LISTEN: A TOOL TO INVESTIGATE THE USE OF SOUND FOR THE ANALYSIS OF PROGRAM BEHAVIOR*

David B. Boardman Vivek Khandelwal Geoffrey Greene
Aditya P. Mathur

February 26, 2001

Abstract

We describe the architecture and use of a tool named `LISTEN`. This is a general purpose tool to instrument computer programs so that during program execution aspects of program behavior are mapped to audible sound. Ongoing research aimed at investigating the usefulness of sound in various programming-related tasks and a lack of supporting tools led to the development of `LISTEN`. This tool is expected to find use in tasks such as program testing and debugging, development of programming environments for the visually handicapped, and data analysis using aural cues. We also report our initial experience gathered during exploratory use of `LISTEN` and provide a summary of ongoing research using this tool.

1 Introduction

Visualization is a widely used means of displaying various aspects of a program's run time behavior. By a program's run time behavior we mean the flow of control and the change of values of various data items computed by the program. For example, during debugging, one may observe the path followed by a program on a given test case by watching on a display screen the movement of a dot through various nodes in the flowgraph of the program. Simulation of an automobile engine is often accompanied by a visualization of the movement of the pistons through various cylinders. More examples of visualization of program behavior can be found in the literature on program visualization.

In comparison to graphics/text based visualization, sound has been used much less in displaying aspects of program behavior. However, we do find examples of research aimed at investigating the use of sound in understanding program behavior and in adding sound to simulations for increased appeal. A survey of the use of sound in understanding program behavior is reported by Francioni and Jackson [7]. In an early work Gaver [8, 9] proposed the use of auditory icons for use as a part of Apple's interface on the Macintosh machines. It is believed, and has been demonstrated in a few cases, that the use of sound can enhance program understanding. Yeung [15] proposed the use of sound to recognize multivariate analytical data. Francioni and Jackson [7] used program auralization to understand the runtime behavior of parallel programs. Brown and Hershberger [5] auralized some of their

*This work was supported in part by an educational supplement from the National Science Foundation No. CCR 9102311 and 9123502-CDA. The authors are with Software Engineering Research Center and Department of Computer Sciences, Purdue University, W. Lafayette, IN 47907. David B. Boardman, Vivek Khandelwal, Geoffrey Greene, and Aditya P. Mathur may be contacted at, respectively, boardman@cs.purdue.edu, vk@cs.purdue.edu, greene@cs.purdue.edu, and apm@cs.purdue.edu.

animations generated using the Zeus animation system. In his doctoral dissertation, Edwards [6] built and evaluated a word processor with an audio interface for use by visually handicapped users.

Programming environments and applications developed so far have attempted to use visual media to a great extent [1]. Though research in the use of audio in workstations [10, 13] has been on the rise, audio remains a distant second to visual media. However, with the availability of relatively inexpensive audio devices such as synthesizer modules and sound digitizers and the provision of sound generators in workstations, audio is within reach of most PC and workstation users.

We have designed and implemented `LISTEN` to provide a general purpose environment for adding sound to a program so as to be able to "listen" to any aspect of its behavior. The approach used in `LISTEN` is intended to simplify the task of adding sound to a program resulting in relative ease of experimentation in this area. In the remainder of this paper we describe the architecture of `LISTEN`, features of a language LSL used in `LISTEN`, and an illustrative example to show the utility of `LISTEN`.

Section 2 describes the architecture of the `LISTEN` system. An intermediate language, named Listen Specification Language (LSL), that serves as the carrier of a programmer's intentions regarding mapping program behavior to sound, is reviewed briefly in Section 3. A simple experiment to evaluate the use of sound in software testing is described in Section 4. Conclusions from and status of the current research are reported in Section 5.

2 Components and processes in `LISTEN`

`LISTEN` instruments a given C program to enable the generation of sound during its execution . Various components of `LISTEN` and the processes executed during its use are shown in Figure 1. A brief description of the system follows; details appear in [3].

As shown in this figure, a user of `LISTEN` begins by editing the source program, if not available, and a specification file. The specification file, hereafter referred to as an Auralization Specification (ASPEC) file, contains commands written in LSL. As described in Section 3, these commands specify the mapping of program behavior to sound. The program source file and the ASPEC are input to lslCC which parses the ASPEC to create an Auralization Database consisting of data obtained from the ASPEC. Next, the source program is parsed and a parse tree is created. This parse tree is now decorated by adding instrumentation using information contained in the Auralization Database. The instrumentation consists of some initialization code and calls to sound generation routines placed appropriately in the tree. These sound generation routines are a part of the LSL Library. The decorated parse tree is then deparsed and instrumented C source code generated.

