Visualizing Java in Action

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

Abstract

Dynamic software visualization is supposed to provide programmers with insights as to what the program is doing. Most current dynamic visualizations either use program traces to show information about prior runs, slow the program down substantially, show only minimal information, or force the programmer to indicate when to turn visualizations on or off. We have developed a dynamic Java visualizer that provides a view of a program in action with low enough overhead so that it can be used almost all the time by programmers to understand what their program is doing while it is doing it.

CR Categories: D.2.6 Graphical environments, D.2.5 Debugging aids.

Keywords: Dynamic software visualization, run-time monitoring, instrumentation.

1 Introduction

Software visualization has not been particularly successful for program understanding. Visualizations that look at the static aspects of a software system are only able to provide limited insights and say nothing about the important and more complex dynamic behavior of the system. Dynamic visualizations have been expensive to use because they require the programmer to run the program in an environment that produces the appropriate trace data, generally slowing program execution by an order of magnitude or worse. The result is that programmers generally don't bother using visualizations even if they would be helpful.

We wanted to provide a dynamic visualization environment that could actually be used for real running programs. Such an environment would provide programmers with the information they needed to understand what their program was doing as it was doing it. The environment had to be simple to use, had to minimize the overhead involved with the visualization, had to work with arbitrary programs, and had to provide immediate feedback to the programmer. Moreover, the resultant system had to be not only informative but also entertaining — we wanted programers to start using visualization just because it was fun.

The requirements for such a visualization system emphasize usability rather than detailed or high-quality visualizations. In particular, we felt that such a system should:

- *Minimize overhead*. The overhead involved in data collection must be such that the program run as fast as possible. Ideally we wanted a slowdown factor of 2 or less.
- *Maximize information*. Given the constraints of minimizing overhead, we wanted to provide as much information as possible. The more information that is provided, the more useful the system is and hence the more it is likely to be used.
- *Emphasize real time*. We wanted a system that would show the programmer what the program was doing right here and now. This meant that we needed to both extract the information and display it in real time.
- *Maximize displayed information*. Not only did we want to maximize the amount of information that was collected, we wanted to be able to display as much of this information as possible at once. This ensures that the programmer does not have to continually adjust the visualization to get information that might be relevant.
- *Provide a compact display.* For the visualizer to be run frequently and with a variety of applications, we needed to ensure that the result would be non-obtrusive. The system should be able to provide as much information as possible while minimizing the amount of screen space that it consumed.

A system meeting these requirements would offer the first step toward making visualizations both useful and used. Moreover, it would demonstrate that software visualization could be an everyday thing rather than something only to be used when problems were so severe that nothing else worked.

2 Prior Work

There have been a large number of different systems that provide visualizations of the dynamics of a program. Ours is different in that it attempts to provide high-level program-specific information in real time.

Perhaps the most prominent effort is IBM's Jinsight [5-7]. Jinsight typically runs by collecting detailed trace data as the program executes and then, after execution is complete, letting

the programmer understand execution at a very detailed level using a variety of views based on the trace. Trace collection, however, is not that efficient,requires a suitably modified JVM (and the program to work with that particular JVM), and is typically not the type of thing one would use all the time. Recent work on Jinsight has been aimed at letting the programmer identify just those portions of the program for which tracing should be done. This provides for almost immediate visualizations, but assumes that the programmer knows what to look for in advance.

The program visualization group at Georgia Tech has implemented several visualizations that provide insights into program execution using program traces [2,4]. Similar systems include PV from IBM [3], and the dynamic aspects of the Bloom system [12-14]. The problem with these trace-based analyses is that they require the programmer to take the extra effort to run the system with tracing and often are both difficultto use and run too slowly to be practical. Our goal was to get as much of the information that these tools provide as possible without the considerable overhead that they incur.

Another set of relevant tools are performance visualizers that provide insight into what the machine is doing while the program is being run. These range from standard operating-system based performance tools such as those incorporated in Sun's workbench toolkit or IBM's PV system, to viewers that concentrate on some specific aspect of execution. In the later category, one finds dynamic visualizations of thread behavior [1], visualizations of heap, performance and input/output in the FIELD environment [8,9], and the large number of different visualization of the behavior of processors and messages in parallel systems culminating in the various MPI visualization tools such as *upshot* or *xmpi*.

Finally, we note that dynamic visualization is nothing really new. Back in the 1960s we (and others) used to try understanding what their program was doing either by looking at the lights or the performance meter of the system (on a GE635) or by placing a radio next to the system and listening to the different types of static that were generated.

3 Getting Java Trace Data

The key to a successful real-time dynamic visualization system is obtaining appropriate trace data with minimal overhead. Our goals here were to determine what information was needed, what information could practically be obtained, and how to obtain as much information as possible with the least overhead.

