Dagger: A Tool to Generate Program Graphs

Yih-Farn Chen

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
chen@research.att.com

Abstract

Dagger is a tool that generates program graphs to selectively visualize a software structure. The design of dagger achieves a strong degree of language independence by exploiting the duality between a class of entity-relationship databases and attributed directed graphs. This paper describes the C and C++ versions of dagger, which map a selected subset of relationships stored in a program database to a graph specification, decorate the graph with display attributes, and then pass it to layout tools or interactive graph browsers. Dagger takes output from database queries to generate a large variety of interesting program graphs, including header file hierarchy, module binding, and type inheritance graphs. This paper describes the graph generation process surrounding dagger and a sample of tools involved in the process. In particular, we describe how a closure operator works in tandem with dagger to control graph complexity by generating reachability graphs where sub-structures are selectively ignored or expanded. Our experience in applying dagger to several software projects has demonstrated its capability in abstracting and visualizing complex software structures without much overhead. All program graphs presented in this paper are tagged with timing statistics.

1 Introduction

Programmers are frequently faced with the task of maintaining complex software systems with inadequate documentation. The situation is analogous to maintaining a complex building with inaccurate or missing blueprints. Tools that abstract and visualize such software systems to regenerate the necessary *software blueprints* can help programmers maintain the structure more effectively.

This paper presents a program visualization approach that focuses on automatically generating abstractions of the static code structure. The visualization process consists of a sequence of mappings. First, an *abstractor* analyzes the source of a program and maps it to a program database according to an entity-relationship model[3]. Next, a *query* or *operator* retrieves a subset of the database, which is then mapped to a directed graph. Finally, the graph specification is passed to layout tools or interactive graph browsers. The complete graph generation process is diagramed in Figure 1.

This paper describes **dagger**, a program visualization tool that controls the mappings from a program database to a directed graph specification. The design of dagger is largely language-independent because of its focus on entities and relationships. The program database used by dagger for C is generated by CIA[4][5], and the one for C++ is generated by CIA++[11][10]. The directed graph specification generated by dagger is acceptable by the graph-drawing tool **dot**[9][8]. By changing a set of mapping functions, dagger can easily adapt to different entity-relationship databases and automatic layout tools.

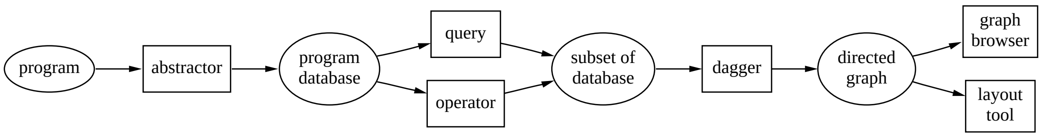

Figure 1: The process of program graph generation is a sequence of selective mappings

Dagger takes the output from database queries and operators designed for the CIA and CIA++ program databases to generate a large variety of program graphs, including header file hierarchy, entity dependency, module binding, type inheritance, reachability, and *focus* graphs (see Section 4). For example, Figure 2 shows the type dependency graph of a C program generated by dagger and drawn by dot with the following simple command pipeline[1]:

```
$ dagger type - type - | dot -Tps
```

The graph generation process, including query execution, mapping to directed graphs, and generation of the PostScript® layout, took 3.88 seconds (1.20 seconds in user time + 2.68 seconds in system time) on a Series5/800 Solbourne® (a Sun4 clone) server running OS/MP 4.1A.3. All pictures in this paper are labeled with timing data collected on this machine. To generate this graph, dagger selects only type-to-type relationships. The picture reveals the mutual dependencies between the two types `struct _sfdc_` and `Sfdisc_t`. It also shows the relative structural complexity of `Ciarec_t` and `Symbol_t`.

Some abstractions such as reachable sets and program layers cannot be readily expressed by traditional database queries and must rely on special operators. For example, the CIA++ tool **Subsys** uses the closure construct in the Daytona data management system[12] to compute all entities and relationships reachable from a particular entity. Dagger is designed as an open-ended tool to optionally accept output generated by these operators to visualize their corresponding abstractions.

Roman and Cox define program visualization as a process of mapping programs to graphical representations and give a comprehensive survey on its various forms[18]. While static program visualization tools have become more available in recent program development environments such as ObjectCenter®[15], SMARTsystem®[17], and Energize®[1], unlike dagger, they usually lack the abstraction power and flexibility needed to selectively visualize facets of complex structures in large software projects.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the entity-relationship model of a C program database. Section 3 describes the mapping process of dagger followed by several examples of program graphs. Section 4 discusses how dagger can be used with a closure operator to provide interesting graphs beyond what typical queries can express. Section 5 describes our experience in applying dagger to large software projects and presents future directions in our program visualization efforts. Section 6 concludes with a summary.

2 An Entity-Relationship Database for C

Dagger is essentially a filter that takes as input a set of binary, directed relationships. To generate program graphs for C and C++ , it relies on information stored in the CIA and CIA++ program databases, which are based on the entity-relationship model. Both CIA and CIA++ record five kinds of non-local program entities (files, functions, types,

[1]The `-Tps` option of dot sets the output language to PostScript®.

