Scripts for Program Trace Visualization

Manos Renieris
er@cs.brown.edu

We believe that program trace visualization systems can benefit from incorporating a scripting mechanism akin to a debugger command language. Such a mechanism should have the ability to narrow the visualized data, to navigate the visualization, and to externalize the data so that the visualization system can interface with other analysis systems.

Standard debuggers like gdb and dbx provide the user with simple command languages with which the programmer can stop the execution, explore the data, and allow the execution to proceed. Debugging tools have the advantage that since the program is loaded in memory, its state can be examined at will. However, the difficult task of correlating temporally distant events is left to the user.

Visualization tools have limited information available. On the other hand they allow the user to navigate freely in the event sequences, highlight specific events of kinds of events, and display, close to each other, events that have semantic, but not temporal, proximity. This navigation is done, almost invariably, in a graphical manner, by points and clicks. We believe that the usefulness of these abilities can be greatly enhanced by adding a command line to trace visualization systems.

To motivate the discussion, suppose we have a debugger that can insert breakpoints at function entry points, and a visualization system whose trace data includes function calls and returns. Some queries we might want to ask the systems are the following:

1. Show me all but the first three calls of `printf`.
2. Show me all the calls to a class's constructor that precede all calls to that class's destructor.
3. Show me all calls of function `A` not followed immediately by a call to function `B`.
4. Show me the last call before, and the first call after, any call to `B`.

Query 1 can be easily answered by a debugger (set a breakpoint on printf; ignore it three times). Query 2 takes a little more work (set a breakpoint of both the constructor and destructor; on the first destructor break, delete the constructor breakpoint). Query 3 cannot be easily answered by a debugger, because it involves state which has to change on every function call. Query 4 requires the display of an event in temporal context and debuggers simply do not support it[1] On the other hand, even if we assume that all these queries are feasible on a debugger, their results are potentially large sets of events so displaying them in an interactive manner becomes impractical. Trace visualization systems with elaborated trace models are very well suited to answer this kind of query, but they do not expose a query mechanism.

To give two examples, the basic trace model of ISVis is graph based [5], and the model presented to the user is very elaborate, specifying interactions between run-time entities, but the is not a 'command line'

[1]Debuggers with a reversible execution (e.g. [2], [1]), when practical, allow one to see events in a temporal context. However, for the scope of this discussion, reversible execution is just a lazily generated trace.

to specify queries. The data model for Ovation [3] has a relational flavor, and it supports very expressive queries, but the user is presented with black box visualizations (although the system is greatly extensible otherwise).

Providing a query language interface has other benefits. Primitives could be provided to *focus* the display on a particular event, to *advance* to the next event of a particular type, to *highlight* or *elide* others. An API can be provided to interface the query system, and through it the whole system, with external analyzers. Lastly, if we have the ability to describe queries textually, they can be stored, re-executed and sent through simple email. At a higher level, exposing the data model to the users will allow them to better understand the limitations and capabilities of the system.

We believe that the ideal language for a visualization system would be declarative, like Prolog or SQL. At the same time, scripting languages like Tcl, Perl and Python are widely used to extend software. All these languages already provide APIs, execution models, and large use bases.

The usefulness of Prolog as a debugger language is demonstrated in Ducassé's Coca [4]. Her system is based on an interface between Prolog and gdb. Prolog is left untouched, except for a small number of added primitives, which include `fget` (forward get), which advances the execution to the next event that matches its argument. Gdb is only changed to interface with Prolog.