The primary objective of our visualization system was to show programmers what their program was doing as it was doing it. To this end, we needed information about what classes were currently executing, what was happening to memory, and what the various threads were doing.

Rather than attempt to show everything that the program was doing, we decided to break the execution into intervals and then display a summary of what the program did during each interval. This let us cut down substantially on the amount of data that had to be conveyed from the application to the visualization tool and made sense since the visualization tool would have to report summary information in any case.

The information we wanted to include for each interval then included:

- What was currently executing. While this could be done at the method level, it made more sense, given limited display space and the need to summarize data, to batch this information by class, or in the case of libraries, by package or collection of packages.
- How much time is spent in each class. Ideally we wanted the class execution information to indicate how much time each thread spent in each class.
- How much time is spent in each class for synchronization.
- What was being allocated. To indicate memory behavior, we wanted to show the programmer how many allocations occur for each class of object. Again, for libraries, we wanted to group this by package.
- What was being deallocated. To get a sense of the total use of memory, one needs this information along with the allocation information. We noted, however, that because Java uses garbage collection rather than explicit freeing, this information will be a bit skewed.
- What threads are in the program. We wanted the programmer to understand the current set of threads created by the application and to show when threads were created or destroyed.
- What each thread is doing. We wanted to show the programmer the state of each live thread. Here we wanted to distinguish between actively executing, blocking, doing I/O, sleeping, and running in a synchronized region. Moreover, we wanted to show the time in each interval that the thread was in each of these states.
- How often does a thread block other threads. Here we wanted to provide an indication of which threads were the cause of other threads needing to block.

These items provided our target set of information. The next issue we faced was how to obtain this information. Here several alternatives were possible.

The simplest option was to use the hooks provided within the Java system for profiling or debugging. Java provides a profiling facility, JVMPI, that is able to invoke user routines whenever profilable events (e.g. method entry or exit, monitor waits, or garbage collection) occur. It also provides a debugging facility, JVMDI, that offers additional hooks to let a debugger control the running program. We experimented with using JVMPI, having had considerable experience with it from our previous work [10-12]. We quickly found out, however, that just turning JVMPI on for method calls (the basic information we wanted), caused a slowdown in performance of well over a order of magnitude, much more than we were willing to allow. Moreover, with JVMPI there is no way of a priori distinguishing events that occur in user code from those that occur in library classes or packages.

The alternative we considered was to patch the Java program in order to insert calls whenever a significant event occurred. The events we were interested in involved method entry and exit, monitor entry and exit, and the state of each thread. Using IBM's Jikes Bytecode Toolkit (http://www.alphaworks.ibm.com/tech/jikesbt), our solution was to instrument the complete application by inserting calls to trace routines for each method entry and exit, for each allocation, and around each synchronized region.

While this is much more efficient, it still isn't perfect. First, we tried to use native code so that our tracing code would not affect the original application and could be as efficient as possible. This was not practical because Sun's Java virtual machine has considerable overhead when calling a native method. In particular, just inserting empty native event calls for the above events slowed the program down by an order of magnitude.

The second problem involved getting information about the current thread. In Java this needs to be done by calling the static method *Thread.currentThread*. Unfortunately, there is considerable overhead associated with this method and using it for each event again slowed the program down by an order of magnitude. This forced us to consider ways of providing the user with appropriate information without knowing the current thread in most cases. This limited the information available since we would not match entry events with the corresponding exit events and could not associate allocation events with appropriate calls.

A third problem was that patching a Java program has considerable overhead, especially if one has to patch not only the user's code but also all the Java standard libraries and any other packages used by the application. To lessen the overhead here, we decided to take into account the set of classes that the user was interested in. It would be impractical from a visualization standpoint, to show information about all the classes in an application. There are just too many library classes, and most of these are not relevant to the programmer. Instead, we decided to let classes be grouped into packages and packages into a package hierarchy. For example, all classes in java.io.* (and all subpackages of java.io.) can be represented by a single visualization object.

In order to minimize overhead and patching time, we divide the classes into three categories. Detailed classes are those directly in the user's application. For these we provide information that consider all methods (private and public) and detail any nested classes for separate visualization. Library classes, on the other hand, are grouped into packages and we typically only generate events for the initial entry into the library. (This is done by creating a stub routine for the library call and replacing all explicit calls to the library from the application with calls to this stub.) We do not generate events for calls within a library class. Similarly, allocations of all objects from the package are grouped together. Finally, classes that are neither detailed nor library are treated at an intermediate level of granularity. Here nested classes are merge with their parent, we count only public methods, and we patch the classes to find out what is going on as they are called.

4 Generating Data for Visualization

We next needed to generate the actual information we wanted to visualize from the event calls. This involved processing each event appropriately, accumulating the necessary information, and then sending it to the visualizer.