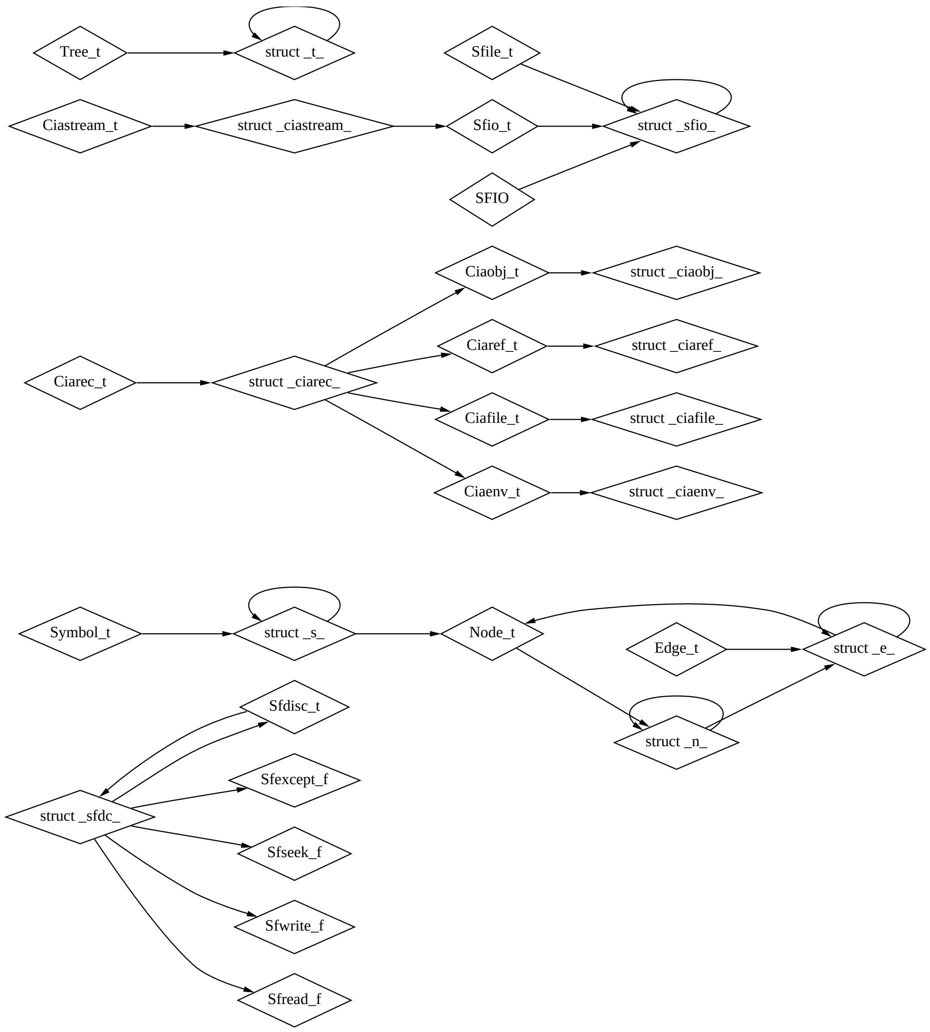

Figure 2: A type dependency graph generated by dagger and drawn by dot. The shape of each node is determined by its entity-kind attribute. By default, C types are mapped to diamonds.

variables, and macros) and their inter-relationships. For example, consider the following C program:

```
/* file ftable.c */
#include "coor.h"   /* defines types COOR and struct coor */
#define TBLSIZE 2
extern COOR *rotate();
extern COOR *shift();
typedef COOR *(*PFPC)();
static PFPC ftable[TBLSIZE] =  { rotate, shift };
```

After processing this piece of code, CIA records nine entity declarations:

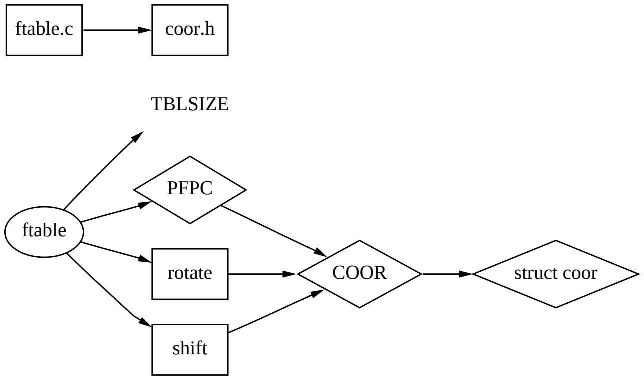

Figure 3: The complete program graph of a small C program. Files and functions are shown in boxes, types in diamonds, variables in ellipses, and macros in plain text.

files: `ftable.c` and `coor.h`
functions: `rotate` and `shift`
types: `PFPC`, `COOR`, and `struct coor`
variable: `ftable`
macro: `TBLSIZE`

`CIA` also records nine relationships between these entities. For example, `ftable` refers to the macro `TBLSIZE`, the type `PFPC`, and two functions, `rotate` and `shift`, which in turn refer to the type `COOR`. In addition, the attributes (storage class, data type, definition/declaration, etc.) of each entity declaration are recorded in the database. In summary, the job of CIA is to selectively map program source text to an entity-relationship database, complete at its abstraction level, by ignoring certain details such as data and control flow information involving only local variables.

The complete database for the above piece of code can be easily mapped by dagger to a graph shown in Figure 3. However, for any real and complex programs, we must rely on abstractions to retrieve only a subset of all the relationships in a program database. In CIA and CIA++, abstractions can be obtained through queries or specialized program database operators. Queries for CIA and CIA++ are based on either **cql**[7] or Daytona[12]. They provide flexible access to information stored in the program databases. For example, the following **cref** query of CIA returns all functions referred to by the the variable `ftable`[2]:

```
$ cref var ftable func -
k1 file1             name1              k2 file2             name2
== ================= ================== == ================= ===============
v  ftable.c          ftable             p  ftable.c          rotate
v  ftable.c          ftable             p  ftable.c          shift
```

All CIA tools (including dagger) that deal with relationships share this interface, where the arguments specify patterns for the two end points of a reference relationship: (*parent kind*, *parent name*) and (*child kind*, *child name*). All relationships that match the specified patterns are retrieved.

In order for other UNIX® commands (such as **awk**) and CIA tools (such as dagger) to further process the reference information, cref also generates unformatted output with

[2]In the CIA database, `p` is used as a shorthand for functions (procedures).

complete attributes for both the parent and child entities[3]:

```
$ cref -u var ftable func -
12;ftable;variable;0;ftable.c;PFPC [];static;7;0;7;def;3118aafa;; \
         6;rotate;function;0;ftable.c;COOR *;extern;4;0;4;dec;bed65a3a;;7
12;ftable;variable;0;ftable.c;PFPC [];static;7;0;7;def;3118aafa;; \
         10;shift;function;0;ftable.c;COOR *;extern;5;0;5;dec;e875f5f4;;7
```

This unformatted form of the `cref` output provides a standard data exchange format between many CIA tools, including subsys and dagger.

3 Generating Program Graphs with Dagger

Conceptually, dagger performs four steps during the mapping from a program database to a directed graph specification:

- *Selection*: Select a subset of relationships from a database by invoking queries or taking output from database operators.
- *Decoration*: Use the attribute values of entities or relationships to determine their corresponding display attributes, such as color, style, and shape.
- *Projection*: Select a subset of the entity and relationship attributes to filter out information unnecessary (such as the checksum of each entity) for applications of the final graphs.
- *Mapping*: Generate an attributed directed graph specification by mapping each entity to a node and each relationship to an edge. Graph attributes such as page size and aspect ratio are then added to control the graph layout.

