

ALMOST: Exploring Program Traces

Manos Renieris and Steven P. Reiss

{er,spr}@cs.brown.edu

October 30, 1999

Abstract

We built a tool to visualize and explore program execution traces. Our goal was to help programmers without any prior knowledge of a program, quickly get enough knowledge about its structure so that they can make small to medium changes. In the process, a number of problems were faced and tackled concerning the efficient use of screen space, interaction with multiple concurrent views, and linking of asymmetric views.

1 Overview

When a programmer is trying to understand a program code, execution traces can be a useful source of information. However, in textual form it is almost impossible to read any but the most trivial traces; there is a need to visualize them. As a domain of data to be visualized, traces are quite specific: they are time series, they have no noise (all the data is significant), and they come from a process that is available (we always have the program, even if in machine code) and deterministic (modulo input, output, timing, and resource allocation, which can also be modeled in a deterministic way [1]). An effective trace visualization will represent data in an intuitive way, directly showing their sequential and nested nature.

Traces, though, are only one of the things we are interested in when we are trying to understand a program. Understanding involves not only the flow of control, but also memory utilization, input and output operations, user interaction, and, most importantly, the connection of all these with the actual code. A useful visualization tool will not only display the information, but will also synchronize the views and allow the user to interact with them, so that he can browse the execution and get back to the code that is executed.

We present our first attempt to visualize trace data for exploration purposes. The rest of the paper is structured as follows. Section 2 motivates our research. Section 3 briefly talks about our data collection mechanism. Section 4 presents our main visualization of the data, and Section 5 our modifications to it to better utilize the screen. Section 6 addresses the problem of having multiple connected views on the screen, while Section 7 describes how this is expanded

to allow for non-homogeneous views. Section 8 discusses what we see as the next step in trace data exploration. Section 9 describes related work, and Section 10 lists the problems we think our future research should tackle.

2 Motivation

Coding is essentially an effort in spanning two domains: code and program behavior. When a programmer writes code, he is actually thinking not of the code itself, but of its execution behavior. When a programmer is debugging, he is first trying to get the reverse mapping: from undesired behavior to the portion of code that causes it. Then he is thinking again in terms of the first mapping: what the code must look like to evoke the desired behavior.

Sometimes coders will have to make small changes to the behavior without having any knowledge of which part of the code does what. Notice that the programmer only needs to know which part of the code implements the portion of behavior that he wishes to change. Since this is not available, the programmer needs to start exploring the direct mapping, from code to behavior, by reading and understanding the code. Sometimes, he will only discover the “guilty” code when this direct mapping is complete: when he has read and understood all the code.

We therefore see the need for a tool that allows the programmer to quickly acquire some knowledge about the behavior, so that he needs to understand the direct mapping of only a small part of the code. Today’s tools cannot, in general, help in this because they all require some prior familiarity with the code. A debugger, a program slicer or a static code analyzer will not do, because the user interacts with them in the domain of code, which we wish to avoid as much as possible. A profiler, though it allows some insight into the causality between code and behavior, does so only at the aggregate level. What we really need is some way of finding out “what happened when” in the program, like an *a posteriori* interaction diagram.

One possible solution is to collect trace information as the program runs then read it, along with the program text. Unfortunately, program traces are too long to be read, and not very easy to understand. We present the first version of a tool, ALMOST, that visualizes execution traces of C programs. Our ambition is that it will eventually provide enough information to guide a programmer to the right lines of code.

3 Collecting Data

The data was collected with a home-grown tool [9], which works in three steps: The first step creates a patched version of the executable to be traced. The second step executes the patched program, recording calls and instruction counts at call points. The third step queries the trace file, giving legible information. For example, tracing the program in Figure 1 gave us the data in Table 1.

The first three columns specify the calling function, the next three columns the called function, the seventh column specifies the level at which the call occurred, and the rest of the columns give time information. Time is always given in instruction counts.

