The BLOOM Software Visualization System

Steven P. Reiss, Manos Renieris

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

ABSTRACT

BLOOM is a system for doing software understanding through visualization. Software understanding involves asking specific questions about the structure and behavior of a software system and getting accurate answers quickly. BLOOM addresses software understanding by providing a complete system that combines facilities for collecting and analyzing program data, a visual language for defining what should be visualized and how it should be displayed, and a powerful back end that supports a variety of high-density, high-quality visualizations. This paper describes the various aspects of the BLOOM system and our initial experiences with it.

1. Introduction

Software understanding is the task of answering questions and gaining insights about software systems. In some cases it involves gathering broad-stroke information about what a system is doing; in other cases it involves understanding a particular aspect of the system; in still other cases it involves addressing very specific issues such as why was this routine called or what will happen if I change this input.

Providing tools to enhance and facilitate general software understanding has always been difficult. While there have been a wide range of efforts, both in the reengineering community and in the software environments community, few of these efforts have been really successful or led to tools that are in active use today. In particular, tools that use visualization as a means to software understanding have been proposed and demonstrated, but have rarely been incorporated into successful programming environments, and, when they have been, have not been used.

Many efforts have been made to use visualization for understanding [18,29]. A lot of early work went into developing a variety of different visual browsers as part of programming environments. These provided the user with visual, often hierarchical views of the call graph, the module structure, the dependency structure, or the class hierarchy. These were included in a variety of programming environments including FIELD, HP Tooltalk, DEC Fuse, and Sun SparcWorks. However, they were not widely used in these environments and have not been duplicated in current environments except in a rudimentary form as a simple tree view. Similarly, there have been a variety of tools that provide visualizations of system performance. Except for specific visualizations of multiprocessor performance such as those incorporated into MPI or tools such as TotalView, these too have seen limited use and commercial development.

1.1 Objectives

As part of our continuing efforts aimed at software understanding, we looked at the reason why visualization systems, which intuitively seemed so obviously useful, were not being used. Our analysis suggested that the primary reason was that they failed to address the actual issues that arise in software understanding. In particular, they provided fixed views of a fixed set of information and were not flexible enough to let the user address the specific questions or varying issues that actually arise. Secondary concerns were the difficulty in using such systems, both in terms of setting up the data for them and in understanding how to get the desired information out of them, the fact that they often do not have or display the relevant information, and the overwhelming amount of information inherent to a real software system.

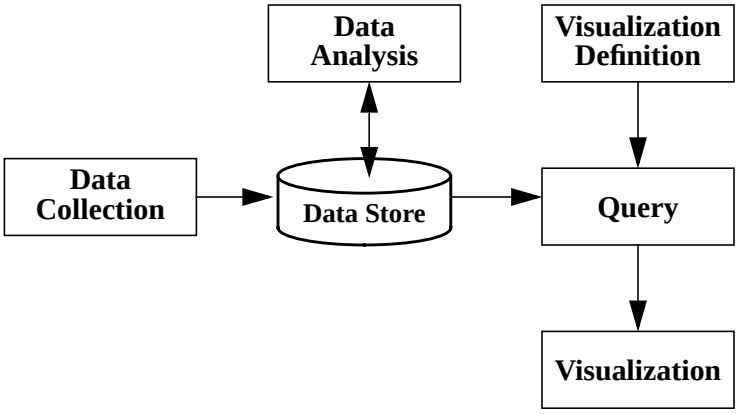

FIGURE 1. Overview of the BLOOM architecture.

To address these problems, we have designed and developed a new visualization system lets the user address specific problems in large software systems. This system, BLOOM, attempts to avoid the pratfalls of previous systems by providing a framework that is designed particularly for software understanding. In particular it provides facilities to:

- Collect a variety of program information including program traces, structural information, and semantic information, all unobtrusively. Software understanding can involves questions about any and all of these aspects of a system and having the data available is a prerequisite for answering such questions.
- Provide an extensible set of different analyses of this information that both summarize and highlight its relevant aspects. Raw data alone is not sufficient for understanding. Instead, one needs to extract appropriate information from this data through different analysis. For effective software understanding a wide range of analyses are needed and new analyses should be easy to add.
- Combine the different analyses and program information in new and potentially interesting ways. Most understanding questions require a combination of data from different sources. Moreover, relationships between the analyses and data sources can provide valuable insights into the behavior of a system.
- Define what analysis or data is interesting to the particular problem at hand and how that data should be visualized. It is important that the user be able to define their particular understanding problem quickly. The easiest way to do this is by specifying what data is relevant to the problem at hand and how this data should be related.
- Visualize large amounts of program-related data in an effective manner. Visualizations of large software systems are generally clumsy because they involve thousands or hundreds of thousands of objects. It is important to design and provide visualizations capable of displaying and navigating through large amounts of data.

Our goal in designing the system was that the programmer should be able to develop a high-quality visualization for a specific understanding problem in five minutes or less. While this goal has not be fully achieved, we feel that the current system goes a long way toward showing that a general purpose visualization system can be used effectively for software understanding.

1.2 System Architecture

Our system is divided into five parts as shown in Figure 1. The first part of the system, *Data Collection*, collects data from both static analysis of the source and through trace data collection. From the source, it builds up a object-oriented database with information about the data types, definitions, references, and files used in the system. The trace data includes a full execution call trace as well as information about memory allocation and synchronization.

Because this trace data is rather voluminous and not very useful in and of itself, the second part of the system, *Data Analysis*, offers a range of analyses that attempt to collapse and provide insights from the trace data. These look at performance, synchronization, memory management, and generalized behavior representations. The results of these analyses are kept as XML files as part of the data store for later visualization.

The third part of the system, *Visualization Definition*, first provides a visual query language whereby the user can quickly specify which source and analysis data is relevant to the particular software visualization question they need to address. This language provides a modified entity-relationship view that offers a unified view of the different data

sources. It lets the user relate the various sources and constrain the information as needed for the particular task. This part of the system then matches the data view specified by the user to the available visualization strategies. It lets the user select from the appropriate strategies and then constructs an appropriate visualization file that both defines the set of queries that should be issued against the various data sources and parameterizes the resultant visualization. It generates a file describing what queries are needed and what visualization should be used.

This file is used by the final parts of the system. The first of these, *Query*, is a generalized visualization engine. This engine obtains data from the various data sources and analyses, merges it as appropriate, and generates a set of objects for visualization. The final part of the system, *Visualization*, provides a variety of 2D and 3D high-density visualizations of these object. The visualization system provides the necessary facilities for browsing over one or more visualizations. These facilities allow the dynamic restriction and grouping of data within the visualization as well as correlations of multiple visualizations. They also let the user change the mappings from the source data to visualization properties such as color and texture.

In the remainder of this paper we describe the current state of the BLOOM system. The next section describes the information gathering phase. Section 3. describes the different analyses upon which visualizations can be built. Section 4. explains the interface for defining visualizations including both the visual query language and the techniques for selecting the appropriate visualization based on the user's data specification. Section 5. then details the visualization engine and the associated browsing facilities. We conclude by describing our experiences with the system and the work yet to be done.

2. Gathering Information

One requirement for successful software visualization is that the necessary data be available. This is difficult in the context of software understanding since the system cannot know in advance what questions the user will want to ask nor what data will be needed to answer those questions. The best that can be done is to gather all possible data either in advance or on demand. However, this in itself is impractical since the amount of such data is very large and much of it is expensive to obtain.

We have attempted to achieve a compromise in the BLOOM system. We gather structural data representing the symbol table and associated information from the source program either directly from the compiler or with a simple source processor. While this does not include detailed semantic information (e.g. detailed control and data flow), it generally provides the information that programmer's are likely to need in software visualization and the information that is likely to be queried about. This information is relatively easy and inexpensive to obtain and can be obtained unobtrusively as the user saves files or compiles.

Obtaining dynamic information is more difficult, both because there is a lot more potential information and because the information is more expensive to collect. Ideally programmers wants to look at a running program without incurring any overhead; moreover, they want to ask multiple questions about that particular run where the questions aren't necessarily formulated until the run is complete or until other questions have been asked and answered. Neither of these ideals is achievable, however, because instrumentation is both selective and expensive. One attempts to minimize the amount and overhead of dynamic data collection by only collecting what is necessary. However, when the questions aren't know in advance its impossible to know what is necessary. Our solution is to collect a broad range of trace data with as little overhead as possible.

The data needed for visualizing program behavior is based in part on previous work in this area. Performance visualizations have been around for 20 years in various forms. (Even the old UNIX *prof* command had a graphics mode to produce histograms.) The FIELD environment, among others, provided dynamic views of the heap, file I/O and performance while the program was running [20]. Various efforts at Georgia Tech and elsewhere have developed a number of different dynamic visualizations including the notion of call dags that we are using [12,29]. More recently, the IBM Jinsight efforts use relatively complete call trace data to provide the user with insights into how the program is running and to address such questions as Java memory leaks [2,17].

2.1 Structural Data

Most of the early work on software visualization dealt with visualizing the structure and organization of large systems, for example showing the call graph or the class hierarchy. Our experience was that such diagrams were helpful for navigation, but, except for some examples from reverse engineering, were not too helpful for software understand-

File:
- File name
- Date last modified
- Dependencies

Dependency:
- Depends on file
- Dependency type

Definition:
- Name
- Definition scope
- Symbol type
- Storage type
- Flags
- Parameters
- Type
- New scope
- Start and end location

Reference:
- Name
- Scope
- Definition
- Flags
- Start and end location

Scope:
- Name
- Parent scope
- Include scopes
- Scope type
- Associated symbol
- Associated type

Type:
- Name
- Type style
- Base type
- Definition
- Flags
- Parameters
- Super type
- Interfaces
- Scope
- Primitive type

FIGURE 2. Overview of the information in the structural database.

ing. However, our previous experiences with trace visualization showed that one often wanted to combine such structural data with the trace data in order to create more meaningful visualizations. As examples, one can look at trace data more compactly by collapsing methods into classes and classes through their class hierarchy, and one can get a better understanding of the execution of a program by seeing how different class hierarchies actually interact at run time.

Thus, our first step was to ensure that the necessary structural information was available. Most previous efforts, including ours, provided such information in the form of a program database that was generated using approximate scanners or information provided by the compiler (.sb files from Sun's C++ compiler for example) [10,14,20,25]. In particular we previously used an in-memory relational database containing relations for references, definitions, scopes, files, calls, the class hierarchy, and class elements.

For our current work, we wanted to both simplify and extend this approach. We moved to a more object-oriented database from a purely relational one. We cleaned up the data and made it more relevant and specific to object-oriented languages, notably Java and C++ which were the primary targets for our visualizations. The resultant sets are shown in Figure 2.

Our previous work had demonstrated that it was important to gather this structural information without bothering the programmer. That is, the information should be gathered automatically and with a minimum of overhead. Our current approach has the programmer define the set of directories and files that constitute the system in question using either an interactive visual tool or a simple text editor. Once this is done, our program database will automatically gather and maintain the database as the specified files or any file in the specified directories changes.

To gather this information accurately and quickly for Java, we took the IBM Jikes compiler and modified it to dump the necessary information from the abstract syntax trees that it produces in the front end. The information is produced on a file-by-file basis and is stored as XML files. These file are then read by our specialized database system that is capable of quickly discarding all previous information from a file and inserting all the new information. The database actually runs the Jikes compiler in its incremental mode so that the compiler shares some of the load of determining what files need to be rescanned and so that updates beyond the first are extremely quick. Our experience with Jikes has been that it is extremely fast and effective.

2.2 Trace Data

The best visualizations for understanding are created by correlating a variety of different data from a single trace. For example, one interesting visualization we have worked with in our previous system involved correlating allocations of objects along with the dynamic call graph. This allowed us to get an overview of how segmented memory was for

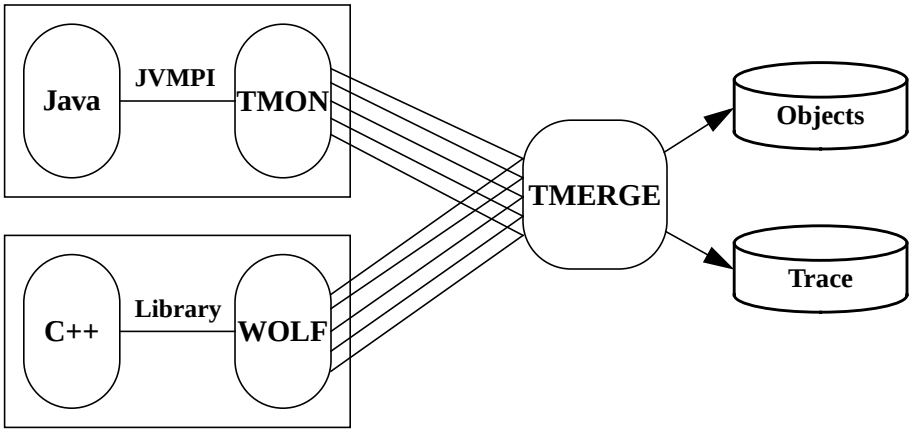

FIGURE 3. Overview of the Java trace data generation architecture.

each portion of a system's execution. Our goal in collecting trace data for visualization was to generate the data so that a number of different analyses could be done after the fact for visualization.

This goal required that our trace collection meet certain criteria. First, it should be as complete as possible. One cannot analyze data that one does not have. Ideally, we wanted to know everything that was possible, including a full call trace of all the active threads, complete control flow information, data access patterns, locking information, input-output behavior, memory utilization data, and performance information. Second, one needs to have a consistent way of relating information from the trace to static program analysis. This means being able to map routines, classes, and code from the trace to the actual program. Third, the data needs to be collected with a minimal impact on the performance of the underlying program. This requires that the execution of the program not be slowed down so much that any interactive portions of it are unusable and that the tracing does not introduce side effects such as synchronizing multiple threads. This requirement conflicts with the first two in that the more data that is collected, the more invasive is the trace procedure. Finally, we wanted to do this in a portable way without any changes to the user's program.