The mapping functions in the decoration step are customized for each language and chosen to enhance the understanding of a program structure. For example, dagger maps different kinds of C or C++ entities to different shapes: file and function entities are mapped to box nodes, types to diamonds, variables to ellipses, and macros to plain text. In the case of C++ inheritance diagrams, the three possible values of the *access-specifier* attribute of each class inheritance relationship (public, protected, and private) are mapped to different edge styles: solid, dashed, and dotted. In addition, an edge is labeled “`v`” if the inheritance relationship is virtual. The decoration, projection, and mapping steps are straightforward, so we will focus on the selection step in this paper. The rest of this section and the next section give examples of graphs that are created through queries and operators, respectively.

While both CIA and dagger have been used to generate databases and graphs for projects that involve millions of lines of code, most program graphs presented in this paper are derived from the CIA program databases of two popular tools: **incl**[20] and **xgremlin**, an X11®version of **gremlin**[16]. Both programs have many users and have evolved over the years to meet user demands. Some metrics on the complexity of these two programs are shown in Table 1. The numbers include all the user and system header files used by these two programs, but they exclude the source files of libraries used. Both source and database sizes are measured in the number of bytes.

program	*lines*	*source size*	*db size*	*entities*	*relationships*
incl	1,957	49,963	26,416	392	517
xgremlin	24,582	620,726	390,430	4,842	6,634

Table 1: Complexity metrics for the two examples of program databases

[3]Due to typesetting constraints, each line of the actual output is split into two lines, separated by a backslash character.

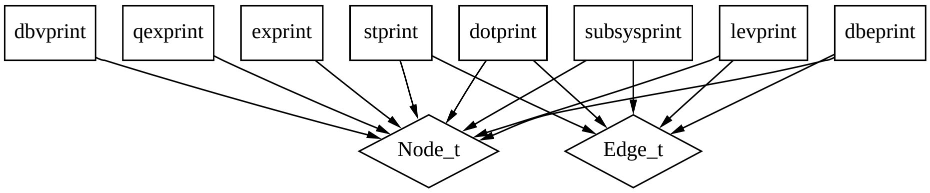

Figure 4: All printing functions in incl depend on `Node_t`, `Edge_t`, or both (generation time: 0.76u + 2.48s = 3.24 seconds)

Besides simple query patterns, dagger and cref support **ksh**[2]-like regular expressions, logical operators, and negations. For example, the following command generated a program graph of incl shown in Figure 4[4]

```
$ dagger -R func '*print' type - | dot -Tps
```

The diagram shows a selected subset of function-to-type relationships. Only functions whose names end with `print` and the types they refer to are displayed. The picture reveals an interesting structural fact that is not obvious in the corresponding relational view below: all functions that depend on `Edge_t` also depend on `Node_t`.

```
$ cref func '*print' type -
k1 file1             name1              k2 file2             name2
== ================= ================== == ================= =================
p  incl.c            dbeprint           t  incl.h            Node_t
p  incl.c            levprint           t  incl.h            Node_t
p  incl.c            dbeprint           t  incl.h            Edge_t
p  incl.c            exprint            t  incl.h            Node_t
p  incl.c            levprint           t  incl.h            Edge_t
p  incl.c            subsysprint        t  incl.h            Node_t
p  incl.c            qexprint           t  incl.h            Node_t
p  incl.c            dotprint           t  incl.h            Node_t
p  incl.c            stprint            t  incl.h            Node_t
p  incl.c            subsysprint        t  incl.h            Edge_t
p  incl.c            dotprint           t  incl.h            Edge_t
p  incl.c            stprint            t  incl.h            Edge_t
p  incl.c            dbvprint           t  incl.h            Node_t
```

Dagger can also be used to visualize array dependencies on macros, which are frequently used to set array sizes. The following command pipeline generates the graph shown in Figure 5, which shows all those array variables that depend on any one of the following macros `NFONTS`, `NSIZES`, or `N*STIPPLES`. The *selection clause* `file1=graphics.c` further limits our interests to only those arrays defined in `graphics.c`. The picture reveals that eight arrays satisfy the criteria and two of them depend on both `NFONTS` and `NSIZES`. These two arrays (`charysizes` and `font_info`) are two-dimensional.

```
$ dagger -R v - m 'NFONTS|NSIZES|N*STIPPLES' file1=graphics.c | dot -Tps
```

As an example for C++ program graphs, Figure 6 shows a typical type inheritance graph generated from a CIA++ program database. Note that the edge styles are adjusted according to the access specification of the inheritance relationship. `WorkTree` has a protected inheritance relationship with `Tree`, while `DispTree` has a private one. All other inheritance relationships are public and virtual ones are labeled with "v". The

[4]The `-R` option of dagger forces the picture to be drawn from top to bottom instead of left to right, the default direction.

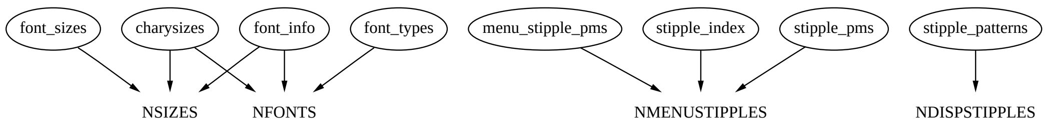

Figure 5: A graph that shows all variable arrays in `graphics.c` whose sizes depend on the selected macros (generation time: 1.28u + 2.23s = 3.51 seconds)

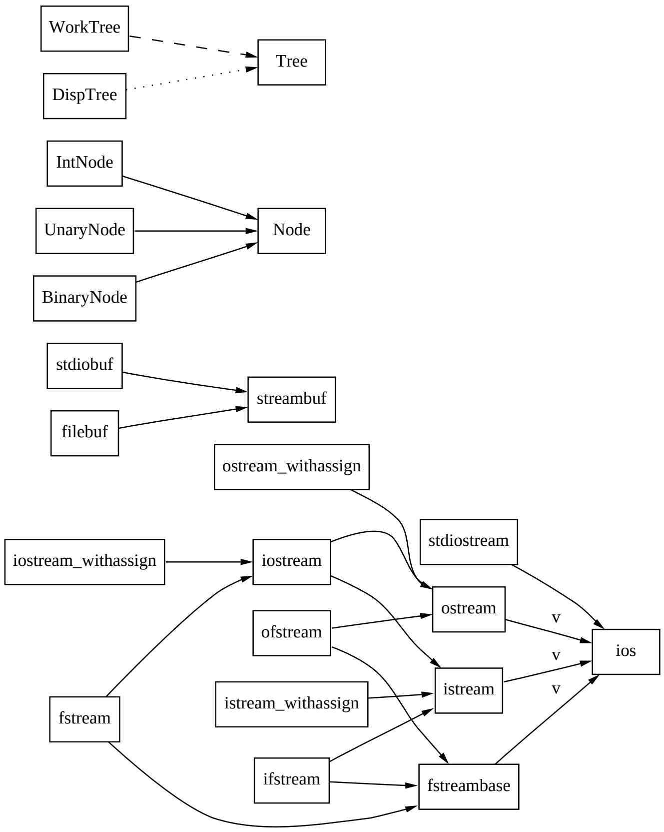

Figure 6: A C++ type inheritance graph (generation time: 1.98u + 4.66s = 6.64 seconds)

graph is generated by Dagger, the C++ version of dagger, with the following command pipeline, by limiting the relationship kind to inheritance (`rkind=i`):

```
$ Dagger type - type - rkind=i | dot -Tps
```