During the execution of the instrumented program calls to LSL library routines generate data sent to a sound generation device. In the current implementation a synthesizer module[1] is used to generate sound. Through the ASPEC one can control several characteristics of the sound so generated. Some of these characteristics are pitch, timbre, time of generation, duration, and rate. A run time system, not shown in Figure 1, allows alteration of these parameters during program execution. A Graphical User Interface allows the easy creation and modification of an ASPEC. An ASPEC is an ASCII file and can also be edited using any text editor.

[1]We currently use a Roland Sound Canvas and a Proteus/3 World as the synthesizer modules to generate sound.

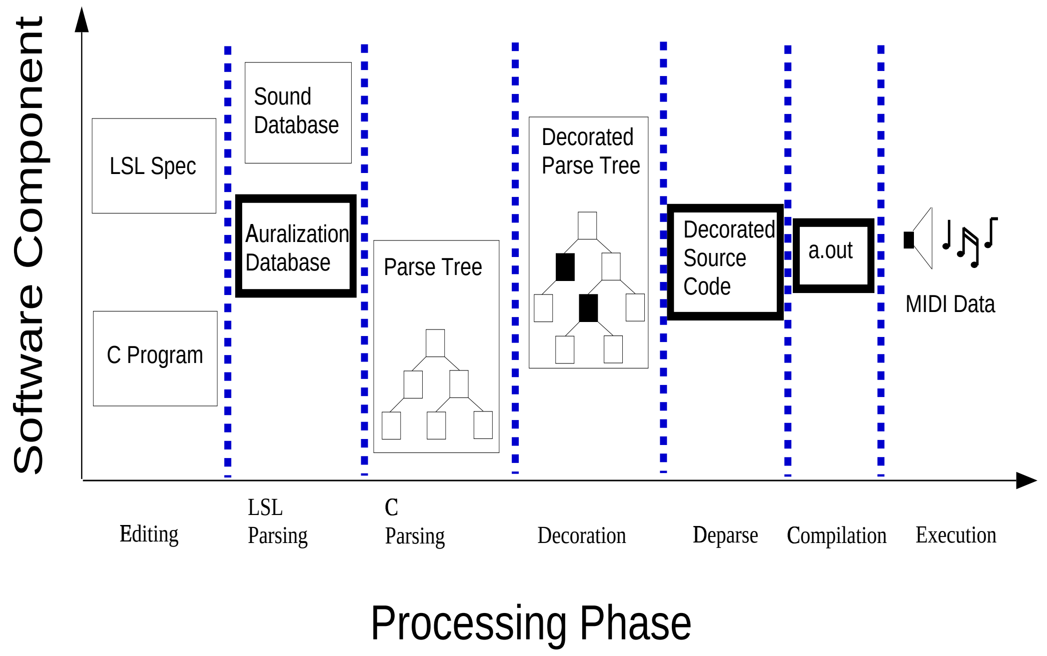

Figure 1: Components and processes in `LISTEN`.

3 An overview of LSL

As mentioned earlier, LSL is a language to specify mappings from program behavior to sound. The design goals for LSL are:

1. *Generality*: It should be possible to specify any auralization using LSL.
2. *Language independence*: It should be possible to use LSL with the commonly used programming languages such as C, C^{++}, Ada, Pascal, and Fortran.

Below we provide a brief explanation of how the above goals were met and review key features of the language. An illustrative example is provided to help in understanding the use of LSL and the `LISTEN` system.

3.1 ASPECs and realizations

To be able to design a language using which one can specify all possible auralizations, a quantification of two domains is established. Let E be the domain of all those occurrences during the execution of any program that one may wish to auralize. The nature of such occurrences is discussed below. Let S be the domain of all possible sound patterns that may be associated with each element of E. A mapping from E to S is an association of sound patterns in S to occurrences in E. Such a mapping is specified as a set of pairs (e, s) where $e \in E$ and $s \in S$. The term *program auralization* for a given program P refers to the set $\{(e_1, s_1), (e_2, s_2), \ldots, (e_n, s_n)\}$, where each $(e_i, s_i), 1 \leq i \leq n$ is an association of an occurrence to a sound pattern. In this paper we refer to a program that emits sound during execution as an "auralized program". The task of instrumenting a program so that it emits sound during execution will be referred to as "program auralization".

