ON PROGRAM AURALIZATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Vivek Khandelwal

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 1995
ACKNOWLEDGMENTS

I am grateful to my advisor, Professor Aditya Mathur, who provided the motivation and guidance without which this work would not have been possible. I would like to thank members of my thesis committee, Professor Vernon Rego and Professor Chandrajit Bajaj for their assistance and encouragement. Thanks are also due to Professor Thomas Kuczak and Michelle McNabb for their help on statistical analysis.

Members of the Listen group, especially, Nate Nystrom, Howard Chen, and Leo Rijadi were involved in various stages of software development. Dave Boardman, Geoff Greene, and Praerit Garg provided several valuable ideas relating to Listen. My thanks go to all these individuals. I express special thanks to Professor José Maldonado for his idea of applying Listen to mutation testing which relates closely to the experimental work described in this thesis.

Dr William Gorman, and Mary Jo Maslin deserve special thanks for guiding me through the administrative procedures and providing a smile and comforting word when the chips were down.

Finally, I thank my parents and my family for their love and emotional support in all my endeavors.
DISCARD THIS PAGE
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Organization of this Thesis</td>
<td>2</td>
</tr>
<tr>
<td>2. AN OVERVIEW OF THE USE OF AUDIO IN COMPUTING</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Terms and Definitions</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Auditory Displays</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Auralization/Sonification Tools</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Behavioral Research Related to Use of Audio in Human-Computer Interface</td>
<td>7</td>
</tr>
<tr>
<td>3. LISTEN V2.0: ARCHITECTURE AND IMPLEMENTATION</td>
<td>8</td>
</tr>
<tr>
<td>3.1 The Listen Preprocessor (ls1cc)</td>
<td>8</td>
</tr>
<tr>
<td>3.1.1 Environment Setup</td>
<td>9</td>
</tr>
<tr>
<td>3.1.2 Syntax</td>
<td>11</td>
</tr>
<tr>
<td>3.1.3 Value Dependent Aural Pattern (VDAP)</td>
<td>12</td>
</tr>
<tr>
<td>3.1.4 VDAP Implementation</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Listen Sound Library</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Sequence Interrupt Handler</td>
<td>19</td>
</tr>
<tr>
<td>3.2.2 Porting Listen to Solaris</td>
<td>21</td>
</tr>
<tr>
<td>3.3 The Graphical User Interface</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1 Screens</td>
<td>23</td>
</tr>
<tr>
<td>3.3.2 Environment Setup</td>
<td>24</td>
</tr>
<tr>
<td>3.3.3 Implementation</td>
<td>25</td>
</tr>
<tr>
<td>----------------------</td>
<td>----</td>
</tr>
<tr>
<td>3.4 Run-Time Audio Controller</td>
<td>47</td>
</tr>
<tr>
<td>3.4.1 Screens</td>
<td>47</td>
</tr>
<tr>
<td>3.4.2 Architecture</td>
<td>48</td>
</tr>
<tr>
<td>3.4.3 Implementation</td>
<td>49</td>
</tr>
<tr>
<td>3.4.4 Limitations and Future Enhancements</td>
<td>50</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>50</td>
</tr>
<tr>
<td>4. EFFECTIVENESS OF AURAL, VISUAL, AND AURAL-VISUAL CUES: AN EXPERIMENT</td>
<td>56</td>
</tr>
<tr>
<td>4.1 Method</td>
<td>57</td>
</tr>
<tr>
<td>4.1.1 Subjects</td>
<td>57</td>
</tr>
<tr>
<td>4.1.2 Input Variables</td>
<td>57</td>
</tr>
<tr>
<td>4.1.3 Response Variables</td>
<td>58</td>
</tr>
<tr>
<td>4.1.4 Hypotheses</td>
<td>58</td>
</tr>
<tr>
<td>4.1.5 Materials</td>
<td>58</td>
</tr>
<tr>
<td>4.1.6 Visual to Aural Output Mappings</td>
<td>59</td>
</tr>
<tr>
<td>4.1.7 Procedure</td>
<td>61</td>
</tr>
<tr>
<td>4.1.8 Training</td>
<td>61</td>
</tr>
