Applications of Graph Visualization

Stephen C. North
Eleftherios Koutsofios
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract

dotty is a customizable graph editor. Its main components are a programmable viewer (*lefty*) and graph layout generators (*dot* and *neato*). *dotty* can run stand-alone, but more importantly, it can be programmed to act as a front-end for other applications. Some interesting examples are *ciao*, a program source code database browser, *vdbx*, a visual extension to dbx for displaying data structures as graphs, and *vpm*, an interactive distributed process monitor.

Keywords: graph browser, visual debugger, process monitor, software information system.

1 Introduction

Graph drawings are one of the best ways to present technical information. Such diagrams are particularly appropriate for showing relations between objects. Finite state machines, networks, program call graphs, and various kinds of object to object dependencies are a few examples of information that can be made easier to understand when presented as graphs.

It is therefore important to have a good set of tools for displaying and manipulating graphs. Such a toolkit should include tools to read and write graphs, to make layouts, and to view and interact with graphs as an interface to other programs. The tools and libraries presented in this paper make up such a toolkit. From the user's point of view, the primary tool is *dotty*. *dotty* can provide high-quality graph layouts and allow the user to operate on them. Figure 1 shows two snapshots of *dotty* in use. Figure 1a shows *dotty* as a stand-alone graph editor. A graph representing an automaton is being edited.

Figure 1b shows *dotty* as a front-end for a process management tool.

dotty can be controlled either through a WYSIWYG interface, or through a textual (procedural) interface. As a stand-alone tool, *dotty* is similar in operation to other systems based on treating pictures of graphs as structured objects. GRAB, EDGE, and GraphEd [24, 22, 11] are some well-known examples. Like these tools, *dotty* provides menu-driven commands for loading or creating graphs, performing editing operations, and saving the changed graphs. Attributed graphs are stored in a data language that is flexible in handling attributes, so new ones can be added to graph files easily without causing incompatibility with existing graph tools.

The procedural interface is convenient for algorithmic operations (*e.g.* set node color as a function of degree). The procedural interface also allows reprogramming the WYSIWYG interface. For example, the left mouse button can be bound to a function that highlights all edges attached to the node under the mouse pointer. The underlying programming language has primitives to start external processes and to establish interprocess communication channels. This makes it possible for *dotty* to operate as a graphical front-end for other processes. In this context, graphs can represent state information maintained by a back-end process, and user actions can be bound to functions that translate graph operations to corresponding state change requests sent to the back-end.

As a front-end, *dotty*'s programming language and the library of functions that accompanies it are higher level than C or C++ graph toolkits. *dotty* has already been used as the front-end for a number of applications:
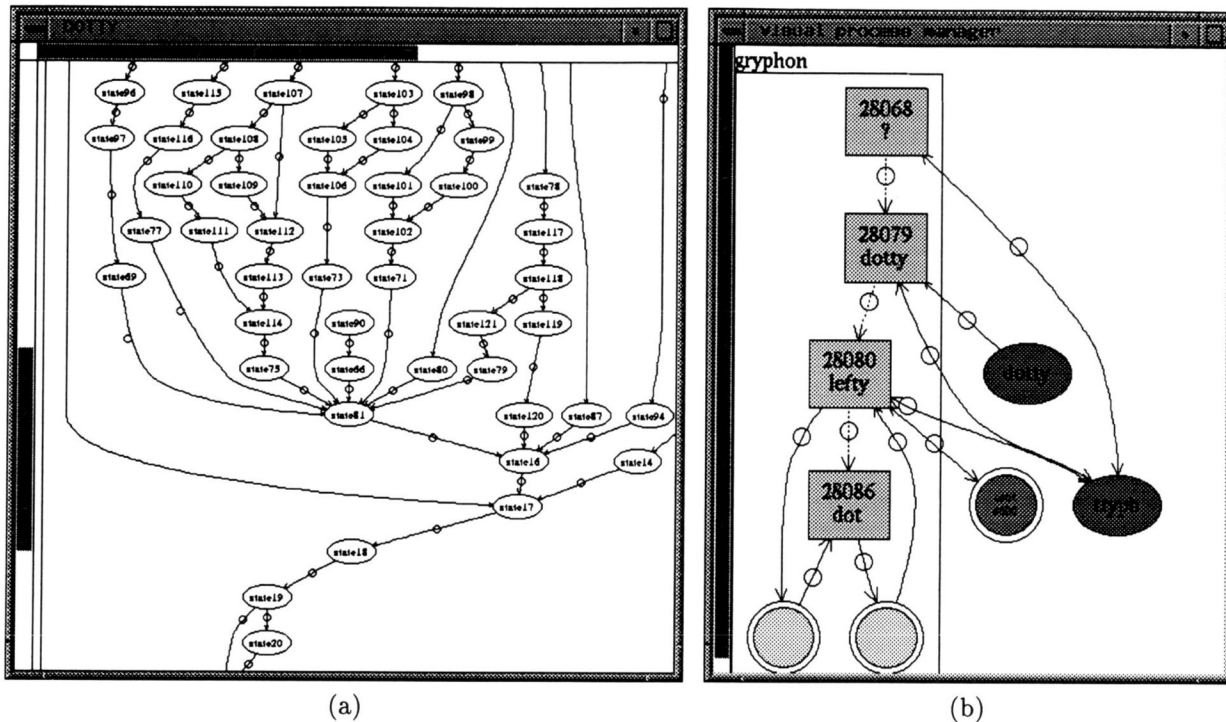
Figure 1: Two snapshots of *dotty* in use