A *language* L for program auralization is a notation to specify any such mapping for any program. A mapping specified using L is referred to as *auralization specification* abbreviated as ASPEC. Specifications are always written with reference to a given, though

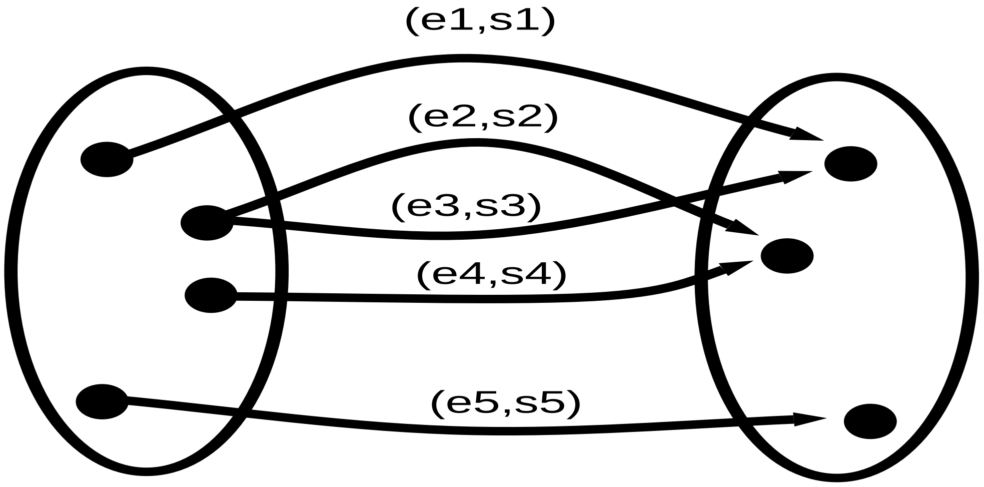

Figure 2: A domain based view of program auralization.

arbitrary, program in some programming language. Figure 2 illustrates this view of program auralization. Note that an ASPEC is a many-to-many mapping.

Let (e, s) be an element of an ASPEC for program P. During the execution of P if each occurrence e is identified by a sound pattern s, it is said that the pair (e, s) has been *realized*. An ASPEC for program P is considered realized if all its elements are realized for all executions of P. An implementation of L for programs in a given programming language PL is said to be correct if each ASPEC, written in L, for any program P, written in PL is realized.

3.2 Occurrence space characterization

Ideally, it should be possible to specify any auralization. To do so, the space of all possible occurrences that might arise during program execution must be defined. Towards this end a three-dimensional space using the orthogonal notions of position, data, and time are selected. Position refers to any identifiable point in a program. For example, in a C program, beginning of a function call, end of a function return, start of a while-loop, start of a while-loop body, and start of a condition, are all positions. In general, an identifiable point is any point in the program at which an executable syntactic entity begins or ends. This implies that a position cannot be in the middle of an identifier or a constant. In terms of a parse tree for a given program, any node of the parse tree denotes a position. For example, the subscripted dot $(\bullet_i)$ denotes seven possible positions in the following assignment: $\bullet_1 X \bullet_2 = \bullet_3 X \bullet_4 + \bullet_5\, 3 \bullet_6 / \bullet_7\, 2$.

Data in a program refers to constants allowed in the language of the program being auralized and the values of program variables. A data relationship is an expression consisting of constants, variables, and function calls. Time refers to the execution time of the program. It is measured in units dependent on the system responsible for the execution of the auralized program. In a heterogeneous system, time is measured in units agreed upon by all elements of the system.

As shown in Figure 3, a three dimensional space is used for specifying occurrences in LSL. Two kinds of occurrences are distinguished: events and activities. LSL allows an arbitrary combination of data relationships, positions, and time to specify an event or an activity associated with program execution.

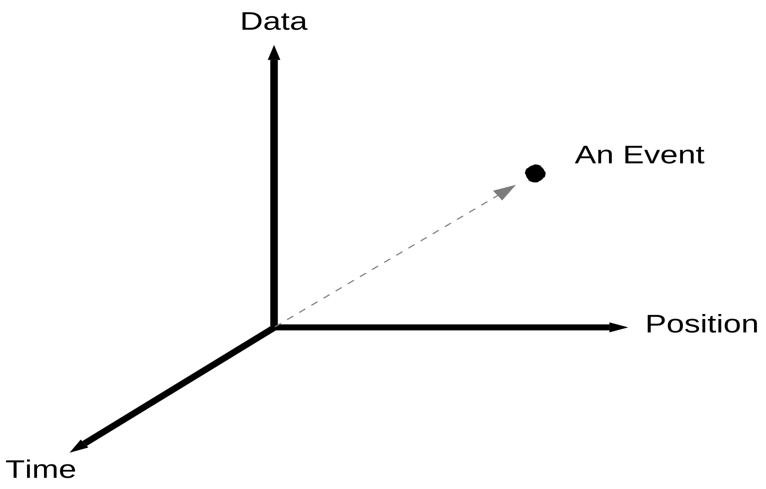

Figure 3: Occurrence space characterization in LSL.