We used method entry and exit events to determine both what was executing and the state of each thread. What was executing was determine by keeping a counter for each class (or package) and incrementing that counter when the appropriate entry event occurred. Ideally, what we wanted here was to use the counter to represent time, tracking the time spent executing in each class. This proved difficult for two reasons. First, it is difficult or impossible to get a timer that is accurate enough to give reasonable information on most machines (on Linux, for example, the best one can do is ten millisecond resolution). Second, we had no way of matching entries and exits in a multithreaded environment since determining the current thread was too costly. Because of this, we approximate usage by just keeping counts of calls.

Entry and exit events are used to determine thread state changes by identifying those routines that affect thread state and treating calls to these routines differently. We use an XML file to identify all routines in the Java libraries that represent a state change. (Additional files can be provided if there are any application-specific routines.) These include all routines that might do blocking input-output, routines that cause the thread to sleep, and routines that cause the thread to wait for an event. When an entry or exit call to any of these routines is detected, the tracing code determines the current thread and changes its notion of the state of the thread accordingly.

This is not sufficient for detecting all thread states. We needed to augment this with information about synchronization and synchronized methods and code blocks. Synchronized methods and blocks are handled differently by the Java virtual machine. Blocks are handled by a set of JVM opcodes that indicate synchronized entry and exit. Here it was easy to insert a call immediately before and after synchronized entry and a call immediately before synchronized exit. This let us identify states where the thread is waiting on (or at least checking to acquire) a monitor, running inside a monitored region, or releasing a monitor. Synchronized methods, however, are handled internally by the virtual machine. In order to make this explicit, we patched the code using stub routines in order to provide the corresponding three calls for synchronized methods. We also use the calls at the start of a synchronized block or method to maintain counters of the number of synchronizations done for each class or package as well as counts of the number of times one thread causes another thread to block.

Finally, we inserted event calls on each allocation, noting the type of object being allocated for each. Again, we felt that we could not determine the thread associated with the call because of the potential cost of doing so. Thus we just gathered information about the total number of objects of each class allocated and the class or package that is the source of the allocation.

```
<STATS TIME='1035559445758'>
  <ENTRY NAME='java' COUNT='1101551' />
  <ENTRY NAME='spr.onsets.OnsetExprSet' COUNT='159197' ABY='95171' />
  <ENTRY NAME='spr.onsets.OnsetCubeSet' COUNT='225' ABY='225' />
  <ENTRY NAME='spr.onsets.OnsetTypeSet' COUNT='94496' ABY='94496' />
  <ENTRY NAME='spr.onsets.OnsetExprSet$SetExpr' COUNT='225' ABY='225' />
  <ENTRY NAME='spr.onsets.OnsetCardSet' COUNT='1509' ABY='1734' />
  <ENTRY NAME='spr.onsets.OnsetCubeBase' COUNT='1729410' />
  <ENTRY NAME='spr.onsets.OnsetBitSet' COUNT='4577700' />
  <ENTRY NAME='spr.onsets.OnsetCardDeck' COUNT='1059' ABY='1059' />
  <ENTRY NAME='spr.onsets.OnsetCubeDeck' COUNT='2929392' />
  <ENTRY NAME='spr.onsets.OnsetExprSet$Expr' COUNT='225' ABY='450' />
  <TOTALS COUNT='10594989' ABY='96680' ABY='96680' />
  <THREAD INDEX='1' NAME='main' SYNC='1016' />
  <THREAD INDEX='2' NAME='Reference Handler' WAIT='1016' />
</STATS>
```

FIGURE 1. Sample trace interval output.

Based on these event calls and processing, we need to actually generate information for the visualizer. This is done in two stages. First, we create a buffer that can hold all the data. This has an entry for the execution counts, allocation of counts, and allocation by counts for each class or package that will be reported. It also has an entry for each thread that indicates the amount to time spent in each possible state by that thread and the number of blocks caused by the thread. The event handlers merely update the information in the current buffer. This is done efficientlyby precomputing indices into the buffer at patch time and passing the indices directly in the event calls.

Second, we create a monitoring thread that wakes up at the end of each interval to generate a report. This thread switches the current trace buffer with an empty one, sets up a new empty buffer for the next interval, updates the execution times associated with the finalthread state of each thread using information from the previous buffer as needed (to determine the previous thread state if nothing changed), and then generates appropriate output for the interval. Note that in order to access the previous buffer and have a clean buffer for the next event, the monitoring code switches between three buffers.

The output is currently generated in XML. It provides detailed information about each class that has non-zero counts and about each thread that was not dead throughout the interval. The trace package then sends this output directly to the visualizer along with general information about the interval such as totals and the time represented by the interval. Currently, the information can be sent either through sockets or through an XML-based message server. A sample interval file is shown in Figure 1.