- finite state machine animator
- C/C++ source code database browser
- distributed process monitor
- debugger with graphical data structure displays
- program trace animator
- GUI for the Yeast event-action specification tool [17]
- GUI for the Provence process modeling tool [16]

dotty itself is constructed as two co-operating processes, *dot* and *lefty*. *lefty* is a programmable graphics editor that takes care of displaying the graphs on the screen and allowing the user to operate on them. Thus *dotty*'s programming language is actually that of *lefty*. *lefty* runs *dot* to make graph layouts. These programs communicate via pipes, as shown in Figure 1b.

Having interactive tools like *dotty* is clearly useful. There are, however, many situations where batch tools are also important. Since all our tools process files in our graph data language, it is easy to compose command pipelines that perform graph filtering operations. Often this is easier than using a system in which operations must always be performed using a point and click interface.

In this paper we describe *dotty, lefty, dot, neato,* and several graph filters. We also present our graph language, *libgraph*. Finally, we present some of the applications where *dotty* is used as a front-end.

2 Related Work

Related graph viewing programs include EDGE, GraphEd, daVinci (U. of Bremen), the XmGraph toolkit (Douglas Young, U. of Iowa) and the Graph Layout Toolkit [26]. Like our system, they allow programmers to add application-specific functions, possibly to communicate with external back-end processes. Another related system is Knuth's Stanford GraphBase [14], which is a C library and data sets for implementing and measuring the performance of graph algorithms. Unlike the other packages, GraphBase has no special graph layout or user interface features. We will compare some aspects of these systems with ours.

Architecture. Almost all these systems take the point of view that the user will run most applications "by hand" from a central GUI. In contrast we envision *dotty* as a complement to a collection of non-interactive tools and filters. Often there are engineering advantages to creating a number of smaller tools that each perform one task well, instead of making one large system that combines many di-

verse functions. Also, other systems have insufficient data language support for graph processing applications. Ideally, the data language should be implemented with a clean library interface and separated from the user interface. (EDGE is an exception to this criticism, but only a prototype parser was implemented). Admittedly, a disadvantage of a less-integrated design is that sometimes a feature must be duplicated in several places, or eliminated with a possible loss in functionality. Examples include the functions to draw node shapes, interpret text labels, or fit edge splines.

Programming Language. Almost all the other systems are C or C++ toolkits. To customize the system, an application programmer writes C or C++ code and links it with the main system. *dotty*'s script language is higher-level than C and should be more productive for user-interface programming. Experience suggests this is true with other high-level user-interface languages such as TCL/tk [21]. A disadvantage of our system is that the language is non-standard, but most of its features, such as functions, scalars and associative arrays, hierarchical name spaces, windows, menus, and text-stream I/O are familiar to experienced programmers. Another issue is that C is more general than *dotty*'s script language. For example, it may be easier to add new graphical widget types to an ordinary C program than to an interpreter having its own graphics model.

daVinci is written in an applicative language called ASpecT, and is customized by creating an external process that treats daVinci as an abstract user interface, using an application protocol to communicate about menus, selections, text dialog boxes, etc. There does not seem to be much further support for client-side graph data structures or operations. It would be interesting if ASpecT and its graph library were made available for writing applications.

Portability. *dotty* makes it possible to write very portable applications because *lefty* hides the host graphics and operating system. Application programmers do not deal with the details of fonts, menus, color maps or other low-level features that impair portability. Consequently we can run the same scripts in the Microsoft Windows and UNIX/X11 versions of *dotty*.

Layout Quality and Robustness. *dotty* benefits from using *dot* to make layouts. Considerable effort was spent on developing *dot*'s layout algorithms, and creating a robust and efficient implementation. This is important because making readable layouts automatically, without requiring user intervention, is a central problem in graph visualization. Robustness is also important because graphs from real life can have multiple edges, self-arcs, degenerate components, etc. Some of the other systems provide a much wider variety of layout algorithms (GraphEd 3.0 has 18), but the implementations are often not very robust or practical.

3 Graph Language and Library

Graphs sent between processes or stored in files in our system use a common format with variable (not fixed) attributes. Attributes are essentially property lists attached to graphs, nodes, and edges. This accommodates a variety of applications without forcing them to agree in advance on the set of attributes. To describe structure within graphs, the graph model allows nested subgraphs. Figure 2a shows a sample graph. Basic graph data structures, operations, and file I/O for C and C++ programming are encapsulated in *libgraph* [20].

