Software Visualization in the Desert Environment

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

ABSTRACT

While software visualization has been widely used for navigation, its use for understanding has been much more limited. Software visualizations are expensive to develop, require large amounts of information that is often difficult to collect, and even then, are good only at addressing the specific task for which they were developed. Our work on software visualization in the Desert environment is aimed at making software visualization a viable approach to understanding. We do this by providing ready access to a variety of information about the system at hand, a range of high-quality, high-density visualizations, and a simple interface that lets the programmer rapidly create new software visualizations for understanding problems as they arise. In this paper we describe the approach we have taken.

1 MOTIVATION

Software visualization is an attempt to provide dynamic pictures of software. It has been used to illustrate program structure, performance, and the behavior of parallel systems. However, there have been few widely used applications of software visualization. Most of the applications that do exist are aimed at providing better navigation tools. This is the purpose of the module and class views in Microsoft's Visual Studio and it is where the call graph and class diagram views provided by the FIELD environment were most effective [14]. Visualization has had limited success in the real problem it was envisioned to address, software understanding.

Software understanding is the task of asking and answering specific questions about a software system. These questions arise during development and maintenance of a software system when the programmer wants to understand the behavior or determine the consequences of a given modification. Answering such questions typically takes data on program structure, program semantics, change history, and program behavior. It is the nature and variability of the questions that are asked here that makes software visualization difficult to apply.

The programming environment Desert provides the benefits of data integration without the costs. It maintains the original source files and augments them with several databases of program information that are readily accessible to programming tools and are automatically updated. The current databases include a fragment repository providing information about logical units (such as functions of class definitions) of the source files, a cross-reference database that can be viewed as a system-wide symbol table, and a database of trace information. Additional databases are planned to hold the system model including both build and change history information and flow information derived from program analysis.

This abundance of system information, current and future, provides a strong basis for visualization. In addition, we have developed a general-purpose visualization framework that specializes in high-quality, high-density information visualization. This framework is designed to present large amounts of data to the user in a meaningful way, to let the user rapidly browse over that data to find items of interest, and to use the current displays for navigation.

On top of this foundation, we have built a system that lets the user rapidly define and customize arbitrary program visualizations. The system uses a visual front end to let the programmer select information to visualize from any combination of Desert's databases. For simplicity, this query language is based on a universal-relation assumption. The front end also gives the programmer facilities to select and parameterize an appropriate visualization style. The back end gathers the data from the various databases and then uses the visualization framework to create a browsable display.

This work integrates a unique combination of facilities for software visualization. It is the first system to use a visual query language based on the universal relation assumption. It provides uniform access to a variety of data sources,

Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by DARPA order 8225, by ONR grant N00014-91-J-4052, and by support from Sun Microsystems and Digital Equipment Corpora-

allowing these sources to be correlated in the visualization. It offers a broad and extensible set of visualization styles. At the same time, it is also one of the few systems that can handle visualization of large software systems. These claims are illustrated in the remainder of this paper.

2 DATA FOR VISUALIZATION

The databases maintained by Desert are designed to be easily accessible by other tools in the environment and for visualization. The principal database used by other tools is the cross-reference database, which obtains information either from the compiler (using Sun's source-browser database) or through approximate scanners. Scanning and updating occur automatically whenever files are changed by the user — no user action is required. The database provides a message-based front end that is accessible to other tools. This front end offers a full SQL interface for accessing the data. The database compiles the SQL into a query, optimizes the query and generates the requested output.

Desert's trace database was designed to provide a variety of information about program execution for software understanding and visualization. The raw data is collected by patching executable files and shared libraries with calls to generate appropriate trace information. The technique is similar to that used by Pixie [32] or Purify [9]. Our package is a simplified version of Warts [2,11]. The raw data we can currently obtain includes real-time performance, dynamic call counts, full dynamic call traces, use of heap memory, load and store traces, basic block counts, and basic block traces.

The data generated from these traces is stored in a common directory. A second system gives visualization tools (and other tools in Desert) access to the trace files. This tool takes specialized requests, performs any appropriate analysis, and returns the result as a relation. Multiple analysis can be done from a common trace file.

