# Research Report:
# Cacti: A Front End for Program Visualization

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910
(401)-863-7641, FAX: (401)-863-7657
spr@cs.brown.edu

## Abstract

*In this paper we describe a system that allows the user to rapidly construct program visualizations over a variety of data sources. Such a system is a necessary foundation for using visualization as an aid to software understanding. The system supports an arbitrary set of data sources so that information from both static and dynamic analysis can be combined to offer meaningful software visualizations. It provides the user with a visual universal-relation front end that supports the definition of queries over multiple data sources without knowledge of the structure or contents of the sources. It uses a flexible back end with a range of different visualizations, most geared to the efficient display of large amounts of data. The result is a high-quality, easy-to-define program visualization that can address specific problems and hence is useful for software understanding. The overall system is flexible and extensible in that both the underlying data model and the set of visualizations are defined in resource files.*

## 1.0 Background and Motivation

Program visualization is the process of providing visual representations of a program and its execution to the programmer. Because software developers typically draw diagrams to describe and help others understand how their software works, the classical motivation for program visualization has been that it is an aid to software understanding. Software understanding is the task of helping a programmer to answer questions about the software during maintenance or development. It is a key to software development since it involves the ability to answer the specific questions that tend to arise in these phases. For example, a developer might want to know why a particular function is called so often or how a particular situation involving timing constraints could arise or what needs to be modified to add a parameter to a given function.

Program visualization efforts have a long history, dating back to a variety of programs that would automatically produce flowcharts from a deck of Fortran cards [13,18], ranging to standard diagrams such as call graphs, dependency graphs, and a class browser, all of which were designed to handle moderately large software systems and which were present in our previous environment FIELD [15] and are now standard in many programming environments. Other visualizations show data structures either statically [12] or through algorithm animation [2,3,8,23], show performance information [1,7,11,22] or show other aspects of the program as the heap visualizer in FIELD or the various views of Seesoft [5].

Program visualization is also related to the more general field of information visualization and data mining. Significant work in developing information visualizations has been done at Xerox PARC [9,20], SGI, and elsewhere. We make use of this work by incorporating the visual metaphors that are incorporated into our generic back end visualization engine. The Visage system represents a more general approach that, like ours, attempts to combine data specification, browsing, and visualization in an easy-to-use framework [21]. Our system differs in providing a more data-centric approach, in concentrating more on abstract data from a variety of sources rather than more concrete visualizations from an object data base, and on managing dynamic as well as static data.

Program visualization has been quite beneficial as an aid to navigation and specific program visualizations have been quite successful in addressing some explicit problems. However, program visualization has not been widely used for understanding. The results of our study [10] showed little if any effect on software understanding from using visual tools. Similar experiences and the fact that these tools, while practical, are not widely used, show the same thing.

Our analysis has identified three reasons why program visualizations have not succeeded in addressing the issues of software understanding: the visualizations do not address the specific questions that are inherent to software understanding; they are not capable of displaying the large amount of information that is inherently needed; and they are too difficult to set up and use.

For Color Plate See Page 120

## 2.0 Overview of our Approach

To create a practical approach to program visualization we need to deal with the three problems cited above. The Cacti system we are developing, and the Desert environment [17] it is a part of, attempt to accomplish this by 1) providing a range of automatic and inexpensive data collection techniques, 2) offering a range of back-end visualization methods, and 3) providing the facilities to allow the user to define high-quality visualizations quickly and easily.

The first component of our solution is a set of inexpensive data collection techniques which have been implemented in our previous environment, FIELD, in our current programming environment, Desert, or in an experimental project, AARD [14]. Access to this information is provided by a set of background database processes that automatically update as the underlying files change [19].

The second component is to provide a wide range of visualization strategies suitable for handling large amounts of data. This allows an appropriate strategy to be used for each specific visualization to maximize the amount of information presented and to highlight the important relationships. Our recent three-dimensional efforts, Valley [16], provides an extensible framework that allows the easy incorporation of a wide variety of visualization strategies. We have currently integrated about ten different strategies and are able to add a new strategy with a day or two's effort.

The third component is the focus of our current efforts. Here we have chosen a strategy that separates the definition of what should be visualized from how it should be visualized. This has allowed us to deal with visualization issues, i.e. different visualization strategies, browsing techniques, and graphical support, independently of the issues of what data should be visualized. The front end generates a set of objects and relationships to visualize and chooses the visualization strategy and the mapping from objects to visualizations.

