Monitoring Java Programs Using Music

C. Collberg, S. Kobourov, C. Hutcheson, J. Trimble, and M. Stepp

Department of Computer Science
University of Arizona
Tucson, AZ 85721
{collberg,kobourov,conrad,trimblej,steppm}@cs.arizona.edu

Technical Report TR05-04
June 15, 2005

Abstract. We present JMusic, a tool to analyze the run-time behavior of a Java program. Unlike other profiling systems, this one presents its results to the user as *music*. Using audio in addition to more standard visual methods of presentation allows more information to be presented. To this end, JMusic is our first attempt at presenting profiling information as computer-generated music. The tool is specification-driven, allowing users complete control over what profiling information to present, and what type of (MIDI) music to generate for different situations. We discuss the system design and implementation, as well as present several examples.

1 Introduction

Successful use of sound in human/machine interaction predates computers: the Geiger counter has been in use since the 1900s and the sonar, since 1916; drivers of shift-stick automobiles know when to change gears by listening to the pitch of the engine; in the early days of computer programming, machine operators would listen to the sound of the hard disk to detect infinite loops. Given the ability of sound to convey information, it is natural to consider its use in computer applications. Audio output can augment the typically visual output by providing debugging and profiling feedback, without necessitating changes in existing graphical interfaces. Human hearing is sensitive to the differences in periodic and aperiodic signals and can detect small changes in the frequency of continuous signals [12]. Rapidly evolving data can be blurred or missed in a graphical display, while it can be easily heard in a simple auditory display. Moreover, certain sound cues are well suited for representing certain types of program information, such as repeated patterns in the control flow of the program. Finally, unlike visual perception, sound perception does not require the focus of the listener, allowing audio cues to augment the perception of a user whose eyes are already busy with a visual task. While there are many advantages in using

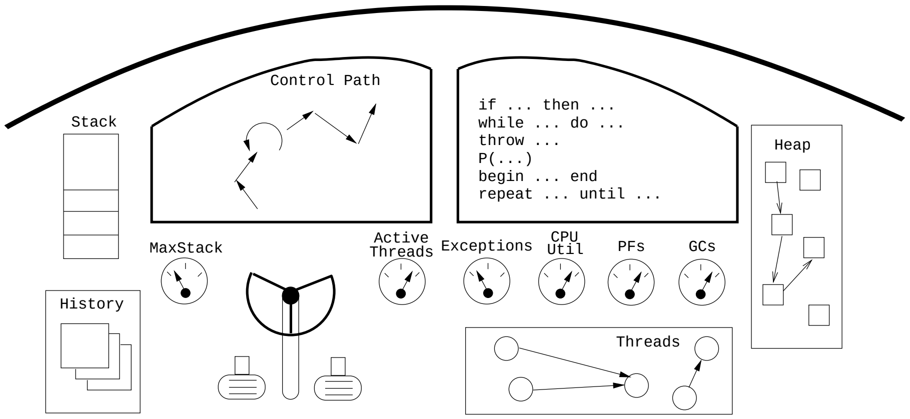

Fig. 1. A mockup of a *Programmer's Cockpit.*

auditory output over visual output, there are few examples of successful use of sounds – or "auralization" – in computer programming.

In this paper we present JMusic, a tool to analyze the run-time behavior of a Java program. Unlike other profiling systems, this one presents its results to the user as *music*. The motivation for this work is our desire to build a *programmer's cockpit*, a system for complete steering, monitoring, profiling, and debugging of computer programs. Like pre-computer airplane cockpits, ours will present all possible information about the state of a running program to the programmer, allowing them instant feedback from possible anomalous behavior.

Figure 1 shows a mockup of what such a system might look like. Shown are dials for monitoring garbage collection activity, page faults, number of active threads, maximum stack depth, and exception activity. Dedicated displays give details about execution stack behavior, heap structures, thread wait-graphs, and the version history of the program. Many other types of information about the program can of course be displayed, depending on the language and the user's interests. Execution is steered using pedals and a steering wheel, allowing control over direction of execution (forwards/backwards), speed, etc. The windows of the cockpit show the execution path and the corresponding source code. We expect the cockpit to be realized as a *cave* or as a collection of large displays, giving the programmer a sense of immersion in the behavior of their program. This will be particularly useful when analyzing the behavior of large legacy systems, where the programmer has little initial understanding of the program's internal behavior.