Both of these databases have been designed to handle large systems. The cross-reference database easily handles the 250,000 lines of Desert code and we expect it can handle million-line systems even in its current prototype form. The database currently provides cross-reference information about C, C++, and Java sources. It is relatively easy to extend it to other languages. The trace generation and analysis is designed to handle arbitrary-sized system and is limited only by the amount of disk space required for eventually storing the traces.

We expect to add other data for visualization. The FIELD system included a configuration management database that provided a limited set of queries over information related to build dependencies and version history [20]. We plan to integrate a new such tool into Desert so that the visualization tools can utilize this information as well. We are also working on obtaining and storing data and control flow information for more complex visualizations. The overall visualization framework is designed so that this new information can be easily integrated and used as soon as it is available.

3 THE VISUALIZATION FRAMEWORK

The visualization framework we have developed is an extension of our earlier work on general-purpose 2D visualizations [21]. It differs from that work in that it is designed to provide a broader range of visualizations, includes both 2D and 3D visualization strategies and richer and more general browsing techniques, makes full use of color and other graphical properties, provides automatic animation, and handles visualization of a much larger number of objects. The current framework is the third generation of this work; each generation has had a simpler application interface, used a different underlying graphics framework, and handled an order of magnitude more objects [21]. The current generation has been used effectively to display visualizations based on 100,000 objects and we would expect it to scale up slightly from there. The current limitation is on the capabilities of the graphics hardware and our efforts here have been aimed at producing new visualizations that provide more abstract views of the visualization data.

Our visualization framework works in two stages. It starts with an arbitrary set of user objects; for software visualization, these are objects constructed from queries on the available databases. These objects are then mapped into objects for a particular visualization. The graphics objects include high-level objects that represent a visualization strategy, middle-level objects that represent components of that visualization, and data objects that represent data available from a user object but do not necessarily have a graphic representation of their own. For example, a graph diagram consists of a Layout object along with any number of Arc objects and Node objects, while a file-based view similar to SeeSoft [6] consists of a FileView object and arbitrary data objects representing information about the file. The current set of visualizations is shown in Figure 1.

4 DEFINING WHAT TO VISUALIZE

The key to producing a visualization environment for software understanding is to provide a powerful front end that lets the user quickly define the appropriate visualization for the task at hand. To be successful this front end must:

- *Provide access to all data sources.*
- *Allow simple selection of the data to visualize.*
- *Utilize a straightforward query language.*
- *Allow easy specification of the visualization strategy.*

Our system achieves these goals in a three step process. It first provides a front end that lets the user specify what should be visualized and provides appropriate mappings from the underlying data sources to a set of data objects and relationships ready for the visualization engine. This front

Visualization	Component Objects	Description
BoxTree	Data	A compact tree representation consisting of boxes packed together that can display four statistics in addition to the tree structure.
FileMap	Data	A file-based display for showing line-related information about a large number of files within a system.
FileView	Data	A simple single-file display based on SeeSoft that can show data about individual lines of a file.
MultiFile	Data	A generalization of FileView that can display multiple files simultaneously.
PointMap	Data	A display starting with a 2D matrix of dots where each data element determines the X and Y location of a dot. It can be extended to let the dots be prisms, using data statistics to set the starting and ending Z location and to take the closure.
Layout	Node, Arc	A display of a 2D or 3D graph consisting of the underlying nodes and arcs.
TimeMap	Data	Display a 2D layout of objects with time represented by the X dimension and some other value being expressed by the Y dimension. It can be augmented with a set of arcs that are tied to the original set of 2D objects but which can have arbitrary Z starting and ending points.

FIGURE 1. Current visualization strategies.

end is parameterized through resource files so it can be readily adapted to multiple data sources. Then it provides the mechanisms necessary for letting the user choose, parameterize, and use a visualization appropriate for the selected data. The system again uses resource files here to let new visualizations be added easily. Finally, it offers a separate tool for viewing, browsing, and navigating in the visualization.

The first stage provides a user interface that lets the user define the set of objects to visualize., and provides a visual query language that lets the user build objects by specifying the fields they contain.

The system uses the universal-relation assumption in which the various databases are viewed as a single relation. The data fields from all source relations are viewed as elements of a variety of domains, with the different domains being arranged hierarchically. For example, information about a function call contains a from routine and a to routine. These separate domains are both instances of the routine domain that is contained in the identifier domain. The user, in building an object, selects which domains are appropriate without having to identify or understand the structure of the underlying database.