Note that the first two graphs in this section have only two layers because their parent and child entity kinds are different. The complexity of diagrams in this class of graphs can be managed easily by using regular expressions and selection clauses to narrow down the subset in each end. In the class of graphs where the parent and child entity kinds are the same, there tend to be many more layers because of their transitive nature. Examples include the type inheritance graph (type-to-type) shown in Figure 6, function reference graphs (function-to-function), and include hierarchies (file-to-file). Getting a good abstraction out of these graphs requires more than simple query manipulations.

As an example, Figure 7 shows the complex include hierarchies in xgremlin that are generated by the following command[5]:

```
$ dagger -F file - file '!(icons/*)' | dot -Tps
```

The diagram has been simplified by using a *negation* operator in the child name pattern to ignore all include relationships involving header files stored in the `icons` subdirectory. Unfortunately, the picture is still too difficult to read on a single page. What the user frequently wants is a single include hierarchy rooted at a particular source file with all other include relationships ignored.

While the labels in Figure 7 are hard to read, the function reference graph in Figure 8 generated by the following command seems to be beyond comprehension:

```
$ dagger -F func - func - | dot -Tps
```

The diagram does show some interesting facts, though:

- A few functions have very large fanins. Further studies on these functions can be performed with the CIA tool **ciafan**.
- Many functions do not seem to be on any of the reference paths from `main` (the root function). There are two possibilities: a) These are dead functions that never get exercised. The CIA tool **deadobj** does detect 11 dead functions in xgremlin. b) These are functions indirectly invoked through variables. In this case, both function-to-function and variable-to-function relationships (such as the one between `ftable` and `rotate` in the C example in Section 2), must be drawn to complete the picture. Xgremlin indeed has many functions defined in a variable array for menu selections. Section 5 discusses how dynamic function reference relationships can be captured for this class of programs during program execution.

A brute-force way to solve the graph complexity problem is to blow up the diagram to such an extent that all symbols become legible and use an option of dot to paginate the diagram. Magnifying complex graphs, however, is usually not satisfactory because the number of edge crossings increases substantially and they become a major distraction. In the next section, we show how dagger can work with a closure operator to limit graph complexity.

4 Managing Graph Complexity

One way to manage the complexity of a large program graph is to concentrate on individual nodes of interest. Typical maintenance tasks might require

- Finding all program entities that an entity depends on directly or indirectly. This information is particularly useful for packaging software components and partitioning complex program graphs.
- Finding all program entities that might be affected by the change of an entity. TestTube[6], a system for selective regression testing, uses such information to determine what test cases must be rerun.
- Displaying a few layers of relationships centered around a particular node.

There is no easy way to specify such subsets of relationships using a simple query syntax. Fortunately, dagger is designed as an open-ended tool to optionally take output from any tools that follow the standard exchange format shown in Section 2. One such tool is the CIA closure operator **subsys**, which computes all entities and relationships reachable (either forward or backward) from selected entities. For example, the include hierarchy of `graphics.c` in xgremlin is generated by the following command pipeline[6]

[5]The `-F` option requests dot to stretch a picture and change its aspect ratio to maximize the use of all the space available within the specified area.

[6]The `-u` and `-e` options force subsys to generate the **cref -u** format.