4 Linear Visualization, Parameters, and Queries

We start by visualizing function calls. The horizontal axis on our plane represents time (which increases from left to right) and the vertical the depth of the function call. A single function activation is visualized by a horizontal bar that extends between the starting and the ending time of that function. The vertical offset of the bar corresponds to its call level. The color of the bar represents the called function, and the name of the function is written in the bar. Figure 2 shows the visualization of the whole execution of the example program. This kind of visualization is particularly intuitive, to the point that it has been used in introductory programming classes. Ullman’s books on ML have a similar drawing (section 3.2.2, in the second edition) to help explain recursion.

There are a number of variables that can be dynamically altered to explore the trace data. The user can zoom in and out, for viewing at different levels of detail. For the mapping of functions to colors, we sort the functions and then assign them equispaced colors on six edges of the RGB cube, going from red to purple. The sorting key can be altered at run time, and can be one of the following: the function’s name, the name together with the file and the function’s position in it, the time of the first invocation, or the time of the last invocation.

We also implemented a proof-of-concept subsystem that allows limited queries on the data, and grays out the portions that do not pass the filters. The filters include one for each of the attributes of the activations: the total time spent in the activation, the function name, or the calling function name.

5 The Spiral View

The main drawback of the linear view is that the screen tends to fill up very quickly: see Figure 3. The key observation is that we are fully utilizing only the horizontal dimension of the screen. We need a mapping of time to two dimensions, that will fill the screen, but will be continuous and easy to follow with the eye. A spiral ([13], [5]) fills these requirements quite well: it fills its bounding rectangle well, and time grows in a single direction (of increasing radius). The result of wrapping the linear view around itself into a spiral is shown in Figure 4. We are not actually drawing a real spiral, but a sequence of aligned semicircles, a well-known geometrical trick.

The spiral has some drawbacks, however. There is no intuitive way of zooming in and out. If we zoom in on the conceptual level and restrict our time window but map it on a new spiral, then we are not zooming visually and we

are disrupting the correspondence between the picture and the data. If we zoom in visually, we will get an arc, losing the benefit of using a spiral in the first place. We cannot display the names of the functions on the graph, because they would be upside down or curved. None of these were a problem with the linear view. We need some way of regaining these features. One solution to this problem is having both the spiral and the linear views on the screen concurrently. This creates a need to coordinate views.

6 Connecting Views

To achieve the best of both worlds, we allow multiple views on the screen concurrently. We need now a way to coordinate them, so that the time windows in separate views are centered around the same instance. In a sense, we need to establish a model-view-controller (MVC) relationship between our views. Since coordination is by *time* level, our internal data structure (the *model*) includes a field that says around which time point the current view is centered.

However, we want to keep the interface direct: the user should be able to interact directly with the picture, to get where he wants, instead of typing or otherwise interacting with some other window. Therefore, our MVC has to allow every view to be a controller. Conveniently, with current workstation environments, the user interacts with only one window at anyone time. Therefore, what we need to do is designate as the current controller, the window that has the user focus at the moment. The rest of the views will follow, centering themselves around the same point in time. In Figure 5, for example, the user has linked a spiral and a linear view.

During our initial experiments we found out that the user should be able to specify dynamically which views are connected, so that we could have different views of the same kind showing different parts of the execution history. This means that views should have names; we use country names as defaults. We also found that automatic coordination was more distracting than useful. If, for example, the user scrolls one view, the views linked with it should not scroll automatically. The user has to explicitly request refocusing of the other views by clicking in the current window.

Connecting views in this way allows for some more interesting interaction scenarios: The user can bring multiple views on the screen and link them, then change their parameters. We can have multiple views with different zoom factors. We can have two views with showing the same time window, but with different color codings, or the same color coding (color coding can also be linked) with different time windows (fig. 6). In these cases, the user is performing a visual *join* on the data, mentally matching things that are at the same coordinates or have the same color.