The architecture for collecting grace data is shown in Figure 3. We provide separate facilities for collecting information from Java programs and from C/C++ programs. These share a common back end that produces two files, one describing the objects used in the trace described with enough detail so they can be used directly or related to source objects, and a second containing the sequence of events of the trace. There are multiple connections between the front end and the back end to accommodate multiple threads; the back end takes care of producing a single event stream that merges the different threads appropriately.

2.2.1 Java Trace Collection

Java provides an interface for generating performance information, the JVMPI, that meets most of these criteria. It provides hooks into the JVM that can be used without modifying the user program or the JVM itself. It provides much of the data that we needed for visualization. It is relatively fast as well, with most of the hooks being integrated directly into the JVM (and compiled during JITing) and introduces little extra overhead other than the call to the JVMPI code.

The JVMPI however, does not meet all the requirements. It first reports addresses rather than true object identifiers for objects, classes, threads, etc. Because of garbage collection, such addresses will change over the course of a run and these cannot be used directly for identification purposes or consistently for visualization. It avoids synchronization of the different threads for the most part, but does frequently disable garbage collection during calls which can affect the program's behavior. Most importantly, it only provides a subset information that might be needed. In particular, while it offers the complete call trace, it provides no additional information about control or data flow within the application. This represents a compromise in that offering such information can put very heavy burdens on the trace collection and instrumentation tools. It provides incomplete information about synchronization behavior, failing to report the acquisition of uncontested locks. It also fails to provide any performance data, although this can easily be obtained by gathering run time and real time through other system calls.

Despite these problems, we decided that the JVMPI was the most effective means for unobtrusively gathering trace data for Java. We built a package, TMON, that used the JVMPI to collect information and then pass that information along to our back end for further processing and storage. To avoid synchronizing threads, TMON creates an output stream for each thread. TMON augments the raw JVMPI data by providing periodic events with the run time and the execution time used by each thread. It also provides a facility whereby the user can interactively control the type and amount of trace data collected.

The output streams used by TMON can take one of two forms. They can first be files. Here TMON is given a directory (through an environment variable at start up) and creates new raw trace files as needed. Alternatively, we have implemented a shared-memory communication mechanism. Here TMON uses a shared memory buffer to communicate with the back end to set up a new shared memory buffer for each new thread. Once the buffer is set up, TMON simply adds information to that buffer as needed. The buffer is designed so that a minimum of synchronization is required. This allows the back end trace generation to run in parallel with trace collection.

2.2.2 C/C++ Trace Collection

To extend our framework to handle C/C++ programs we developed a parallel trace collection facility, WOLF, that works with standard UNIX applications. Here we used object-code patching technology similar to EEL [13] to generate calls to a dynamically loaded shared library. We are able to patch not only the user's program, but all the shared libraries that the program uses (including system libraries). Moreover, by simply tracing all calls and returns along with the first argument on a call and the result on a return, we are able to collect trace information similar to that available from the JVMPI for Java. In particular, by looking for calls to the system memory management routines, one can generate events for allocation and freeing; by looking at calls to constructors one can determine the type of allocated objects; by looking at the first argument on a method call, one can determine the associated object; and by looking at calls to the synchronization and thread library routines, one can obtain information about thread utilization and interaction. In collecting C/C++ trace data, we generate periodic events about the execution time and real time used as we do with Java. In addition, our object-code patching library lets us keep track of the number of instructions executed by each thread and we generate events that provide such instruction counts for each call and return to provide fine-grain resource analysis.

The trace data generated by this framework is designed to look and act like the Java trace facility. It generates trace files in the same format as TMON. Moreover, it can run by either generating raw data files or using shared memory buffers.

2.2.3 Generating Trace Data

The output from TMON or WOLF, either from the set of files generated or from the set of shared memory buffers, is processed by the next phase system, TMERGE. TMERGE produces two files from the raw trace data. The first is a database of information about the objects used in the user program. For Java, this information is gathered from the trace records generated by the JVMPI when a class is loaded or when an object is created. TMERGE ensures that the each references to these objects are unique and consistent for the generated trace. This is needed to accommodate the changing identities that are used by the JVMPI. For C/C++, this information is obtained by analyzing the symbol table information that is stored with the object file and shared libraries. This information, aimed at the debugger, allows TMERGE to map addresses in the trace to actual symbols and to provide more detailed information about those symbols such as signatures. The particular types of objects and the information stored with each is shown in Figure 4.

The second file TMERGE generates contains the actual run time trace. This file consists of a simple sequence of event entries, each of which contains the event type and the thread id. Additional information with each entry is dependent on the event type and can be seen in Figure 5. Note that the thread execution time is not stored with each entry, but is stored as occasional trace records indicating additional run time. This follows from the scheme whereby this information is only recorded periodically. It can be used by the program processing the trace data to generate approximated execution times for each call or other events.

2.2.4 Results

The current system is able to generate large amounts of trace data relatively efficiently. For Java, it generates about a gigabyte of trace data for every 10 seconds of JITed execution time. (The exact amount depends on the type of pro-

Thread:
- Thread id
- Total execution time of thread
- Total real time used by the thread
- Total instructions executed by the thread
- Thread name
- Thread group name
- Thread parent name

Object:
- Object id
- Class id
- Array type
- Object size
- Freed flag

Class:
- Class id
- Class name
- Source file name
- Number of interfaces
- Number of methods
- Number of static fields
- Number of instance fields

Method:
- Method id
- Class id
- Method name
- Method signature
- Source file
- Start and end line

Field:
- Field id
- Class id
- Field name.
- Field signature

Monitor:
- Monitor id.
- Monitor name.

FIGURE 4. Information stored for trace objects.

Class Load and Unload
- Class id

GC Start and Stop
- Number of used objects
- Total used object space
- Total object space

Initialize and Termination

Method Entry:
- Method id
- Object id

Method Exit:
- Method id

Monitor Enter and Exit:
- Object id

Monitor Wait:
- Object id
- Timeout or time waited

Object Allocate and Free
- Object id

Thread Start and End

Run time:, Real Time, and Instruction Count
- Time.

Memory Allocation
- Block address
- Block size

Synchronization
- Lock type
- Lock address

FIGURE 5. Summary of trace file entries.

cessing done in the application, particularly the number of calls and allocations.) The overhead of using the system is about a factor of 3 over running the program without JITing or 20-40 with JITing. Moreover, merging the different traces and generating the resultant files can be done in parallel with trace collection without any additional slowdown.

The results for C/C++ are not quite as good at this point. The overhead of trace collection yields a slowdown of 30-50 over raw execution time. While most programs can be run effectively at this speed (user interaction is still acceptable), the overall slowdown is too great to look at long running or computationally intense applications. Also, the amount of trace data that is generated is considerably more than for Java, about 1G for every 1-2 seconds of uninstrumented execution time. Finally, because TMERGE needs to do considerably more work for C/C++ in terms of looking items up using the symbol table in the object file, it is generally run after the fact since it slows the trace collection by an additional factor of two.

Beyond performance issues, the coarseness and incompleteness of the trace data has been a problem. Many of the questions that arise in software understanding require looking at detailed data and control flow information at particular points in the program. Since this information is not available (and would be quite voluminous if it were), we are currently unable to address these problems. This is discussed more completely in Section 6.

3. Data Analysis

While we can easily generate very large trace files, such files are not practical for understanding or visualization. Instead of looking at a hundred gigabytes of raw data, we want to understand and visualize the essential details encoded by that data. Thus, rather than attempting to visualize the trace data directly, we first analyze it in different ways depending on what the user wants to understand. These analysis can either be done immediately after the trace is collected or on demand when the user specifies a particular analysis as the source of a visualization.

To be useful for understanding, a wide variety of different analyses need to be provided. We developed a generic framework for such analyses, TFILTER, and have developed a variety of different analyses that are useful for understanding through visualization.

3.1 Trace Data Analysis

The first set of analyses provide abstractions of the trace suitable for obtaining performance information. These summarize the trace in different ways, grouping together all calls that are similar, and then provide resource (time and memory) utilization information for each grouping. The particular analyses available here include:

- ***Dag Encodings***. One effective abstraction of the dynamic call tree is to view the tree as a directed acyclic graph (dag). This dag is built from the tree in a bottom up fashion by constructing a string representing each node and then creating a dag node for each unique string. The mapping is defined recursively over the tree so that the dags for all children of a node are computed before the dag for the node. The string representing a node is composed of the routine called at that node followed by an encoding of the sequence of calls. By varying this encoding, we can vary the coarseness of the abstraction. The different encodings that can be used are described in Section 3.2. This analysis can be used to provide relatively compact complete performance visualizations of the execution of the system.
- ***String Encodings***. An alternative to encoding the call tree as a dag is to encode it as a single string. This could be done by looking only at the sequence of calls that are done, but such an encoding would be ambiguous. For example the sequence ***ABC*** could represent A calling B calling C or A calling B and then A calling C. In order to make the string unique, we insert markers representing returns in addition to routine names representing calls. Thus, these two alternatives would be encoded ***ABC***^^ and ***AB^C***^ respectively. The raw strings that are generated in this way are generally going to be too long to be considered. However, we again use one of the various encodings described in the next section to provide a more compact description of the string. Note that each thread of control generates its own string. For Java where we have thread names, we allow threads with similar names to be grouped for encoding purposes. Resource information is accumulated here as part of the encoding. This analysis is useful when used with abstract encodings to provide a high level view of the behavior of the system.
- ***Class Encodings***. As an alternative to looking at the overall trace, one can look at particular classes. Here the system form a model of class usage and then augment this model with resource information. To do this the system looks at the sequence of calls made for each particular object and then groups these sequences by class. The set of grouped sequences are then encoded using one of the encodings described in the next section. The sequences can either consider all calls or can be restricted to top-level calls when one is only interested in the external view of the object and not its internal behavior. This analysis is useful for understanding the behavior of each class and then visually picking out unexpected or unusual behaviors.
- ***Package Encodings***. A similar approach could be taken to provide abstractions of the use of libraries, packages, or other entities. While this is easy in principle, the difficulty in practice comes at attempting to define what is meant by a single use of a given library or entity. We use an external specification given in XML to let us specify a set of system abstractions based on libraries and packages and to let the user specify abstractions that are particular to their system. This analysis can be used for understanding how a library is used or for visually identifying unusual behaviors at a more abstract level than classes.
- ***Call Encodings***. Performance data is generally collected by grouping all the calls to a single routine into a single node and collecting statistics for that node. Using the trace data, we provide a simple n-level generalization of this. Two-level call encodings consider accumulate performance data for all instances of routine A calling routine B. Three-level call encoding accumulate data for all instances of A calling B calling C. Here the system actually computes statistics for each tuple, yielding not only total counts but also averages and standard deviations. We also provide a call encoding that, like *gprof* [9], forces the resultant graph to be a tree by constructing artificial nodes to represent cyclic or recursive calling sequences. This analysis is the most useful for straightforward performance visualization, offering a good compromise between detail and abstraction.
- ***Interval Encodings***. Another way of abstracting the data is to consider the program trace in chunks. We call this *interval analysis*. We break the execution down into a number of intervals (e.g. 1024), and then do a simple analy-

sis within each of the intervals to highlight what the system is doing at that point. Within each interval we summarize both the calls and the allocations made by the program. Rather than looking at calls to individual methods, this analysis combines all methods of a given class into a group statistic for that class. Here, in addition to combining the standard statistics from all the calls to that class over the interval, it keeps the average stack depth of routines in the particular class over the interval. It also tracks time spent waiting while executing in a class. These statistics are kept separately for each thread. Allocation information is also kept on a class basis. For each interval the system records the number and size of allocations of objects of each class by each thread. This analysis is useful for getting a high-level visualization of the overall behavior of a system.

- ***Trace Sampling***. Another way of abstracting the trace is to sample it. We provide an analysis that reduces the trace by sampling at different levels of coarseness with intervals ranging from none to each second. The trace that results is designed to look like a valid trace sequences (all calls needed for the sampled routines are present and all calls have returns) and to provide relevant resource statistics. This analysis is good for more detailed visualizations of the behavior of a system, especially single-threaded systems.
- ***Event Sampling***. Here the trace is abstracted into a sequence of events and resources are accumulated for each event separately. Events in this context refer to conditions that trigger computation, for example, input events from the user interface, messages coming from a socket, or remote method calls coming from Java RMI. We are able to trace resource titillations for the processing associated with the event even when that processing is split between multiple threads at different times in the program. External resource files are used to define global events and to let programmers define events that are specific to their systems. This analysis is particularly good for understanding and visualizing the performance of reactive systems.

In addition to encoding the trace data directly, we provide several orthogonal analyses. These include:

- ***Potential Race Conditions***. Here the trace data is analyzed to find places in the trace where multiple threads are executing in methods of a common object (or class) and where synchronization is not used. The result is a set of points in time along with a set of classes, threads, and objects for each point, where there might be problems. These can then be visualized and inspected in greater detail to determine if actual problems exist.
- ***Memory Allocation***. Here the trace data is analyzed to get information about how memory is used over the course of the application. Analysis can be done in one of two modes. In object-mode, the system reports the lifetime of each object over the course of the program. This lets us produce a visualization of memory titillation over time. In class-mode, the analysis gathers statistics about objects by class. Here it provides statistics on the size, lifetime, number of garbage collections, and number of moves of the objects for each class. This is useful for understanding the overall memory behavior of a system and visually spotting anomalies or problems.
- ***Lock Analysis***. This analysis attempts to find potential deadlock situations. It provides a table of all the nested locking sequences that are used throughout the application. By visually or automatically inspected conflicting lock sequences, the programmer can detect possible deadlocks.