<tr>
<td>4.1.9 Measurements</td>
<td>65</td>
</tr>
<tr>
<td>4.2 Evolution of the Main Experiment</td>
<td>65</td>
</tr>
<tr>
<td>4.2.1 Aural Output</td>
<td>65</td>
</tr>
<tr>
<td>4.2.2 Visual Output</td>
<td>65</td>
</tr>
<tr>
<td>4.2.3 Aural-Visual Output</td>
<td>66</td>
</tr>
<tr>
<td>4.2.4 Pilot Study</td>
<td>66</td>
</tr>
<tr>
<td>4.2.5 Design Decisions</td>
<td>67</td>
</tr>
<tr>
<td>4.3 Results and Analysis</td>
<td>67</td>
</tr>
<tr>
<td>4.4 Discussion</td>
<td>69</td>
</tr>
<tr>
<td>4.4.1 Limitations of Our Study</td>
<td>69</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>70</td>
</tr>
<tr>
<td>5. CONCLUSIONS</td>
<td>73</td>
</tr>
<tr>
<td>5.1 Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>73</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>76</td>
</tr>
<tr>
<td>APPENDICES</td>
<td></td>
</tr>
<tr>
<td>Appendix A: LSL Grammar</td>
<td>80</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Table of means showing effect of cue type on correctness</td>
<td>67</td>
</tr>
<tr>
<td>4.2 Table of means showing effect of cue type on time-taken</td>
<td>68</td>
</tr>
<tr>
<td>Appendix</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>A.1 Language Dependent Terminals in LSL Grammar.</td>
<td>93</td>
</tr>
<tr>
<td>C.1 Mutants for Cal program</td>
<td>136</td>
</tr>
<tr>
<td>C.2 Mutants for Look program</td>
<td>136</td>
</tr>
<tr>
<td>C.3 Mutants for Sort program</td>
<td>137</td>
</tr>
<tr>
<td>C.4 Raw Data on correctness by cue type. Each entry is the value of</td>
<td>138</td>
</tr>
<tr>
<td>correctness for a given subject using a given cue type.</td>
<td></td>
</tr>
<tr>
<td>C.5 Raw Data on time-taken by cue type. Each entry is the value of</td>
<td>139</td>
</tr>
<tr>
<td>time-taken for a given subject using a given cue type.</td>
<td></td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Auralization using Listen. Components in Listen and their interaction.</td>
<td>9</td>
</tr>
<tr>
<td>3.2 File <code>rockets.lsl</code>. Contains an LSL specification to track <code>dist_remain</code> and <code>rock_remain</code> using a VDAP in conjunction with a <code>dtrack</code>. VDAP function contains one LSL command <code>play</code> with several parameters which depend on the variables being tracked. Each such command is replaced by an appropriate function call. Syntax of the C code inside a VDAP function is not checked for correctness by <code>lslCC</code>.</td>
<td>13</td>
</tr>
<tr>
<td>3.3 File <code>rockets.c</code>. Contains the main program which defines two variables <code>dist_remain</code> and <code>rock_remain</code> whose dynamic relationship is to be tracked using a VDAP in conjunction with a <code>dtrack</code>.</td>
<td>14</td>
</tr>
<tr>
<td>3.4 VDAP implementation. In this example, two temporary files containing VDAP code get generated. The total number of temporary C files is communicated to <code>lslCC</code> which constructs the names and includes them for compilation.</td>
<td>16</td>
</tr>
<tr>
<td>3.5 Sample data structures from file <code>vdap.h</code></td>
<td>17</td>
</tr>
<tr>
<td>3.6 Sample VDAP processor routines from file <code>vdap.c</code></td>
<td>18</td>
</tr>
<tr>
<td>3.7 Some functions from file <code>lib.c</code>. This file contains the upper layer routines for Listen library.</td>
<td>20</td>
</tr>
<tr>
<td>3.8 Workstation and sound-production hardware used in Listen. A MIDI synthesizer module is connected to a workstation via a serial port (<code>/dev/ttya</code>). A computer serial port to MIDI interface (MIDIATOR) is used. MIDI output is connected to a wireless speaker system and headphones.</td>
<td>22</td>
</tr>
</tbody>
</table>
Figure