This component consists of a unified data model to describe the variety of data sources, a common model for representing the target objects, and an algorithm that allows us to define a query building the target objects from the underlying data sources. To simplify the user's task in defining objects, we decided to use a variation of a universal relation model with a visual front end. While there are a number of visual query languages [4,6,24], those that have been proposed required too much knowledge of the underlying databases for our purposes. By hiding the structure of the underlying data, we intend to greatly simplify the user's query definition. However, doing so requires a more elaborate underlying data model and a heuristic algorithm for actually defining the query from the user's specification.

## 3.0 An Example of Cacti

Digital Equipment Corporation had a problem with understanding C++ programs. Inefficiency and potential bugs are introduced in such programs when the compiler creates class temporaries for call parameters or within an expression. They wanted a tool that could find all instances of such temporaries over a large system and allow the user to browse over the result to find code fragments that should be changed.

Rather than creating a separate tool, we can define an appropriate visualization using Cacti. A temporary can be identified by the existence of a constructor call and a destructor call for the same class at the same line in some source file. We use Cacti to create two classes, one to represent constructor calls and one to represent destructor calls. Figure 1 shows the first step in this process, with the `ConstructorCall` class of objects defined. This was built by first selecting new class and then choosing the file name, to name, and line number from the set of known fields displayed in a dialog window, then defining the `class` field as a computed field based on the `ToName`, and restricting the `ToName` field to be a constructor name. The set of known fields is the union of data fields from all available data sources. Note that the user does not need to know the structure of this data or its form in order to use the system.

The second step in creating the visualization is to create a similar class for destructors, repeating the above operations but this time restricting the `ToName` field to be a destructor name. After this is done, the two classes are related using a reference field as shown in Cacti Color Figure 1 in the color plates at the end of the proceedings. Reference fields provide a way of relating information in one class to that of another. In this case, we specify that a constructor must be found with a matching file name, line number, and class name or the corresponding destructor call entry should be deleted. Here we have also designated the `ConstructorCall` class as passive so that only the remaining destructor calls, those that reflect temporaries, will be display.

The next step is to request that this data be visualized. The user does this by clicking on the Visualize button. At this point, Cacti uses a variety of built-in mappings and heuristics to determine how to map the class specification given by the user into a set of queries on the underlying databases. If the result cannot be determined or if it is ambiguous, the user is asked for clarifications. Otherwise, Cacti looks through the set of available visualizations and determines which of these might be appropriate for the given set of objects. If more than one visualization is appropriate, Cacti asks the user to select the desired visualization style. In this case, we choose a `FileMap` style that provides a mapping of information to source files and works over a large domain.

At this point, Cacti puts up the dialog box shown in Figure 2 to allow the user to specify parameters for the visualization and the mapping from fields to visualization information. In this case we have specified that the backdrop color be white and that the remaining parameters be assigned default values and presented to the user as dynamically settable options as part of the visualization. Moreover, we have specified which fields of the `DestructorCall` class should be viewed as containing the file name, line number, and data statistic that is required by a `FileMap` visualization. In this case, we specified that the statistic should be the class name.

Once we accept these parameters, Cacti runs the visualization engine Mirage with a data file describing how to get the appropriate data and how to display the result. This data file can be saved to allow the same visualization to be applied to different systems or at different times without having to rebuild the description in Cacti. Mirage contacts the necessary databases, extracts and combines the data from possibly multiple sources, and then uses the underlying visualization engine to put up a display.

The display corresponding to this example can be seen in Cacti Color Figure 2 in the color plates at the end of the proceedings. The visualization displays each file as a row split into buckets for the different lines. In this case, because there are a large number of files, multiple files are shown in each row. An instance of a temporary is shown as a box on top of the file displays. If multiple instances occur within the same bucket, these are stacked up on top of each other in such a way that a top-down view would show all the different colors within a bucket. The statistic field is used to determine color. Mirage automatically determines that the value specified in this case, the class name, is a string, and maps this into integer values for the visualization which then use the values to determine the color of the different nodes. Thus each class for which temporaries are created is given a different color. The relative preponderance of one color indicated a large number of temporaries of that class type. This is particularly useful in this case since temporaries of some classes are benign which those of others indicate real or potential problems. By simply looking at what colors correspond to what classes, the user can quickly identify where the actual problems might be.