7 Linking Back to the Code

Our original goal was to find some way of tying the execution history back to the code. To this end, we first implemented a SeeSoft-like [3] view. A SeeSoft view is a view of the code text, where every character (or line) is mapped to a single pixel. From there, the user can (again with a single mouse click) bring up an emacs window with the cursor at the position in the source. The colors of functions on the SeeSoft view maintain the same options as in the other views, which means that we have the same ability to perform visual joins on the data. Also, this view allows for one of the orderings (by name and position in the file) to become much more meaningful and intuitive: it provides semantics for the function calls that are directly derived from the code, so that for example, the red functions correspond to code in the first few files. This can be easily expanded to allow for code-coloring with all the original SeeSoft metrics, which would enable to see, for example, in the execution history how much of, for example, new or old code is actually executed. Figure 7 show all four of our views together on the screen and linked.

However, there is one question of what it means for a code view to follow a trace view: the code view represents static information, and therefore we cannot map our original notion of focus, which included only time. We devise two solutions for that: one involves augmenting the code view, and the other augmenting our *model* internal structure.

In the first solution we change the code view so that it displays, not only the code, but also the call stack. Every active function call is represented by a line that connects the caller with the callee, and the thickness of the line increases as the call level deepens. The second solution involves augmenting our *model* with a stack level. Hence, when the user clicks on some point on a trace view, he does not click on just a point in time but also on a specific call level. The corresponding function gets highlighted by a frame. Notice that none of the solutions can reintroduce the lost parameter of time: if the user clicks on a code view, there is no unique corresponding spot on the trace views to focus.

These two solutions differ radically in their approach, and have different implications for the corresponding more general cases. Following the paradigm of the first solution, we have to make all views equivalent; we are trying to show as much information as possible on each of them, even if there is not a natural way to do so. This is not always beneficial. Our code view, when augmented with a lot of lines has no most salient point, no one point of focus. The ambiguity in this solution is evident in the views, and therefore imposed on the user. The second solution goes in the opposite direction: eliminate the ambiguity by making the *model* as specific as possible, then allow each view to depict as much of it as is appropriate for the view. This however, breaks the symmetry of the views, as each of them shows different slices of information. It also makes linking more awkward, because many links become impossible: there are no 1 – 1 mappings between a lot of views.

8 Visual Data Mining

In Figure 5, functions are colored according to the time of their first activation. This allows us to see an initialization phase where the colors are close to red. We can also make out loops and repetitions, even sequences of function calls that were approximately the same only at the deeper levels (they are memory requests that were actually forwarded to the operating system). There is a trade-off between the smallest thing that can be seen and the size of the time window. If the trace of Figure 5 was longer and the memory allocation sequences far apart, by the time they would be together on the screen they would not be visible.

Figure 8 is an example that really stresses the limits of what can actually be shown on a single screen. The yellow triangle on this picture needs some explanation: it is 900 levels of recursion of a single function. This fills the screen vertically, (which has height slightly more than a thousand pixels). However, this was one of the most interesting pictures produced; it raised the question of why the right edge of the triangle is so close to being vertical. Pure inspection of some seconds allowed us to foresee that this means that the function is tail-recursive, and therefore a good candidate for optimization. The conclusion was verified easily by bringing up an editor window. The important point here is that looking at the picture allowed us to discover things about a piece of code we had never seen before. It pointed to ten specific lines in the code (about 800 lines in total) that were significant. What we want to do is first define a number of such interesting features, then equip the user with tools that can expose them even in programs a thousand or ten thousand times as large. The whole process here would be very similar to that of knowledge discovery in databases [4], which resembles to the usual processes of debugging and understanding code.

9 Related Work

The work closest in spirit to ours is that by Lieberman and Fry [8]. They are not using traces but reverse execution, which is feasible because they are working with a subset of LISP. A lot of work with visualization of long sequences has been done by Jerding, Stasko, and Ball ([6],[7]). Among other things, they deal with visualization of traces with more or less the same goals as ours, and they develop rendering techniques to accommodate the large amounts of data. Their visualizations are stand-alone however, and there is no notion of integration between them. Technically, we are closest to JINSIGHT from IBM research [2]. They build a number of visualizations (our Linear View could be regarded as “spreading out” their Stack View; it looks like their Execution View, but we are more space efficient), but they do not concern themselves with screen space limitations except for scaling. They also do not integrate or coordinate their views, and although their goals are very similar to ours they do not provide a link with an editor.