3.9 Main Screen. The text editor shown here is the main screen of the Listen Graphical User Interface. The menu bar on top contains standard X application features such as File and Edit. Buttons along the bottom provide quick-click options. On the top right, names of the C and LSL files (Cal.c and Cal.lsl in this example) are displayed. A special note icon, located near the top right, is used to create an aural breakpoint during program execution. .......................... 32

3.10 Compiler Preferences Screen. Default compilation command is ls1cc. Options to ls1cc may be specified. The options are described in Section 3.1.2. A make command may be specified. ......................... 33

3.11 Add-Sounds Screen. All specification commands present in the current LSL file are listed. Selective commands may be listed by choosing one or more of the options Track Data, Track Activity, and Notify Event. 34

3.12 New Specification Screen. A new specification command may be created or an existing one may be modified. The type of command must be chosen from the list near the top right corner. The sound to be used must be specified using another screen. ................................. 35

3.13 Data Track Screen. A new specification command to track data values may be created or an existing one may be modified. Start and end events must be specified. Variable name (cp in this figure) whose value is to be tracked is specified. More than one variable names may be given by separating them with keyword and. The sound to be used must be specified using another screen. ................................. 36

3.14 Activity Track Screen. A new specification command to track an activity may be created or an existing one may be modified. Start and end events must be specified. The sound to be used must be specified using a screen not shown here. ................................. 37

3.15 Notify Screen. A new specification command to notify the occurrence of an event may be created or an existing one may be modified. The event to be notified must be specified. The sound to be used must be specified using a screen not shown here. ................................. 38

3.16 Sound Pattern Screen. This screen displays a list of predefined sounds loaded from the sound database. A sound can be played by highlighting it and clicking Play. ................................. 39
3.17 Edit Class Screen. Classes are used to group one or more specification commands together. Every specification command can be included in one or more classes. A new class may be added or an existing one may be modified using this screen.

3.18 Event Screen. Listen provides pre-defined special events based on programming language constructs. One such category of special events is Program which contains Special Syntactic Entities `prog_begin` and `prog_end`. These correspond to the start and end of program execution respectively. Here `prog_end` is selected by highlighting it.

3.19 Event Screen. Another category of special events is Functions which contains Special Syntactic Entities `function_call`, `function_entry` and `function_return`. Here `function_entry` is selected by highlighting it. There is a subtle difference between `function_call` and `function_entry`. The event `function_call` occurs *before* the call whereas `function_entry` occurs *after* control has reached inside the function.

3.20 Event Screen. Another category of special events is Selection which contains Special Syntactic Entities related to `if` and `switch` statements of C. Here `if_then_part` has been selected by highlighting it.

3.21 Synchronization Type Screen. An auralization may be synchronized to a metronome or to the program. In case of metronome, tempo must be specified. The default tempo is 120 beats per minute (bpm). `Heartbeat` can be turned on/off. When turned on, it generates a note using a default sound after each statement gets executed thus providing a relative sense of timing during execution.

3.22 Autosave Screen. The text editor saves the C and LSL files periodically to disk. User can specify the duration after which buffers should be saved. Default autosave duration is 15 minutes.

3.23 Search Screen. The text editor provides a facility to do simple string search inside the currently loaded C file.

3.24 Annotated Main Screen. Each arrow points to a component of the GUI screen and shows the name of the source file which implements that component. For example, `menu.c` contains the source to implement the menu bar on top.