3.2 Encoding Sequences

Many of the above encodings rely on the ability to effectively encode a sequence of items. There are two different approaches that can be taken to such encodings. The first is to provide a more concise but exact representation of the sequence. This typically involves some form of compression where repeated subsequences are only output once. The second approach is to provide an approximation of the sequence. This could be done by ignoring repetition counts greater than a certain value or by converting the sequence into a finite state automata that accepts not only that sequence but other, similar sequences.

The approximation of sequences is particularly useful when one is attempting to identify similarities or, for our case, when one is attempting to encode a group of related sequences using a single encoding. Several of the encodings of the previous section, for example, looking at all call sequences for a class or the different call sequences of a single routine, attempt to group such sequences. In these cases an approximation is often better for understanding or visualization.

Our analysis framework provides a variety of encoding techniques. These include both approximations and exact encodings. Some are simple and others are more complex. The correct one to use will depend on the particular program execution encoding method that is being used and the reason for the doing the encoding, i.e. what exactly needs to be visualized or understood. The particular encodings include:

- ***Run-Length Encoding***. The simplest approach that we provide (other than no encoding) is to find immediate repetitions and replace them with a count followed by the item being repeated. Thus, the string ***ABBBBBBBBBCDDDBCDC*** is encoded as ***A 9:B C 3:D BCDC***. This is very fast and often quite effective in terms of compression.

The run-length encoding algorithm also takes a parameter k indicating the longest repetition to be flagged. Any repetition of size longer than k will look the same. Thus, if k = 3, the above sequence would be encoded as ***A *:B C 3:D BCDC***. This provides a degree of abstraction.

- ***Grammar-Based Encoding***. An alternative to simple run-length encoding that finds immediate repetitions, it to find all common subsequences and to encode the string using a grammar where each such subsequence is represented by a rule. One such approach is the Sequitur algorithm [15,16]. This algorithm builds a context-free grammar representing the original string with the property that no pair of adjacent symbols appear more than once in the grammar and every rule is used more than once. The standard Sequitur algorithm provides an exact encoding of a single sequence. Our implementation of the algorithm provides for encoding groups of sequences. We have also modified the basic algorithm in two ways. First, we find sequences and represent them using much as we do in run-length encoding. This lets the grammar representation reflect an abstraction rather than the exact string and tends to identify and bring out cyclic behavior. Second, we modified the algorithm to produce grammar rules that are balanced. This is useful for encodings such as the string encodings where a single sequence of calls and returns is generated for each thread of control.
- ***Finite State Encoding***. An alternative method of encoding a sequence is to build a finite-state automaton that recognizes that sequence. Here one can vary the accuracy and precision using different means for constructing the automaton. At one extreme, one can build an automaton to represent exactly the sequence or sequences that need to be defined. This can be done trivially by just building a long sequence of states, one per input symbol. At the other extreme, one can build an automaton with just one state and have all transitions on valid inputs be self arcs to that state. Neither of these approaches is useful, however. What is needed is an automaton that provides good intuition of what sort of sequences are valid. In other words, the automaton should in some way generalize the sequence to show us its underlying structure. Where multiple sequences are provided, the automaton should generalize from the set of sequences.

  There has been significant previous work on inferring finite state automata from input sequences. Most of this work has concentrated on the use of positive and negative examples, i.e. providing a set of inputs that are valid sequences and a set of inputs that are not. In our case, we only have positive examples which makes the problem more difficult. We offer two different finite state encoding methods [24]. The first assumes that each transition leads to a new state and then attempts to merge states based on the set of transitions that extends from them. This is a modification of the algorithm proposed in [4]. The second approach assumes that all transitions for a given token go to the same state and then does a statistical comparison of the output paths associated with each different input path to a state to split the state when differences become significant.

  These two approaches do a reasonable jobs of modeling the input sequence or sequences. Example of the merging approach are shown in Figure 6. The first automaton shows the how the solution class is used in a knight's tour program. The second looks at the sequence of calls made by the main search routine of this program shown in Figure 7. Figure 8 then shows the same sequences encoded using the splitting algorithm. The splitting algorithm tends to provide more detail which is often what is desired. However, there are cases where it provides too fine a resolution..

3.3 Summary

The use of a variety of analysis of the raw trace data makes the visualization of the behavior of a program practical. The different resource-based analysis have been used extensively to provide insight into the performance of a variety of programs. The encodings of class utilization have been used both to understand system behavior and to identify unexpected behavior and hence problems. We have used each of the methods to provide different visualizations.

Moreover, we have demonstrated that the general framework for doing analyses for trace visualization is both practical and achievable. We are able to add a new analysis is less than a day of work. Once the analysis is present, it becomes available to the rest of the system and can be used for a variety of different visualizations.

4. Combining Analyses

The combination of structural data along with the above analyses of trace data provide a broad foundation for software visualization. The next step in a visualization framework is to let users select the data that is relevant to their particular problem. This step must at least let the user understand and select from the different analyses. Just as important, however, it must let the various data sources be combined in arbitrary ways.

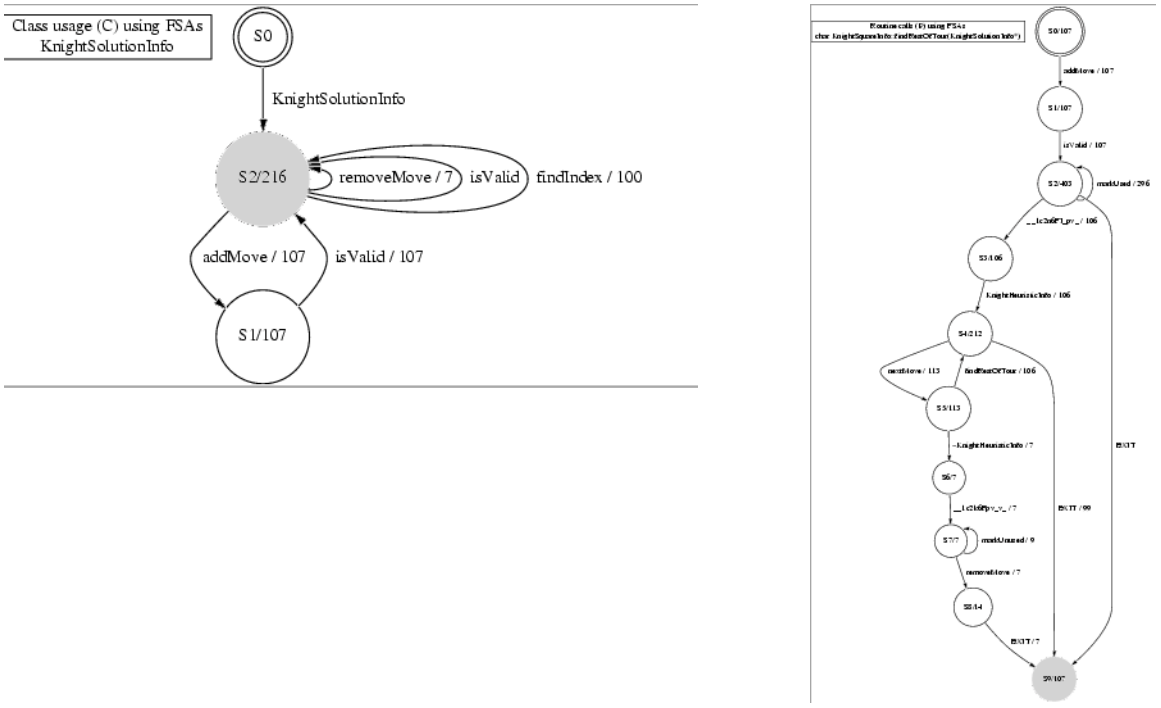

FIGURE 6. Example automata generated by node merging.

```
Boolean
KnightSquareInfo::findRestOfTour(KnightSolution sol)
{
        KnightSquare sq;
        KnightHeuristic heur;
        Integer i;

        sol->addMove(row_number,column_number);
        if (sol->isValid()) return TRUE;

        for (i = 0; i < num_moves; ++i) legal_moves[i]->markUsed(this);

        heur = new KnightHeuristicInfo(num_moves,legal_moves);
        while ((sq = heur->nextMove()) != NULL) {
                if (sq->findRestOfTour(sol)) return TRUE;
         }
        delete heur;

        for (i = 0; i < num_moves; ++i) legal_moves[i]->markUnused(this);

        sol->removeMove(row_number,column_number);

        return FALSE;
}
```

FIGURE 7. Search method code that is encoded by the automata.

4.1 The Data Model

BLOOM does this through a data manager that provides access to data in a uniform way and that lets new objects be created from existing ones. The data manager is based on a entity-relationship based data model and query system.

The model starts with a notion of domains of data to provide a common basis for multiple data sources. Each domain represents a basic type of data such as integer or string. Domains can be arranged hierarchically with lower levels of the hierarchy acquiring additional semantics. Thus a subdomain of string is *filename* which represents a valid file-

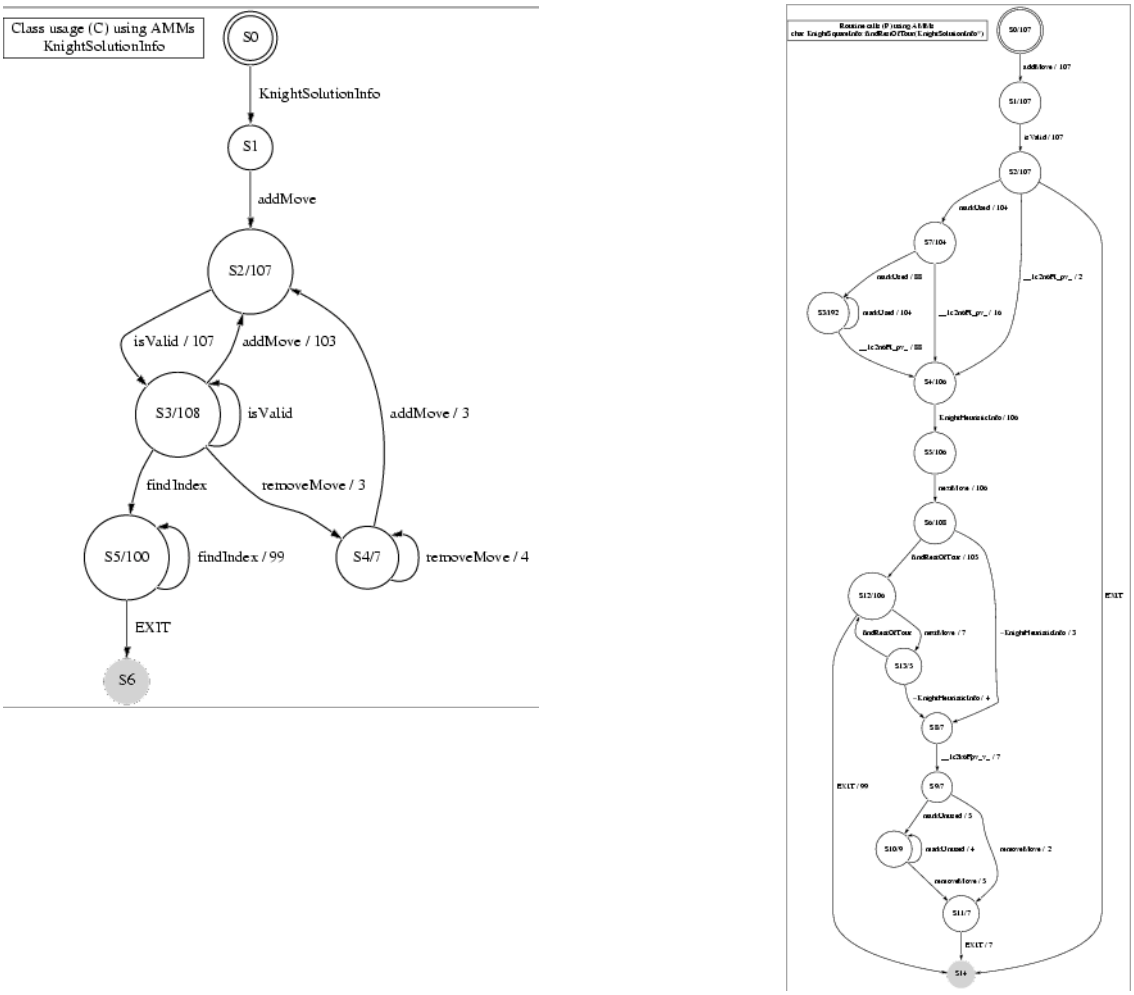

FIGURE 8. Example automata generated by node splitting.

name string and a subdomain of integer is *line* which represents a line in a file. Domains are also used to represent references, either direct pointers or indirect unique identifiers, to a particular entity.

The data model uses entities to represent the base data from the different data sources. Each data source is viewed as a set of homogeneous entries. Each entry consists of a set of fields where each field has a fixed domain. This model is general enough to handle all the data sources previously described, although some of the encodings need to be viewed using multiple entities. The data manager also permits constraints to be associated with any entity. Such constraints restrict the entries associated with that entity. XML-based definitions are then used to define each data source in terms of entities and fields.

A sample from a model definition file is shown in Figure 9. These files let the user define new domains as references to existing domains, as subdomains of existing domains, or as arrays of existing domains as shown by the top definitions in the figure. The user can also define new entities representing information from a data source. Each data source can have multiple such entities associated with it. Each entity specifies both how to access the data source and the fields associated with the entity. The example shown represents the result of trace analysis providing statistics for all allocations grouped by class. The analysis is obtained from an XML file which is produced by running the TFILTER program, and each entity corresponds to a CLASS element in the XML file. This information is described in the ACCESS element of the domain definition. The various fields of the entity are each given a name and a domain and rules for obtaining that field from the source XML file either from an attribute or from a related element.