The example in Figure 2 comes from a software understanding problem in industry. The company was concerned with efficiency in C++ programs and wanted to locate all places in a system where the compiler generated class temporaries. Class temporaries, they felt, often represented incorrect or inefficient code. We accomplish this in our front end by finding all lines that contain a call to a constructor and a destructor for the same class. The first set of objects represents constructor calls, the second destructor calls.

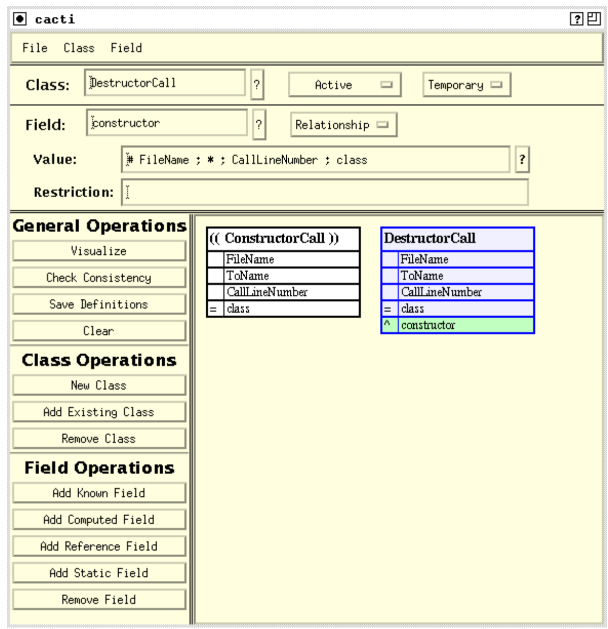

FIGURE 2. Defining the information to visualize.

The user defines what should be visualized by defining classes containing fields, both of which are displayed visually in the interface. Classes are defined by specifying a class name and the display properties of the class. The current display properties are *hidden* for classes that are not to be displayed, *visible* for classes to be displayed, and *sequential* for classes to be animated in the display. In Figure 2, the DestructorCall class is visible while the ConstructorCall class is hidden.

Once a class is defined, the user can add fields to it. The basic added fields represent the domains that are defined for each type of data. The user selects these domains from a domain browser. In the ConstructorCall class of Figure 2, the user has selected three domain fields, FileName, ToName, and CallLineNumber. Note that the use of domains isolates the user from any knowledge of the underlying database schema. For example, even here the ToName domain has multiple sources (dynamic call information and static call information), and file names are kept in a separate relation that must be joined with the static call data using a file identifier field.

The user can also define computed values as part of the class. In this case, the user has defined a class field that is the class name derived from the ToName of the call. The system provides a full expression language, including regular expression matching, for this purpose. Both domain fields and computed fields can be restricted by arbitrary Boolean expressions, which can use values from any other field of the class. In Figure 2 such a restriction is used to insure that the ToName of the ConstructorCall class actually represents a constructor name.

The third type of field the user can add to a class is a reference field, which is used to relate one class to another. A reference field is defined by matching each field in the referenced class. The match can be either wildcarded or against a field of the referencing class. In Figure 2, the constructor field of the DestructorCall class is a reference to the ConstructorCall class. Here the specification says that the corresponding filename, line number, and class fields must match. Reference fields can be defined as either mandatory (as here) to require that a match be found, optional to allow but not require a match, or constructive if a new tuple should be created in the referenced relation if none is found.

A fourth type of field represents a static value that the user can change dynamically when the data is being visualized. This can be used to parameterize the resultant visualization for later browsing.

The specification given by the user is mapped into a query when the user either asks to visualize the data or to check its consistency. Since the visual front end uses the universal-relation assumption, the user's definition is generally ambiguous. For example, the given query could be viewed as a request for either static or dynamic call information, and a filename can be associated with a call relation in various ways (for example, the file containing the definition of the called routine).