4 Graph Layout Tools

dot makes hierarchical layouts of directed graphs [8]. It was written as a successor to *dag* [9], which incorporated results of Warfield, Carpano, and Sugiyama *et al* [27, 1, 25]. *dot* makes good layouts and has an assortment of shapes, styles, and colors appropriate for software-related diagrams. For example, *dot* can draw data structure graphs, displaying records as nested box lists, with node ports for connecting pointers. *dot* also incorporates a new algorithm for drawing graphs with *clusters* or recursive node set partitions [19]. Clusters at the same level are drawn in non-overlapping rectangles. This has applications in diagrams of hierarchical structures, such as nested source code modules. Figure 2b shows the *dot* layout for the graph in Figure 2a. *dot* has the ability to emit graphs either in our graph language, or in several graphical languages such as PostScript.

For undirected graphs (such as the computer network graph of Figure 3), hierarchical layouts may not be as informative as other types that emphasize connectivity. *neato* is an undirected graph embedder that uses virtual physical models of Kamada and Kawai [13]. It is compatible with *dot* to the extent of accepting the same input files and command line options. In fact, *dotty* can switch between either of the two using just a simple path name change.

```
digraph G {
  size ="4,4";
  main [shape=box]; /* comment */
  main -> parse [weight=8];
  parse -> execute;
  main -> init [style=dotted];
  main -> cleanup;
  execute -> { make_string; printf}
  init -> make_string;
  edge [color=red];
  main -> printf [
    style=bold,label="100 times"
  ];
  make_string [label="make a\nstring"];
  node [
    shape=box
    style=filled
    color="blue"
  ];
  execute -> compare;
}
```
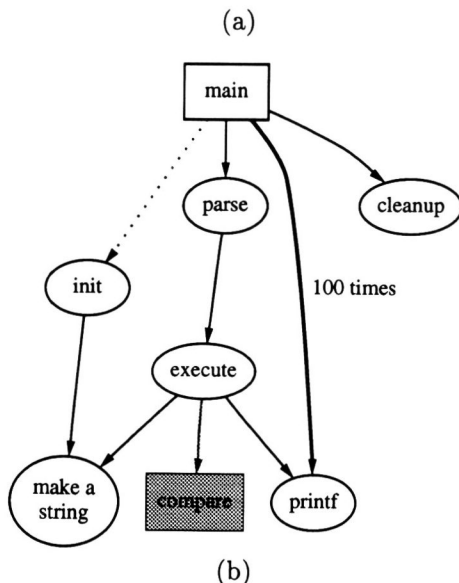
(a)

(b)

Figure 2: A sample graph description and its layout

Figure 3: A sample layout from *neato*

5 Graph Filters

Layouts of large graphs are often complex and difficult to read. Though good layout algorithms help, sometimes a graph is simply too large or dense to understand visually. Appropriate techniques of filtering, partitioning, collapsing, and applying color can often help to convey properties of interest. We have written utilities for some of these operations.
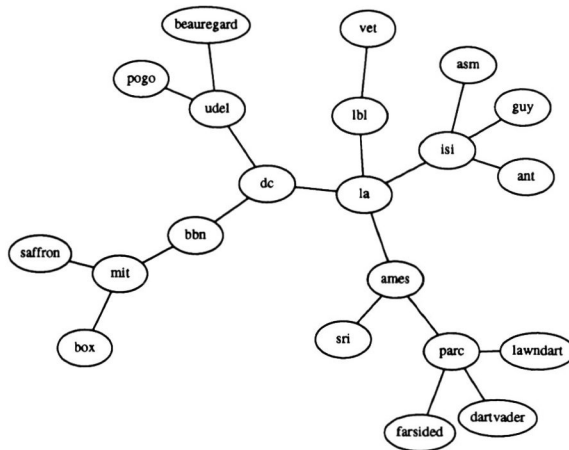
tred computes transitive reductions of directed graphs. When applied to dense graphs, this operation removes many edges without modifying the property of reachability between nodes.

unflatten adjusts lengths of leaf edges or wide fanout fan-in patterns. When applied to bushy graphs, this yields layouts having less extreme aspect ratios.

gpr (for "graph processor") applies a given predicate expression on node or edge attributes to select a subgraph, that is emitted. A command line option enables path contraction on non-selected nodes and edges.

colorize allows setting "seed" colors on some nodes, and propagates colors along edges to help highlight nodes that are logically related, even when dispersed geometrically. This takes advantage of the capability of the human eye to quickly locate similarly-colored objects in a collection.

6 *lefty*

lefty [15] is a two-view graphics editor for technical pictures. This editor has no hardwired knowledge about specific picture layouts or editing operations. Each picture is described by a program that contains functions to draw the picture and functions to perform editing operations that are appropriate for the specific picture. Primitive user actions, like mouse and keyboard events, are also bound to functions in this program. Besides the graphical view of the picture itself, the editor presents a textual view of the program that describes the picture. The language implemented by *lefty* is inspired by the language in the EZ system [7]

Programmability and the two-view interface al-

low the editor to handle a variety of pictures, but are particularly useful for pictures used in technical contexts, e.g., graphs and trees. Also, *lefty* can communicate with other processes. This feature allows it to use existing tools to compute specific picture layouts and allows external processes to use the editor as a front-end to display their data structures graphically.