The data model supports a number of different types of relationships between these entities. Relationships can be explicit through links or IDREFs in the XML file, implicit through fields in one entity that have domains that match

```
<DOMAINS>

<DOMAIN NAME="TfileClassRef" TYPE="REF" REFER="TfileClass" />
<DOMAIN NAME="TfilterThread" TYPE="NAMED" DOMAIN="ThreadName" />
<DOMAIN NAME="TasteTypeRef_Array" TYPE="ARRAY" BASE="TasteTypeRef" />

<DOMAIN NAME=”AllocClass” DESCRIPTION=”Allocations by class” TYPE=”ENTITY”>TTIP=”Allocation statistics organized by
                                class”>
    <ACCESS TYPE=’XML’
        RUN=”$(TFILTER) -d $(TRACEDIR) -AC”
        FILE=”$(TRACEDIR)/AllocClass.tf”
        ELEMENT=”CLASS” />
    <FIELD NAME=”class” DOMAIN=”TfileClassRef” XMLATTR=”ID” />
    <FIELD NAME=”number_of_objects” DOMAIN=”int” XMLATTR=”OBJECTS” GROUPING=”Sum” />
    <FIELD NAME=”number_freed” DOMAIN=”int” XMLATTR=”FREED” GROUPING=”Sum” />
    <FIELD NAME=”average_span” DOMAIN=”double” XMLATTR=”TIME” GROUPING=”Sum” />
    <FIELD NAME=”average_moves” DOMAIN=”double” XMLATTR=”MOVE” GROUPING=”Sum” />
    <FIELD NAME=”average_gcs” DOMAIN=”double” XMLATTR=”GC” GROUPING=”Sum” />
    <FIELD NAME=”stddev_span” DOMAIN=”double” XMLATTR=”TIMESD” GROUPING=”Sum” />
    <FIELD NAME=”stddev_moves” DOMAIN=”double” XMLATTR=”MOVESD” GROUPING=”Sum” />
    <FIELD NAME=”stddev_gcs” DOMAIN=”double” XMLATTR=”GCSD” GROUPING=”Sum” />
  </DOMAIN>
```

FIGURE 9. A sample entity definition. This definition represents statistical information about allocations by class. It is obtained from an XML file using the specified access rules and XML information specified for each field. The *class* field contains a reference to a *TfileClass* object.

keys in another entity, or user-defined by associating values in one entity with values in the second. Relationships can be used to restrict the entries in each entity by only considering those where a valid relationship exists. Relationships can also be used to create new virtual entities and more complex relationships. Virtual entities can be formed by combining existing entities using any of the relationships. New relationships can be formed by combining sequences of relationships, either directly or using various forms of transitive closure. The implementation of the data model and query system is part of our visualization back end described in Section 5.

4.2 The User Interface

The user interface for defining software visualizations is essentially a query language for this data model. This interface was designed to make it easy for the user to select and combine the appropriate data sources for a broad range of problems. Its particular requirements included:

- Providing a query language over multiple data sources. If the available data is viewed as a (possibly virtual) database, then asking questions over this data is essentially querying that database.
- Providing independence from the underlying data formats and structures. Programmers are already burdened with the complexities of the language, environment, and the particular system. It is not reasonable to make them learn the wide variety of formats that will be used in collecting and analyzing the data that is needed for understanding.
- Allowing the easy addition of new data sources. As systems get more complex, new data sources become relevant. As software understanding problems become more complex, more data sources and data analyses are needed to address them. Any system should be able to easily accommodate such changes.
- Supporting a variety of different data formats. Some of the data will be available in relational databases. Other data might be available as linked objects, for example symbol tables or abstract syntax trees. Other data might be available in XML format, for example program analysis data through GXL [11] or the results of various analysis tools. Other data will be available dynamically through requests of existing systems, for example configuration data can be obtained dynamically using appropriate commands to CVS or RCS.
- Providing full query capabilities. The questions that arise in software understanding can be quite involved and the data sources can be quite complex. In order to get the proper answers from these sources, the front end must provide a powerful and flexible query language.
- Making common queries simple. While many understanding problems are complex and unique, a significant number of the queries that need to be addressed occur repeatedly. Such repeated queries, no matter how complex they are, should be easy to ask.
- Offering an intuitive and easy to use interface. Programmers are harried enough and are loathe to learn or use a new tool. A visualization tool that is not easy to use will simply not be used. Programmer’s don’t yet understand the potential value of visualization and will not take the time to learn a difficult or obscure tool. A consequence of this is that ideally the front end should be integrated into an existing programming environment.

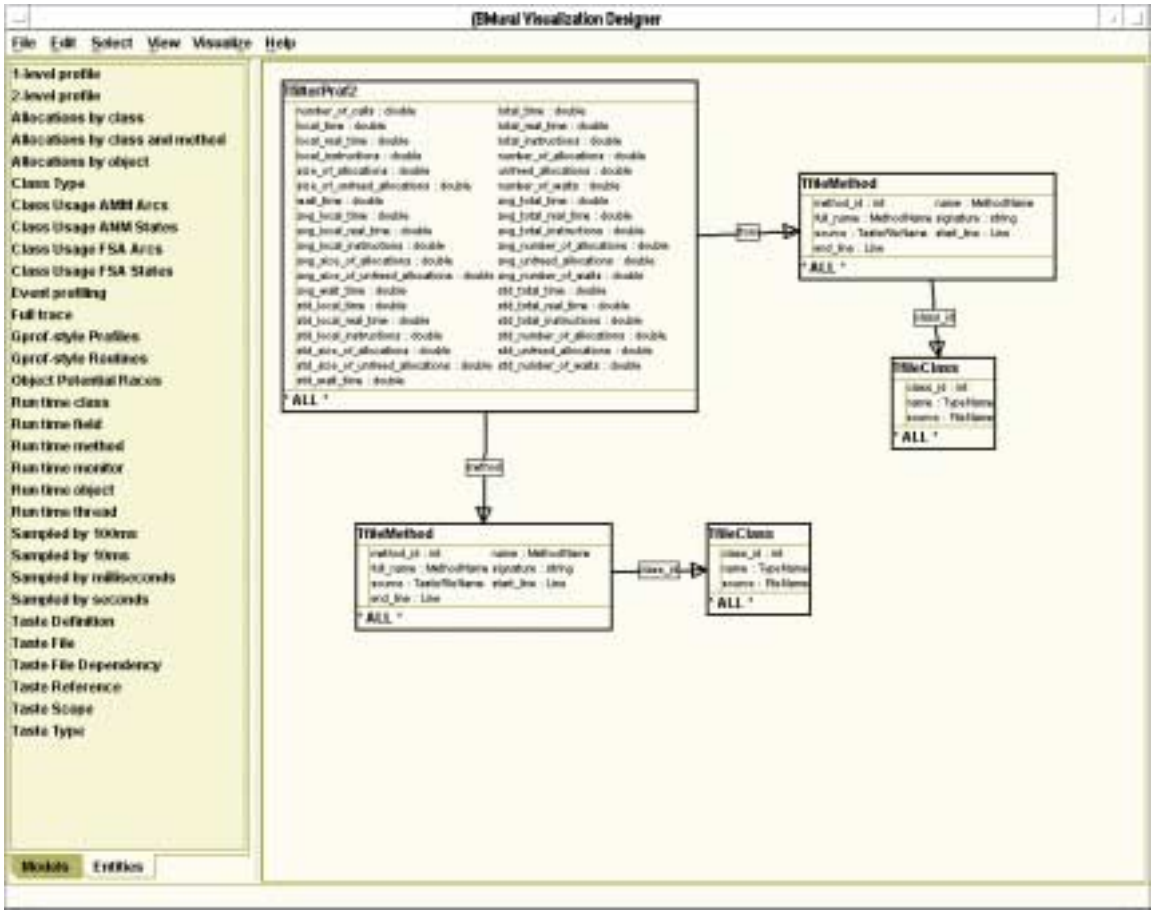

FIGURE 10. Snapshot of MURAL showing the setup of a query that correlates 2-level call analysis (statistics about routine A calling routine B) with static information about the two routines and their classes.

- Providing easy integration with a visualization system. The front end should let the user define the question that needs to be addressed and, at the same time, should let the user define what the visualization should look like.

The result of our efforts is reflected in the MURAL front end. MURAL provides a powerful visual query environment that attempts to meet the above criteria. This front end matches the underlying data model. An example of the MURAL interface is shown in Figure 10. The query shown here combines profiling statistics accumulated for each pair of calling-caller routines along with static information about those routines and their classes.

4.2.1 Entities and Relationships

MURAL uses boxes to represent entities. At the top of the box is the name of the entity. Entities can be restricted to only be part of the query and not be available for visualization. This is indicated by graying out the name here. At the bottom of the box is a display of any restrictions that apply to the entity. In between these, the box lists the data fields that comprise the entity and that will be available for display For each field it lists the name and type. Internal fields, those containing pointers, are not displayed here since they are not directly available for visualization. The user selects entities from the list of data sources on the left of the drawing area.

Relationships are represented as labeled arcs between the entities. Simple relationships, those that are implicit in the data model either through direct pointers or that can be accessed through indices, can be created by simply connecting two entities. If the user starts drawing an arc in one entity and ends in empty space, the system will provide a series of dialog boxes that list the available relationships and, once the user selects the appropriate one, will add both a new target entity and the corresponding relationship. This is a convenient means for exploring the data space. More complex relationships can be created by selecting the entities to be related and then selecting *Combine* from the *Edit*

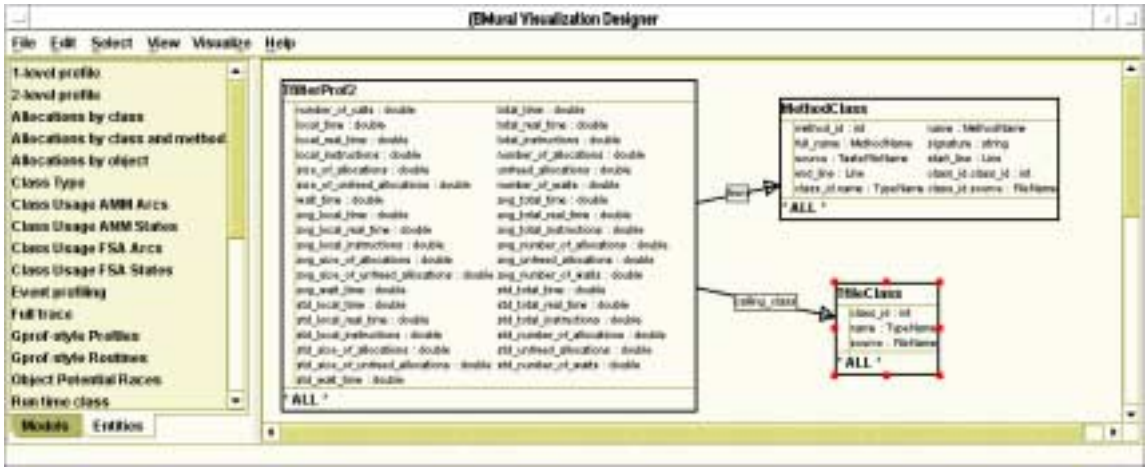

FIGURE 11. MURAL supports grouping operations. Here the *Method* and *Class* entities that are separate in the original query are combined into a single entity for the *from* relationship while the *method* and *class_id* relationship along with the *TfileMethod* entity are combined to form the *calling_class* relationship.

menu. At this point the system will provide a dialog box that lets the user relate fields of the source entity with fields of the target entity that should match for the relationship to hold. The user can also create multiple relationships between two entities. When this is done, the user is prompted as to whether these should be merged into a single relationship, hence representing an AND of the conditions of the original relationships, or if they should be left alone, representing an OR of the relationship conditions.

4.2.2 Combined Entities

To facilitate complex queries and the definition of visualizations, MURAL introduces several concepts. The first is the notion of *combination*. Entities and relationships can be combined into new entities or relationships. Two or more entities along with the relationships that connect them can be combined into a single entity. The result is similar to taking a join of the underlying sets using the relationship as the join expression. For example, Figure 11 shows the result of selecting the entities *TfileMethod* and *TfileClass* entities for the *from* relationship along with their accompanying *class_id* relationship and combining them into a single entity labeled *MethodClass*.

Combined entities are useful in a variety of ways. Their primary use is to let the user define new data objects as logical combinations of existing objects based on the relationships among the existing objects. This is necessary for defining what should be visualized in our overall framework. Second, they make the specification of complex queries simpler by providing levels of abstraction. A combined entity essentially is an abstraction of the various entities and relationships that went into it.

Combining entities also provides the means to define queries involving transitive closure. MURAL treats the case where two identical entities are combined using a single relationship as special. It asks the user to specify the range of times, from zero to infinite, that the relationship can be followed. This provides a natural and intuitive way of both expressing and denoting transitive closure operations in the visual query. Note that transitive closure is essential to queries in the software domain since it is needed for asking about class hierarchies, scope containment, and call containment (A calls routines that eventually call B).

4.2.3 Combined Relationships

Just as multiple entities can be combined using the relationships between them to form new entities, a series of relationships and their intervening entities can be merge to form a combined relationship. The implication here is that the data in the selected entities is not needed, just the relationship. For example, Figure 11 also shows the effect of combining the *method* and *class_id* relationships along with the intermediate *TfileMethod* class to create a *calling_class* relation.