To overcome these ambiguities, the mapping algorithm uses a branch-and-bound approach to determining the appropriate query, assigning different weights to different ways of associating domains with fields and joining two relations. The universal-relation assumption is not widely used because of the inherent ambiguities and difficulty in determining the proper query in the general case. These problems are alleviated in this case by assuming the domains are reasonably well known and by predefining basic mappings between relations in the appropriate resource file. The system generally chooses the correct query when there is an obvious one. When the result is ambiguous, the system reports it and lets users clarify what they actually mean.

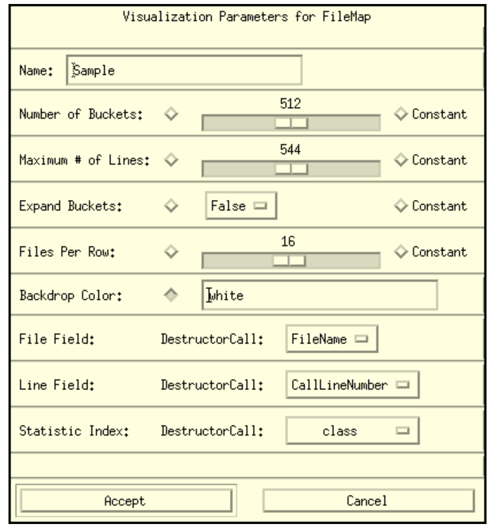

FIGURE 3. Defining the visualization of class temporaries.

This approach to visual querying can be contrasted to previous visual query languages [5,7,33], which either require too much knowledge of the underlying databases or are too schema-oriented for our purposes.

5 SPECIFYING THE VISUALIZATION

The second stage in generating complex software visualizations rapidly is to let the user select and define the appropriate visualization. The front end we provide does this in two steps. It again uses a resource file to define the available visualization techniques so additional techniques can be easily added. Each technique is characterized by an abstract definition of what classes are appropriate, a set of parameters that control the visualization, and a set of parameters that describe how fields in the original data map into the visualization.

The first step in specifying the visualization is taken by the front end. When the user hits the visualize button and the front end determines that the class definitions are consistent, it looks through the set of available visualizations and determines from the abstract definitions of how they are used which visualizations might be appropriate to the given data. If more than one visualization is applicable, the user is asked to select the desired one.

The second step is done in conjunction with the user. The front end constructs a dialog box to let the user name and parameterize the selected visualization. For example, the dialog box for the example of Figure 2 is shown in Figure 3 where the user has assigned it the name "Sample". In addition to the name, the dialog box contains two types of settings. First, it allows the user to set the parameters that characterize the visualization. Here the user can set a value (other than the default) and indicate whether this parameter should be fixed or changeable by the user at visualization time. The first five parameters in Figure 3 are of this type. The remaining settings allow the user to associate data from the classes being displayed with data characterizing the visualization. Here the front end constructs a list of applicable fields based on the field and parameter types and lets the user choose among them. In Figure 3 the user has specified that the file specification should come from the FileName field, the line specification from the CallLineNumber field, and the statistic value from the class field.

6 DISPLAYING THE VISUALIZATION

Once the visualization has been specified, the front end creates a visualization description file and passes it to the back end. The file defines both the query to obtain the data and the parameters of the visualization, and can be used to recreate the same visualization on this or another system. This is a convenient way of saving software visualizations for later reuse.

The back end reads this data file and then builds the visualization. This again is done in two steps. The first corresponds to defining the data to visualize. Here the back end uses the query information in the visualization file to obtain the data. For each database referenced by the query, the back end makes a pass over the overall query and determines what parts of it can be handled by the local database. It then builds a query for that database and requests the resultant data. The back end next puts together a query that combines the information from multiple databases by looking at those parts of the original query that were not handled by local databases. This is evaluated internally using a optimizing query-evaluation engine. The result is the basic set of objects corresponding to the user-defined classes. Once the data is obtained, a separate pass is made to handle any reference fields. The logic for this pass is able to remove and add tuples as appropriate.

Once all the data is created, the back end uses the visualization framework described earlier to put up a visualization. Here it makes a pass over the class data to construct the appropriate visualization objects and assign appropriate values to the parameters of these objects.

The visualization that results can support be browsed and navigated in various ways.The user can change parameters of the visualization itself and the values of any static fields that were defined as part of the data. Whenever such a change is made, the visualization automatically updates itself and animates any changes. The user can also use the inherent browsing facilities of the visualization engine to zero in on any particular aspect of the display. A text window shows what the user is currently pointing to. Clicking on the display causes an editor associated with the Desert environment to display the corresponding source, thereby providing browsing facilities.