In previous papers we have explored the visualization of the version (CVS) history of a program, as well as the visualization of changing dynamic (heap-based) data-structures [5]. Using audio in addition to more standard visual methods of presentation allows more information to be presented. To this end, JMusic is our first attempt at presenting profiling information as computer-generated music. The tool is specification-driven, allowing users complete control over what profiling information to present, and what type of (MIDI) music to generate for different situations. Results are presented in real-time allowing users to map the current state of their program to the music being played.

The run-time behavior of a Java class is of critical interest in questions of optimization and performance. For some classes, this behavior may be determined entirely from the source code in cases where the class is deterministic. However, for complex classes, particularly in event-driven situations, the run-time behavior cannot be determined ahead of time, since external input will determine the order of code execution. Therefore, it is necessary to analyze the operations performed by the class as they occur. The output of this analysis may take any of several forms. A textual output is easiest for a mathematical treatment of the results, and there are several existing tools of this kind. One such tool is the built-in profiler, which is invoked by including the `-prof` argument to the Java virtual machine [13]. This allows the user to see which methods are called most often, but the report provides limited information and is also only available after execution. Visual and audio output provides the user with a more intuitive feel of how the class is performing as it executes. Some research has been done in producing visual analysis tools [16]. However, there is little progress toward answering the question, "How does a good program sound?" To this end, we seek to implement a program that will analyze the run-time behavior of a Java class according to user-defined goals and specifications and produce real-time results in an audio format. Sample auralizations can be found at

2 Implementation

2.1 System Overview

Figure 2 shows a schematic layout of the JMusic system. The input to JMusic consists of three files:

- A Java program (a collection of `.class` files in the form of a `.jar` file) to be monitored;
- An *auralization* script that describes the actions to be taken (what music to play) for different types of events (changes to the execution pattern for the monitored program);
- An *annotation* script that describes the types of events in the monitored program that are of interest.

Before executing the monitored program, it must be *annotated*. This involves modifying the class files of the program with code that monitors its behavior at runtime. For any specific pattern of instructions that is describable by a regular expression, the annotation script can increment a counter whenever such a pattern executes. For example, code could be inserted that counts the number of method calls, the number of array manipulations, the use of Java monitors,

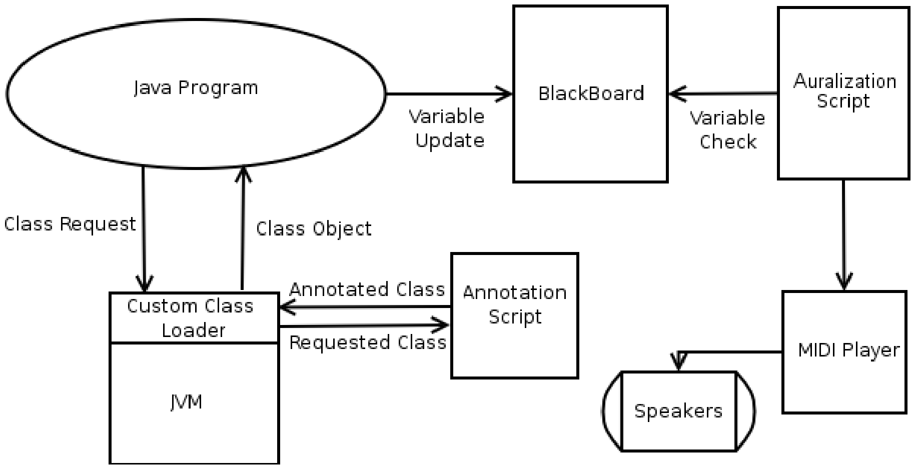

Fig. 2. A Schematic Layout of the system

etc. In addition, the annotation script can mark all of the loops in the program, detecting when the loop is entered, exited, or iterated.