3.25 Architecture of GUI shown at source file level.
3.26 File Menu available from the main screen. Options in this menu deal with the executable file, LSL file, and sound database. Names of the executable file, and LSL file may alternatively be specified on the command-line.

3.27 Specification commands listed according to their names. Each item in the list represents one specification command. There are five specifications in this figure, namely for_loop, go, stop, tmp, and while_loop. The sound name is displayed in the second column. The button with a + sign on it indicates that this specification is turned on. A specification can be turned off by clicking on its button.

3.28 Specification commands listed according to the classes they belong to. Each class may contain one or more specifications. Similarly, a specification may belong to one or more classes. In this figure, three classes are shown, namely data, endpoint, and loop.

3.29 Change Sound screen. This screen is used to change a sound corresponding to a specification or add new ones.

3.30 Edit Existing Sound screen. This screen is used to edit a sound. This figure shows the parameter settings for a predefined sound flute1.snd. New sounds so defined can be saved to the sound database by selecting the option to do the same from File menu. User is prompted in case the sound database has changed during the current session.

4.1 A fragment of the shell script run_sort used by subjects in group G3. The fragment shown here corresponds to the aural cues stage of the experiment. It executes the program Sort on a predetermined input and redirects stdout and stderr to /dev/null. This redirection disallows any visual output to appear on the screen.

4.2 LSL specification file for auralizing UNIX cal program. Command syncdio synchronizes the audio output to a metronome clicking 300 beats per minute. Command dtrack i tracks the value of variable i. The tracking starts after the function cal is entered and tracking ends just before control returns from it. Command notify rule = function_entry:"pstr" and notify rule = function_return:"jan1" notify the two said events by generating a simple note in either case using flute1.snd and jump0 respectively.
4.3 LSL specification file for auralizing UNIX `look` program. Command `syncto` synchronizes the audio output to a metronome clicking 360 beats per minute. Command `dtrack cnt` tracks the value of variable `cnt`. The tracking starts after the function `canon` is entered and ends just before control returns from it. Tracking is defined by a Value Dependent Aural Pattern (VDAP) specified by function `track_cnt`.

4.4 LSL specification file for auralizing UNIX `sort` program. Command `syncto` synchronizes the audio output to a metronome clicking 240 beats per minute. Command `dtrack cp` tracks the value of variable `cp`. The tracking starts after the function `sort` is entered and tracking ends just before control returns from it. Tracking is defined by a Value Dependent Aural Pattern (VDAP) specified by function `out_str_track`.

4.5 Mean values of `correctness` plotted by group and cue type.

4.6 Mean values of `time-taken` plotted by group and cue type.

Appendix

Figure

C.1 Nested Factorial Q-Q Plot for `correctness` to test if residuals are normally distributed. A Shapiro-Wilk test confirms the normal distribution.

C.2 Nested Factorial Q-Q Plot for `time-taken` to test if residuals are normally distributed. A Shapiro-Wilk test confirms the normal distribution.
ABSTRACT


Listen is a general-purpose tool for program auralization. In the first part of this thesis, we describe the architecture and implementation of Listen v2.0 (v1.0 is described elsewhere). Listen provides a preprocessor for auralizing C programs. It also provides graphical user-interfaces to create auralizations and to control audio output at run-time. It allows flexible control of sounds by means of Value Dependent Aural Pattern (VDAP) to track data.