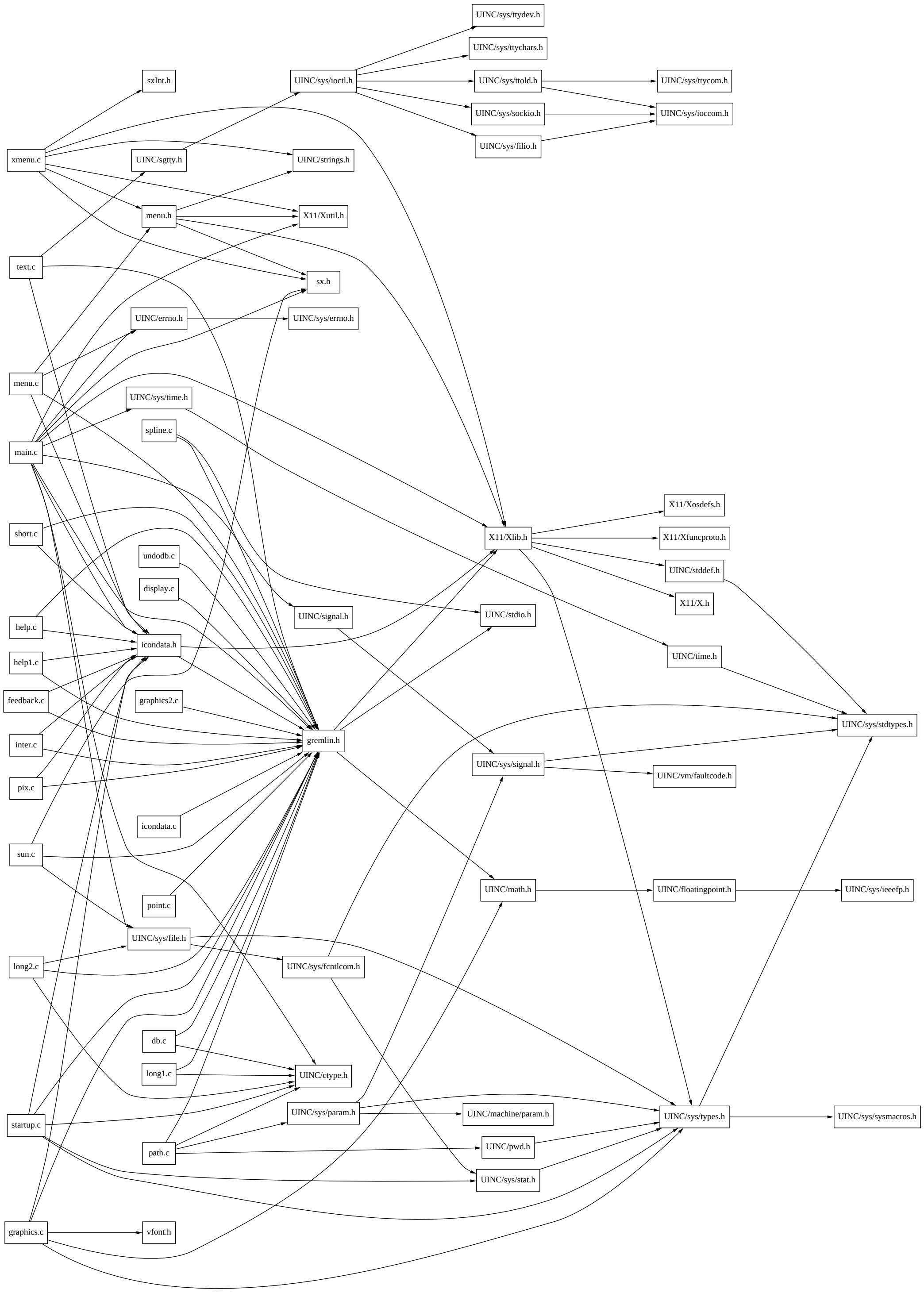

Figure 7: The simplified include graph of xgremlin (generation time: 5.30u + 6.01s = 11.31 seconds)

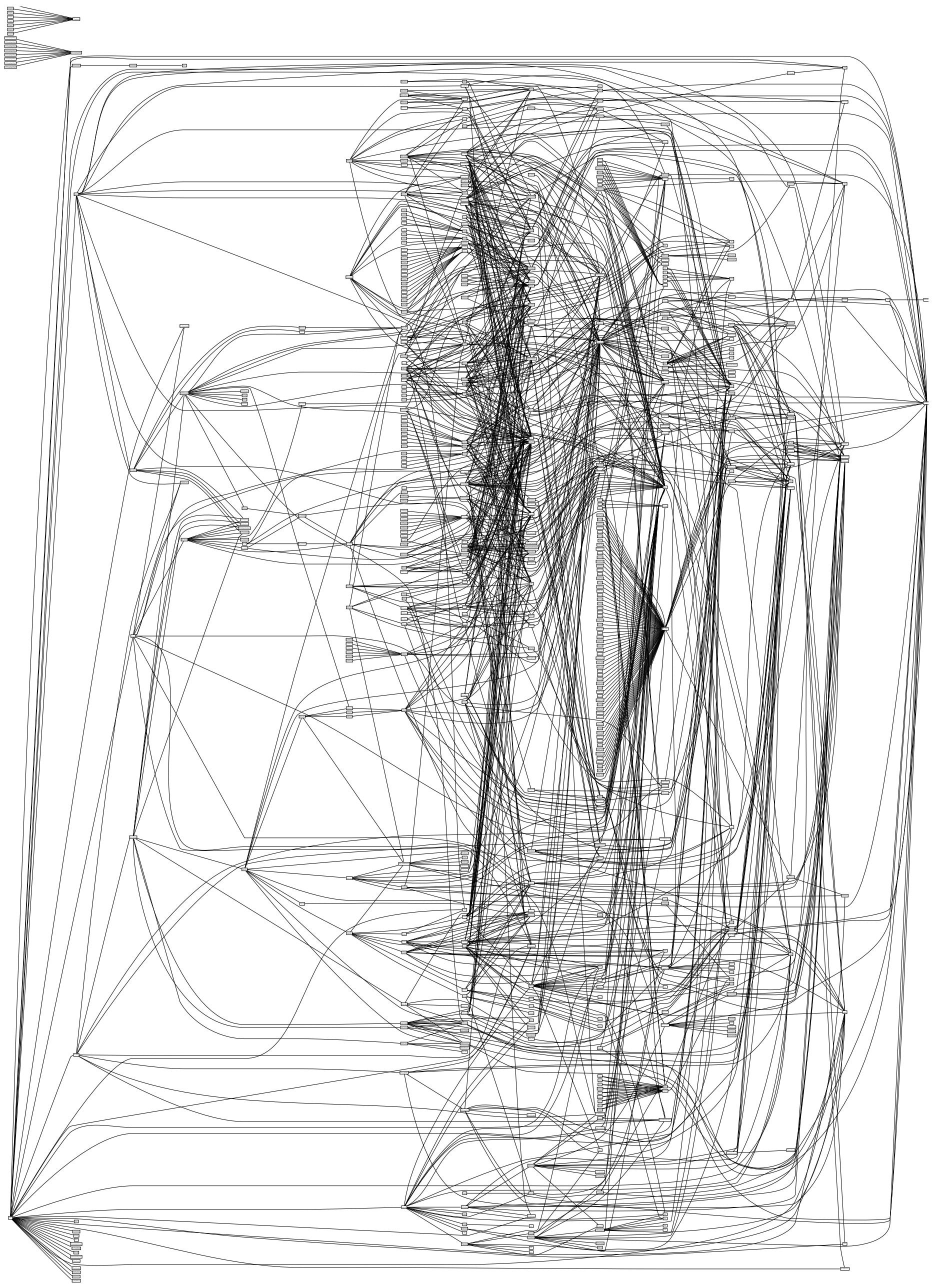

Figure 8: The complex function reference graph of xgremlin (generation time: 242.76u + 48.90s = 291.66 seconds)

and shown in Figure 9. The graph generation time includes the closure computation by subsys, the directed-graph generation by dagger, and the graph layout by dot.

```
$ subsys -u -e file graphics.c | dagger -i | dot -Tps
```

By default, subsys traces only reference relationships of *kind-to-kind*, where *kind* is the root entity kind. In the previous example, only file-to-file relationships are traced. However, it is frequently necessary to compute a *complete* reachable set, where all kinds of relationships are considered. For example, the following command (using the `-k` option)