Combined relationships provide many of the benefits of combined entities. They simplify both the visual image of a query and its definition by providing a level of abstraction. They let the user make explicit what relationships should

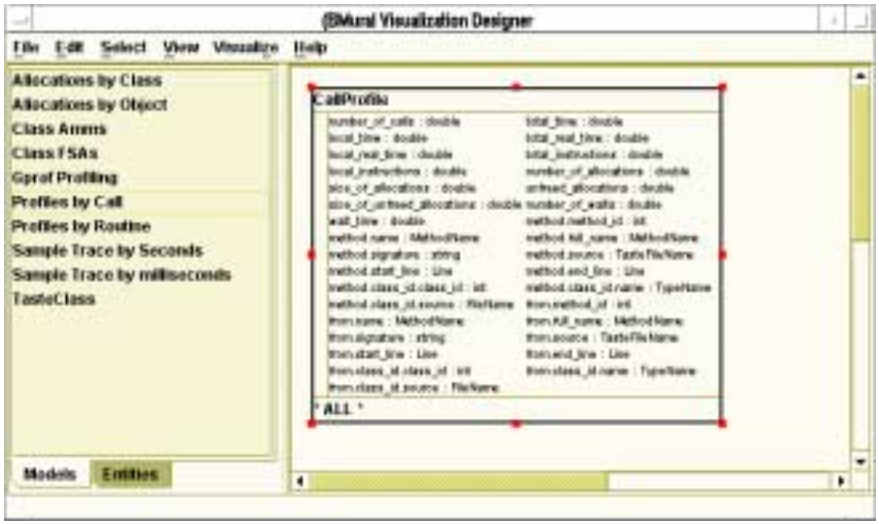

FIGURE 12. The call-profile query of Figure 1 shown as a submodel. This view also shows the current set of available submodels which cover the most common visualizations that a user might want to see.

be the focus of the visualization that is being defined. Also, they make it easier to use logically AND together multiple relationships.

4.2.4 Restrictions and Fields

The second concept that MURAL uses to facilitate complex queries is the notion of restrictions. The user can define arbitrary restrictions on each entity in the form of Boolean expressions. These restrict which objects in the set implied by the entity actually are represented by the entity on the display. Restrictions can be defined either on base or combined relationships. Note that by defining restrictions on combined relationships, the user is actually defining more complex, cross-entity conditions. This is another way that combined entities allow easier and more intuitive definition of sophisticated queries.

The third concept that MURAL uses is to let the user restrict and modify the set of fields that an entity contains. Individual fields can be removed either from the visual display of the query or from consideration in the construction of the visualization or both. This can be used to save screen space and to make large and complex entities more understandable while building queries. It is also useful in defining a restricted and possibly more relevant set of fields for a particular visualization. This is needed to let the user specify what is important in the visualization that is ultimately being defined.

In addition to limiting the set of fields, MURAL lets new fields be defined for an entity. This is done by providing a new field name and an expression over the other entity fields that provides the new field's value. Computed fields let the user define new data elements that should be used in the visualization. They also allow the stepwise definition of restrictions and relationships since the new fields can be used to define new restrictions and relationships.

4.2.5 Submodels

In addition to making it possible to construct complex queries, MURAL attempts to make it easy to ask common queries. Once a query is constructed, it can be saved as a submodel. Then the user can instantiate such submodels as needed when building the query. When the most common queries are stored as submodels, such queries can be invoked with a simple series of clicks. For example, Figure 12 shows the query of Figure 10 defined as a submodel. The initial display provided by MURAL shows the user the set of available submodels to encourage the user to select an existing model if one is relevant.

4.2.6 Expressive Power

The query language that results is at least as powerful as the relational algebra or calculus. The basic operators that need to be provided to accomplish this are product, select, and project. The query language handles products through relationships in general. While most relationships express restricted products or joins, the use of arbitrary, field-based

```
<VIZDATA>

<DOMAINS>
<DOMAIN NAME="Color" COLOR="TRUE">
      <USE DOMAIN="int" />
      <USE DOMAIN="long" />
      <USE DOMAIN="float" />
      <USE DOMAIN="double" />
      <USE DOMAIN="string" />
      <USE DOMAIN="enum" />
      <USE DOMAIN="flag" />
    </DOMAIN>
</DOMAINS>

<VISUALIZATIONS>
<VISUALIZATION NAME="SpiralFlavor" DESCRIPTION="Spiral" >
       <REQUIRES>
         <REQUIRE>
           <ENTITY NAME="Object">
             <FIELD NAME="FromTimeIndex" DESCRIPTION="From Time" DOMAIN="Index" MAP="Range
             <FIELD NAME="ToTimeIndex" DESCRIPTION="To Time" DOMAIN="Index" MAP="RangeMap"
             <FIELD NAME="ZFromIndex" MAP='LevelMap' DESCRIPTION="From Height" DOMAIN="Ind
             <FIELD NAME="ZToIndex" MAP='LevelMap' DESCRIPTION="To Height" DOMAIN="Index"
             <FIELD NAME="WidthIndex" MAP='WidthMap' DESCRIPTION= "Width" DOMAIN="Index" O
             <FIELD NAME="ColorSpec" DESCRIPTION="Color" DOMAIN="Color" OPTIONAL="1"/>
            <FIELD NAME="TextureSpec" DESCRIPTION="Texture" DOMAIN="Texture" OPTIONAL="1"
           </ENTITY>
          </REQUIRE>
         </REQUIRES>

</VISUALIZATION>
</VISUALIZATIONS>
</VIZDATA>
```

FIGURE 13. Sample visualization definition. The first part shows the definition of visualization domains as sets of possible data domains. The second part provides a description of our spiral visualizations first by specifying the data model to be visualized and second by defining the parameters of the visualization.

relationships allows the definition of arbitrary products if these are needed. Selects can be done either through relationships which imply a selection of the corresponding product, or through restrictions on the entities. Projects are done by limiting the set of fields in the entity or by eliding entities or relationships when doing a combination operation. Note that this limitation does not affect the fields that are available for restriction or for relationship definition.

The query language offers additional capabilities that are only found in extended relational databases, logic database, or object databases. The ability to treat the combination of a class with another instance of the same class using a relationship as a transitive closure provides the ability to define transitive closure-based queries that cannot be defined in a relational framework.

4.3 Specifying Visualizations

Once users have selected the data of interest by choosing an appropriate set of entities constrained by a combination of relationships and restrictions, they will want to get a visualization of the result.

MURAL provides an internal mechanism that finds appropriate visualizations and offers the user an ordered choice of what it deems appropriate for viewing the specified data. This mechanism starts with definitions of each of the available visualization strategies. These definitions provide MURAL with a model of what type of data are used in the visualization, a brief description of the visualization, the set of parameters that control the visualization, and the set of visualization fields to which the data described in the user's query can be mapped.

An example of such a definition is shown in Figure 13. The definitions are in two parts. First the definitions specify the set of data domains that are used in the various visualizations. Each of these is defined as a set of possible source domains.

The second part of the figure shows the definition of our spiral visualization. The first part of this definition describes the object model. This particular visualization is driven by a set of objects each of which must have a from and a to time fields. These fields must have a data type that corresponds to the visualization domain *Index*. In addition, there are optional visualization fields for the height, width, color and texture associated with the visualization More complex models allow the specification of multiple object types and relationships between these types.

When the user finishes building the data model, he clicks the visualization button in MURAL. At this point the system attempts to match the user's model with each of the visualization data models. The matching attempts to associated each entity with a visualization object type in such a way that all required fields are present and as many of the optional fields as possible are available. Multiple entities can be mapped to a single visualization object type.

The matching process is actually more complex than this. In doing the matching, not only does MURAL consider the explicit user entities, but it also considers possible operations on those entities including:

- Combining multiple entities using intervening relationships as is supported by the query language.
- Combining multiple relationships into a single relationship as is supported by the query language.
- Merging two identical entities into a single entity using a union operation. This is needed to handle self-loops in the visualization model since such loops are not directly representable in the query model.
- Omitting entities and relationships by not associating them with any visualization entity or relationship.

The system uses heuristics to assign a cost to each of these operations and a value to each of the matching fields. The result of these values is used to sort the list of potential visualizations for the given user data model. This list is pruned by eliminating strategies that are not within 50% of the best selection or within 90% of the best selection for a given visualization.

Once the system has provided a list of potential visualizations and the user has selected one of these, MURAL generates the information needed for visualization. It does this by generating an XML file that describes both the query that is needed to obtain the data from the various data sources and the visualization that should be used along with initial settings of the various visualization parameters and fields. Once the file is complete, MURAL will run the visualization back end described in the next section to actually provide the visualization.

4.4 Related Work

The visual query language itself is built on top of numerous other efforts aimed at providing visual database interfaces. These start with Query by Example [32] which defined a table-based interface. Other examples include PSQL [26], various Entity-Relationship query languages [8,27,31], SeeQL [30], G+ [5], DOODLE [6], and VISUAL [1]. Our query language differs from these both in its features and its focus. Most of these languages are aimed at specifying exact queries. Our language is oriented toward specifying a broad collection of data for visualization with the assumption that any specific data element will be isolated as part of the visualization process.

The systems that are closest to our work include Query by Diagram (QBD) [27]. QBD uses an entity-relationship base to construct relational queries, with both entities and relationships containing information. It starts with the overall ER database schema and lets the user select paths within that schema. It provides bridges to allow connections based on arbitrary conditions. It can translate loops into generalized transitive closure. Our system, on the other hand, does not assume an initial entity-relationship schema or even use a true entity-relationship database model. Instead, it lets the user connect entities that represent data sets dynamically. Our relationships are essentially links rather than the traditional ER-type relationships. Moreover, QBD has no equivalent to our use of combinations of both entities and relationships.

We also used experiences from our previous work where we used a much simpler query visual language based on a universal relation assumption that became too bulky and complex for specifying relevant queries [23]. Rather than pursuing this previous model, we used the lessons we learned in making the MURAL query language both more powerful and simpler.

The handling of multiple data sources builds on the broad body of work in federated databases. Our work is different in that it is designed to handle a wide range of data sources beyond databases as well as database-related sources. At the same time, we have not done a lot of the optimization work that previous systems do to make federated queries run fast. The front end described here assumes that there is an efficient back end to obtain and visualize the data. We have a start at that back end, but not one that handles arbitrary queries very efficiently.

Finally the work on finding appropriate visualizations is based loosely on the various efforts at finding appropriate data display formats for different types and combinations of data.

4.5 Experience with MURAL

The MURAL system goes a long way toward meeting the requirements for software visualization we previously defined. It provides a common model of multiple data sources that hides the nature and structure of the source and its data from the user and makes it easy to accommodate new or additional sources. It provides a full query language oriented toward specifying the data and relationships that should be visualized. This language makes use of the notion of combination to let the user quickly build new data objects and relationships that should be the basis for the desired visualization. The system uses submodels to make specifying common visualizations simple. Moreover, the language is designed to be intuitive and easy to use. Finally, the whole framework is designed so that the system can select the appropriate visualization and thus integrate easily into the overall visualization framework.

Using MURAL we have demonstrated that it is possible to define a wide range of different visualizations quite easily using the front end. Moreover, these visualization typically interrelate data from several different sources, demonstrating the utility of a general purpose query language. The visualization range from views of the stack during execution to views of allocations by class and method to views of the statics structure of the file and class dependencies of a system.

The problems we have had with the query language are two fold. First, restrictions and user-defined operations are still a little difficult to use and are not adequately reflected in the visual representation. Second, we have found a need to be able to specify aggregate queries. Right now aggregation is available in the back end as a feature of the visualization engine. It seems logical and some users have asked that it also be available as part of the query language.

We have not had any problems with new data sources and have been able to match all our available data to the entity-relationship form that MURAL provides. Adding new data sources to MURAL itself is relatively trivial. Adding new access methods to the back end to support these sources typically takes one to two days of programming.

The major problem to date with MURAL itself has been the heuristics used in selecting the appropriate visualization. Our experience here is that the best visualization is always one of the ones chosen as appropriate, but is generally not the one that is chosen as most appropriate. We are currently looking into determining a better set of heuristics and a better set of values for selecting the best visualization.

5. Visualizing the Result

Once users have defined what information is relevant to their immediate software understanding task, they need to see the information in an understandable and meaningful manner. Using visualization here puts a number of requirements on the underlying visualization system. Software understanding implies being able to view a software system from many perspectives and at many levels of detail. It means getting overview information at times and answers to very specific questions at other times. It requires the ability to ask a wide range of questions and get the answers to those questions reasonably fast and efficiently. All this requires that the visualization system provide:

- *Multiple visualizations*. No one visualization strategy can handle displaying information about the wide range of questions that software understanding implies. Instead, different types of visualizations and different visualization strategies are going to be most suitable for different types of data. The back end must offer a range of such strategies so that the one most appropriate to the question at hand can be used.
- *Browsing facilities*. The questions involved in software understanding are generally broad at first and then narrow down to specific issues. For example, one might want to know why some operation takes so long, and then narrow this down to why a particular routine takes so long and then why a particular line in that routine is executed so frequently. Software visualization should facilitate such searches. Moreover, specifying very specific questions for visualization is often difficult; it is generally much easier to specify a more general question and then use browsing facilities to isolate the particular data in question.
- *Coordinated views*. Many of the issues that arise in software understanding involve relationships between different analyses and program information. While such relationships can often be expressed in a single visualization, it is often easier to use multiple visualizations with one being used to navigate in the other. The visualization system must allow and facilitate such coordination.
- *Multiple data sources*. Finally, software visualization requires acquiring and correlating data from multiple data sources to answer the particular questions that arise. The back end system must be able to gather and combine such sources and then organize the resultant information for visualization.

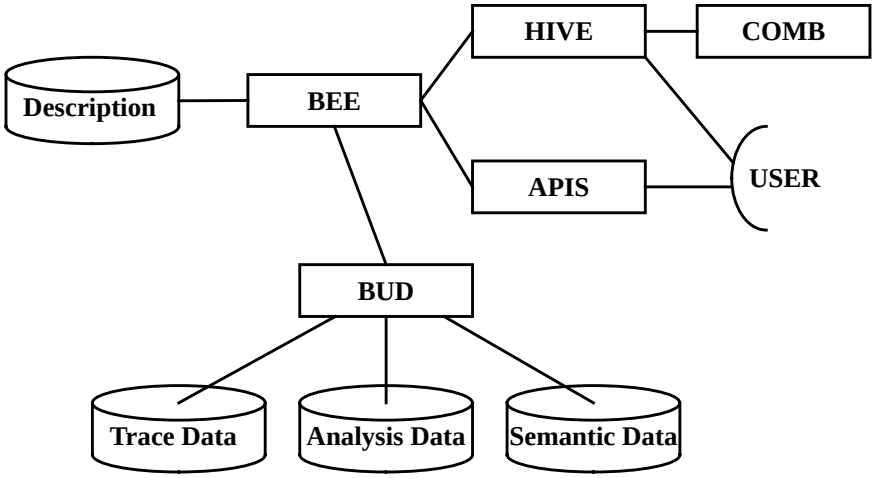

FIGURE 14. Overview of the back end visualization architecture.

