Graphs, Masters Thesis, Department of Computer Science, University of Toronto.

Consens M.P. and Mendelzon, A.O. (1990). GraphLog: A Visual Formalism for Real Life Recursion, Proceedings of 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 404-416.

Consens M., Mendelzon A. and Ryman A. (1991). Visualizing and Querying Software Structures, IBM Canada Laboratory, Technical Report TR 74.053.

Consens M.P., et al. (1994). Architecture and Applications of the Hy+ System, IBM Systems Journal

Eigler F.C. (1993). GXF: A Graph Exchange Format, contained in (Mendelzon, 1993).

Gamma E., et al. (1993). Design Patterns: Abstraction and reuse of object-oriented design, European Conference on Object-Oriented Programming, Kaiserslautern, Germany.

Harel, D. (1988). On Visual Formalisms, Communication of ACM, Vol. 31, No. 5, pp. 514-530.

Mendelzon A.O. (1993). Declarative Database Visualization: Recent Papers from the Hy+/Graphlog Project, Computer Systems Research Institute, University of Toronto, Technical Report CSRI-285.

Meyers S. (1992). Effective C++ 50 Specific Ways to Improve Your Programs and Designs, Addison Wesley.

Meyers S., et al. (1993). Constraining the Structure and Style of Object-Oriented Programs, Brown University, Technical Report No. CS-93-12.

Müller H.A., et al. (1993). A Reverse-engineering Approach to Subsystem Structure Identification, Software Maintenance: Research and Practice, Vol. 5, No. 4, pp. 181-204.

Sametinger J. (1992). DOgMA: A Tool for the DOcumentation & MAintenance of Software Systems, VWGÖ, Vienna.

Tsalidis C., et al. (1992). ATHENA: A Software Measurement and Metrics Environment, Software Maintenance: Research and Practice, Vol. 4, No. 2, pp. 61-81.

Weinand A., et al. (1989). Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No.2.