10 Future Work

We still need to provide views of memory utilization and input/output, perhaps similar to our older work on the theme [11]. We need to work on coordinating them with our execution and code views.

The first thing that became obvious while we were building our system was that the size of program traces becomes prohibitively large very quickly. We need to find ways to manage them. Most of the possible directions address at least one other problem at the same time. They include collecting snapshots at specific intervals and enough data to replay the program from there, compressing the data as they are getting collected, compressing the tuples column-wise (regarding each column as a regular time sequence and compressing it), and using the static call graph as a dictionary for encoding.

The other thing we realized was that visualizing the raw data is, most probably, a futile process. A regular, high end computer screen has a resolution of about a million pixels, and three bytes of information per pixel. That is, on a screen, we can see about three megabytes of raw data, in the best case. Therefore, we need to somehow process and summarize the data, allowing overviews not at the visual, but at the conceptual level.

A whole new set of problems is introduced in handling Java, instead of C. Java is object-oriented and multithreaded at the same time; this introduces two more free variables, which will have to be handled. It is possible that we could retreat to 3-D visualizations ([10]) to accommodate one of them, but 3-D pictures are much harder to comprehend, since they require manipulation to reveal hidden surfaces. Nonetheless, threads and classes and objects provide good ways to group subtraces, since they are of a higher conceptual level than mere functions. Software designers already exploit this when they are going from code to behavior; we plan to do the same for the reverse process.

References

- [1] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, 1998.
- [2] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 1999.
- [3] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft — a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, pages 957–968, Nov. 1992.
- [4] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, Nov. 1996.

- [5] C. L. Jeffery. A Menagerie of Program Visualization Techniques. In Stasko et al. [12], chapter 6, pages 73–79.
- [6] D. F. Jerding and J. T. Stasko. The information mural: Increasing information bandwidth in visualizations. Technical Report 97-24, Georgia Institute of Technology. Graphics, Visualization and Usability Center, 1997.
- [7] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 1997 International Conference of Software Engineering*, Boston, MA, USA, May 1997.
- [8] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In I. R. Katz, R. Mack, L. Marks, M. B. Rosson, and J. Nielsen, editors, *Proceedings of the Conference on Human Factors in Computing Systems (CHI'95)*, pages 480–486, New York, NY, USA, May 1995. ACM Press.
- [9] S. P. Reiss. Trace-based debugging. In P. Fritzson, editor, *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, volume 749 of *Lecture Notes in Computer Science*, pages 305–314. Springer, 3–5 May 1993.
- [10] S. P. Reiss. Software visualization in the Desert environment. *ACM SIGPLAN Notices*, 33(7):59–66, July 1998.
- [11] S. P. Reiss. Visualization for Software Engineering — Programming Environments. In Stasko et al. [12], chapter 18, pages 259–276.
- [12] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. M.I.T. Press, Feb. 1998.
- [13] E. Tick and D.-Y. Park. Kaleidoscope visualization of fine-grain parallel programs. Technical Report CIS-TR-91-18, University of Oregon, Department of Computer and Information Science, Oct. 1991.