To meet these requirements, we developed a general purpose and extensible visualization back end. This subsystem is composed of the five components as shown in Figure 14. The central component, BEE, controls the visualization process. It reads a visualization description file generated by the MURAL front end. This file describes both the data that is needed for the visualization and the visualization to be used to display the data. BEE uses a data manager, BUD, to actually access the data from the various data sources that are available. BUD actually implements the data model and query system described in the previous section. The back end next provides a user interface, APIS, that lets the user specify how to correlate the data with the visualization and offers the appropriate browsing and view coordination facilities. Finally, it uses a common visualization back end, HIVE, to actually create the appropriate visualization. HIVE is a generic visualization engine that can support a wide variety of different visualizations. The actual visualization implementations are provided as plug-ins through the COMB package. We consider these packages in more detail in the next sections.

5.1 HIVE and COMB

HIVE provides a generic framework to support a wide variety of 2D and 3D visualizations. It provides facilities for managing data objects representing information to visualize, for displaying and updating the visualization window, for letting the user pan, zoom, and otherwise manipulate the resultant visualization, for correlating positions with objects displayed in the window, for managing visualization parameters, and for supporting the actually drawing that the different visualization methods require. Its development was based on our previous experiences in developing a generic visualization back end [21-23].

HIVE offers two different interfaces. The first, used by BEE, is designed to let a client program easily create and use visualizations. This interface provides the abstractions of a visualization window and a visualization object. The client creates an appropriate set of objects to visualize which implement the visualization object interface. This interface gives HIVE a handle on the object and provides it with access to data associated with the object. It also allows individual objects to be quickly removed from or selected by the visualization. The visualization window interface provides the client with the ability to control the visualization. It offers facilities for editing the set of objects, for setting and getting properties of the actual visualization, for handling events using appropriate callbacks, and for handling selections.

The second interface is designed to support a variety of different plug-in visualizations. This interface provides a basic abstract class that the visualization plug-in actually needs to support. In addition, it provides support for maintaining the set of visualization objects, for drawing through a set of high-level primitives, for accessing parameters, f and for managing colors and textures. This interface was designed so that creating a new visualization would be relatively simple.

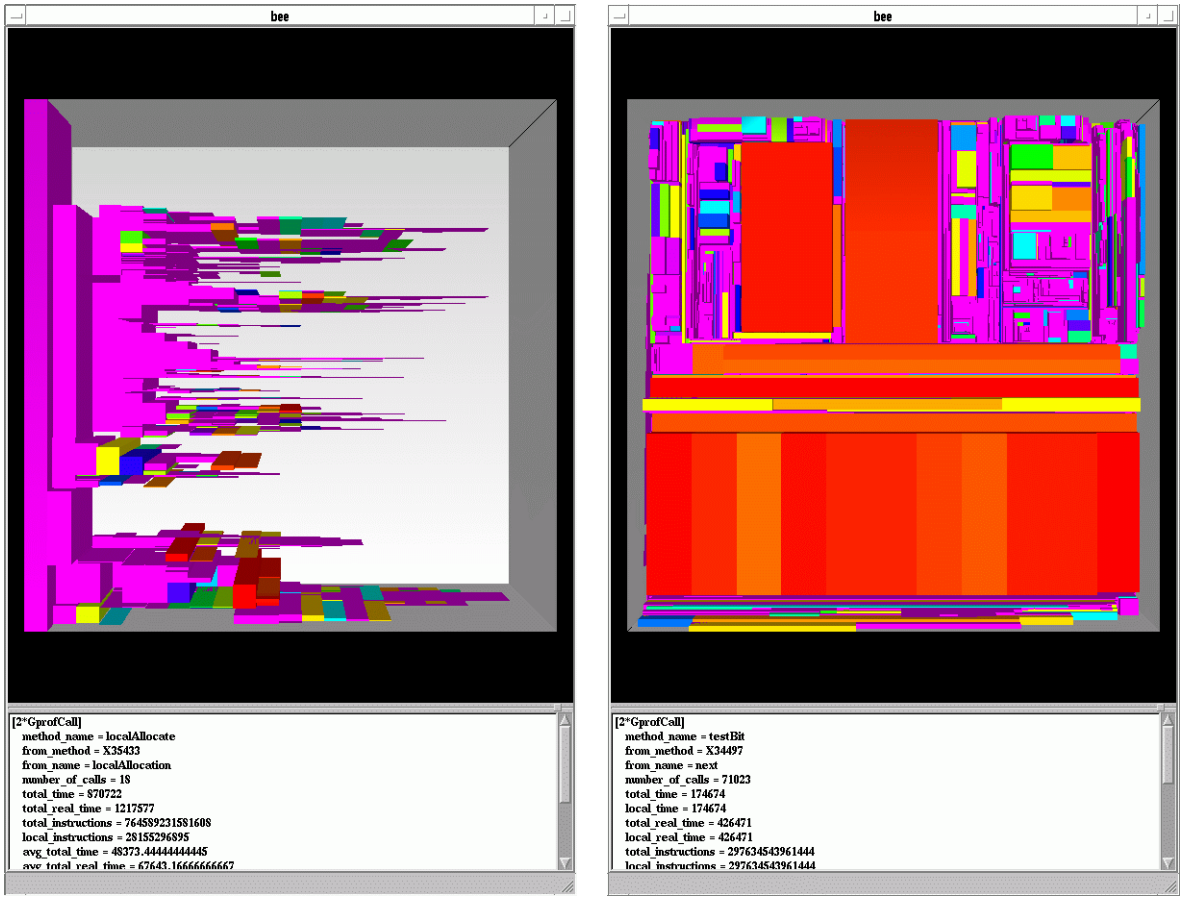

FIGURE 15. Two tree views of gprof-style performance data. The left view is a box tree; the right view is a tree map.

The actual visualization strategies are encoded in the COMB package. Actually, HIVE allows arbitrary plug-ins that can be specified and loaded at run time. However, we have developed an initial set of visualizations that cover a broad spectrum of software understanding problems. These include:

- *Box trees*. This is a tree representation where the nodes are drawn adjacent to their parents without having to display the arcs. Color, height, depth, and size can be used to represent various dimensions of data. For example, the first view in Figure 15 shows a box tree view of *gprof*-style performance data. Here each routine is represented as a box. Depth of the gprof tree is shown in the X dimension. The size of the box in the Y dimension represents the fraction of the time the routine takes from its parent. Color is used to highlight routines that take significant amounts of time. The Z dimension is used to reflect the number of calls (up) and the real time spent in the routine (down). This visualization makes it quite easy to determine where the program is spending its time and then, using the other visual cues, to determine why it spends its time there.
- *Tree maps*. This is an alternative tree representation where nodes are nested inside one another, alternating between the horizontal and vertical. Ours is a 3D variant of the display originally developed by Schneiderman [28]. The right-hand view in Figure 15 shows the an alternative view of a gprof-performance tree. Here the size of each box represents the amount of time spent in that routine, color is again used to indicate the routines that take the most time, and the Z dimension is used to reflect the number of calls.
- *File maps*. This is a representation that is roughly equivalent to SeeSoft [7]. It can be used to display information relevant to a single file or a set of files. Color, height, and texture can be used to represent information that is associated with lines in the files. Figure 16 shows two different file maps. Both illustrate performance data over the set of files in a system. The one on the left uses color to indicate the number of allocations done by a routine and height to indicate the run time spent in the routine. Lines within the file are grouped into blocks and thus a column might indicate multiple routines. The view on the right is a 2D view that shows the lines in the various files. Here

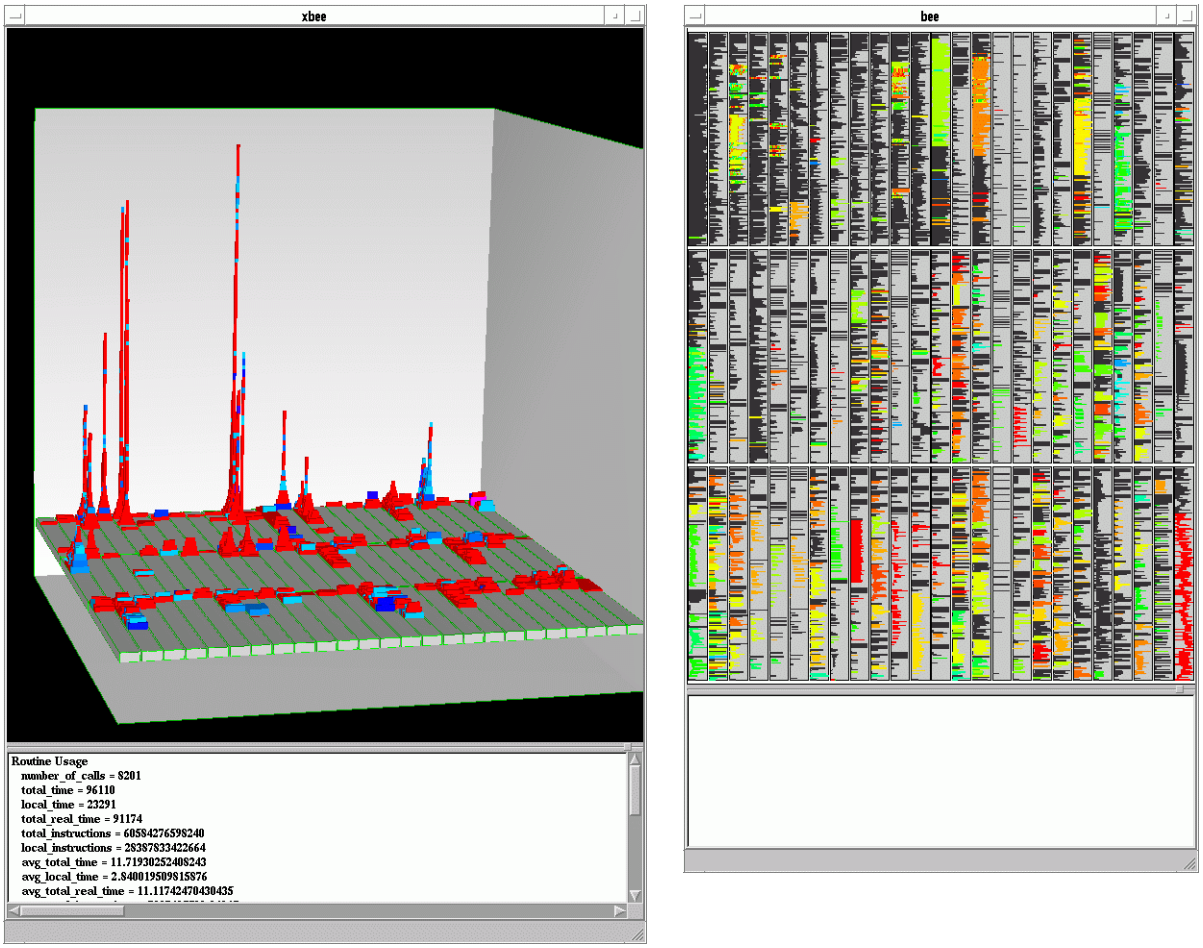

FIGURE 16. Two different file maps. Both show performance over a set of files in a system.

color is used to indicate run time spent in the routine. These visualizations are well suited for relating information back to source files. They are capable of displaying information over a large number of source files.

- *Layouts*. We include the ability to display 2D or 3D graphs consisting of nodes and arcs where this is appropriate. These typically are only effective when the number of nodes is less than a hundred and we have not used them extensively with the system because of this restriction.
- *Point maps*. These are generalized 2D or 3D scatter or dot plots [3]. Position, hue, saturation, texture, size (X,Y, and Z separately), and rotation can be used to represent arbitrary information along with two or three positional dimensions. A pair of point map visualizations are shown in Figure 17. The first view shows 2-level call performance data for a moderate sized system. The X and Y axes represent the from and to routine respectively. Color is used to represent the amount of time spent in the called routine; the size of the node represents the number of allocations done by the called routine. The fact that the graph clusters along the diagonal indicates that routines are likely to call other routines in the same or related classes since the routines are sorted by their full names including the package and class prefix. The second view shows allocation information by class. Here the X axes represents the class name, the Y axes the number of garbage collections, the Z axes the number of objects freed, color the average span of an object of that class, and texture indicating the average number of moves.
- *Connected point maps*. This is an extension of point maps where we allow lines to connect related points. This provides a simple layout capability that can be used to illustrate time sequences or other relationships. The user can specify what fields are used to determine the relationship expressed by the lines. Figure 18 shows an example of a connected point map. Here we are displaying the set of finite state automata that result from analyzing class usage. The set is restricted to the classes in a particular package. Color is used to indicate the number of times a state is reached and an arc is traversed.