5 Box Display Visualization

Once the data is available, we needed to have a visualization for the data. In particular we wanted a visualization that could show a large number of objects (e.g. all the relevant classes and packages or all the application's threads) and several pieces of information about each object (e.g. for a class, the number of entries, the number of synchronization calls, the number of allocations, and the number of allocations by methods in this class; for a thread, the time spent in each of the possible states) in a small display area. We also needed something that was simple, fast and easy to understand since we wanted the visualization to run in real time.

We settled on what we call a box display. Here each class or thread is represented as a box on the display. Within the box we can display one or more colored rectangles. The various statistics can be reflected visually in the vertical and horizontal sizes of the colored display, in the color (hue, saturation and intensity as separate items) of the displayed region, or through textures where the density of the texture is used to represent the corresponding statistic.

For the class display, we typically display five simultaneous values. First, the height of the rectangle is used to indicate the number of calls. Second the width of the rectangle is used to represent the number of allocations by methods of the class. Third, the hue of the rectangle is used to represent the number of allocations of objects of the given class. Fourth, the saturation of the rectangle is used as a binary indicator as to whether the class was used at all during the interval. Finally, the brightness of the box is used to represent the number of synchronization events on objects of this class. Currently, we do not use texture as part of the display because we found that Java could not display textures in real time, however the system includes the capability to tie them to a property if the user wishes.

Each of these statistics is treated a little differently. First, the height and width have a minimum value so that a zero or trivial count (in either dimension) does not cause the display to disappear. Second, we provide a variety of color models for mapping hue including red to violet, yellow to red, and green to red. Next, both brightness and saturation are done over a limited range so that the effect does not obscure the hue value or hide the drawing altogether. Finally, we invert the sense for brightness so that the more common low values have high brightness and the occasional high values standout by being darkened.

For threads, we create a stack of color rectangles inside each box. Here we vary the height of each rectangle based on the percent of time within the interval the thread is in the corresponding state and the width of each rectangle as the percent of time in this state represented by the given thread. The hue is used to denote the actual thread state. In addition, we currently use the brightness of

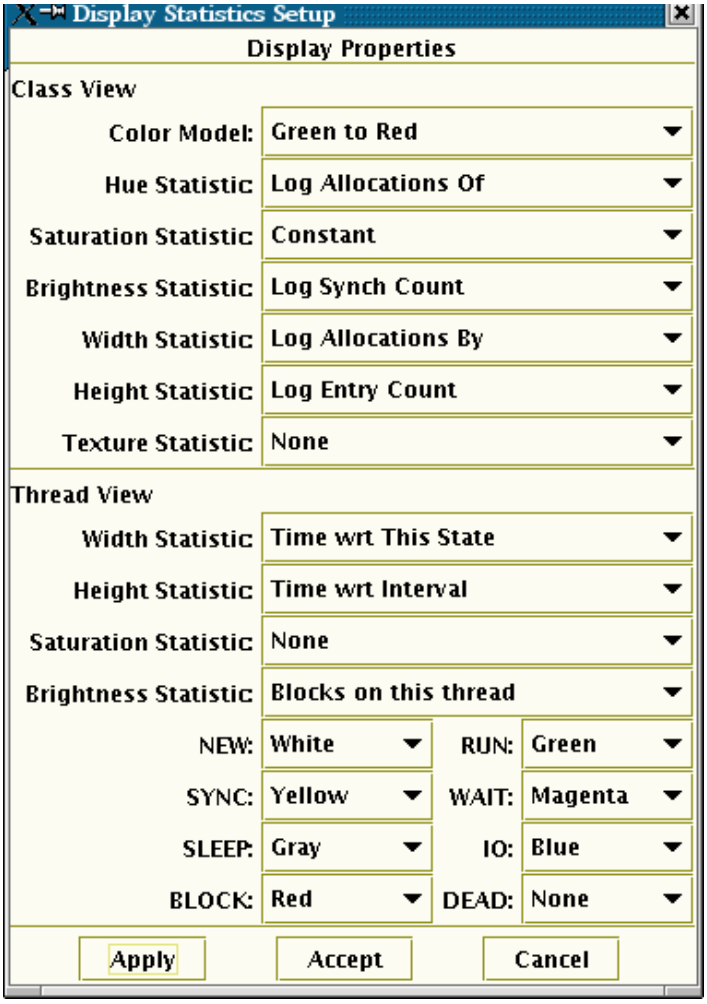

FIGURE 2. Dialog box for setting the mapping between statistics and graphical properties.

the display as an indicator of how many threads are blocking on this particular thread.

All of these parameters can be changed dynamically by the user through the dialog box shown in Figure 2. Moreover, the user can choose, for each of the count statistics, whether to use a linear scale or a log scale The latter is often more appropriate is considering performance statistics.