Figure 4 shows a typical snapshot of *lefty* in use. The editor has been programmed to edit delaunay triangulations. The window on the left shows the actual picture. The user can use the mouse to insert or move cites and the triangulation is kept up to date by the editor (which uses an external process to compute the triangulation). The window on the right shows the program view of the picture.
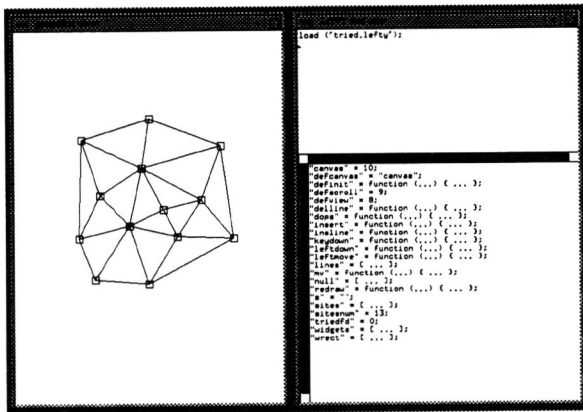

Figure 4: A snapshot of *lefty* in use

7 dotty

dotty is implemented as two cooperating processes (*lefty* and *dot*) and a program in the *lefty* language that customizes *lefty* so that it can handle graphs and their components. The program includes functions to insert and delete nodes and edges, as well as to draw these objects according to attributes such as color, shape, and style. There is also a function that computes the layout. This function sends the graph to a *dot* process running in the background. The *dot* process computes the layout and sends the graph (with layout information inserted as graph attributes) back to the *lefty* process. *lefty* then updates the node and edge coordinates and redraws the graph on the screen. Figure 1b shows how the two processes are connected.

The *lefty* program is organized in two layers. The lower layer (called the dot layer) implements the necessary data structure operations such as inser-

tion and deletion of nodes, edges, and subgraphs. This layer includes functions for reading and writing graphs from files, internet sockets, or UNIX pipes. The functions that perform the actual input and output operations are implemented as C functions and are accessed as *lefty* builtins. This allowed us to use pieces of *libgraph*. The higher layer (called the dotty layer) implements the necessary graphical operations. For example, function dotty.insertnode inserts a new node by calling function dot.insertnode and then drawing the node on the display using the node's color and shape attributes. Figure 5 shows the main parts of these two functions.

```
dot.insertnode =
        function (graph, name, attr) {
    ...
    graph.nodedict[name] = nid;
    graph.nodes[nid] = [
        'nid'   = nid;
        'name'  = name;
        'attr'  = copy (dot.nodeattr);
        'edges' = [];
    ];
    return graph.nodes[nid];
};
dotty.insertnode =
        function (gt, pos, name, attr) {
    ...
    if (~(node = dot.insertnode (gt.graph,
            name, attr)))
        return null;
    node.pos = pos;
    node.size = size;
    dotty.drawnode (gt.views, node);
    return node;
};
```

Figure 5: Functions for inserting a new node

Overall, the *lefty* program implements the following operations:

- create or destroy graphs.

- create or destroy views of graphs (a graph may have several views).

- load or save graphs to files (or sockets and pipes)

- insert or delete nodes, edges, and subgraphs

- pan or zoom within a view
- search for a node by name
- geometrically move a node (and have all its edges follow)
- edit attributes of an object

User actions can be bound to graph operations. For example, pressing the left mouse button can be bound to a function that inserts a new node at the position of the cursor. This can be done by writing a function called `leftdown` that calls `dotty.insertnode` with the appropriate arguments. By default the left mouse button is bound to inserting or moving nodes, the middle button is bound to inserting edges between existing nodes, and the right button brings up a menu for selecting the rest of the operations mentioned in the list above. There are also node- and edge-specific menus.

Because of its design, *dotty* took very little time to build. Though the IPC overhead between *dot* and *lefty* would be avoided if *dot* were called as a library, this cost is small compared to the layout time, and so not noticeable, and we gain the flexibility of having smaller, compatible tools instead of one excessively complicated graph viewing program. Any approach where the graph editor is constructed from scratch (as a single C or C++ program) would take significantly longer to build. It would also be harder to debug and improve. In *dotty*, the individual tools can be tested and fixed independently. Both *lefty* and *dot* can be driven through scripts and this makes testing much easier.

Though the decomposition of *dotty* into separate *lefty* and *dot* processes has advantages, it does also create some limitations. Because the processes communicate at a high level (through the graph language), low-level features, such as typesetting of text labels, formatting of records, and color name lookup need to be replicated. Also, *dot* is a batch program and does not presently handle incremental layout. The disadvantage of this is that layouts of very similar graphs are not necessarily close topologically. We intend to address support of interactive and incremental layout in future versions of *dotty*.

An important feature of *dotty* is that it can be easily customized. Customizing *dotty* amounts to modifying its *lefty* program. For example, the user interface functions (such as `leftdown`) can be redefined to perform different actions. Alternatively, the functions that operate on the graph data structures (such as `dot.insertnode`) could be modified to only

allow operations that are appropriate for a specific type of graph. The most interesting class of customizations is the one where *dotty* is programmed to act as a front-end for another process. In this context, a tool that generates and maintains information that can be expressed as a graph can use *dotty* to display this information graphically. *dotty* provides high quality layouts and a simple way to implement a user interface. Graphs are first class citizens in such an interface; they are used not only as a way to view information, but also as a way to operate on this information. An added advantage of this approach is that it requires little or no change to the original back-end tools.