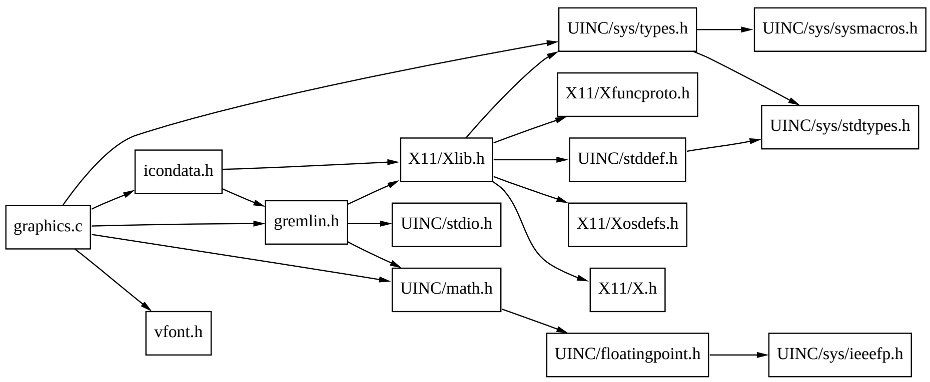

Figure 9: The include hierarchy rooted at `graphics.c` (generation time: 0.86u + 1.91s = 2.77 seconds)

generates a complete subsystem rooted at the function `mustuse` of `incl`. The results are shown in Figure 10.

```
$ subsys -u -k -e func mustuse | dagger -i | dot -Tps
```

Since dagger is just a filter, the set of graphs generated can be further extended by mixing an interesting collection of raw `cref` records and presenting the resulting set to dagger. The following simple shell script concatenates the output of two subsys commands and generates a *focus* graph with the specified entity at the center. The first subsys command traverses the graph starting from the root in a forward direction and the other one traverses backwards.

```
# Usage: focus [options] entity_kind entity_name
subsys -u -e "$@" > $$.1
subsys -u -e -r "$@" > $$.2
cat $$.1 $$.2
rm -f $$.1 $$.2
```

Figure 11 shows a focus graph centered at the incl function `mksymbol`, which is created by the following command. The `-l` option of subsys (and focus) limits the graph traversals to only two levels deep in each direction.

```
$ focus -k -l2 fu mksymbol | dagger -i | dot -Tps
```

Even the reachable set of a selected entity may be so complex that further simplifications are necessary. One common technique is to recursively partition a reachable set into trackable pieces. During the generation of a subsystem, we can stop the graph traversal at certain selected nodes, ignore their sub-hierarchies, and later expand these nodes as necessary. For example, Figure 12 shows the subsystem of the xgremlin function `GRArc` by ignoring the relationship hierarchy rooted at `GRVector` (with the `-i` option). The graph is generated by the following command pipeline, in which a simple **sed** script **markn** is used to change the color and style of the ignored node.

```
$ subsys -u -e -k -i GRVector fu GRArc | dagger -i | markn GRVector
         | dot -Tps
```

It turns out that the subsystem of `GRVector` is still so complex that we have to ignore two entities during the graph traversal of itself: the two types `GC` and `Display`. Following is the command that generates the partial subsystem graph of `GRVector` shown in Figure 13. It is conceivable that another shell script can be written to generalize the partition process.

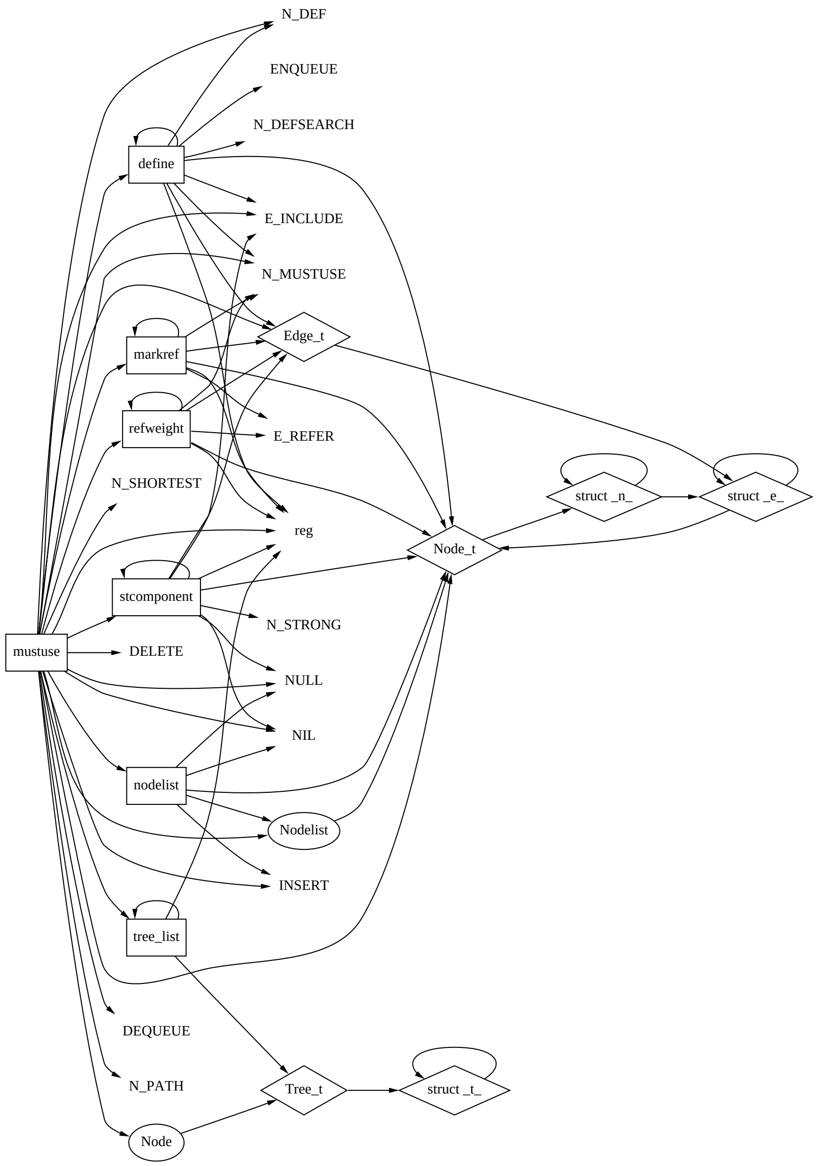

Figure 10: A complete subsystem graph rooted at the incl function `mustuse` (generation time: 2.66u + 2.36s = 5.02 seconds). Functions are shown in boxes, types in diamonds, variables in ellipses, and macros in plain text.

```
$ subsys -u -e -k -i Display -i GC fu GRVector | dagger -i |
        markn Display FC | dot -Tps
```

5 Experience and Future Directions

Dagger has been exercised by dozens of AT&T software projects. Many features in dagger and subsys, such as the style of mapping in the inheritance graphs and the partition process of subsys, are direct results of user feedback. The selective nature of dagger makes the generation of a variety of program graphs a simple task during software