Mirage actually provides a range of different visualization strategies including `PointMap`, `BoxTree`, `Layout`, and `MultiFile`. `Layouts` represent graphs consisting of nodes and arcs laid out in two or three dimensions. `BoxTree` layout is a compacted tree layout that is able to show four statistics in addition to the tree structure. `PointMap` uses an array of points or bars to represent the relationship between two quantities. For example Cacti Color Figure 3 in the color plates at the end of the proceedings uses a `PointMap` layout to show a call graph. Here the X and Y axis represent the various routines (in alphabetical order), color represents the depth of call, and Z represents different classes. If a call goes between two classes, it has a bar going from one class to another. Finally, the MultiFile layout allows information to be shown on a line-by-line basis inside a file. For example, Cacti Color Figure 4 in the color plates at the end of the proceedings shows a small project with different types of scopes identified by color.

## 4.0 Implementation and Experience

Cacti is currently implemented as part of the Desert software development environment. It uses the Desert cross reference databases as its data sources. The front end allows the user to easily define visualizations by defining the data to be used and then selecting and parameterizing the visualization strategy to be used. At this point Cacti generates a file describing the visualization and runs a back end, Mirage, with this file. Mirage is responsible for reading in the data and providing the actual visualization. Cacti and Mirage are implement in about 23,000 lines of C++.

This two-step approach and the use of resource files throughout the process provides a great deal of flexibility. We have extended the resource file definitions for Cacti to allow the definition of classes as well as data sources. This allows Cacti to provide a wide range of predefined data visualizations and simplifies the user's task. In addition, definitions themselves can be saved in such a resource file. The visualization description file output by Cacti for Mirage can also be saved and reused. This allows the system to be used to defined fixed visualizations while still offering the user the flexibility of parameterizing the result.

While our experiences with Cacti and Mirage have been limited (like the overall Desert project, these packages are early prototypes and are not widely used), we are optimistic that this approach will be successful. We have been able to define a wide range of visualizations using the front end. These have been defined quickly and logically. The heuristic algorithm for converting the class and member definitions into a query has worked quickly and has generated the right query for all our tests.

## 5.0 References

1. Bill Appelbe, Kevin Smith, and Charlie McDowell, "Start/Pat: a parallel- programming toolkit," *IEEE Software* Vol. **6**(4) pp. 29-38 (July 1989).

2. Marc H. Brown and Robert Sedgewick, "Techniques for algorithm animation," *IEEE Software* Vol. **2**(1) pp. 28-39 (1985).

3. Marc H. Brown and Marc A. Nojork, "Algorithm animation using 3D interactive graphics," DEC Systems Research Center (1992).

4. I. F. Cruz, "DOODLE: a visual language for object-oriented databases," *ACM SIGMON Intl. Conf. on Management of Data*, pp. 71-80 (1992).

5. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft - a tool for visualizing software," AT&T Bell Laboratories (1991).

6. K. Goldman, S. Goldman, P. Kanellakis, and S. Zdonik, "Isis: interface for a semantic information system," *Proceedings of the ACM SIGMOD*, (1985).

7. Vincent A. Guarna, Jr., Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur, "Faust: an integrated environment for parallel programming," *IEEE Software* Vol. **6**(4) pp. 20-27 (July 1989).

8. Ralph L. London and Robert A. Duisberg, "Animating programs using Smalltalk," *IEEE Computer* Vol. **18**(8) pp. 61-71 (August 1985).

9. Jock D. Mackinlay, George G. Robertson, and Stuart K. Card, "The perspective wall: detail and context smoothly integrated," *Proc. CHI'91*, pp. 173-179 (April 1991).

10. Scott Meyers and Steven P. Reiss, "An empirical study of multiple-view software development," *Software Engineering Notes* Vol. **17**(5) pp. 47-57 (December 1992).

11. Thomas G. Moher, "PROVIDE: a process visualization and debugging environment," *IEEE Trans. Soft. Eng.* Vol. **14**(6) pp. 849-857 (June 1988).

12. Brad A. Myers, "Incense: a system for displaying data structures," *Computer Graphics* Vol. **17**(3) pp. 115-125 (July 1983).

13. B. A. Price, I. S. Small, and R. M. Baecker, "A taxonomy of software visualization," *Journal of Visual Languages* Vol. **4**(3) pp. 211-266 (Dec. 1993).