From function	From file	From line	To function	To file	To line	At level	At time	Return time	Local time	Total time
_start	*	*	main	main.c	19	2	7294	7288	0	0
main	main.c	19	foo	main.c	14	3	7297	0	0	0
foo	main.c	14	trivial	main.c	3	4	7300	0	0	0
foo	main.c	14	trivial	main.c	3	4	7300	7305	5	5
main	main.c	19	foo	main.c	14	3	7297	7309	7	12
main	main.c	20	bar	main.c	7	3	7311	7309	0	0
bar	main.c	7	trivial	main.c	3	4	7314	*	0	0
bar	main.c	7	trivial	main.c	3	4	7314	7319	5	5
bar	main.c	8	trivial	main.c	3	4	7321	7319	0	0
bar	main.c	8	trivial	main.c	3	4	7321	7326	5	5
bar	main.c	9	trivial	main.c	3	4	7328	7326	0	0
bar	main.c	9	trivial	main.c	3	4	7328	7333	5	5
main	main.c	20	bar	main.c	7	3	7311	7337	11	26
main	main.c	21	foo	main.c	14	3	7339	7337	0	0
foo	main.c	14	trivial	main.c	3	4	7342	*	0	0
foo	main.c	14	trivial	main.c	3	4	7342	7347	5	5
main	main.c	21	foo	main.c	14	3	7339	7351	7	12
_start	*	*	main	main.c	19	2	7294	7355	11	61
_start	*	*	exit	*	*	2	7357	7355	0	0
exit	*	*	_exithandle	*	*	3	7360	*	0	0
_exithandle	*	*	mutex_lock	*	*	4	7368	0	0	0

Table 1: Trace data from the example program

```
int trivial()
{
}

int bar()
{
    trivial();
    trivial();
    trivial();
}

int foo()
{
    trivial();
}

int main()
{
    foo();
    bar();
    foo();
}
```

Figure 1: Example program

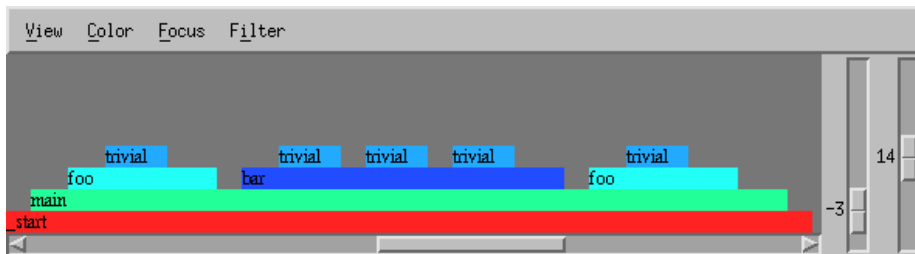


Figure 2: Linear view of the example program

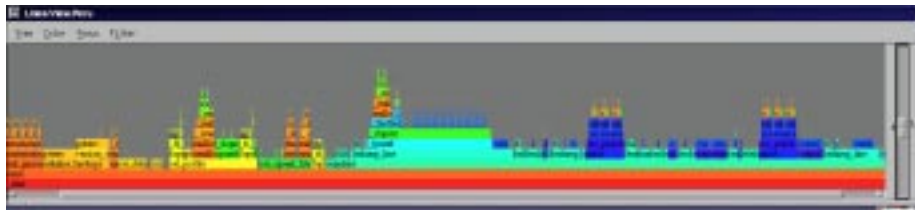


Figure 3: Linear view of a larger trace. The screen is crammed.