Figure 4 shows a variety of visualizations from this system. The visualization in the upper left shows the result of the sample query described in the previous sections, i.e. the location of compiler-generated class temporaries, using a FileMap display. In this case, the visualization is applied to Desert itself and shows an analysis of about 250,000 lines of code. Each row in the X direction represents a file. (Actually, because of the number of files in this display, each row is divided into three parts and represents three files.) The Y position indicates the line in the file. Color indicates the class type of the temporary.

The visualization in the upper right of Figure 4 was done to find relationships within a large call graph. Here we used a PointMap display. Routines are represented by an X (and hence Y) coordinate and a call is indicated by a dot. The visualization did a transitive closure and the color of the dot indicates the level of the call relationship between the items. The Z dimension is used here to indicate classes, with a rod showing a call between classes. The rod blocks on the left represent a large set of interclass calls, indicating a high degree of coupling between the two classes. Isolated blocks represent classes with high cohesion.

The visualization in the lower left was the result of a request to find uses of global variables outside the files where they were defined. Here we have used a MutliFile view to show the file locations of such references within the Desert software. Color coding here indicates which variable is being used.

The fourth visualization in the figure was used to understand the memory behavior of a game-playing program. The display shows a map of memory usage over time based on trace data using a TimeMap display. Time runs in the X dimension, memory address runs up the Y axis. Color here encodes the class of the data being allocated.

7 RELATED WORK

Software visualization work has e a long history [18,22]. More recent efforts have aimed at providing interactive visualizations of large software systems in FIELD [20] and commercial systems such as ObjectWorks, DEC's Fuse, Sun's Workbench, HP's Softbench, and SGI's CodeVision. These tools provide views of the static program structure. They typically exploit the implicit hierarchies to provide browsing capabilities and use a variety of information-encoding techniques to provide substantial information to the user. Other related work can be found in visual approaches to reengineering, such as the Rigi system [16].

More dynamic visualizations can be seen in the display of data and related program structures. This work has taken two forms. The first involves data structure display, starting