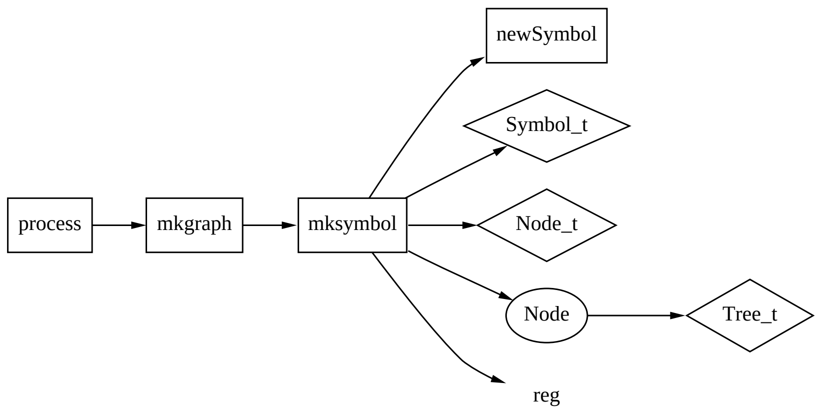

Figure 11: A five-layer focus graph centered at the incl function `mksymbol` (generation time: 1.61u + 3.85s = 5.46 seconds)

maintenance.

Several new directions in our program visualization efforts related to dagger and dot are emerging:

- *Cluster Graphs*:

  The type dependency graph shown in Figure 2 can be modified by using an explicit, labeled cluster to group each set of types belonging to the same header file. Such a graph provides information on both header file and data structure dependencies. The notion of *subgraph* in dot is ideal for this purpose. A subgraph in dot is a subset of the graph's nodes and edges with its own graph attributes (such as its own graph label and default node and edge values). In general, dagger can use the value set of any entity attribute to create clusters. For example, one clustering approach is to group variables in a program graph according to their storage classes.

- *Records for Containment Relationships*:

  Another desirable clustering method is to group all members of a C structure or a C++ class. A natural representation for the containment relationships between an aggregate structure and its members (which may themselves be aggregate structures) is dot's notion of *record*, a recursive array of labeled boxes. Dagger should be extended to visualize interactions between C++ class members using record nodes.

- *Undirected Graphs*:

  So far, this paper discusses only directed program graphs, but some program structure information is best represented as undirected graphs. The CIA tool **share**, which is based on Daytona's query language Cymbal, computes the number of shared child entity references between any pair of parent entities. A high degree of sharing frequently implies that the two entities operate or depend on similar entities. Such a pair can be considered *strongly coupled* because a change in one entity is likely to affect the other. We can use the undirected-graph layout tool **neato**[14] to adjust the distance between any pair of nodes by mapping the coupling strength to the weight (or spring constant) of an edge. By tuning the mapping function, we attempt to create natural clusters where strongly coupled entities are drawn to each together.

- *Customized Mapping*:

  Dagger uses pre-defined functions to map entity and relationship attributes in C and C++ to display attributes. This approach is not satisfactory because a program-

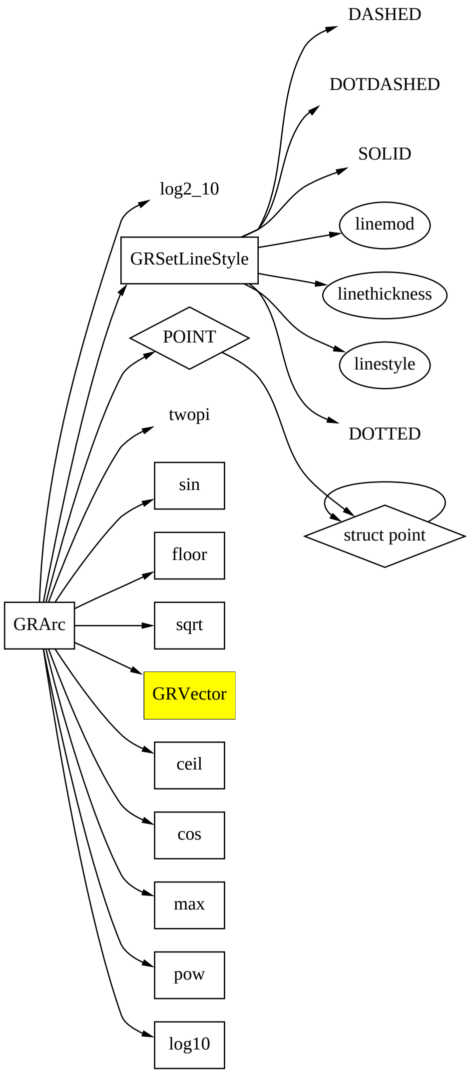

Figure 12: A partial subsystem graph rooted at the xgremlin function `GRArc` (generation time: 1.33u + 2.65s = 3.98 seconds)

mer may decide to change the mapping to emphasize certain aspects of a program graph. For example, in a general C++ type dependence graph involving inheritance, friendship, and reference relationships, the edge styles (solid, dashed, dotted) may be used to represent these three types of relationships rather than the three kinds of access specifications (public, protected, and private) of inheritance relationships. We are considering a simple mapping mechanism based on dynamically changing cql's schema to support customized mapping.

- *Dynamic Program Graphs*:

  The static function reference graph shown in Figure 8 only reveals the static function dependency relationships, which do not necessarily reflect the actual function calls during an instance of program execution. Work is underway in a program animation project to merge dynamic function reference information produced by a version of **app**[19], an annotation preprocessor enhanced with instrumentation capabilities,