3.3 Sound space characterization

The sound space is characterized by sound patterns comprised of notes, durations, play styles, and instruments. Notes of arbitrary durations can be combined to form sound patterns. Each note can be associated with one of several play styles and with an arbitrary instrument. For example, a note can be played staccato on a piano with a specified volume. Combining notes in various ways gives rise to a domain consisting of an infinity of sound patterns. Digitized sound, such as human voice, is considered a sound pattern.

3.4 Programming language independence

The second requirement stated above is significant as LSL should be usable by programmers regardless of their preference for one programming language or the other. Adherence to this requirement has produced a language which in the strict sense should be considered as a meta-language. One can therefore adapt LSL to specific programming languages. However, in the implementation for our research the LSL/C, an adaptation of LSL for the C programming language [11] is implemented.

3.5 Features and syntax of LSL

Next, we briefly review the features of LSL; details may be found in [2, 4]. An LSL program is known as a specification. Each specification is composed of one or more specification modules. Each specification module is composed of zero or more specification definitions and one main specification. A specification module, a specification definition, and a main specification are analogous to, respectively, a module, a procedure, and a module body in a Modula-2 [14] program.

Event specification

As mentioned above, one can map program events, activities, and data to sounds using LSL commands. Event specification is achieved by the `notify` command. `notify` is a generic command and can be adapted to a variety of procedural languages. In examples below we assume that programs being auralized have been coded in C.

1. `notify all rule`=while_loop_body_begin `using` body_begin;

`notify all rule`= while_loop_body_end `using` body_end;

These specify two event types, namely the beginning and end of a while-loop body using two general purpose syntactic specifiers. It also indicates that all positions in the program where such events could occur are to be auralized. Thus, a C program auralized using the above `notify` will generate the sound corresponding to the variables body_begin and body_end, respectively, whenever the beginning and end of a while-loop body are executed.

2. `notify selective label` = special_loop `rule`=while_loop_body_begin `using` body_begin;

   This is the same as Example 1 except that the event selection is selective. Thus, any loop body labeled by *special_loop* will be auralized. Any syntactic entity can be labeled in the program being auralized by placing an LSL `label` command in front of that entity.

3. `notify all instance`= "++count" `and` "search(x)" `using` count_or_search `in func` = "search", "report";

   This specifies the execution of the statements *++count* and *search(x)* as the events. When any of these two events occur, *count_or_search* is played. However, these events are to be recognized only inside functions *search* and *report.*

4. `notify all assertion` = (x<y || p≥q) `using` assertion_failed;

   This specifies an event which occurs whenever the condition
   (x<y || p≥q) is not satisfied. Note that this condition is based on variables in the program being auralized. When this condition is not satisfied, *assertion_failed* is to be played.

5. `notify all rule` = conditional_expression `and assertion` = odd(x) `using` cond_sound `in filename` = "myfile.c";

   This example shows how to specify the auralization of all conditional expressions that occur in file *myfile.c* only when condition *odd(x)* is not satisfied.

6. The `all` and `selective` tags can restrict any event selection. Multiple labels are used within one `notify` command as in the following.

   `notify selective label` = loop_1, loop_2 `rule`=while_loop_body_begin `using` body_begin;

   `notify selective label` = special_loop `rule`= while_loop_body_end
   `using` body_end;

Data tracking

Event notification consists of specifying one or more events and reporting them aurally during program execution. There are applications wherein changes to values of variables need to be monitored. It is certainly possible to specify assignments to such variables as events and then report the execution of these assignments aurally. Such reporting is, however, independent of the data being assigned. To obtain data dependent auralization, LSL provides the `dtrack` command. A few examples of `dtrack` use appear below.

1. `dtrack` *speed*; will track variable *speed* using an initial value of 0 and default sound parameters such as note pitch and volume.