6 Dynamic Java Visualization

The actual visualization window is divided into two panes as shown in Figure 3 and Figure 4. The left half of the fi guresshow class and package usage information. Each class or package is displayed using a box display visualization. In Figure 3 height reflects the log of the number of entries to the class and width reflects the number of allocations done by the class or package. The darker displayed rectangle indicates where synchronized calls occur. The red rectangle indicates what is primarily being allocated at this point in the execution. All the very light rectangles indicate classes or packages that were not used during the interval.

In order to provide stability to the class display, the visualizer gets information about the complete set of classes and packages that might be displayed when the application starts. This lets it create a layout that includes all necessary items and that will remain the same through the execution. Moreover, the system labels each box in the display with an abbreviation of the class name. Tool tips are then used to provide the full class name to the user.

The right half of the display in Figure 3 uses additional box displays to indicate the status of the two threads in the system. The green box on the left indicates that the thread is actively running while the magenta box on the right indicates the thread is waiting for an event. The number of threads used by the application cannot be known in advance. Thus the visualizer will add new box visualizations to the thread half of the display as the threads are created.

Figure 4 shows another view of the visualizer, this time on an application that uses Java's Swing package. Here one can see the state of the various threads in the application and Swing, in particular, one can quickly detect which threads are running (green

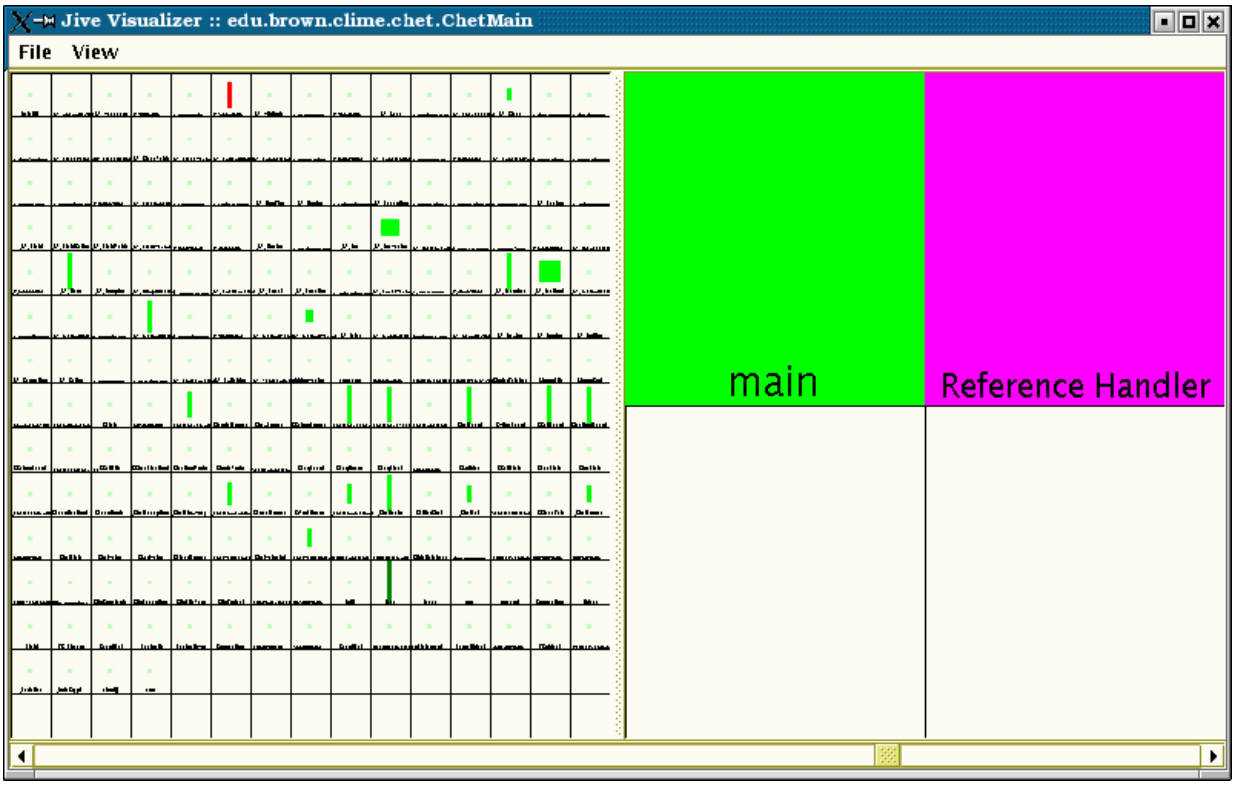

FIGURE 3. Dynamic display of a Java program.

for normal, yellow for synchronized), and whether the remaining threads are waiting (magenta) or blocking for I/O (blue). The class pane here indicates that almost all the execution is occurring in the *ApiColorChooser* class.