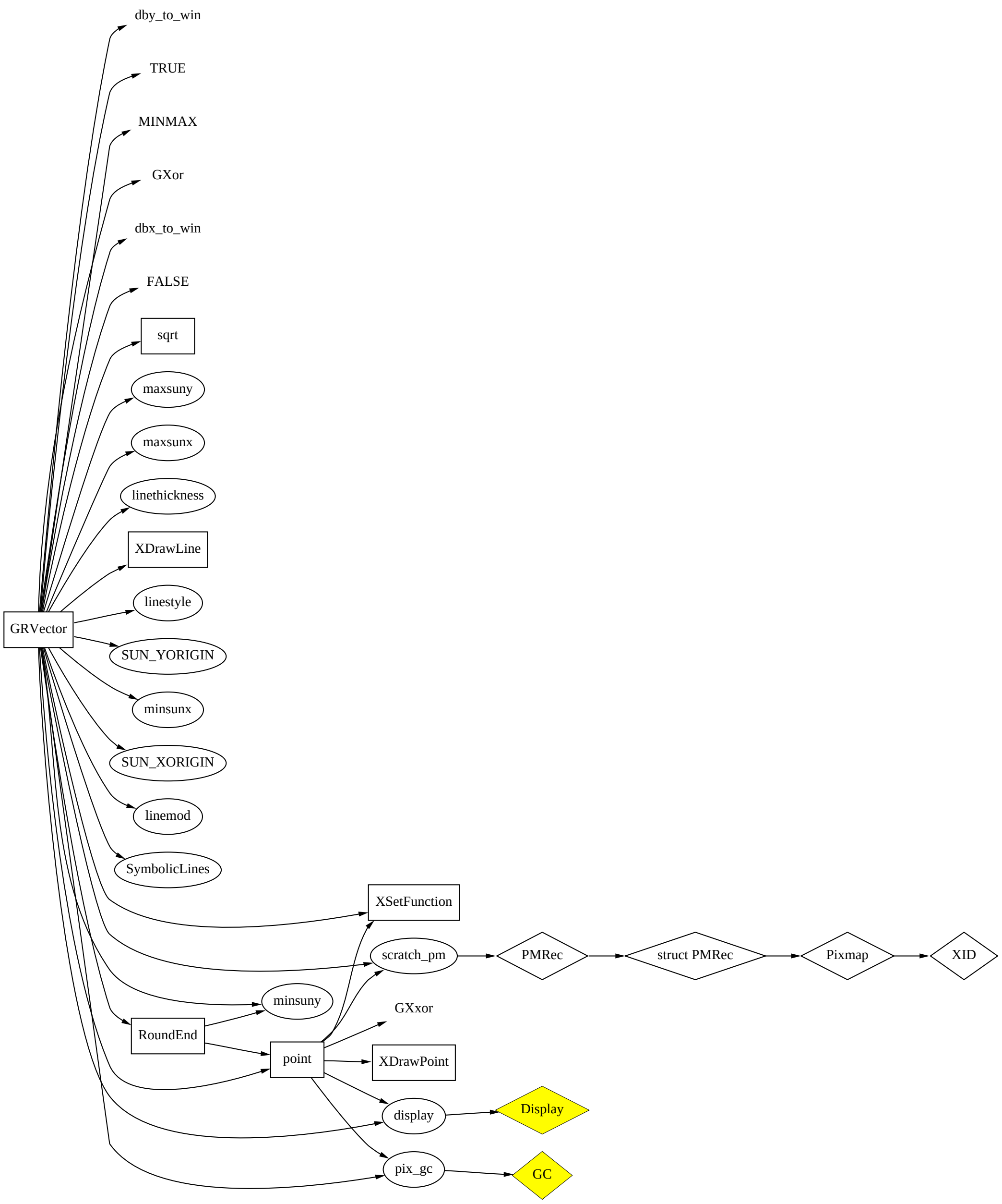

Figure 13: A Partial Subsystem Graph Rooted at the Xgremlin Function `GRVector` (generation time: 1.86u + 3.10s = 4.96 seconds)

with the static information recorded in the CIA database.

- *Program-Graph Browser*:

  Instead of serving merely as passive displays, the program graphs generated by dagger and dot can become active interfaces for interactions with the CIA program database. **Dotty**, a graph editor based on dot and **lefty**[13], has allowed us to build an interactive graphical interface called **ciao**[7] for the CIA and CIA++ databases.

[7] Joint work with Eleftherios Koutsofios.

Programmers can generate textual, relational, and graphical views with great ease from pop-up menus without memorizing the cryptic dagger and subsys options. The menus are attached to program graphs and their individual nodes and edges.

6 Conclusion

By using a sequence of selective mappings through program database queries and closure operators, dagger implements a simple yet expressive mechanism to generate a large variety of interesting program graphs. The design of dagger exploits the duality between a class of entity-relationship databases and attributed directed graphs. The mapping and complexity management techniques described in this paper should be applicable to program databases for other languages – as long as they are constructed with a proper entity-relationship model. We expect program visualization tools like dagger to be widely used in the future to help programmers study, preserve, and improve structures in software systems.

7 Acknowledgements and Availability

Thanks to Susan Ashmore, David Belanger, Karen Brown, Emden Gansner, David Korn, Eleftherios Koutsofios, Steve North, and Chris Rath for their helpful comments on earlier drafts of this paper. Thanks to all those colleagues in the Software Engineering Research Department who have provided well-designed and reusable software components to make this work possible. All the dagger-related tools described in this paper can be obtained by contacting **cia@mozart.att.com**.

REFERENCES

[1] Jr. Andrew D. Wolfe. Three Touches of Class. *UnixWorld*, July 1993.

[2] Morris I. Bolsky and David G Korn. *The KornShell – Command and Programming Language*. Prentice Hall, 1988.

[3] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[4] Yih-Farn Chen. The C Program Database and Its Applications. In *Proceedings of the Summer 1989 USENIX Conference*, pages 157–171, June 1989.

[5] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, March 1990.

[6] Yih-Farn Chen, David Rosenblum, and Kiem-Phong Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th Internation Conference on Software Engineering*, May 1994.

[7] Glenn Fowler. cql – A Flat File Database Query Language. In *Proceedings of the USENIX Winter 1994 Conference*, January 1994.

[8] E. R. Gansner, S. C. North, and K. P. Vo. DAG – A Program that Draws Directed Graphs. *Software: Practice and Experience*, 18(11), November 1988.

[9] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, March 1993.

[10] Judith Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems*, 5(1):5–67, 1992.

[11] Judith Grass and Y. F. Chen. The C++ Information Abstractor. In *Proceedings of the Second USENIX C++ Conference*, April 1990.

[12] Rick Greer. Daytona User's Manual, October 1993.

[13] Eleftherios Koutsofios and David Dobkin. Lefty: A two-view editor for technical pictures. In *Proceedings of Graphics Interface*, pages 68–76, May 1991.

[14] Eleftherios Koutsofios and Steve North. Applications of Graph Visualization. In *Proceedings of Graphics Interface*, May 1994.

[15] Steve Manes and Tom Murphy. C++ Development. *Unix Review*, June 1993.

[16] Mark Opperman, Jim Thompson, and Yih-Farn Chen. A Gremlin Tutorial for the SUN Workstation. Technical Report UCB/CSD 322, Computer Science Division, University of California, Berkeley, December 1986.

[17] Tim Parker. Pick a Pack of CASE. *Software Review*, 9(12), December 1991.

[18] G.-C. Roman and K. C. Cox. Program Visualization: The Art of Mapping Programs to Pictures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 412–420, May 1992.

[19] David S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.

[20] Kiem-Phong Vo and Yih-Farn Chen. Incl: A Tool to Analyze Include Files. In *Proceedings of the Summer 1992 USENIX Conference*, pages 199–208, June 1992.