8 *dotty* applications

8.1 *ciao* — A source code database front-end

ciao is a graphical interface to the *cia* and *cia++* program databases [2, 3]. These are source code analysis tools for large programs written in C or C++ respectively. For simplicity, we will refer to both as *cia*.

cia constructs databases of C or C++ entities, such as files, macros, types (or classes), functions, and global variables. *cia* has text commands to execute database queries and updates. For example, the following command lists all the call sites of a given function.

```
> cref function - function fprintf
k1 file1      name1     k2 file2      name2
== ========= ========= == ========= =======
p  peekf.c   main       p  <libc.a> fprintf
p  xmalloc.c xrealloc   p  <libc.a> fprintf
p  xmalloc.c xmalloc    p  <libc.a> fprintf
```

Some query types generate simple lists of objects. Most of the queries, however, generate lists of tuples. These lists can be represented pictorially as graphs. Each line in the example above can be represented as a graph edge between the two objects in that line. Users find such graphical representations can help to clarify relationships between program entities.

A number of program browsers integrate text views of source code with layouts of procedure and data graphs, and allow navigation between graphs and source files. *cia*, on the other hand, has UNIX text commands instead of such browsing features. Given the size of the programs *cia* is intended to analyze, its authors made a reasonable decision in

emphasizing data-handling capabilities over an interactive interface. In fact, interactive browsers often have efficiency limitations in loading source programs having tens or hundreds of thousands of lines. One reason is that such systems usually keep significantly more detail about individual source statements, down to the token level. The thesis of *cia*'s design is that for large programs, the cost of maintaining this information is not justified.

Although the user interface for *cia* is mostly text-based it does provide tools for converting the output of queries into graphs. The graphs can then be processed using either *dag* or *dot* to compute their layouts. The disadvantage here is that after the layout is made there is no way to navigate between graph views and text views, or otherwise operate on the pictures as structured objects. We felt that we could match many of the advantages of the integrated systems without incurring their cost or complexity by creating an interface with *dotty*.

The main customization to *dotty* was the specification of a table that provides a mapping between node types (which correspond to *cia* entities) and operations appropriate for each type. (The *cia* tool that generates the graphs includes the entity type of each node as an attribute of that node). The user interface functions were then modified so that when the user tries to bring up a menu over a graph node, the menu that appears contains exactly those operations that are appropriate. Some menu selections result in new graphs that are displayed in separate windows, while other selections produce text output that is appended to a *message* window. There is also a global menu that can be used to generate complete graphs, such as the full function to function reference graph or the file to file inclusion graph.

To generate one of these graphs, *dotty* composes a UNIX command pipeline. The first part of the pipeline is the appropriate *cia* query. The second part of the pipeline is the *cia* tool that generates graphs from query output. *dotty* then runs this command and collects the output (which is a graph). Big graphs, such as the full function to function graphs, can also be cached by storing them in files.

ciao consists of approximately 500 lines of *lefty* code. No changes were made to any of the *cia* tools and it took only a couple days to put a prototype together. Figure 6 shows a snapshot of a *ciao* session. It shows several windows, each containing a different graph. Some show full graphs while others were created by selecting an object on a full graph and performing a *cia* query.
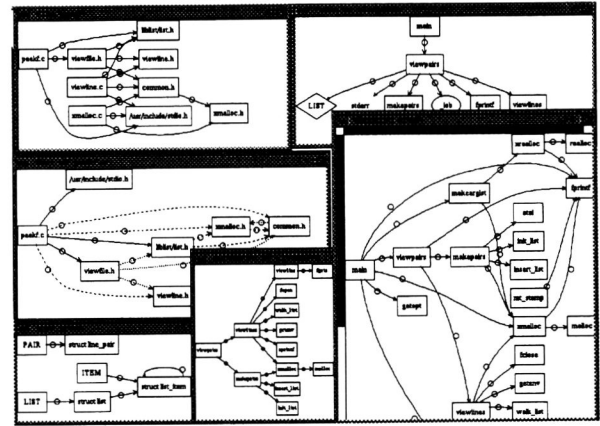

Figure 6: A snapshot of *ciao* in use

8.2 *vdbx* — A graphical debugger

Although graphical users interfaces are commonplace, debuggers lack useful graphical data structure displays. Several previous efforts have involved displays of lists or simple graphs [12] but lack generality. A great deal of other work has concentrated on algorithm animation [18, 23]. Generally this relies on adding extra code to abstract data types, such as arrays, stacks, lists, queues, and geometric data structures, to create on-line graphical displays, or at least emit traces for subsequent post-processing and animation. This is quite general, and well designed algorithm animations can be highly informative. These techniques, however, are not directly applicable to general-purpose debugging using pictures; they are usually intrusive and the pictures often do not map well to the lower-level or concrete views needed in practical situations.