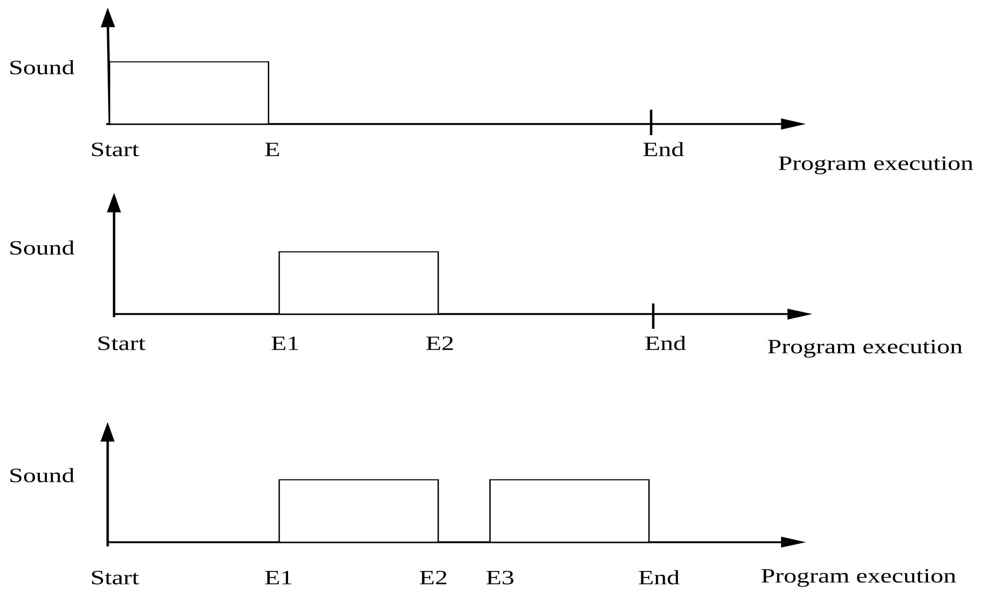

Figure 4: Sample activity patterns specifiable in LSL. Start, E, E1, E2, E3 and End denote events that occur during program execution. The vertical axis indicates the presence or absence of sound.

2. `dtrack` *crash* `init=false`; will track *crash* assuming an initial value of `false`.
3. `dtrack` *x* `capture`=x_reset; will track *x* after capturing its initial value at the assignment labeled by the LSL label *x_reset*
4. `dtrack` *mouse* `and` *color* `using` *color_mouse_melody* (&mouseval, &colorval) ; will track variables *mouse* and *color* using a user defined function named *color_mouse_melody* with two parameters.
5. `dtrack` *speed* `when` speed>65 `until` x$\leq$65 `mode=continuous`; will begin tracking *speed* whenever its value exceeds 65 and will stop tracking it immediately after its value becomes equal to or less than 65. Tracking will resume if the start event occurs again. The discrete mode can be used to avoid resumption of tracking of *speed.*

Activity tracking

An activity is a sequence of actions between two events. An activity begins at the occurrence of an event and ends at the occurrence of a later event. As mentioned earlier, start and termination of program execution are considered as events. LSL allows the specification of tracking arbitrary activities using the `atrack` command. Using the start and terminating events one may specify a variety of activity tracking patterns as shown in Figure 4.

4 Experience with `LISTEN`

We have conducted several simple experiments with `LISTEN`. Below we describe an experiment aimed at exploring the use of sound in understanding the behavior of a program and determining if one can obtain information about certain attributes of the program, specifically efficiency and correctness, simply by listening to it during execution.

4.1 Listening to an inefficient selection sort

In one experiment a selection sort[12] program was coded by an individual. A suitable ASPEC for this program was also written by the same individual. Two other members of our research team, who had not seen the code listened to the selection sort for various input data values. While experimenting with this sort routine we posed the following questions: *What would we learn when we hear the program*? We then decided to auralize the program so that during its execution we can listen to (a) the start and end of program execution, (b) the elements that are being exchanged, (c) the start of the body of the for-loop inside which comparisons are made between data being sorted, and (d) the execution of each program statement. Item (d) above is also known as a *heartbeat*. While listening to a program a hearbeat provides a sense of timing. Our selection sort program sorts an input array consisting of N integers into ascending order. Note that there are many program-sound mappings. We selected one of these mappings arbitrarily.

An ASPEC written in LSL that meets the above goal is shown in Figure 6. The `notify` command specifes the beginning of the for-loop body as an event. This event is mapped to a `Sticks` sound. This is one of a large number of sounds that are a part of the sound database in the `LISTEN` system[2] The `dtrack` command specifies that the variable `temp`, used inside function `exchange` is to be tracked using the `Flute2` sound, also a part of the `LISTEN` sound database. This implies that whenever the value of `temp` changes, the `Flute2` sound will be emitted. The `syncto mmrel` command specifies that the heartbeat will be 120 beats per minute. Thus, when an event occurs, the emitted sound will be synchronized to this heartbeat. An alternative would be to synchronize sound generation to program execution. The latter scheme is not useful for this example as it results in a sound pattern that lasts for an interval too short for the human ear to identify distinct notes because the program executes very fast, and therefore is unable to convey any useful information.