Another feature shown in Figure 4 is the use of tool tips to show the particular class or thread that the mouse is over. This lets us shrink the display to the point where the text is unreadable while still providing a meaningful display. For example, Figure 5 shows a miniaturized version of the display for a static analysis tool. To make this view more telling, we changed all the class statistics to use the log of the associated values.

Figure 5 also illustrates the use of the browser to show execution totals rather than incremental information. In this mode the values shown for each class and each thread represent the total counts or utilization from the start of the execution rather than just during the last interval. This is selectable in the View menu and provides useful summary information in attempting to understand an execution.

The dynamic visualizer normally shows the execution as it is happening. To achieve this it establishes a socket connection with the trace package that is loaded into the application, reads data for each interval, and then updates its display based on the data. For most applications it is able to maintain a real time display with up to one hundred frames a second, although the resultant intervals are often too small to be meaningful to the viewer.

In addition, the visualizer keeps a record of all the previous entries and provides facilities to let the user browse over the execution, either as it is happening, or more commonly, after the fact. The scroll bar on the bottom of the window is used to let the user scroll over time for the execution. The display updates dynamically as the user scrolls. This facility is useful for going back and viewing transient events and getting a better understanding of the application's execution. It also lets the user stop the display and zero in on any unexpected or unusual occurrences.

7 Running the Visualizer

In order to make the visualization tool as easy to use as possible, we developed a simple front end that lets the programmer run the application and tune the visualization. The front end is invoked by using the *jive* command in place of the standard *java* command when running the application. This command is designed to take all the same arguments programmers would normally use in running their program and hence can be substituted with a minimum of effort.

Running this command starts up the interface for the visualization and automatically starts the application running as well. Beyond this, the front end provides the programmer with a number of additional capabilities.

First, the programmer can rerun the application directly from the interface with the same or modified arguments using the Run command on the File menu to display the dialog box shown in

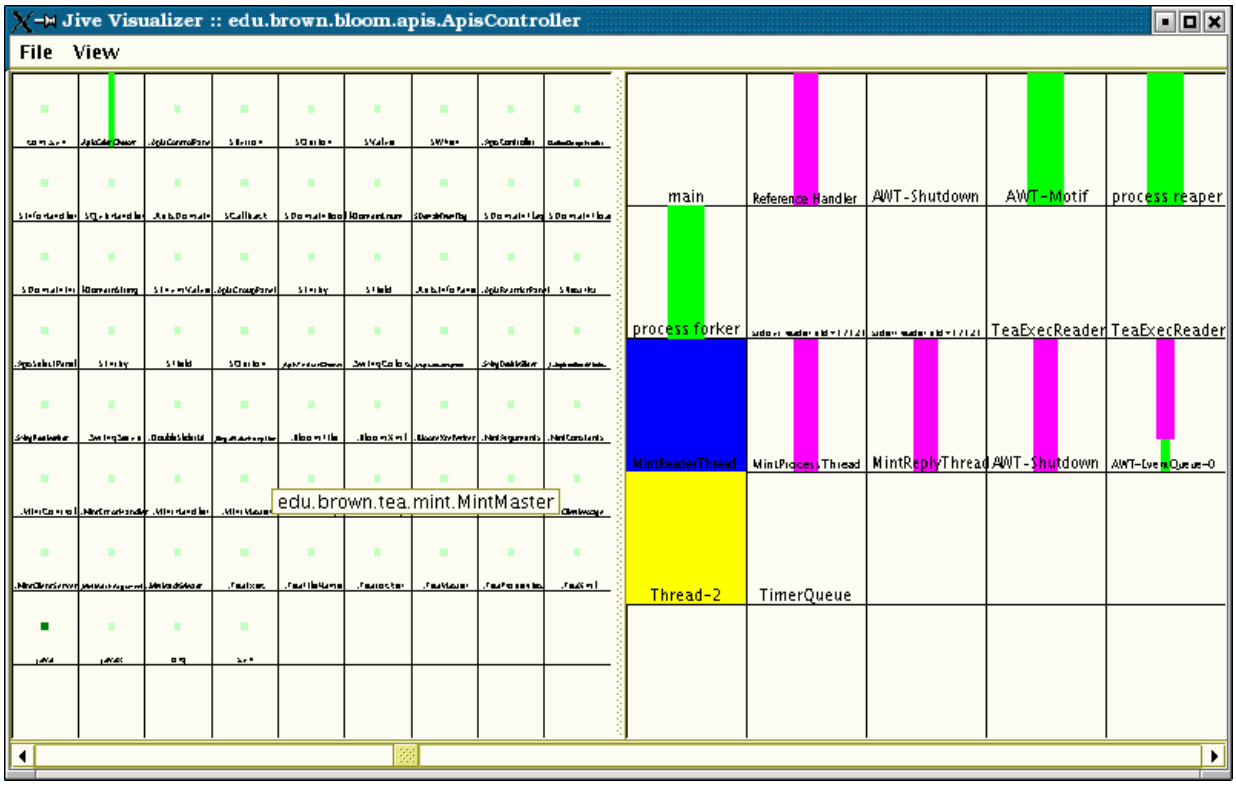

FIGURE 4. Dynamic view of a Java program using Swing.