The annotation script describes which execution events the user is interested in. Given this script, JMusic automatically inserts the annotation code into the program to be monitored. The annotation of the program actually occurs at class load time, which has the advantage that only classes that are actually executed will be annotated.

The monitored program is also augmented with a *blackboard*, a collection of variables which the program will update as it executes. For example, the blackboard may contain an `invoke` variable that gets incremented whenever a Java method is called.

The auralization script is used to watch the blackboard for any variable updates. In response to a variable change (such as the `invoke` variable being incremented) the script may specify that some aspect of the music should change. For example, the tempo could increase, the key could change, etc. The MIDI player that is responsible for producing audio output is updated accordingly.

The implementation consists of four independently executing Java threads:

1. One thread executes the user's Java program.
2. One thread processes the auralization script, monitors the blackboard for any changes, and updates the MIDI player accordingly.
3. One thread manages the blackboard, accepting updates from the monitored program, and notifies the auralization thread of any changes.
4. One thread manages the MIDI player.

Although each of these sections of the program execute in a separate thread, they will interact and modify each other as described above.

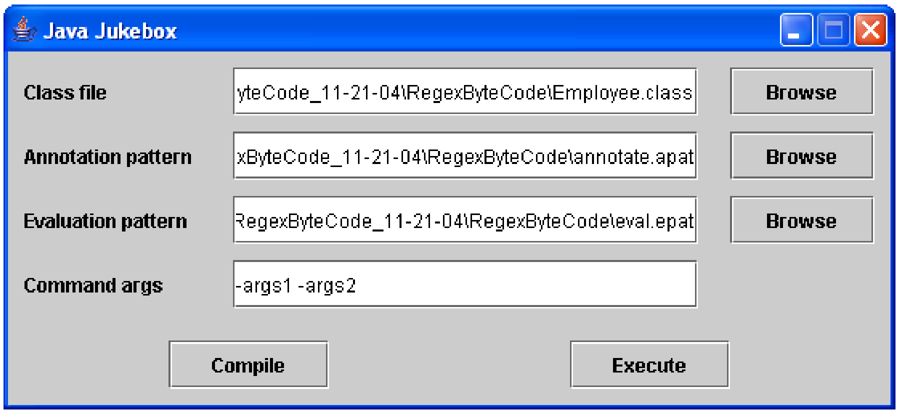

Fig. 3. A screen-shot of the JMusic GUI just before user-class compilation.

Using a script-based approach to specification allows advanced users complete control over both monitoring and music production. Future versions of the system will contain a set of pre-written scripts that cover most common monitoring situations needed by casual users.

2.2 The User Interface

To run JMusic the user must instantiate the `jcaas.util.ClassRunner` class, providing it with the two required scripts, the user program to execute, and the arguments to the user program. The format of the command is as follows:

```
java -classpath jcaas.jar:ProgramClasspath jcaas.util.ClassRunner
     script.apat script.epat MainClass arg1 arg2 ... argN
```

The annotation script file (indicated by the extension `".apat"`) details how the main Java class will be altered to detect significant program events. The auralization script file (indicated by the extension `".epat"`) details how the music should react to the program events specified in the annotation script.

This will allow the given Java program to use JMusic's custom class loader. All arguments after the name of the main Java class will be passed directly to its main method.

The user runs JMusic through a simple graphical user interface. Figure 3 shows a screen-shot of the interface. The first input box allows the user to specify the executable Java `class`- or `jar`-file they wish to execute. The last input box allows the user to specify arguments for the main method of the program. These arguments are treated the same as command line arguments to a Java program.

```
/"ArithmeticInstruction"/ {
   update("math", 1);
}

/"InvokeInstruction"/ {
   update("invoke", 1);
}
```

Fig. 4. The `.apat` file for the anagram solver.

The second and third input boxes allow the user to specify the annotation pattern file and the auralization script file. Finally, the user can run the annotated class file with the `Execute` button. This button simply invokes the main method from the modified Java program `jar`-file with the specified arguments from the fourth input box.