In the second part of this thesis, we describe an experiment to investigate the fault-detection effectiveness of aural, visual, and aural-visual cues. A randomized complete block design was used. UNIX utilities `cal`, `look`, and `sort` were selected as sample programs. Eighteen subjects from Purdue university were given one correct and several incorrect versions of the above three programs. They were asked to classify a given program as correct or incorrect on the basis of the output. The number of correct answers determined their correctness scores. Subjects received mean correctness scores of 0.978, 0.867, and 0.622 using, respectively, aural-visual, visual, and aural cues. The time each subject took to complete the exercise was also recorded. The mean time (in minutes) a subject took in the three cases was, respectively, 2.78, 3.28, and 4.06. An ANOVA revealed that subjects did significantly better using aural-visual cues than the remaining two cue types.
1. INTRODUCTION

Use of audio in computer systems has been on the rise for the past few years [Gav86, Gav89, GS90, LPC90, FJ92, BH92, IEE94, Bla94, Coh94]. Given the ability of sound to convey information in real world, it is natural to consider it as a medium to present information in the world of computer systems. One major technical obstacle to widespread use of sound in computer programs has been the unavailability of general-purpose tools for adding sounds to programs. The Listen system has been developed to automate and thus ease the task of auralizing programs. One can use Listen to map program events and data to sounds, or in other words, to create program auralizations.

1.1 Problem and Motivation

Listen is based on a language for specifying program auralization named Listen Specification Language (LSL) [BM93]. Listen v1.0 implemented a minimal working subset of LSL. Details of this implementation can be found in [Boa94]. Development of Listen v1.0 demonstrated that it was possible to build a general-purpose program auralization tool. However, being a prototype implementation it was far from complete. In order to make Listen more usable, the following enhancements were determined.

1. A more powerful and flexible mechanism to specify data-dependent auralizations. This mechanism should allow a user to specify various parameters related to sound such as pitch, timbre, volume, duration, velocity, and pan. These parameters should depend on information generated at run-time.
2. A system for run-time control of audio output. Examples of desired controls in such a system include pausing and resuming audio output, and modifying sound-related parameters at run-time.

3. An enhanced graphical user-interface which enables the use of various features of LSL without having to learn its syntax.

Once a working version of this tool was available, the next step was to evaluate its usefulness. More specifically, it was important to investigate the applications where Listen could be used and determine if its use enhanced the applications. The questions we decided to answer are:

1. How does audio compare with the visual medium to convey the output of a program?

2. Is audio combined with visual output more effective in conveying the output of a program?

3. How can we auralize commonly-used application programs to make them usable by the visually-impaired?

1.2 Organization of This Thesis

The remainder of this thesis is organized as follows. An overview of the use of audio in computer systems is presented in Chapter 2. This includes a description of some program auralization applications where special-purpose tools were used. Examples of such applications are program debugging tools, simulation tools, and user-interfaces for the visually-impaired. Most of these tools are only suited to one specific application or a specific class of applications. The chapter also describes more recent attempts toward creating general-purpose program auralization tools. The last section mentions past work relevant to the experiment described in Chapter 4.

Chapter 3 begins with a brief description of the architecture of Listen v2.0. It identifies the main components in Listen and how they interact. It then provides
implementation details for each of its subsystems. The subsystems are the Listen
preprocessor (IslCC), the MIDI\textsuperscript{1} sound library, the graphical user interface, and the
run-time controller system. Several screens from the interfaces are shown to describe
their functionality.

Chapter 4 describes an experiment conducted to investigate the effectiveness of
aural, visual, and aural-visual cues in differentiating “correct” and “incorrect” pro-
grams.

A summary of the work, our conclusions, and directions for future research are
presented in Chapter 5.

\footnote{MIDI (Musical Instrument Digital Interface) is a world-wide standard that
provides a way for
electronic musical instruments to communicate. Instruments that have MIDI
connectors can be
connected to any other MIDI device, regardless of the manufacturer or model, and
exchange musical
data as “MIDI messages”.}
2. AN OVERVIEW OF THE USE OF AUDIO IN COMPUTING

Until recently sound has been an under-utilized medium in the computing world. Even with the explosion of multimedia, the focal point has been graphics, animation, and video, with sound sometimes added as an afterthought. Yet people in fields outside of computer science use sounds in many ways to help diagnose problems. A driver knows when to shift gears by listening to the pitch of a car engine. A mechanic can diagnose problems with that car engine by the sounds it makes.