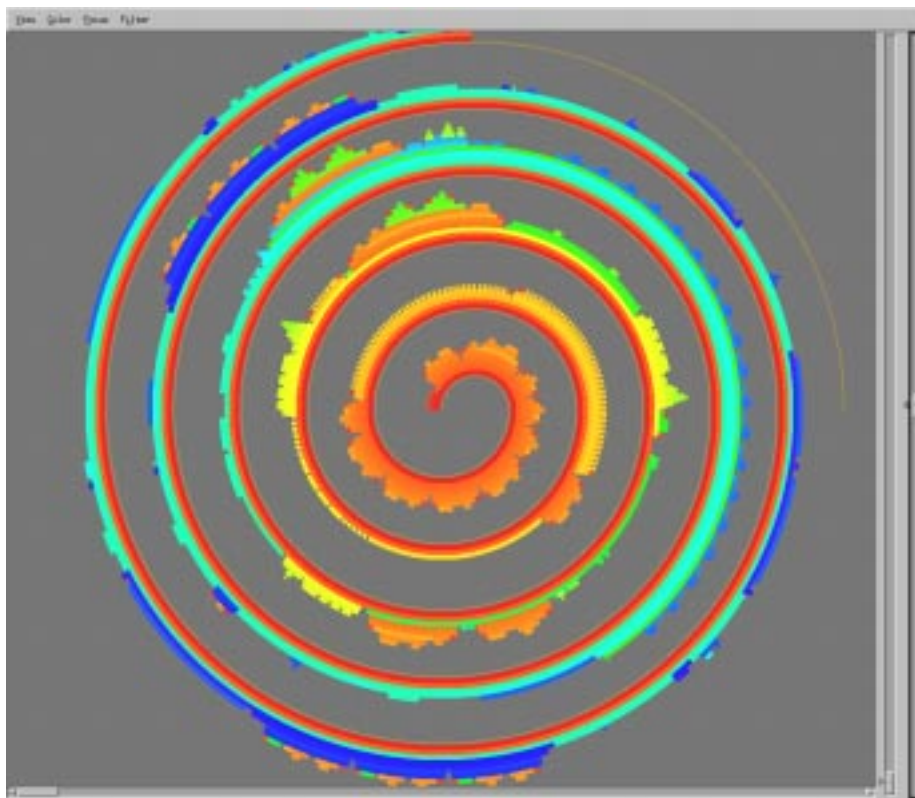


Figure 4: Spiral view of the same trace

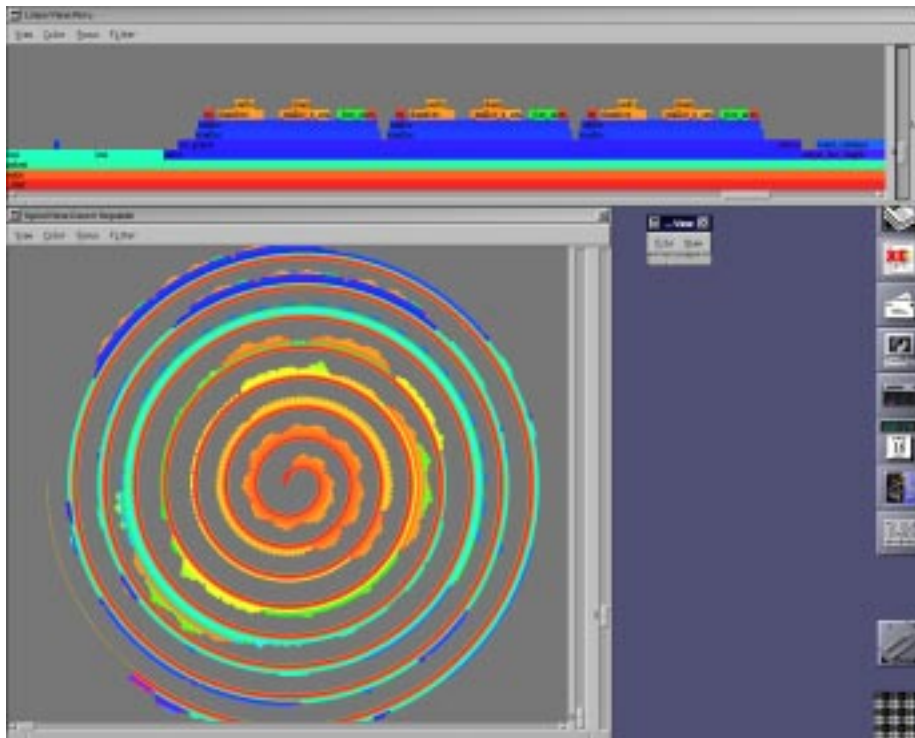


Figure 5: A linear and a spiral view, linked. The linear view shows what is on the top of the outermost semicircle of the spiral.