2.3 Annotation

The annotation scripting language is roughly similar to AWK in both semantic and syntactic qualities allowing for easy identification of patterns within class files. Lexical analysis of the scripts is accomplished using JFlex, which allows for the auto generation of a scanner from a set of regular expressions [11]. Parsing the scripts is performed using Java CUP, which builds a parser from a provided grammar [8]. Figure 4 shows an example of the annotation script. The scripting language provides some basic support for flow control structures, integer and double variables, and calling of Java library methods. Variables are dynamically typed, do not need to be declared, and are always global. Support for arrays, boolean types, and data structures is not currently implemented.

The annotation script consists of a set of rules describing what operations to perform when a given pattern is discovered in a list of Java bytecode instructions. Each rule consists of a condition – which will generally be a pattern – and a code block to execute when the condition is true:

```
/pattern/ {
        code block
}
```

A pattern is a string constant encapsulated within two forward slashes. It can be any regular expression [14]. This pattern will be applied to a string representation of an instruction list. This instruction list is found and manipulated by the BCEL package [6]. The most common operation to perform when a pattern is matched is to insert code into the instruction list that will update a variable in the blackboard. The annotation is done in real-time in memory as classes are loaded into the Java virtual machine.

For example, the script in Figure 4 specifies that for every arithmetic instruction in the program, code should be inserted that increments the `math` variable on the `blackboard` by one. Similarly, the script states that for each of Java's `invoke` instructions (there are four: `invokestatic` used for static methods, `invokevirtual` for virtual methods, etc), the blackboard's `invoke` variable should be incremented.

2.4 Auralization

Figures 5 and 6 show an example of the auralization script. The auralization script is similar to the annotation script. However, instead of searching for regular expressions in the class bytecode, it examines the state of the blackboard to notify the MIDI player thread when user-defined conditions are met:

```
[condition] {
        code block
}
```

In addition, blackboard variables may be examined directly, by referencing them with a '$' character at the beginning. These variables are updated by the execution of the program, and hence and cannot be altered by the auralization script.

The auralization script is capable of manipulating

- tempo, with the `setTempo()` function
- volume, with the `setVolume()` function
- instrument, with the `setInstrument()` function
- chord, with the `setChord()` function, and
- current song (selected from one of several songs builtin to JMusic).

If more control is necessary, songs can also be generated on the fly. A song contains all the functionality that may be expected of a musical staff, including full range of notes, the ability to produce chords and other polyphonic sounds, and variable note lengths. The auralization script can switch to a pre-defined song, or dynamically create new songs. This allows the user to specify a melody, tempo, and chord to be played on each non-rest note. The user may also manipulate these properties as the song is played, without needing to create a new song. The melody may either be manipulated by replacing a single note, or by replacing the entire melody with a new one. These changes will not cause the music to restart, whereas other methods of manipulating the melody – such as creating a new song – will cause it to restart.

```
INIT {
   tempo=150;  volume=600;  songSet=0;
   old_math=0;  old_mem=0;  old_invoke=0; old_stack=0;

   # amount of time between testing of conditions
   setCycleDelay(20);

   # tell the class loader not to annotate
   # some library packages
   ignorePackage("java.");
   ignorePackage("sun.");
   ignorePackage("javax.");
   ignorePackage("org.apache.bcel.");
   ignorePackage("org.apache.regexp.");

   # Very important to ignore this one!
   ignorePackage("jcaas.");

   setSong("The Imperial March");
}
```

```
# This procedure evaluates whenever the
# number of invoke instructions increases.
# It maps the change in invoke instruction
# calls to various instruments.
[ $invoke >= old_invoke ]{
   temp=$invoke;
   if( temp > old_invoke + 100000 ) {
      old_invoke=temp;
      setInstrument(20);
   } else {
      if( temp > old_invoke + 10000 ) {
         old_invoke=temp;
         setInstrument(4);
      } else {
         if(temp > old_invoke + 1000) {
            old_invoke=temp;
            setInstrument(47);
         } else {
            old_invoke=temp;
            setInstrument(42);
         }
      }
   }
}
```