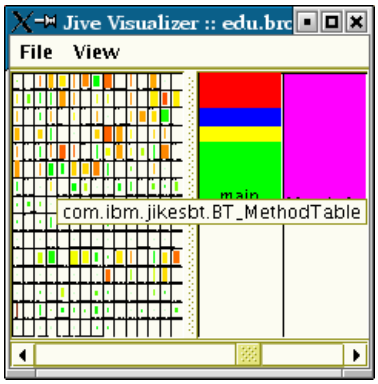

FIGURE 5. The visualizer showing a miniaturized display.

Figure 6. This is convenient since it lets the system avoid having to repatch the program for the additional runs. Second, the programmer can dynamically specify what portions of the application should be viewed as detailed and library classes using the second dialog box shown in Figure 6. Third, the front end hides the actual process of patching the program, running this in background whenever necessary. Finally, the front end provides the user with a separate window for the text input and output of the application.

8 Experience and Future Work

While not perfect, our efforts show that dynamic visualization of real applications is possible and may be practical as a default way of running the application. The program runs with a slowdown of a factor typically between 2 and 3 depending on the structure of the application. Given the wide performance range of today's machines, this seems to be quite acceptable. The drawback to our approach here is that patching the application typically takes 20 to 30 seconds, and thus there is considerable startup overhead in using the tool. However, our approach is totally Java-based and hence portable to any platform that runs Java.

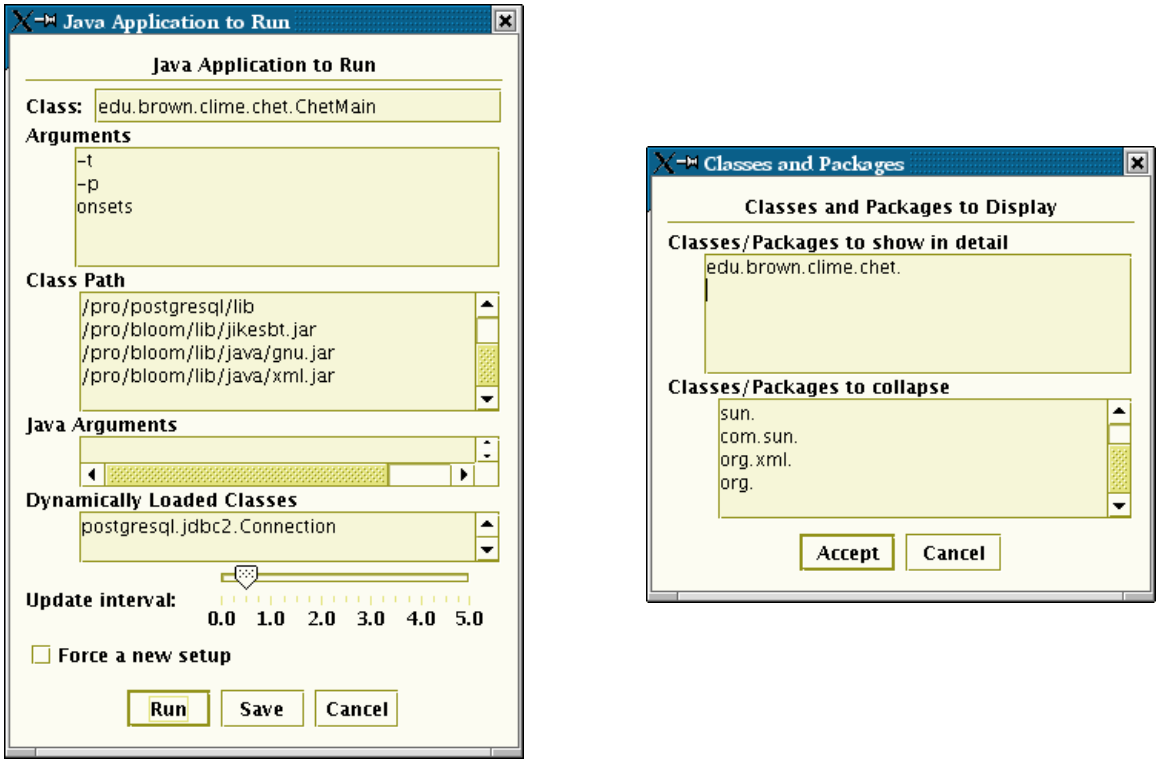

FIGURE 6. Dialog boxes for controlling the execution and specifying detailed and library classes.