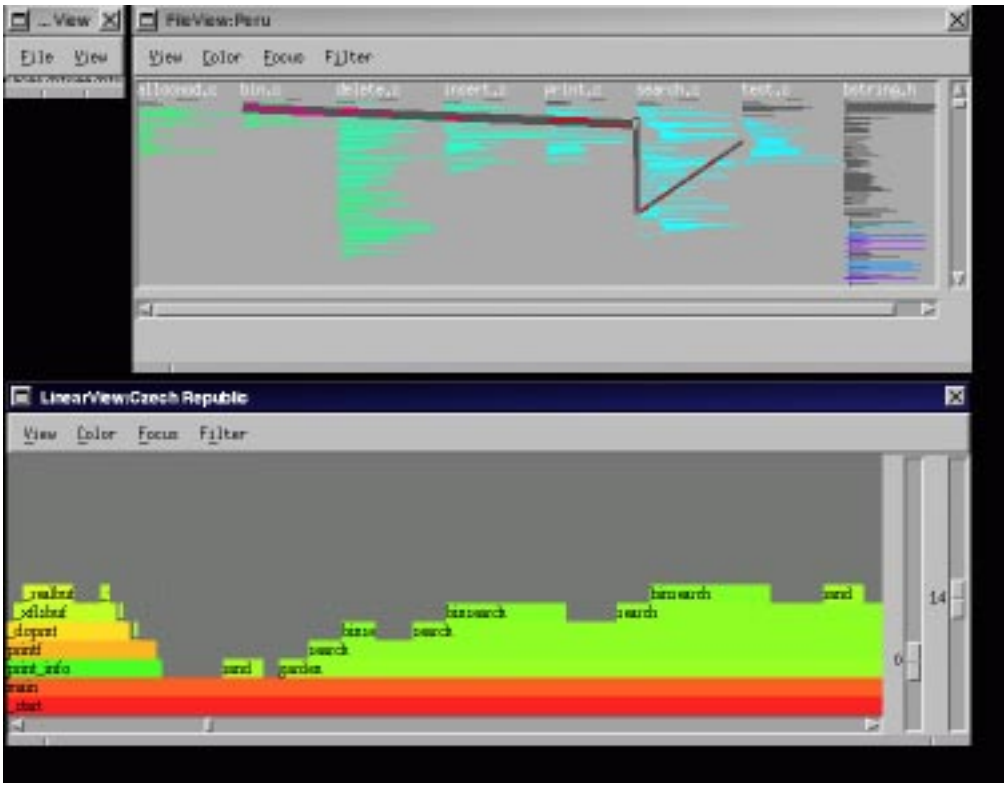
We plan to add a similar facility to our trace visualization system, Almost [6]. Almost reads the source of a program and a trace file and shows the trace as a visibility graph and the program with a Seesoft-like view (see figure 1). Views are connected in a Model-View-Controller fashion, so clicking on a particular function in a trace view will highlight the corresponding portion of program text. Almost's trace model is relational. The basic relation contains function activations. It includes the following fields:

- FunctionId
- Level (depth of activation stack)
- Thread
- StartTime (when the function was called)
- EndTime (when the function returned)

In Prolog, an event would be represented by a fact like `activation(eventId, functionId, level, thread, startTime, endTime)`. We can then write predicates that specify whether an activation caused another (`caused(EV1,EV2) :- activation(EV1,_,_,_,S1,E1), activation(EV2,_,_,_,S2,E2), S2 > S1, E2 < E1.`), whether it directly caused a another (same as caused, but with the added constraint that `L1 + 1 = L2`) or whether it precedes another (the constraint would be `E1 < E2`).

Every Almost view has a *focus* which is the event in the middle of the view. The focus can be modified from any other part of the system, through a specific callback function. If Almost is connected with a Prolog interpreter, `fget` can be implemented as a simple matching function on the events, followed by a call to the callback.

Almost also contains the notion of a filter, which allows the user to highlight events that satisfy specific simple conditions. The existing conditions include simple pattern matching on the function name, and simple interval containment for the time fields. Augmenting this with conditions expressed in Prolog would allow us to highlight events like:



- `:- findall(EV,(activation(EV,F,_,_,_,_), name(F) = "printf"),P), [_,_,_|T] = P, highlight(T).`
  (Find all invocations of `printf`, throw away the first three, and highlight the rest.)
- `:- findall(EVD, (activation(EVD,FD,_,_,_,_), name(FD) = "d"), [D|_])), findall(EVC, (activation(EVC,FD,_,_,_,_), name(FC) = "c", precedes(EVC, D)), CL), highlight(CL).`
  (Find the first invocation of function `d`, find all the invocations of `c`, and highlight all of them that are before all calls to `d`).

(The other queries we mentioned at the beginning can be done similarly).

We are currently developing a set of primitive primitives, together with an algebra over them to handle this kind of query in a more effective and intuitive way.

REFERENCES

[1] Bob Boothe. Efficient algorithms for bidirectional debugging. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 299–310, 2000.

[2] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, 1998.

[3] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. *Lecture Notes in Computer Science*, 821:163–??, 1994.

[4] Mireille Ducasse. Coca: A debugger for C based on fine grained control flow and data events. In *Proceedings of the 21st International Conference on Software Engineering*, pages 504–515. ACM Press, May 1999.

[5] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the 1997 International Conference of Software Engineering*, Boston, MA, USA, May 1997.

[6] Manos Renieris and Steven P. Reiss. Almost: Exploring program traces. In David S. Ebert and Christopher D. Shaw, editors, *Proceedings of the 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, 1999.