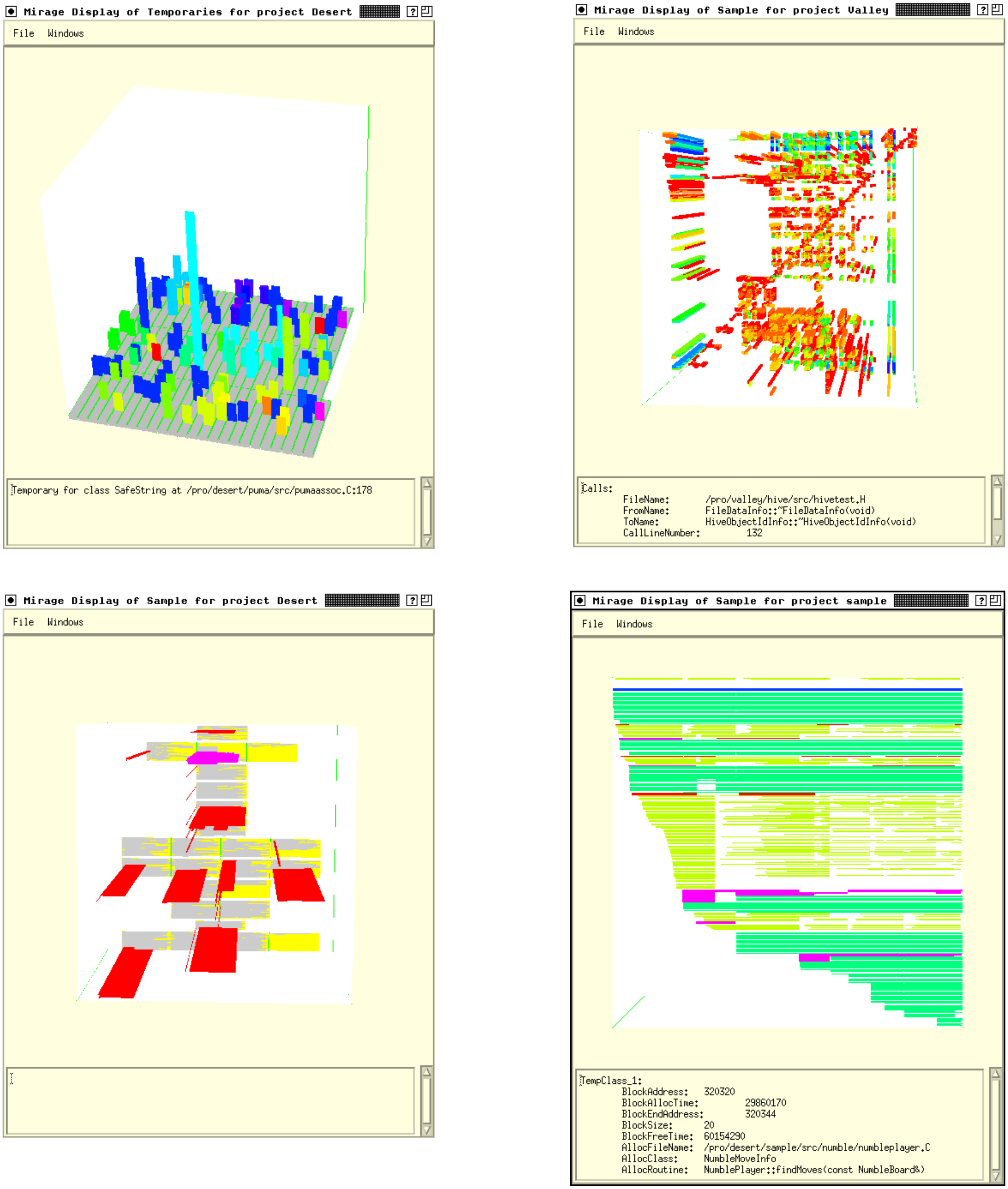

FIGURE 4. Software visualizations in Desert. The upper left display uses a FileMap to show the location of temporaries (color-coded by type) in all of Desert. The upper-right display shows a PointMap view of a call graph, with color indicating depth of call and Z indicating the class. The display at the lower left uses a MutltiFile display to show the use of external variables in Desert. Finally, the display at the lower right shows a map of heap memory over time using a TimeMap display, with color encoding the class of the allocated block.

with the work of Brad Myers [17]. More recent work here ranges from FIELD, where full data structures are displayed automatically using standard or user-customizable graphics and with automatic or semiautomatic update as the program executes, to the displays found in Sun's Workbench or ObjectWorks, where only standard displays are available and the user decides what information to display and its screen location. Other work looks at dynamic visualization of object-oriented programs [10]. This work uses a 2D visualization similar to our TimeMaps but provides compression techniques to allow browsing at different levels of detail.

The various efforts at algorithm animation are a second approach to this problem [3,4,12,24,31]. Here the implementor crafts detailed displays that describe the underlying behavior of a specific program and its data. These systems generally let the user customize the display (with varying degrees of difficulty) and tie the display to the underlying algorithm. For example, Balsa and Tango view the algorithm as generating a sequence of (user-defined) interesting events that then drive the visualization. These approaches are generally useful for small-scale examples, but do not scale up to provide information about large systems.

Another category of software visualizations deal with understanding performance. Displays for sequential programs range from the typical histogram displays found in UNIX *prof*, FIELD, and most current software environments, to the sophisticated graph displays provided by Pure Software Corporation or the 3D displays we developed for the AARD project [19]. A large body of visualization research has been devoted to understanding the behavior of parallel or distributed programs [1,8,15,29]. Displays here have typically concentrated on one aspect of the problem, either processor utilization or message passing.

Finally, there are a number of more recent software visualizations that are more difficult to classify. One set of these is heap visualizations, such as that provided by FIELD, where the system displays a picture of how heap memory is being used, using colors to encode information about each allocation and its use. Another set involves relating information to large bodies of source code, as in Seesoft [6].