We have used the visualization tool on a wide variety of Java programs ranging from simple student programs, to complex single threaded applications (a static checker), to user-interface based applications, to a multithreaded web crawler. We have found that it provides useful information and that watching it makes one think more about what the application is actually doing.

The visualization clearly shows where the application under examination goes through different phases. Each phase has its own distinctive set of active nodes, it is easy to tell the phases apart, and the phase transitions are obvious. When viewing the static analyzer, for example, we could clearly see where the tool was loading the class fi les,where it was doing the initial analysis, and then where it cycled between setting up a program abstraction based on the analysis and checking that program abstraction.

The visualization also is helpful in providing rudimentary performance information. When looking at totals, it becomes obvious very quickly which classes are executed the most, which do the most allocations, and which are allocated the most. It also provides a rapid view into which classes contained the most synchronized methods and hence caused synchronization delays and blocking. The corresponding thread view shows what each thread has done and provides some indication of the amount of time spent in blocking, in synchronized regions, and in general execution. It can also show which threads are the most active in a multithreaded application. Note however, that the tool was not designed for showing the detailed performance information needed for actually tuning the application.

Where the tool has really demonstrated it usefulness is in highlighting unexpected program behaviors. For example, a signifi - cant portion of the cost of patching is just loading the class fi les. The tool shows that about 15% of the time here is spent in synchronization, mainly from standard Java classes. Looking at the code, we fi ndthat it uses synchronized Hashtables rather than unsynchronized HashMaps for key components.

Another example occurred when we took a gas station simulation program and replaced the wait call that was used to provide a time interval during which gas was being pumped, with what we thought was a more appropriate sleep call. While the program seemed to ran normally, looking at the visualization quickly showed that with the sleep call, all threads were blocking rather than just the one thread that was supposed to. It was then obvious that sleep should not be used in a synchronized region.

While the visualization has proved useful, it still has some significant problems. First, the statistics that are reported are a bit biased because we don't have timing information and we report only public calls for some classes or packages and all calls for other packages. Second, because we are only patching Java code, there are some inaccuracies in that we sometimes miss a thread state transition if it occurs either in native code or is buried in a library routine we didn't catch at fi rst.Third, the fact that the tracing code is Java code as well means that some artifacts show up in the trace. For example, the cost of synchronization tends to be over emphasized since the event handlers determine the thread

and thus add considerable overhead before and after synchronization occurs.

The system could also be made more useful by including additional information and by letting the programmer dynamically adjust how the information should be grouped. Easy additional information that could be provided would include the classes in which synchronization occurs, and what threads are blocking other threads. More complex information would include actual timings and tracking of deallocations. Dynamic grouping would let the programmer look at their program at a high level and then zero in on the any interesting behavior, for example determining the specifi cclasses in a package that were causing synchronizations.

Other useful features would be to let the programmer set markers in the visualization to indicate known points in the program. This would simplify using the program history visualization since programmers could readily identify what part of the program they were looking at. A complementary feature would let the programmer save and reload the data from a run so that the execution could be reviewed at a later time.

9 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. Signifi cantadvice and feedback was provided by Manos Renieris.

10 References

1. Bryan M. Cantrill and Thomas W. Doeppner, Jr., "Threadmon: a tool for monitoring multithreaded program performance," *Proc. 30th Hawaii Intl. Conf. on Systems Sciences*, pp. 253-265 (January 1997).

2. 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).

3. Doug Kimelman, Bryan Rosenburg, and Tova Roth, "Visualization of dynamics in real world software systems," pp. 293-314 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine A. Price,MIT Press (1998).

4. Eileen Kraemer, "Visualizing concurrent programs," pp. 237-256 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine A. Price,MIT Press (1998).

5. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object- oriented software execution," pp. 329-346 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine A. Price,MIT Press (1998).

6. 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).

7. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proc. ICSE Workshop of Software Visualization*, (May 2001).

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

9. Steven P. Reiss, "Visualization for software engineering -- programming environments," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

10. Steven P. Reiss and Manos Renieris, "Generating Java trace data," *Proc Java Grande*, (June 2000).

11. Steven P. Reiss and Manos Renieris, "Encoding program executions," *Proc ICSE 2001*, (May 2001).

12. Steven P. Reiss, "An overview of BLOOM," *PASTE '01*, (June 2001).

13. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).

14. Manos Renieris and Steven P. Reiss, "ALMOST: exploring program traces," *Proc. 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, (October 1999).