Fig. 5. The `.epat` file for the anagram solver (part 1).

```
# This procedure evaluates whenever there is an
# increase in the number of math instructions
# executed. It maps the change in math instruction
# execution to different tempos.
[ $math >= old_math ] {
  temp=$math;
  if( temp > old_math + 100000 ) {
    setTempo(500);
  } else {
    if( temp > old_math + 50000 ) {
      setTempo(475);
    } else {
      if( temp > old_math + 25000 ) {
        setTempo(425);
      } else {
        if( temp > old_math + 12000 ) {
          setTempo(350);
        } else {
          if( temp > old_math + 6000 ) {
            setTempo(250);
          } else {
            if( temp > old_math + 300 ) {
              setTempo(100);
            } else {
              setTempo(50);
            }
          }
        }
      }
    }
  }
  old_math=temp;
}
```

Fig. 6. The `.epat` file for the anagram solver (part 2).

2.5 The MIDI Player

Music is produced in MIDI format using the system's default MIDI interface. The MIDI player links the auralization script to Java's MIDI implementation. It combines the settings described in the auralization script to dictate note length, pitch, volume, and instrument of the desired audio output and inputs them into the system's MIDI interface. The MIDI player also keeps track of the songs defined by the auralization script. To avoid consuming excessive system resources, for each note played it determines the amount of time until the next note will be played and causes its thread to sleep for that duration. The MIDI player may

```
INIT {
   # Variables to store current state
   current_nesting=0; current_mel=0;
   current_instr=0; current_loopstart_iteration=0;
   # Variables to store old BlackBoard values
   old_loop_enter=0; old_loop_exit=0; old_loop_iteration=0;
   switch_melody=0; switch_instr=0;
   setMelody("1:F+2;.5:G+2;.5:A+2;.5:B+2;1:C+3;....");
   setModeGenerate();
}
[ $loop_enter-$loop_exit>current_nesting ||
  $loop_enter-$loop_exit < current_nesting ] {
   current_nesting=$loop_enter-$loop_exit; switch_instr=1;
}
# entered a loop
[ $loop_enter>old_loop_enter || $loop_exit>old_loop_exit ] {
   switch_melody=1;
   old_loop_enter=$loop_enter; old_loop_exit=$loop_exit;
   current_loopstart_iteration=$loop_iteration;
}
[ $loop_iteration-current_loopstart_iteration >0 ] {
   val=$loop_iteration-current_loopstart_iteration;
   setTempo(100 + val*10);
}
[ switch_instr == 1 ] {
   setInstrument(current_nesting * 10); switch_instr=0;
}
[ switch_melody == 1 ] {
   if( current_mel == 0 ) {
      switchMelody(".5:A+2;.5:B+2;1:C+3"); current_mel=1;
   } else {
      if( current_mel == 1 ) {
         switchMelody(".5:A+2;.5:B+2;...."); current_mel=2;
      } else {
         if(current_mel == 2 ) {
            switchMelody("1:E;1:D;1:C;...."); current_mel=3;
         } else {
            switchMelody("1:F+2;;...."); current_mel=0;
         }
      }
   }
   switch_melody=0;
}
```

Fig. 7. The `.epat` file for the loop example.

also reset the song at any point, causing the next note played to be the start of the new song.

3 Example 1: Auralization of an Anagram Generator

To illustrate the ability of JMusic to modify existing compositions, consider the example of a simple recursive backtracking problem, in this case an anagram solver. Two strings are anagrams of each other if they contain all of the same characters in the same amounts (ignoring case and whitespace). There is an additional restriction that the string must be composed of words from a given dictionary, separated by whitespace. For example, one anagram for the string "Republican National Convention" would be "Innocence until naval probation" since both contain the same letters, and the same number of each letter. This example will take a phrase and construct every single anagram for it, given a dictionary of words.