vdbx nonintrusively adds graphical data structure displays to dbx. Neither dbx nor the target program are modified. *vdbx* uses *dotty* to draw data structures. *dot* already has good capabilities in this area, so the main issues are handling communication between *dotty* and the debugger. A multiplexing process is used to link *dotty* to the debugger. This multiplexor contains a parser to translate dbx-style output into graphs. It also allows the debugger to be accessed from a virtual terminal window (shown in Figure 7).

vdbx is compatible with any debugger that uses the dbx syntax for C structs. The user can access dbx through the virtual terminal window just as if dbx was being run directly. The user can set breakpoints, inspect values, and give other debugging commands in the usual way.
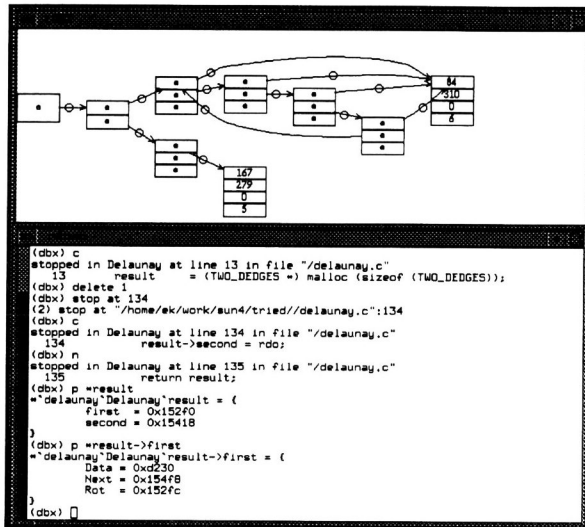
```
(dbx) c
stopped in Delaunay at line 13 in file "/delaunay.c"
    13      result    = (TWO_DEDGES *) malloc (sizeof (TWO_DEDGES));
(dbx) delete 1
(dbx) stop at 134
(2) stop at "/home/ek/work/sun4/tried//delaunay.c":134
(dbx) c
stopped in Delaunay at line 134 in file "/delaunay.c"
    134         result->second = rdo;
(dbx) n
stopped in Delaunay at line 135 in file "/delaunay.c"
    135         return result;
(dbx) p *result
*'delaunay'Delaunay'result = (
    first  = 0x152f0
    second = 0x15418
)
(dbx) p *result->first
*'delaunay'Delaunay'result->first = (
    Data = 0xd230
    Next = 0x154f8
    Rot  = 0x152fc
)
(dbx) []
```

Figure 7: A snapshot of *vdbx* in use

At the same time, the user can issue graphical queries in the *dotty* window. There are two kinds of queries. One kind prompts the user for the name of a variable, and displays it as a new root node. Non-null pointer fields are marked by a special symbol, unexpanded. The other kind of query follows an unexpanded pointer. When executing either case, *dotty* translates the graphical query to a text command that is sent to the multiplexor. The multiplexor communicates with dbx to augment the data structure graph. The new graph is returned to *dotty*, drawn by *dot*, and displayed. This communication takes place invisibly to the user. The graphical interface is not only easier to use than dbx text queries, but it also makes it easier to understand how the various data structures are inter-connected.

We have found *vdbx* to be practical for visualizing data structures while debugging. For example, one of our colleagues found it useful for debugging a program with geometric data structures, which tend to form complex pointer inter-connections. *vdbx* shows promise, but its user interface needs refinement before it is practical for everyday use. One issue is that in practice, records often have many fields. Displaying them all wastes valuable screen area and can overwhelm the user with irrelevant information. Programmers are usually interested in only a few fields during a given debugging run, so only these should be drawn. Another issue involves displays of dynamic arrays and unions in C. Because programmers must define their own runtime conventions to store array sizes or define which members of

unions are in use, this information is not automatically available to dbx for generating data structure graphs. One solution would be for programmers to advise *vdbx* of such conventions, using a debugging language extension kept in a startup file [10]. Another limitation is that our data structure graphs are static snapshots, and are not automatically updated as a program runs and modifies its data. To have automatic updates, the debugger needs to track changes in the data structure. Typically this is done with watchpoints, but they are expensive on most computers. A related issue is that we need stable incremental layouts from *dot*.

8.3   *vpm* — Visual Process Manager

Operating systems such as UNIX make it easy for users to create multiple processes and have them cooperate by exchanging messages via pipes or sockets. If tools have good interfaces, then they can often be combined to create powerful new applications, achieving a high degree of software re-use at the process level. Success depends on two factors: a good selection of components, and powerful support tools, such as debuggers, program analysis tools, and facilities to trace processes. Generally in the second area, most existing debugging techniques break down when there are multiple distributed processes.

vpm was designed to monitor and debug communicating processes in a distributed system. It creates a graphical view of their interactions. Figure 8 shows a sample snapshot. It is, in fact, a snapshot of *vpm* itself. Rectangles represent processes, while ellipses represent files. Double circles represent either pipes or sockets. The bigger rectangle in this figure represents a host computer; the processes enclosed in this rectangle are the processes running on that machine. *vpm* can monitor activity across machines. Figure 9 shows a snapshot of using *nmake*[4] and *coshell*[5] to run several compilations over a network of machines. The processes of each machine are grouped together into *dot* clusters so that they appear next to each other in the layout.