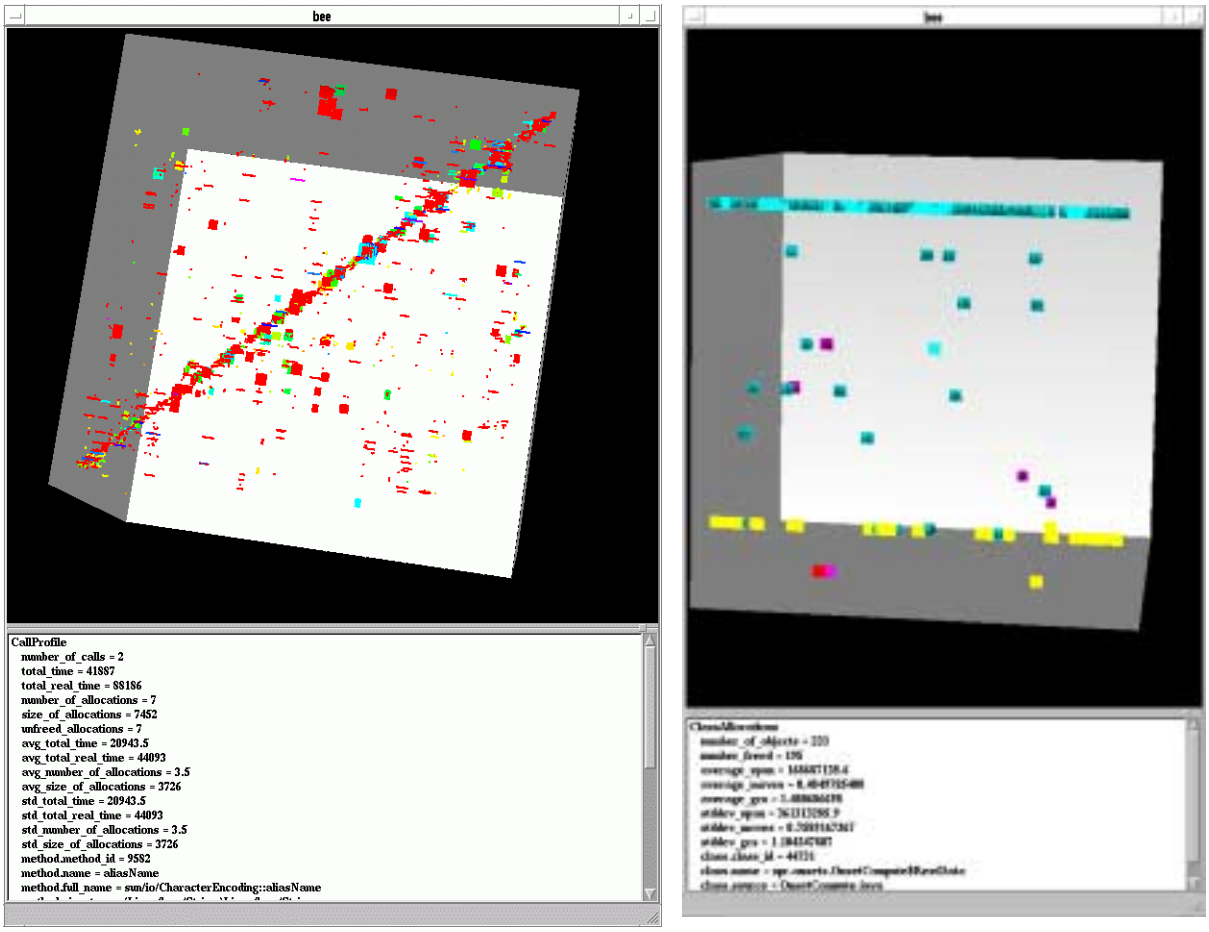

FIGURE 17. Point map visualizations showing the call graph and class-based allocation information.

- *Spirals*. We use spirals to show time series data in a compact and space-efficient manner. Hue, saturation, height, and width can be used to show information relating to time. The spiral can be formed so that equal amounts of time are represented either by equal arcs or by equal lengths. Figure 19 shows two spiral visualizations showing the stack during a run of an optimizing compiler. The first shows the full stack of a sample trace. Here color indicates the routine being called, height indicates stack depth, and width indicates total run time. The second view is a restricted view shown in 2D which excludes library routines. Here it is relatively easy to see the different phases of the compiler.

5.2 BUD

BUD provides the actual implementation of the underlying data model used by MURAL described in Section 4.1. It contains the logic both for accessing the various data sources (using their XML specifications) and for creating the set of objects to be visualized using restrictions and relationships over these sources.

BUD is organized in a modular fashion so that new data sources can be easily accommodated. The XML specifications for a data source provide access information. The top level access information is used to define which BUD module is responsible for accessing that data source; the more detailed access information is then used by this module to map from the data source to an internal representation of the model. The modules are responsible for taking the overall query and locally optimizing it for their particular data source. This allows queries based on indexing or restrictions that are best handled by an underlying database system to be handle efficiently.

The common part of BUD provides facilities for implementing arbitrary restrictions and relationships over the retrieved data. This provides an effective and efficient query system. BUD uses this system to generate a set of data

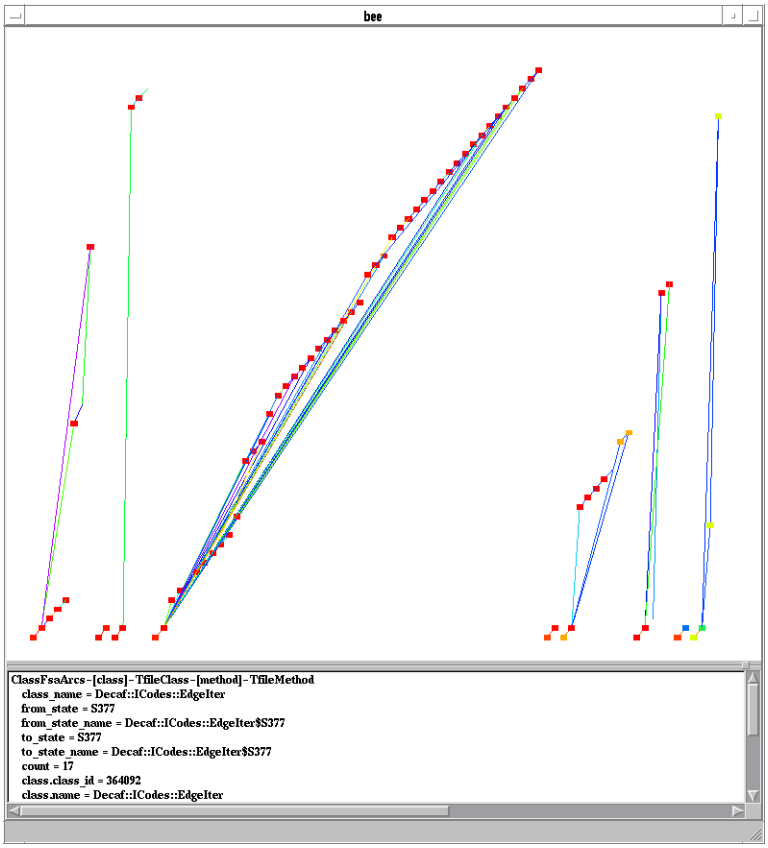

FIGURE 18. Connected point map visualization of class automata.

objects that represent the user's defined data objects for visualization. These objects can then be accessed by HIVE to actually get the data to be displayed in the visualization.

5.3 APIS

The user interface to the visualization display is provided through APIS. APIS provides a set of control panels for each visualization. These include:

- *Visualization Control*. This panel lets the user control the visualization by both setting properties of the visualization and by specifying what data fields are associated with what visualization properties. There are two types of properties listed here. The first are parameters of the visualization itself. These let the user control how the visualization is displayed and how it should be interpreted. For example, a spiral visualization has parameters to let the user specify the number of cycles and whether the spiral dimension is represented by distance or by angle.

  The second type of property are data-based properties of the visualization such as position and size that can be related to fields of the visualization objects. Here the front end lets the user select which of the appropriate fields from the different visualization objects should be used to obtain the appropriate values. APIS provides two special field types. The first is for specifying colors. Here the user can specify separate fields for hue, saturation, and intensity or can set particular values for any of these properties. The second special field type handles textures. The user can select which texture should be used and how it is used. Data fields can be used to either select a portion of the texture by specifying an X and Y position and size. from within the texture, they can be used to specify a repeat count for the texture, or they can be used to specify a rotation. The first case is useful for using a texture to define a range of values and then to use data to indicate a portion of that range. The second case is useful to use the density of the texture to indicate data in either the X or Y direction. The third case allows an arrow or similar texture to be used to convey another dimension of information. An example of this panel is shown on the left in Figure 20.
- *Restriction Control*. This panel provides browsing support by letting the user restrict what data objects should be displayed and which should be elided. The panel allows the user to set upper and lower bounds for numeric data

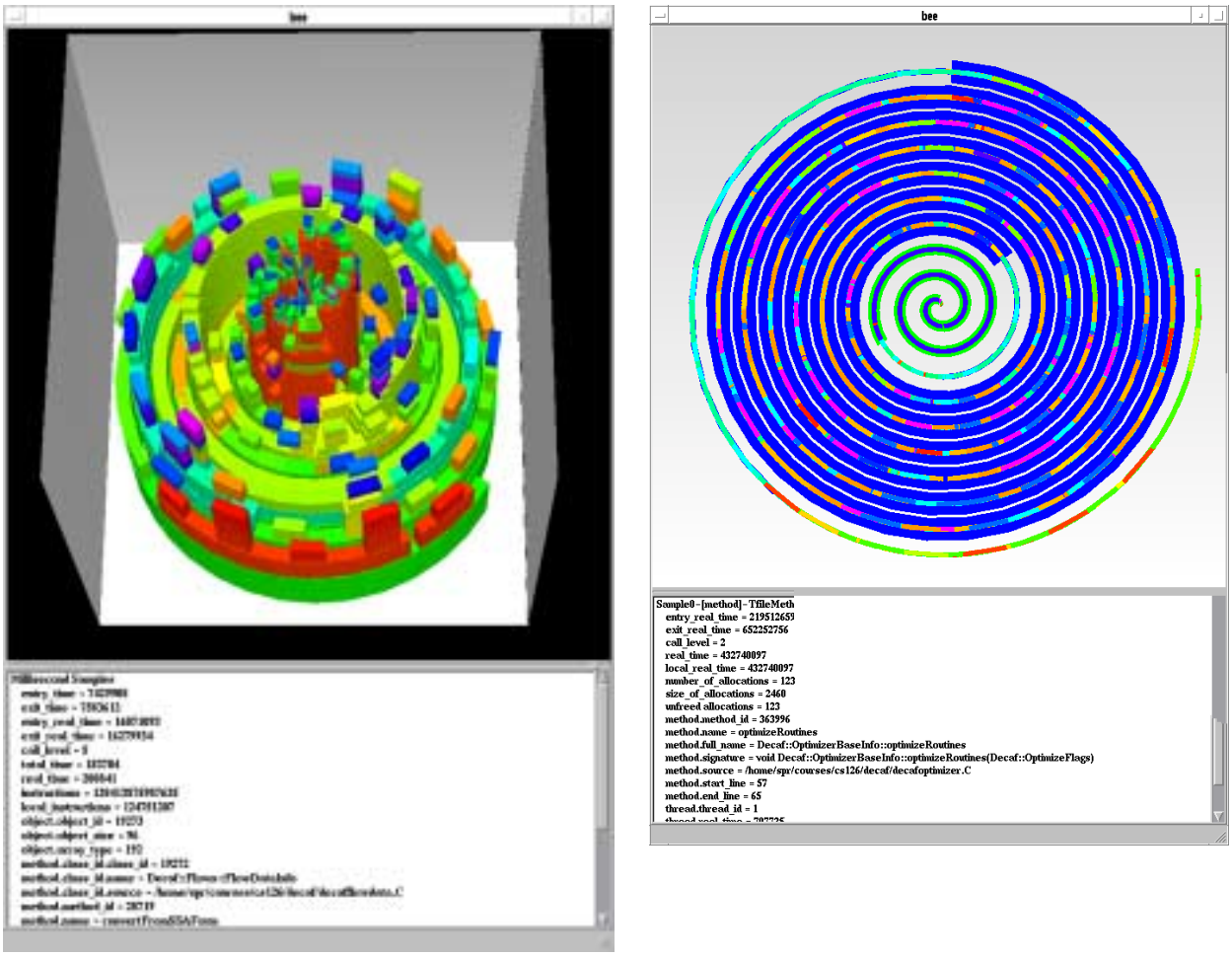

FIGURE 19. Two spiral stack visualizations.

fields and to specify regular expression patterns of items to either include or exclude for string fields. A special two-sided range widget is used for specifying value bounds. An example of this panel is shown on the right in Figure 20.

- *Interaction Control*. This panel provides support for correlating multiple visualizations using semantic information. It lets the user specify what properties of the current visualization should be sent to other visualizations when the visualization is clicked on. It also lets the users specify whether and how the focus and selection of the current visualization should change when information is received from other visualizations.
- *Grouping Control*. This panel provides another form of browsing support. It lets the user dynamically specify groupings of the data objects, for example grouping all data for methods by their class. The user can specify how the various fields should be treated when grouped as well as the actual pattern-based criteria for grouping. This is based on the visualizations that were supported by the FIELD environment for call graphs and the class hierarchy [19,20,22].

These control panels are designed to let the user effectively and efficiently browse the visualization. Any changes made in the panels are immediately reflected in the displayed visualization. Thus the user can continually restrict a value setting and stop when the result looks appropriate. Additional browsing facilities are provided by HIVE through the visualization itself. These let the user pan and zoom over 2D visualizations and pan, zoom, move in and out, and rotate over 3D visualizations.

APIS also provides the user interfaces needed to select the system to be visualized and the particular trace to be examined as appropriate.