Sound has another useful property compared to vision. Sound waves travel around corners. In other words, sound can be used to draw attention to some event even without the need for visual focusing.

In recent years, however, this potential has been recognized and the use of sound in the computing world has been on the increase. This chapter provides a brief overview of the past research relevant to this work. We first define some of the terms used throughout this thesis.

2.1 Terms and Definitions

Auralization is a term first used by Francioni and Jackson [FJ92] to refer to the use of sound to represent different aspects of the run-time behavior of a program. Sometimes it is used as an audio counterpart to visualization.

Sonification refers to the use of data to control a sound generator for the purpose of monitoring and analysing data [Kra92a]. It is sometimes used as synonymous with auralization. Other terms used for more or less the same purpose are audification and audition.
2.2 Auditory Displays

Auditory display research applies the ways we use sound in everyday life to the human-machine interface and extends these uses via technology. The function of an auditory display [Kra92a] is to help a user monitor and comprehend whatever it is that the sound output represents. The display medium could be speech or non-speech sound. If the display medium is non-speech sound, the auditory display will exploit acquired environmental adaptations, including cognitive and preattentive cues.

Yeung [Yeu80] presented an objective recipe, based on the statistical distribution of data, for audio display of multivariate analytical data. Each measurement in the data vector was translated into an independent property of sound. In his words, “Excellent results were obtained when this method was applied to the pattern recognition of a test data set.”

Gaver [Gav86, Gav89, Gav93] proposed the use of auditory icons for use as a part of Apple’s interface on the Macintosh machines. He used the term auditory icon to describe an icon that depicts an object by mapping it to an “everyday sound”. The audio-enhanced interface which Gaver called SonicFinder was not empirically evaluated, however, it was reported that the interface was considered nicer to use.

Sound-graphs [MBJS85] uses pitch to describe the shape of a graph. Edwards [Edw89] built and evaluated a word processor with an audio interface for use by visually-handicapped users. Ludwig [KEE90, LPC90] prototyped an audio window display for workstations to present multiple audio cues simultaneously. In ARKola [GSO91] bottling plant simulation, subjects used auditory icons to obtain information about other subjects and events inside the plant.

Kramer describes some organizing principles for representing data with sound in [Kra92b]. He discusses techniques for auditory data representation, and some perceptual issues they raise.

Brewster et al. [B+92] reported results comparing the effectiveness of earcons [BSG89] with that of simple tones.
2.3 Auralization/Sonification Tools

Brown and Hershberger [BH92] auralized some of their programs using the Zeus algorithm animation system. They used different instruments to represent significant events in sorting algorithms. They found this technique to be effective for signalling exception conditions, suggesting execution patterns, and reinforcing visual information.

Jameson [Jam92] built a sound-enhanced debugger. The tool, Sonnet, provided a visual programming interface to specify the triggering of sounds during code execution. The facilities such as tracking trends in the value of a variable, and detecting read/write access to a variable are similar to the dtrack command in LSL.

Madhyastha and Reed [MR92, MR95] built and used the Porsonify toolkit to create data sonifications. Porsonify allows a user to define data-to-sound mappings by means of widget-control files. Each sound device is represented with a panel of knobs and a list of buttons; the knobs are turned to appropriate settings and the buttons are pressed to manipulate the sound device. By mapping data to specific knobs and buttons, the data can be displayed by turning one or more knobs and pushing one or more buttons. For example, each data value might push a button to play a note and turn a knob using its magnitude to select the note’s pitch. Listen provides these metaphors for sound control through the run-time audio controller.

Bock [Boc94] describes a language for auditory domain specification. This language lets users define software component sound domains. A domain consists of a set of program constructs and their associated individual audio cues. This approach allows one to use sound to isolate program errors.