Having edited the ASPEC file, we compile the sort program and the ASPEC using `lslCC`. This results in an executable code. When the executable code is executed on a given test input one hears the notes, emitted by the program, via the synthesizer module and a set of speakers. Figure 7 shows the score, in western staff notation[3], the notes that are emitted when the sort program is executed. These notes were generated for an input consisting of four elements 1, 3, 5, and 6. Notice that this is an already sorted array. The division of the score into different measures and the key and time signatures is arbitrary in this example. Note, however, that LSL has the ability to specify these parameters to control the various characteristics of the generated sound pattern.

There are four staves in Figure 7 one corresponding to each of the different event or data being tracked. These staves have been labelled for clarity. In this example, we assigned, respectively, woodblock, tubular bell, stick, and flute sounds to the four staves labelled `Heartbeat`, `Program`, `for body begin`, and `Exchange`. The `Hearbeat` staff has a total of 77 notes. Each of these corresponds to the execution of a program statement. A hearbeat note is generated when any statement of the user program is executed. Heartbeat notes are not generated when a system library routine (e.g. `printf`) is invoked. The `Program` staff has just two notes, that are aligned with, respectively, the first and the last hearbeat notes. The `for body begin` staff has a total of six notes aligned with heartbeat notes 29, 30, 31, 41, 42, and 52. The first three notes correspond to the first execution of the loop, the next

[2]A sound in the Listen sound database consists of a type which could be MIDI sound or voice. In the current version of `LISTEN` only MIDI sounds are supported. For each MIDI sound the database contains the channel number, instrument code, and pitch code. MIDI is an acronym for Musical Instrument Digital Interface. It is a serial interface to connect computers and musical instruments.

[3]We are assuming that the reader is familiar with the staff notation. For those who are not and would like to listen to the sound generated in our experiments, a cassette tape is available from any of the authors.

```
#include <stdio.h>

#define MAX                     100
#define TRUE                    1
#define FALSE                   0
#define NORMALIZE_FACTOR        70

int A[MAX];
static int ecount=0;
static int num_of_elements;

void main()
{
        int i;

        read_numof_elements();
        read_array();

        print_array(A);
        selection_sort(A);
        print_array(A);

        printf("\nTotal exchanges = %d\n", ecount);
}

void exchange(x, y)
int *x;
int *y;
{
        int temp;

        ecount++;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

Figure 5: (a) Functions `main` and `exchange` for the selection sort routine.

```
void selection_sort(int A[MAX])
{
        int i;
        int j;
        int position;
        int smallest;

        i = 0;

        while (i<num_of_elements)
        {
                position = i;
                smallest = A[position];

                /* find smallest from i+1 th element to the last */

                for (j=i+i; j<num_of_elements; j++)
                {
                        if (A[j] < smallest)
                        {
                                position = j;
                                smallest = A[position];
                        }
                }

                /* put the smallest element thus found at i'th place */

                exchange(&A[position],&A[i]);
                ++i;
        }
}
```

Figure 5: (b) Function `selection_sort`. Functions for input/output of array elements are not listed here.

```
begin auralspec
specmodule paper-example
begin paper-example

        syncto mmabs q = 120;

        notify rule = prog_begin using Begin_snd;
        notify rule = prog_end using End_snd;

        notify rule = for_body_begin
                using Sticks_snd
                in func = "bubble_sort" and func = "selection_sort";

        dtrack temp
                when rule = function_entry:"exchange"
                until rule = function_return:"exchange"
                using Flute2_snd;

end paper-example;
end auralspec.
```

Figure 6: Sample ASPEC for the selection sort program.

two to the second execution, and the last one to the third and final execution. Lastly, the `Exchange` staff has four notes aligned with heartbeat notes 35, 46, 56, and 65.

Members of our development team listened to the sounds generated during the execution of selection sort program on different test inputs. What surprised us, at least the ones who had not coded the sort program, was the presence of the four notes corresponding to the exchange function for a sorted input. Obviously, we had not expected to hear the flute sound which was assigned to the exchange operation. When we checked with the individual who had coded the program we discovered that the program was correct but inefficient and was performing the exchanges anyway.

4.2 Listening to an erroneous selection sort

The individual who coded the selection sort had another experience to report. A typing mistake occured during the initial coding of selection of selection sort. As a consequence there was an error in the `for` loop inside the selection sort function. The incorrect loop was