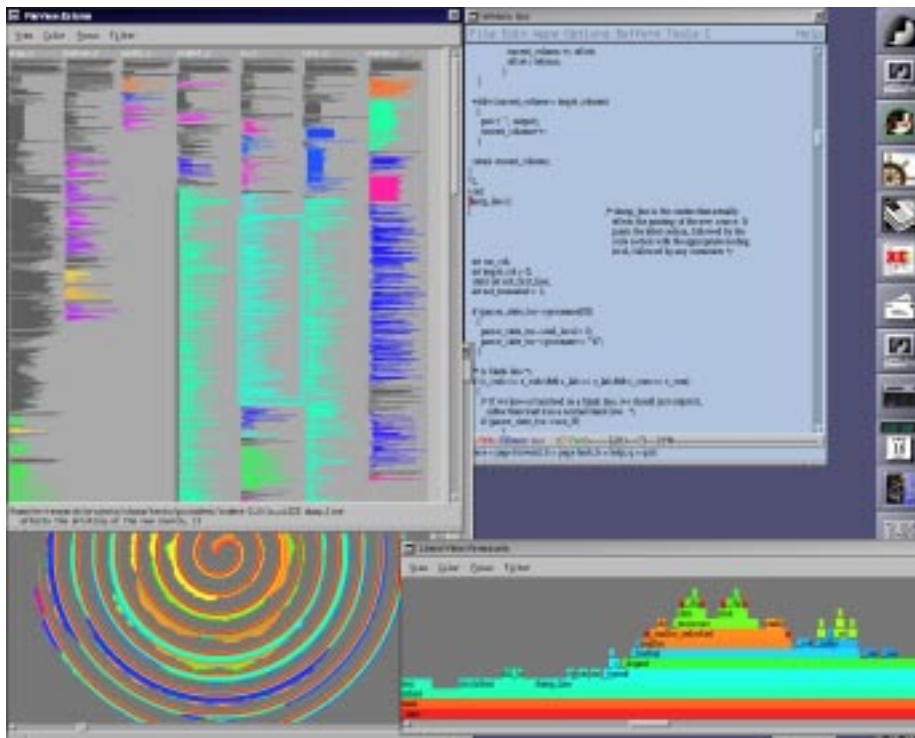


Figure 7: All four views together.

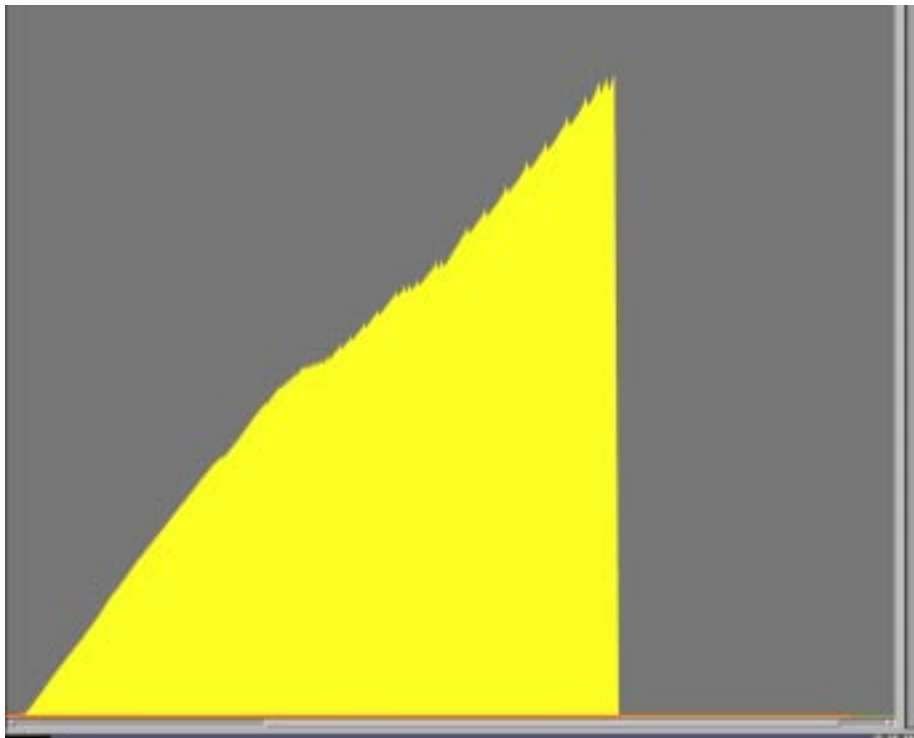


Figure 8: A visual data mining example