To auralize this program, the example script maps operations of the program to various simple manipulations of the composition "The Imperial March" by John Williams. In particular, blocks of code with many method invocations change the instrument and blocks with heavy use of arithmetic operations increase the tempo.

Figure 5 and Figure 6 show the auralization script used. Four instruments are assigned to different levels of method invocation frequency, listed in ascending order: Viola, Orchestral Harp, Honky Tonk Piano, and Church Organ. Since MIDI does not provide a standard mapping from instrument number to actual instrument, our current implementation specifies instrument directly by number. Arithmetic operations can change the tempo to seven different levels, listed in ascending order: 50 beats per minute (bpm), 100bpm, 250bpm, 350bpm, 425bpm, 475bpm, and 500bpm.

The anagram program starts with the "Imperial March" playing on a Viola with a slow tempo since the program is initially blocked waiting for user input. The user enters the name of the dictionary file, which normally causes no noticable change in the music. As dictionary words are added to the internal word list occasionally a brief instrument change occurs. After this, the music returns to its initial state as the program is once again blocked waiting for user input. Entering the phrase "recursion is cool" results in the tempo shooting up noticeably and the instrument changing to Church Organ. After a few seconds the music returns to its previous state, when the program has finished finding all of the anagrams. Entering the phrase "anagrams are fun" results in the tempo shooting up just as before, but in this case there are occasional switches from the Church Organ to the Honky Tonk Piano. Presumably, this is because the rate at which method invocations occur decreases from time to time during the computation on this input.