```
for (j=i+i; j<num_of_elements; j++)
```

where the initialization `j=i+1` was incorrectly typed in as `j=i+i`. This resulted in a sound for two exchange operations without an intervening sound for any comparison. This deviation from expected aural behavior of the program alerted the individual who coded the program and led to the discovery of the error. The sound pattern that led to the discovery of this above error resulted from the exchange operations. Several exchange operations were heard without any sound from corresponding to a comparison. This aural behavior of the program surprised the listener, who had coded the algorithm, and led him to the error.

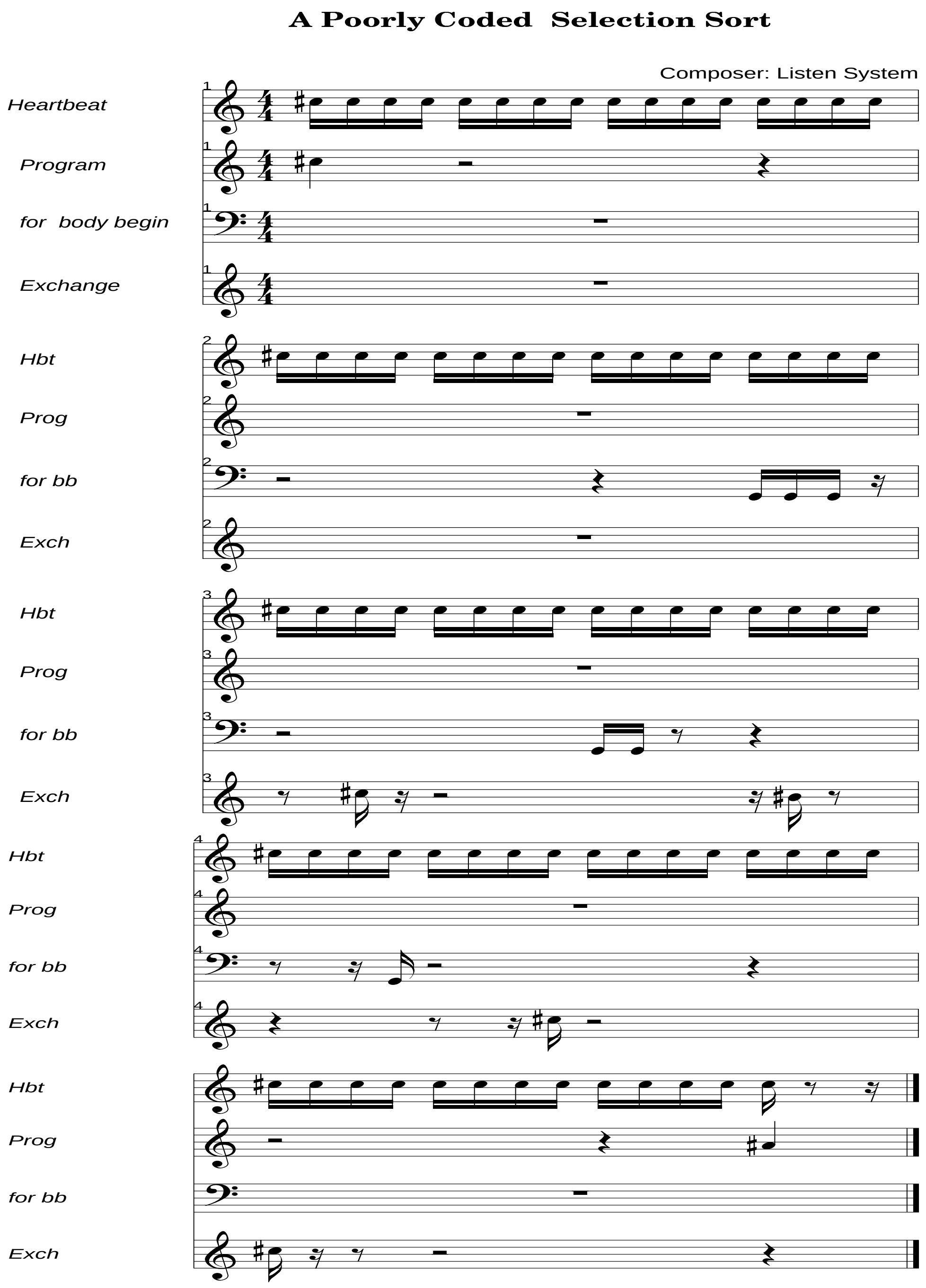

Figure 7: Score representing the notes generated by the auralized version of the selection sort.

4.3 Listening to other sort programs

We made the following observations while "listening" to a few other sorting programs which include bubble sort and insertion sort:

- The sounds generated during the execution of a program on a given input, form a distinctive pattern. This pattern is known as the *aural signature* of the program on the given input.
- Based on its aural signature, an efficient implementation can be differentiated from an inefficient one. By "efficient" we mean efficient in terms of any measurable quantity associated with program execution e.g. execution time, memory usage, number of disk accesses etc.
- The "goodness" of an aural signature depends, among other factors, on the sound specification (the ASPEC). Here "goodness" refers to the amount of useful information that the aural signature reveals about the program's behavior.

5 Summary, status, and conclusions

We have described a system named `LISTEN` that has been designed and prototyped to experiment with the use of sounds in understanding and analyzing run time behavior of computer programs. The system allows an experimenter to specify a mapping of program related various events, activities, and program variables to almost an infinite variety of sound patterns. It is expected, and has been our initial experience, that such a mapping will allow the listener of a program to begin expecting certain sounds from the program. When one does not hear as per expectations one begins to suspect some anomaly in the code. Such a suspicion could be merely a misunderstanding on the part of the listener or could lead to the discovery of a program error. We believe the basic approach used in our work will be useful in the development of software systems that provide program auralization in addition to or as an alternative to the visual method of displaying program behavior. In the case of a user with vision related disabilities, such an alternative might be attractive.

Further development of the `LISTEN` system is continuing. The focus of current work is to (a) develop a run time graphical interface for altering program-sound mappings during program execution, (b) develop an auralized version of a debugging tool for use by blind users, and (c) explore the use of aural cues in program testing and debugging.

From our interactions with programs we have arrived at the following conclusions:

1. Repeated listening to an auralized program creates an "aural signature" in the mind of the listener. The listener's mind tends to question the existence of patterns when they do not conform to an initial expectation. It is this questioning that may lead to one or more of the following (a) improved understanding of the program and (b) discovery of an error.