The LogoMedia programming environment [DBO93] contains facilities similar to the activity tracking, and data tracking in Listen. Cohen [Coh94] describes a system for monitoring background activity using musical sounds. Stevens et al. [S+94] designed and evaluated an auditory glance at algebra for blind users.
All of the above tools except Zeus algorithm animation system, were developed for specific applications. They are not suitable for use as general-purpose auralization tools. In some cases described above, a more general approach as used in Listen could significantly reduce the amount of effort to create effective auralizations. The auditory glance, and Cohen's background monitoring system could use Listen to create their auralizations. Bock's prototype system is similar to the run-time controller system in Listen wherein different classes of auralizations can be created and controlled.

2.4 Behavioral Research Related to Use of Audio in Human-Computer Interface

Mynatt [Myn92, Myn94] describes the development of GUIs for the visually-impaired. In his doctoral dissertation, Brewster [Bre93] suggests guidelines in the design of auditory interfaces for the visually-impaired. Ballas [Bal92, Bal94] conducted experiments comparing accurate identification of everyday sounds. We use guidelines reported in the above studies to design aural cues for our experiment.

Portigal [Por94] conducted an experiment which compared the effectiveness of aural, visual, and combinations cues to convey the structure of a hyper-text document. Subjects in his experiment demonstrated an equivalent level of understanding of the document structure and its content with either a visual cue or a combination cue. Subjects required more time to answer questions in the combination condition than in the visual condition. The aural cues did not appear promising for the particular task. It was noted, however, that the experiment involved only one possible aural cue. The results showed there may not generalizable to any type of aural cue. The observations may reflect more strongly on the design of the cues rather than on the effectiveness of the cues.
3. LISTEN V2.0: ARCHITECTURE AND IMPLEMENTATION

Listen is a general-purpose tool for program auralization. The current version (Listen v2.0) can be used to auralize programs written in the C language [KR88]. The approach taken in Listen requires that an auralization be specified formally using a language. This is done by creating a specification in LSL (Listen Specification Language). A specification is written to a file. Both a C file and corresponding LSL specification file are input to a preprocessor lslcc which parses the specification and creates an auralization database. Using this auralization database, the parse tree for the C source program is decorated. An instrumented C source file is generated from the decorated tree by a process called deparsing. The instrumented code is then input to a standard C compiler and resulting object code is linked with sound-generation libraries to produce an executable. In v2.0 these libraries contain generic routines which provide an interface to a MIDI [DS88] device. The graphical user-interface to Listen can be used to generate an LSL specification file thereby obviating the need to program in LSL. The run-time audio controller system can be used to control the run-time state of auralizations. Components and processes described above are shown in Figure 3.1.

In the following sections, we describe the implementation of enhancements made in v2.0.

3.1 The Listen Preprocessor (lslcc)

A preprocessor, named lslcc, forms the back-bone of the system. lslcc is responsible for the parsing of C source, parsing of LSL source, creation of auralization database, deparsing of C source, and creation of instrumented C source. Implementation details are not described here, see [Boa94] for internals of Listen v1.0. Among
Figure 3.1: Auralization using Listen. Components in Listen and their interaction.

the new features available in v2.0 preprocessor are enhanced data-tracking mechanism through the use of Value Dependent Aural Pattern (VDAP), and an improved MIDI library to support run-time control of audio. Suspension and resumption of audio output, turning on/off classes of auralizations, modifying sound-related parameters such as timbre (instrument), pitch (note value), volume, duration, pan, and tempo at run-time are examples of run-time control. Run-time audio controller is described in Section 3.1.1.