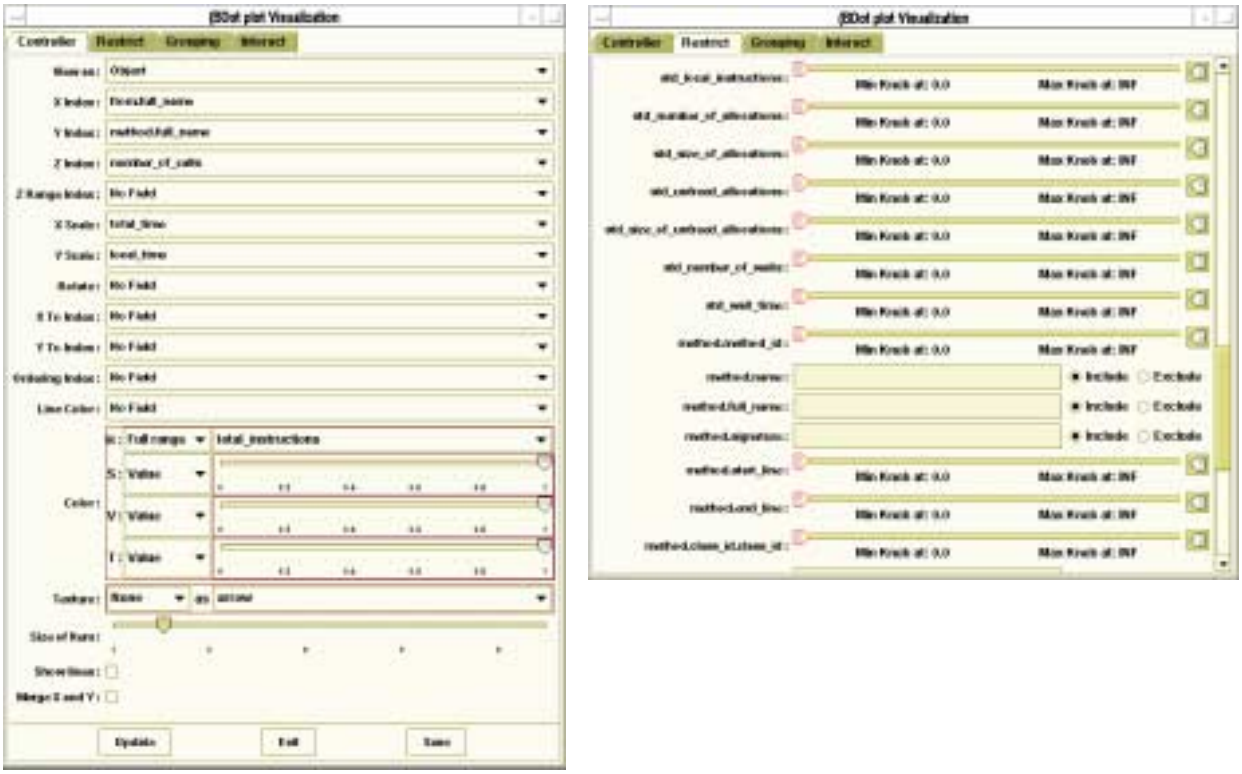

FIGURE 20. Two of the APIS control panels. The left one is the visualization control panel. The right one is the restriction control panel.

5.4 BEE

BEE is the glue that holds the above packages together. It reads a file generated by the visualization front end that describes what data should be visualized and what visualization strategy should be used. It sets up BUD to access and combine the required data sources after using APIS to let the user specify what program and trace files should be viewed. It then uses HIVE to set up the actual visualization window, plugging in the appropriate visualization method from COMB. Finally, it puts up and manages an appropriate APIS interface for the given data sources and the given visualization. It also takes care of passing events from the visualization through APIS to either the display panel of the visualization or to other visualizations.

5.5 Experience

The back end visualization system is a key part of the BLOOM system. It successfully demonstrated that one could design a powerful but general purpose visualization engine that could handle a wide variety of different data sources and a variety of different visualizations.

The power of the system comes from the ease with which it can be extended. New visualizations can be added either by implementing a new module in COMB or through a separate shared library that will be loaded dynamically into the system by BEE. Our experience shows that a new visualization method can be incorporated in one or two days using the facilities provided by HIVE. New data sources can be added to BUD in a day or less. Once new visualizations or data sources are added, they become available for general use with the rest of the system.

We have concentrated on 3D visualizations in what we provide because the extra dimension makes it easy to display more data within the same space. However, while we use 3D in general, we have been very careful to ensure that each of the visualizations is first a 2D visualization. For example, Figure 19 shows a spiral visualization in both 2D and 3D. With current processors and graphics cards the more complex visualizations we provide can both be created rapidly from tens of thousands of objects and then manipulated in real time for browsing purposes.

6. Current and Future Work

We have been working on the BLOOM system for about two years. The system has been complete for about six months. During this time we have had some opportunity to use it both as a demonstration vehicle for what can be done with software visualization and in our day to day work.

The system has been used to create a broad range of visualizations covering all aspects of the program structure and behavior. However, most of the visualizations have concentrated on various aspects of program behavior, in particular performance. Here visualizations have been used to provide insight into locking problems, thread interaction problems, and performance bottlenecks. We were able to use the visualizations to quickly determine, for example, that the reason that one of our graphical editors ran slowly when dragging an arc was because it was continually doing large numbers of object allocations. Similarly, we were able to illustrate where the optimizing compiler used in our compiler class spent its execution time. In particular we were able to very convincingly show how little time was spent on lexical and syntactic analysis compared to either semantic analysis or optimization or even code generation.

Our experiences to date have convinced us that this general approach is the correct one. Visualizations are easy to define and bring up. Moreover, despite the common framework used for all visualizations and the use of 3D, the visualizations are generally easy to interpret and gather information from. The process for gathering trace information, while somewhat slower than desired, is noninvasive, letting the programmer collect data without having to do any modifications to the program or code. The visual query language is easy to use and fairly intuitive. We have been able to quickly teach it to a number of undergraduates who had no previous database experience and have them use it successfully. The visualizations look nice and are capable of display ten to fifty thousand data elements in a single view.

However, we feel the system is still incomplete and not at the stage where it would be a practical to offer as a general tool for software understanding to everyday programmers. The reasons for this are varied, but can be divided into a set of major problems that need substantial work to address and a set of minor problems that can be addressed relatively easily. The major problems deal with data collection and analysis for understanding and not with visualization. These are:

- *Data availability.* The major problem we have had in applying our tools to everyday software understanding has been the lack of the data necessary to address the problem. Many of the problems that we come across need to be viewed at the level of source lines rather than functions. Moreover, variables and data flow are often important elements of the understanding problems. Visualization is useless for understanding unless the appropriate data are available. Our current data collection facilities do not provide this information first because it is not available from the JVMPI and second because the amount of data would be too voluminous. We are working on different approaches to address these difficulties. These involve need trace collection facilities, selective tracing both in time and by source location, and efficient and effective encodings of the relevant data based in part on semantic analysis.
- *More analysis*. Many day to day understanding problems require specific data analysis. While we currently provide a range of different analyses, these are often not sufficient. Moreover, as more data becomes available, a broader range of analyses will be needed. While it is relatively easy to add new analyses to the current framework, this requisite time to do so far exceeds our five minute limit.We plan to address this problem in two ways. First, we continue to add new analyses whenever we come across a new problem that requires analysis. Second, we are working on developing extensions to the MURAL query model that will allow the specification of a range of useful analyses at a high level. This involves adding aggregation operations to the query language as well as defining a separate query language over traces using intervals.

The less serious problems cover all aspects of the system. These include:

- *Data Collection Overhead*. The overhead for trace data collection is still too large for some applications, especially long-running ones. We feel we can speed up C/C++ trace collection by at least a factor of two. We are also concentrating on performance overhead as we expand our trace data collection as described above.
- *Project Definition*. In order to do static analysis of a system, the tools need a definition of the what files constitute the system. This currently requires that the user create a separate project description file. Most users are reluctant to learn how and then provide this file. We hope to eliminate this need by extracting the project information from existing program development tools.
- *Visualization Selection*. The heuristics currently used by MURAL for selecting appropriate visualizations do not always select what we would consider to be the correct visualization. New heuristics need to be added and the weighting parameters used by MURAL need to be adjusted to get this correct.

- *Simplified Visualizations*. The current set of visualizations are quite general. Point map and file map visualizations in particular can both be used as the basis for a variety of different visualization styles. This can become confusing to the novice user and to the heuristics involved in selecting a visualization. We plan to provide simplified forms of these visualizations which provide a clearer interface to the user and to MURAL while using the existing code base.
- *Graphics Performance*. Some of the displays, especially complex ones, cannot always be navigated in real time. This is in part a problem with the old graphics cards we are using and should be alleviated with new hardware. We also have not put much effort into optimizing our graphics code. We are also working on making updates based on changing parameters, which require significantly more computation, fast enough to generally work in real time.
- *Grouping and Interaction Interfaces*. The current APIS interfaces for grouping and interaction, while complete and powerful, are not particularly user-friendly. We need to add some intelligence to these interfaces so that they choose appropriate defaults for the many options and make it easy for the users to perform common tasks.
- *More Visualizations*. The current set of visualizations places a heavy emphasis on performance-style data. Additional visualizations are needed to handle better display of structural data, and such domains as thread synchronization and message passing.

BLOOM represents a strong first step toward a practical system for using visualization for understanding. While we have some problems with the data collection and analysis that are required for software understanding, the complete system demonstrates that the overall approach is viable. The particular contributions of the system include the visual query language and its associated data model for specifying visualizations and the general purpose back end that supports a broad range of 3D visualizations. We believe that once the above problems are addressed this framework will provide a powerful and useful tool for software understanding.

7. Acknowledgements

This work was done with support from the National Science Foundation through grants ACI9982266, CCR9988141, and CCR9702188 and with the generous support of Sun Microsystems. Parts of the code were written or modified by undergraduates Andrew Hull, Robert Hux, Joshua Levin, and Noah Massey.

8. References

1. N. Balkir, G. Ozsoyoglu, and Z. Ozsoyoglu, “A graphical query language: VISUAL,” Case Western Reserve University (1997).

2. Jong-Deok Choi and Harini Srinivasan, “Deterministic replay of Java multithreaded applications,” pp. 48-59 in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, (1998).

3. Kenneth W. Church and Jonathan I. Helfman, “Dotplot: a program for exploring self-similarity in millions of lines for text and code,” *Journal of Computational and Graphical Statistics* Vol. **2** pp. 153-174 (1993).

4. J. E. Cook and A. L. Wolf, “Discovering models of software processes from event-based data.,” *ACM Trans. on Software Engineering and Methodology* Vol. **7**(3) pp. 215-249 (July 1998).

5. I. F. Cruz, A. O. Mendelzon, and P. T. Wood, *Proc. 2nd Intl. Conf. on Expert Database Systems*. 1989.

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

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

8. R. Elmasri and J. Larson, “A graphical query faciliy for ER databases,” *Proc. 4th Intl. Conf. on Entity-Relationship Approach*, pp. 236-245 (October, 1985).

9. S. L. Graham, P. B. Kessler, and M. K. McKusick, “gprof: A call graph execution profiler,” *SIGPLAN Notices* Vol. **17**(6) pp. 120-126 (June 1982).

10. Judith E. Grass and Yih-Farn Chen, “The C++ information abstractor,” *Proceedings of the Second USENIX C++ Conference*, pp. 265-275 (April 1990).

11. Richard C. Holt and Andy Schurr, “GXL: toward a standard exchange format,” *Workshop Conference on Reverse Engineering 2000*, (November, 2000).

12. 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).

13. James R. Larus, “EEL guts: Using the EEL executable editing library,” University of Wisconsin-Madison Computer Science Department (November 1996).

14. Moises Lejter, Scott Meyers, and Steven P. Reiss, “Support for maintaining object-oriented programs,” *IEEE Trans. on Software Engineering* Vol. **18**(12) pp. 1045-1052 (December 1992).

15. C. G. Nevill-Manning, “Inferring Sequence Structure,” *Ph.D. Dissertation, University of Waikato, New Zealand*, (May 1996).

16. Craig G. Nevill-Manning and Ian H. Witten, “Indentifying hierarchical structure in sequences: a linear-time algorithm,” *Journal of Artificial Intelligence Research* Vol. **7** pp. 67-82 (September 1997).

17. Wim De Pauw and Gary Sevitsky, “Visualizing reference patterns for solving memory leaks in Java,” in *Proceedings of the ECOOP ’99 European Conference on Object-oriented Programming*, (1999).

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, Scott Meyers, and Carolyn Duby, “Using GELO to visualize software systems,” *Proc. UIST ’89*, pp. 149-157 (November 1989).

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

21. Steven P. Reiss, “Cacti: a front end for program visualization,” *IEEE Symp. on Information Visualization*, pp. 46-50 (October 1997).

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

23. Steven P. Reiss, “Software visualization in the Desert environment,” *Proc. PASTE ’98*, pp. 59-66 (June 1998).

24. Steven P. Reiss and Manos Renieris, “Encoding program executions,” *Proc ICSE 2001*, (May 2001).

25. David S. Rosenblum and Alexander L. Wolf, “Representing semantically analyzed C++ code with Reprise,” *Proceedings of the Third USENIX C++ Conference*, pp. 119-134 (April 1991).

26. Nicholas Roussopoulos and Daniel Leifker, “An introduction to PSQL: a pictorial structured query language,” *Proc. IEEE Workshop on Visual Languages*, pp. 77-87 (1984).

27. Guiseppe Santucci and Pier Angel Sottile, “Query by Diagram: a visual environment for querying databases,” *Software Practice and Experience* Vol. **23**(3) pp. 317-340 (1993).

28. Ben Schneiderman, “Tree visualization with tree-maps: a 2-D space-filling approach,” *ACM Transactions on Graphics* Vol. **11**(1) pp. 92-99 (January 1992).

29. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, *Software Visualization: Programming as a Multimedia Experience*, MIT Press (1998).

30. Bosco S. Tjan, Leonard Breslow, Sait Dogru, Vijay Rajan, Keith Reick, James R. Slagle, and Marius O. Poliac, “A data-flow graphical user interface for querying a scientific database,” *IEEE Symp. on Visual Languages*, pp. 49-54 (August, 1993).

31. Z. Q. Zhang and A. O. Mendelzon, P. Ng, and R. Yeh, “A graphical query langauge for entity-relationship databases,” in *Entity-Relationship Approach to Software Engineering*, ed. S. Jajodia,North-Holland (1983).

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