14. Steven P. Reiss, "Trace-based debugging," *Proc. AADEBUG '93*, (May 1993).

15. Steven P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

16. Steven P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages*, (December, 1995).

17. Steven P. Reiss, "Simplifying data integration: the design of the Desert software development environment," *Proc. 18th Intl Conf on Software Engineering*, pp. 398-407 (March, 1996).

18. Steven P. Reiss, "Software tools and environments," in *Software Visualization: Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

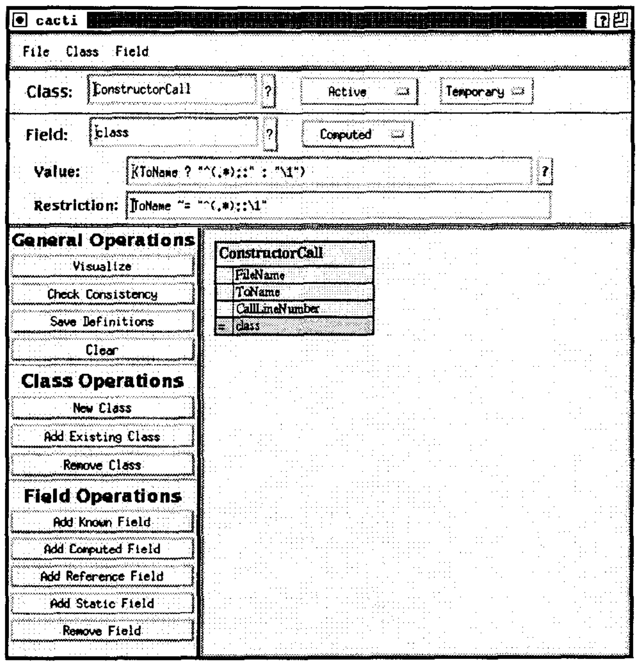
19. Steven P. Reiss, "Dynamic management of the Desert program data store," Brown University CS Tech Report (1997).

20. George G. Robertson, Jock D. Mackinlay, and Stuart K. Card, "Cone trees: animated 3D visualizations of hierarchical information," *Proc. CHI'91*, pp. 189-194 (April 1991).

21. Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Phillip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire, "Visage: a user interface environment for exploring information," *Proc. Information Visualization*, pp. 3-12 (October 1996).

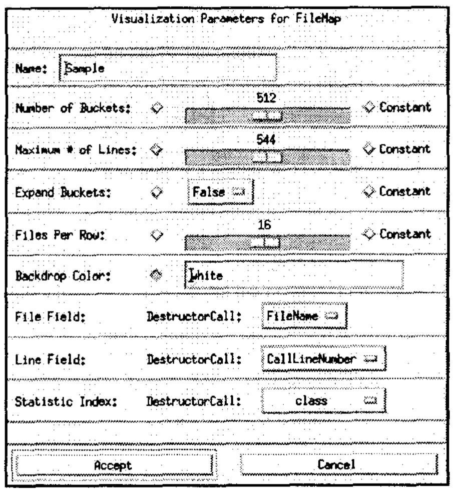
22. Lawrence Snyder, "Parallel programming and the Poker programming environment," *IEEE Computer*, pp. 27-36 (July 1984).

23. John T. Stasko, "TANGO: a framework and system for algorithm animation," *IEEE Computer* Vol. **23**(9) pp. 27-39 (September 1990).



Class: ConstructorCall ? Active Temporary
Field: class ? Computed
Value: (ToName ? "^(.*)::" : "\1") ?
Restriction: ToName ~= "^(.*)::\1"

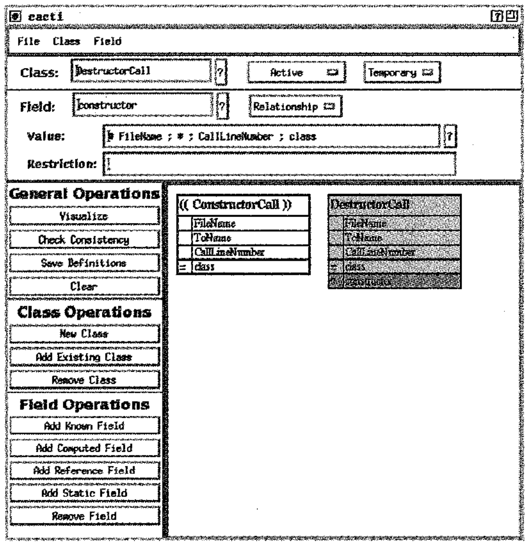
FIGURE 1. A view of cacti showing the definition of a class of objects corresponding to constructor calls within a system.