2. The benefit of sound in a computing environment will be directly related to the quality of the ASPEC. One must choose sound patterns which portray appropriate information with respect to a given occurrence mapping.
3. With respect to debugging or analyzing behavior, it appears that a developer must spend significant time becoming familiar with the sound of a system or program. If this aural training period is not realized it appears a user gains less information from

the sound. This suggests that there is a training time or sensitizing period that a developer must go through to gain the greatest benefit from sound in a computing environment.

4. It may be possible to establish a general ASPEC which creates a distinctive aural signature for a C program in an application domain. An aural signature may provide insight into program behavior. Based on initial experiences a developer may determine what part of the code is executing, how the program behaves on a given input, and narrow the search space when debugging.

Acknowledgements

Bob Liu and Jack Lawry helped in the development of the LSL compiler. Bob Horgan provided support in the use of ATAC. Verna Abe and Helen Brown taught the rudiments of music theory that shaped the design of LSL. Will Montgomery and Neil Herzinger answered endless MIDI related questions. Our thanks go to all these individuals.

References

[1] A. L. Ambler and M. M. Burnett. Influence of visual technology on the evolution of language environments. *IEEE Computer*, 22(10):9–22, 1989.

[2] *LSL: A Specification Language for Program Auralization*, 1994.

[3] D. B. Boardman. Listen: An environment for program auralization. Master's thesis, Purdue University, Department of Computer Science, W. Lafayette, IN 47907, 1994.

[4] D. B. Boardman and A. P. Mathur. Preliminary report on design rationale, syntax, and semantics of LSL: A specification language for program auralization. Technical Report SERC-TR-143-P, Software Engineering Research Center, Purdue University, W. Lafayette, IN, USA, 1993.

[5] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, December 1992.

[6] A. D. N. Edwards. Soundtrack: An auditory interface for blind users. *Human-Computer Interaction*, 4(1):45–66, 1989.

[7] J. M. Francioni and J. A. Jackson. Breaking the silence: Auralization of parallel program behavior. Technical Report TR 92-5-1, Computer Science Department, University of Southwestern Louisiana,, 1992.

[8] W. W. Gaver. Using sound in computer interfaces. *Human-Computer Interaction*, 2:167–177, 1986.

[9] W. W. Gaver. The sonicfinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4(1):67–94, 1989.

[10] R. Kamel, K. Emami, and R. Eckert. Px: Supporting voice in workstations. *IEEE Computer*, 23(8):73–80, 1990.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[12] D. E. Knuth. *Art of Computer Programming: Sorting and Searching Algorithms*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1975.

[13] L. F. Ludwig, N. Pincever, and M. Cohen. Extending the notion of a window system to audio. *IEEE Computer*, 23(8):66–72, 1990.

[14] R. Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley Publishing Company, Reading, MA, 1989.

[15] E. S. Yeung. Pattern recognition by audio representation of multivariate analytical data. *Analytical Chemistry*, 52(7):1120–1123, June 1980.