This example presents a simple method by which existing compositions can be changed to provide audio information about what the program is doing. The next step is not just to modify existing music, but to generate music on-the-

```java
public class LoopExample {
    public static void main(String[] args) {
        for(int i=0; i < 10; i++) {
            sleep(2000);

            for(int j=0; j < 100; j++) {
                sleep(50);
            }

            sleep(50);
        }
        sleep(10);
        for(int i=0; i < 100; i++) {
            sleep(100);
        }
        System.exit(0);
    }

    public static void sleep(int mil) {
        try {
            Thread.sleep(mil);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 8. The user-code for the Loop Example

fly as well as analyze more complex properties of a program other than simple bytecode operations.

4 Example 2: What do loops sound like?

The next example shows how music generation can be used to auralize the control flow of a program, in particular the control flow of simple loops. Transitions into and out of loops result in changes in the melody, series of iterations of a single loop (uninterrupted by exiting the loop or transitioning to a different loop) result in a tempo increase, and nested loops result in a change of instrument.

The simple example auralized here consists of two "for" loops in series with a third loop nested inside the first. The user-program is shown in Figure 8 and the auralization script in Figure 7. The music begins with a Glockenspiel and then quickly transitions to a Church Organ accompanied by a change in melody as execution moves into the nested loop. The tempo then begins to increase as the nested loop iterates. The melody soon changes once again and the instrument changes back to the Church Organ as the inner loop finishes. This process repeats until the first "for" loop is finished. Finally, the last loop is executed, which results in a melody change and a tempo that keeps increasing until the loop exits and the program terminates.

Now a small bug is introduced into the program. The inner loop is altered so that its counter will be decremented with each iteration, thus it will never reach its termination condition. Upon entering the nested loop, the tempo increases to the point that separate notes are no longer distinguishable. Thus, by the auralization alone, the user knows not only that the program appears to have stalled, but that it has stalled in an infinite loop (as opposed to waiting for user input, thread deadlock, etc.).

The above example shows how auralization can be used to inform the user about the flow of a program, as well as to detect undesired behavior of a program without the need of examining the source code.

5 Future Work

The annotation and auralization scripts are ready to be extended beyond regular expressions to allow the user to analyze bytecode in ways not allowed by the current scripting language. In addition, other ways of manipulating the current song, such as by changing the key, rhythm pattern, or clef, may be implemented. With these extensions, JMusic will provide an easily recognizable method of determining status of the executing program when combined with an appropriate script set.

Adding a set of standard annotation and auralization scripts would allow first-time users to immediately get audio from their programs. Creating such generic templates will also be a part of our future work on the Programmer's cockpit (Figure 1). In addition to incorporating several of the existing pieces of the system, we will study the mapping of data to visual, audio, and tactile feedback.

6 Related Work

While we are not aware of existing programs for monitoring the execution of a Java program using music, there is a considerable body of research in program visualization as well as auralization and sonification. Several studies point out the advantages of using musical cues in human-computer interaction [9] and the question "How does a program sound?" has been asked before [1]. Even though sound is more often used in the computing world, it still remains an under-utilized medium. The notion of program "auralization" was introduced by Franconi and Jackson [7] in the context of the use of sound to represent different aspects of the run-time behavior of a program. The notion of "sonification" is broader, in that it refers to the use of non-speech audio to convey information.

The Zeus system introduced algorithm auralization in 1991, by using different musical instruments to represent significant events in sorting algorithms [3, 4]. The Sonnet sound-enhanced debugger triggered various sounds during code execution [10]. It has since evolved into a visual programming language for development of real-time applications.

A number of the existing sonification and auralization tools were designed with specific applications in mind, such as sound-graphs [15] and auditory algebra [17] for visually impaired users. There are also holistic efforts, such as Listen [2], which is a general purpose tool for programs in the C language.

7 Conclusion

We have presented a simple program that analyzes the run-time behavior of Java programs. The user provides script files that determine the action of the analysis and how results are reported. Through an extension of this system's design and implementation, we can provide a new method of interpreting the behavior of programs in real-time in a way that is immediately useful to the user.

References

1. R. Baecker, C. DiGiano, and A. Marcus. Software visualization for debugging. *Commun. ACM*, 40(4):44–54, 1997.
2. D. Boardman, G. Greene, V. Khandelwal, and A. P. Mathur. Listen: A tool to investigate the use of sound for the analysis of program behavior. In *Proceedings of the 19th International Computer Software and Applications Conference*, pages 184–193, 1995.
3. M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. Technical Report 75, 28 1992.
4. M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *COMPUTER*, 25(12):52–63, 1992.
5. C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *ACM Symposium on Software Visualization (SoftVis)*, pages 77–86, 2003.
6. A. S. Foundation. BCEL, 2003. `http://jakarta.apache.org/bcel`.
7. J. M. Francioni and J. A. Jackson. Breaking the silence: auralization of parallel program behavior. *J. Parallel Distrib. Comput.*, 18(2):181–194, 1993.
8. S. Hudson. Cup parser generator for java, 1999. `http://www.cs.princeton.edu/~appel/modern/java/CUP`.
9. C. Hummels, P. Ross, and K. Overbeeke. In search of resonant human computer interaction: Building and testing aesthetic installations. In *9th International Conference on Human-Computer Interaction (INTERACT)*, pages 399–406, 2003.
10. D. H. Jameson. Sonnet: Audio-enhanced monitoring and debuggin. In *Proceedings of the 1st International Conference on Auditory Display*, pages 253–265, 1992.
11. G. Klein. Jflex-the fast scanner generator for java, 2004. `http://jflex.de`.
12. Kramer *et al.* The sonification report: Status of the field and research agenda. 1999. NSF Report by the members of ICAD.

13. E. Larson. J2me optimization tips and tools, 2002. `http://developers.sun.com/techtopics/mobility/midp/ttips/optimize`.
14. J. Locke. Jakarta regexp, 2004. `http://jakarta.apache.org/regexp`.
15. D. L. M. Mansur, M. M. Blattner, and K. I. Joy. Sound-graphs: A numerical data analysis method for the blind. In *Proc. the 18th Hawaii Int. Conference on System Sciences*, pages 198–203, 1985.
16. S. P. Reiss. Visualizing java in action. In *ACM Symposium on Software Visualization*, pages 123–132, 2003.
17. R. D. Stevens, S. A. Brewster, P. C. Wright, and A. D. N. Edwards. Design and evaluation of an auditory glance at algebra for blind readers. In *Proceedings of the Second International Conference on Auditory Display.* Addison-Wesley, 1994.