There have been efforts in the past to let the user create customized visualizations from program and other data. The DOODLE system and its successors provide a simple visual query language that specifies both the query and the presentation [5]. The various approaches to algorithm animation generally include a front end for defining the visualization as in DANCE [30] or the rule-based approach of Pavane [25]. Another approach uses a knowledge base with information about the program in a software model and information layouts in a view model to create customized visualizations [28]. This approach, while more general than ours, is also more restrictive in having a limited repertoire of visualizations (graphs) and a limited domain. The resultant system can generate graph-based views of relationships within the software, provided the user understood the software model and explicitly coded in the connection between the model and the visualization. Another approach is represented in work comparing design and implementation [27]. This work uses a Prolog query over a limited set of static and dynamic program data to build graphical views. Although the visualizations here have a definite purpose and are limited to graphs and the query language requires significant work on the part of the programmer, this approach and ours does have common elements.

Software visualization is also related to the more general field of information visualization. This problem has received considerable attention in recent years both for visualizing graphs and hierarchical structures and for displays for data mining. Significant work in developing information visualizations has been done at Xerox PARC [13,23], SGI, and elsewhere. We make use of this work by incorporating the visual metaphors 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 [26]. Our system differs in providing a more data-centric approach, in concentrating more on abstract data from a variety of sources than on more concrete visualizations from an object data base, and in managing dynamic as well as static data.

8 IMPLEMENTATION AND EXPERIENCE

The visualization package 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 lets the user easily define visualizations by specifying the data to be used and then selecting and parameterizing the visualization strategy to be used. The package is implemented in about 24,000 lines of C++. The code for patching files, generating trace data, doing symbol lookup, and doing the various trace analyses currently in a separate package, totals 20,000 lines of code.

Our visualization approach and the use of resource files throughout the process provide a great deal of flexibility. We have extended the resource file definitions to allow the definition of classes as well as data sources. This lets the front end 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 the front end can also be saved and reused. This lets the system be used to define fixed visualizations while still offering the user the flexibility of parameterizing the result.

While our experiences with the packages 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 in all our tests.

We plan to continue working on these packages. Our current goals are to include more data sources, provide additional types of visualizations, and further simplify the front end for defining visualizations to make it more intuitive. In addition, as the system becomes more reliable we plan to evaluate its potential for software understanding more rigorously by developing a user base among our students and running one or more controlled experiments on how easy the system is to use and understand, and how effective its visualizations are for software understanding.

9 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. Thomas Ball and James R. Larus, "Optimally profiling and tracing programs," *19th ACM Symp. on Principles of Programming Languages*, pp. 59-70 (January 1992).

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

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

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

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

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

8. 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).

9. Pure Software Inc., *Purify 2 User's Guide*, Pure Software Inc. (1993).

10. Dean Jerding, John T. Stasko, and Thomas Ball, "Visualizing interactions in program executions," *Proc 19th Intl. Conf. on Software Engineering*, pp. 360-370 (May 1997).

11. James R. Larus, "EEL Guts: Using the EEL executable editing library," University of Wisconsin-Madison Computer Science Department (November 1996).

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

13. 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).

14. 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).

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

16. H. A. Muller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji, "A reverse engineering environment based on spatial and visual software interconnection models," *Software Engineering Notices* Vol. **17**(5) pp. 88-98 (December 1992).

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

18. 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).

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

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

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

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

23. 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).

24. Gruia-Catalin Roman and Kenneth C. Cox, "A taxonomy of program visualization systems," *IEEE Computer* Vol. **26**(12) pp. 11-24 ().

25. Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plun, "Pavane: a system for declarative visualization of concurrent computations," *J. Visual Languages and Computing* Vol. **3**(2) pp. 161-193 (June 1992).

26. 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).

27. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell, "Monitoring compliance of a software system with its high-level design models," *Proc. 18th Intl. Conf. on Software Engineering*, pp. 387-396 (March 1996).

28. Diana Sidarkeviciute, Enn Tyugu, and Ago Kuusik, "A knowledge-based toolkit for software visualization," *Proc 11th Knowledge-Based Software Engineering Converence, KBSE'96*, pp. 125-133 (1996).

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

30. John Stasko, "Using direct manipulation to build algorithm animations by demonstration," *Proc. CHI '91 Human Factors in COmputing Systems*, pp. 307-314 (1991).

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

32. MIPS Computer Systems, Inc., *RISCompiler Languages Programmer's Guide*. December 1988.

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