Name: Sample
Number of Buckets: 512 Constant
Maximum # of Lines: 544 Constant
Expand Buckets: False Constant
Files Per Row: 16 Constant
Backdrop Color: White
File Field: DestructorCall: FileName
Line Field: DestructorCall: CallLineNumber
Statistic Index: DestructorCall: class

FIGURE 2. Cacti's dialog box for defining the visualization of class temporaries.

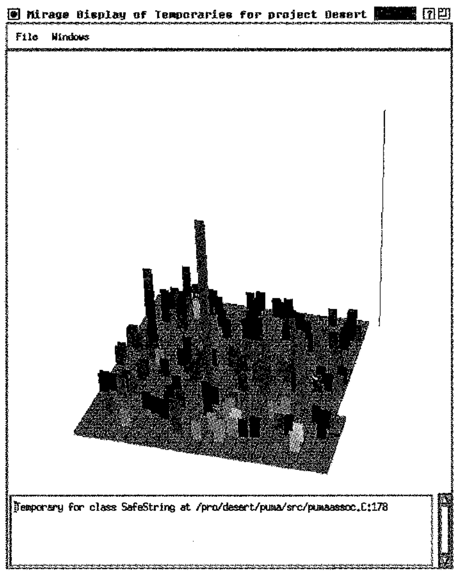
24. M. M. Zloof, "Query by Example: a data base language," *IBM Systems J.* Vol. **16**(4) pp. 324-343 (1977).



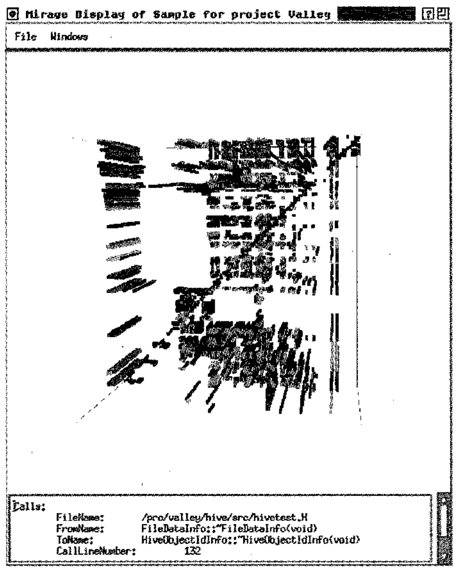
```
Class:       DestructorCall
Field:       constructor
Value:       # FileName ; * ; CallLineNumber ; class
Restriction:
```

```
(( ConstructorCall ))
   FileName
   ToName
   CallLineNumber
=  class
```

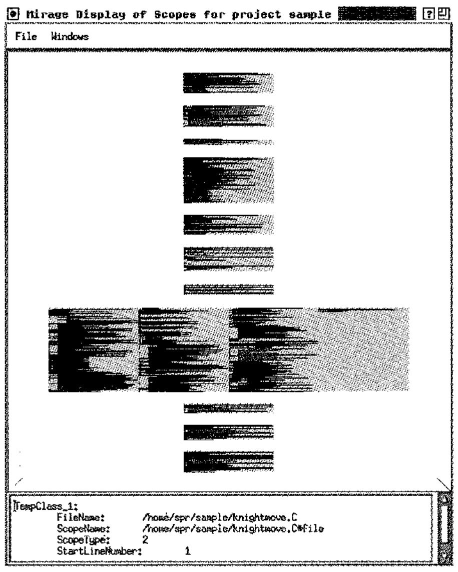
```
DestructorCall
   FileName
   ToName
   CallLineNumber
=  class
   constructor
```



```
Temporary for class SafeString at /pro/desert/puma/src/pumaassoc.C:178
```



```
Calls:
        FileName:        /pro/valley/hive/src/hivetest.H
        FromName:        FileDataInfo::~FileDataInfo(void)
        ToName:          HiveObjectIdInfo::~HiveObjectIdInfo(void)
        CallLineNumber:       132
```



```
TempClass_1:
        FileName:        /home/spr/sample/knightmove.C
        ScopeName:       /home/spr/sample/knightmove.C#file
        ScopeType:       2
        StartLineNumber:      1
```