vpm uses *nDFS*[6] to monitor process activity at the system call level. *nDFS* stands for multidimensional file system. Its features include the ability to attach servers at various points in the file system. Accessing a file under such an attachment point results in making a service request to the attached server. Besides servers, one can also attach monitoring processes. *vpm* is implemented by instructing *nDFS* to monitor process activities.
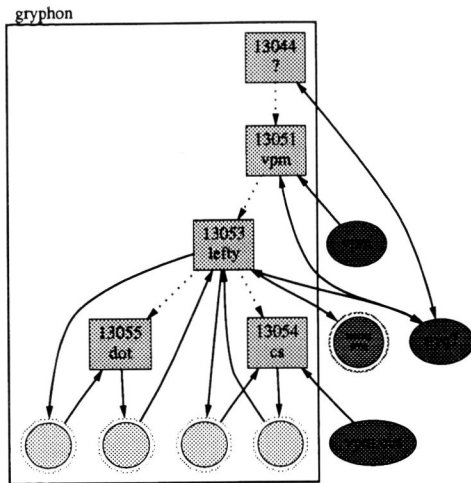
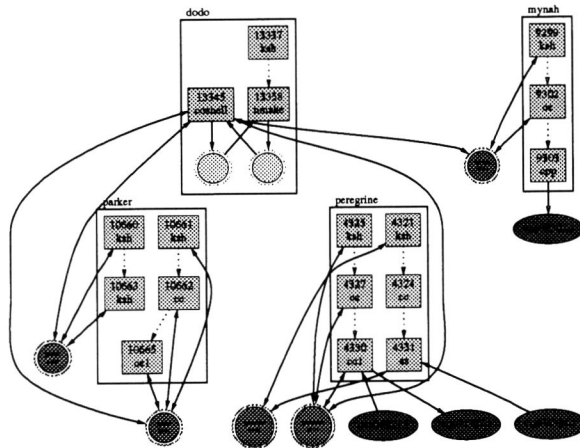Figure 8: Monitoring processes on a single machine

Figure 9: Monitoring processes on a network of machines

nDFS can selectively intercept several kinds of system calls. Currently, *vpm* monitors the following classes of system calls.

- process management system calls, e.g. `fork`, `exec`, and `exit`. In Figure 8, when process vpm forks and creates process 13053, this is shown by creating a new node (for 13053) and linking that node to all the files that the parent had open. In this case, 13053 is linked to `ttyq7`. When process 13053 execs a program called `lefty`, the name of 13053 changes to `lefty` (it was vpm up to that point).

- I/O channel management system calls, e.g. `open`, `close`, `dup`, and `pipe`. Network connections are modeled using `open` and `close` calls.

```
[13051 3750] fork()=13053
[13053 3750] fork()=13051
[13053 3750] exec("/home/ek/bin/lefty")=0
[13053 3750] close(1162+295 0x2bafb)=0
```

Figure 10: *vpm* trace sequence

For example, the four nodes at the bottom of Figure 8 are UNIX pipes. They were created by process `lefty`. `lefty` first created two pipes, then it `forked` and `execed` process `dot`, which inherited these pipes. `lefty` then created the other two pipes and used them to connect to process `cs` (which is used in *vpm* to collect the *nDFS* monitor output). The arrows in process to file edges show whether the files are opened for read or write or both.

- I/O operations, e.g. `read` and `write`.

Which system calls are intercepted can be defined on a per-process level; each new process inherits its parent's mask, but this mask can be changed during the lifetime of the process.

Figure 10 shows an excerpt from the monitoring information generated by *vpm*. It corresponds to having process vpm in Figure 8 spawn a process that eventually runs program `lefty`. When the vpm process performs a `fork` we get two fork messages; one from the parent and one from the child. The first number in brackets is the pid of the process performing the action. The second number identifies the computer where the process runs. The final `close` happens because process vpm had one of its file descriptors marked as close-on-exec.

vpm can work either in real time, or in single step. In real time, *vpm* reads the activity log as it is being generated and updates its graphical representation periodically. Some amount of control over processes is available to the user; the user can send signals to individual processes, start up and attach a source level debugger on them, and hide processes or files that the user considers unimportant. We are currently working on adding more functionality to *vpm*. One important feature is to allow the user to put breakpoints, so that when a process executes a specific system call it stops and waits for user acknowledgment before proceeding. In single step, *vpm* reads an already generated log and updates its graphical view after every step (or after a sequence of steps, depending on user interactions). The real

time view is good for seeing things as they happen. The single step mode provides a more detailed view and is better suited for debugging.

An important feature of *vpm* is that it requires no changes to the applications to allow them to be monitored. On systems that provide shared libraries, the *nDFS* shared library is prepended in the list of libraries to load when starting a program. (On systems without shared libraries, applications have to be linked with the static version of the *nDFS* library).

The real-time performance of the system is noticeably limited by *dotty*'s throughput. When new processes or files are inserted in the graph, *dot* has to re-layout the whole graph. The need for increased throughput suggests work on incremental layout algorithms. We need such algorithms to improve layout stability, but we are cautious in viewing this as a solution to throughput problems, because maintaining stability in incremental layouts may well incur additional overhead. It does seem likely, though, that further tuning of the cluster layout code will yield significant improvements.

9 Conclusions

We have described how new software visualization applications have been created from smaller, general-purpose tools: a programmable 2-D graphics editor, graph layout tools, and text-based applications. An important practical aspect of these systems is that they are unintrusive. *vpm* and *vdbx*, for example, work with unmodified programs. The ease with which they were created suggests that small, well-focussed tools can be a better starting point than large C or C++ libraries.

We are currently working on incremental graph layouts. Our main goal is to provide layout stability: if a small, localized change is made to a graph, the layout of the graph should also change only slightly. Our current techniques will sometimes result in major changes to the layouts and this is disorienting to users. At the same time, we want the quality of incremental layouts to be as good as the batch layouts, making this problem very challenging.

References

[1] M. Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Software Engineering*, SE-12(4):538–546, Apr. 1980.

[2] Y.-F. Chen. The C Program Database and Its Applications. In *USENIX Baltimore 1989 Summer Conference Proceedings*, pages 157–171, 1989.

[3] Y.-F. Chen and J. Grass. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, pages 265–278, 1990.

[4] G. Fowler. The fourth generation make. In *USENIX Portland 1985 Summer Conference Proceedings*, pages 159–174, 1985.

[5] G. Fowler. The shell as a service. In *USENIX Cincinatti 1993 Summer Conference Proceedings*, pages 267–278, 1993.

[6] G. Fowler, Y. Huang, D. Korn, and H. Rao. A user-level replicated file system. In *USENIX Cincinatti 1993 Summer Conference Proceedings*, pages 279–290, 1993.

[7] C. W. Fraser and D. R. Hanson. High-level language facilities for low-level services. In *12th ACM Symp. on Prin. of Programming Languages*, pages 217–224, 1985.

[8] E. Gansner, E. Koutsofios, S. North, and K. Vo. A technique for drawing directed graphs. *IEEE-TSE*, Mar. 1993.

[9] E. R. Gansner, S. C. North, and K. P. Vo. DAG—A program that draws directed graphs. *Software—Practice and Experience*, 18(11):1047–1062, Nov. 1988.

[10] M. Golan and D. R. Hanson. Duel - a very high-level debugging language. In *Winter USENIX Conference Proceedings*, pages 107–117, 1993.

[11] M. Himsolt. Graphed: An interactive graph editor. In *Proc. STACS 89*, volume 349 of *Lecture Notes in Computer Science*, pages 532–533, Berlin, 1989. Springer-Verlag.

[12] S. Isoda, T. Shimomura, and Y. Ono. VIPS: A Visual Debugger. *IEEE Software*, pages 8–19, May 1987.

[13] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, Apr. 1989.

[14] D. E. Knuth. *The Stanford Graphbase: a platform for combinatorial computing*. Addison-Wesley, 1993.

[15] E. Koutsofios and D. Dobkin. Lefty: A two-view editor for technical pictures. In *Graphics Interface '91, Calgary, Alberta*, pages 68–76, 1991.

[16] B. Krishnamurthy and N. Barghouti. Provence: A Process Visualization and Enactment Environment. In *Proceedings of the Fourth European Conference on Software Engineering*, pages 151–160, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag. Published as *Lecture Notes in Computer Science* no. 717.

[17] B. Krishnamurthy and D. S. Rosenblum. An event-action model of computer-supported cooperative work: Design and implementation. In K. Gorling and C. Sattler, editors, *Proceedings of the International Workshop on Computer Supported Cooperative Work*, pages 132–145. IFIP TC 6/WG C.5, 1991.

[18] M. Lee. An algorithm animation programming environment. In *Lecture Notes in Computer Science*, volume 602, Berlin, 1992. Springer-Verlag.

[19] S. C. North. Drawing ranked digraphs with recursive clusters. In *Proc. ALCOM International Workshop PARIS 1993 on Graph Drawing and Topological Graph Algorithms*, 1993. ftp /pub/papers/compgeo/gd93-v2.tex.Z from wilma.cs.brown.edu.

[20] S. C. North and K.-P. Vo. Dictionary and graph libraries. In *USENIX Winter Conference*, pages 1–11, 1993.

[21] J. K. Ousterhout. An x11 toolkit based on the tcl language. In *USENIX Winter Conference*, Jan 1991.

[22] F. N. Paulish and W. Tichy. Edge: An extendible graph editor. *Software–Practice and Experience*, 20(S1):1/63–S1/88, 1990.

[23] S. P. Reiss. Program visualization: Where we go from here. In *IFIP Tran A*, volume V12, pages 218–227, 1992.

[24] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software–Practice and Experience*, 17(1):61–76, Jan. 1987.

[25] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, Feb. 1981.

[26] Tom Sawyer Software, 1824B Fourth Street, Berkeley, CA 94710. *Graph Layout Toolkit*, 1.08 edition.

[27] J. Warfield. Crossing Theory and Hierarchy Mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-7(7):505–523, July 1977.

Author Index