3.1.1 Environment Setup

To use Listen, environment variables named \texttt{MIDI DEVICE} and \texttt{LSL HOME} must be set to appropriate values.
Below is an example from a `.tcshrc` file which defines these variables.

```
#
# Listen environment
#
setenv MIDI_DEVICE    PROTEUS
setenv LSL_HOME       /homes/uk/u15/ls1pub/sparc
```

Environment variable `MIDI_DEVICE` should be set to the name of the MIDI device (synthesizer module) that will be used. Listen v2.0 handles two makes of MIDI devices namely PROTEUS/3 WORLD and ROLAND Sound Canvas-55. The values of `MIDI_DEVICE` corresponding to these are, respectively, `PROTEUS` and `ROLAND`. A sound database file must exist for the MIDI device being used. This file lists default sounds that are loaded at run-time. Each record in this file contains fields to specify channel number, patch or instrument number, and a value to specify the note to be played. The first three fields are not used. Below is a set of records from the file `ls1.proteus.snds`. For example, the first record in this file defines a sound named `piano_snd` to be the note produced by using channel 5, patch 76, and note 52. The fields containing zeros are currently unused.

```
piano_snd  0 0 0 5 76 52
shaker_snd 0 0 0 6 89 80
flute1_snd 0 0 0 7 15 90
flute2_snd 0 0 0 7 15 78
flute3_snd 0 0 0 8 24 80
flute4_snd 0 0 0 14 22 50
flute5_snd 0 0 0 15 22 40
pbo_snd    0 0 0 4 8 52
pe_snd     0 0 0 4 8 50
speech     0 0 0 4 8 60
```

A user may add more entries to the sound database file. Run-time controller provides a facility to add, delete, or modify a sound in a sound database through a graphical interface.

Environment variable `LSL_HOME` must be set to the complete path name of the directory containing the executables and Listen libraries. `ls1CC` searches this directory to include the required libraries at compile time.
Appendix A contains a complete grammar for LSL (a modified version of the original grammar as given in [BM93]). It also contains the grammar rules alphabetically sorted on the non-terminal on left side of each production rule. The last section in the appendix lists the subset of the grammar used in v2.0.

3.1.2 Syntax

Described below is the command-line syntax of lslcc with all the options:

```
lsllc [-nodelete] [-runtime] [-syncaudio]
     [-compiler <compiler>] <c_file> <lsl_file>
```

- **-nodelete**: To save instrumented source files. The intermediate files are saved in a directory within `/usr/tmp`. The name of this directory is generated by appending a random string to `/usr/tmp/lslcc`. When invoked with this option, lslcc displays the name of the temporary directory. This option is useful for viewing the decorated source code. By default, lslcc deletes any intermediate files created after compilation is over.

- **-runtime**: Allows the use of run-time audio controller to control audio output. This is described in detail in Section 3.4.

- **-syncaudio**: To synchronize visual output with audio. By default, audio output is generated and buffered to a MIDI sequencer which plays notes at a tempo specified by the user. However, this may cause visual and audio output to go out of sync. Use of this option causes audio output to be sent directly to the MIDI device. This blocks program execution until audio output is finished thus synchronizing the two modes of output.

- **-compiler <compiler>**: To specify a compiler to compile the instrumented C file(s). By default, lslcc uses GNU gcc.

- **<c_file>**: Name of the C source file.

- **<lsl_file>**: Name of the corresponding LSL file. The name must end in `.lsl`.
Any other options are simply passed to the compiler as in the first example given below.

```
llslCC -nodelete Cal.c Cal.llsl -o Cal
llslCC -syncaudio -compiler /usr/ucb/cc Sort.c Sort.llsl
```

3.1.3 Value Dependent Aural Pattern (VDAP)

A sound pattern whose characteristics depend on program generated data is referred to as a Value Dependent Aural Pattern (VDAP) [BM93]. To obtain data auralization, LSL provides the `dtrack` command. The syntax of `dtrack` appears below.

```
dtrack <track-id-list> <sound-specifier>

  {<mode-specifier>} {<start-event>} {<term-event>}
```

The `using` clause of `<sound-specifier>` specifies the name of a function, say `f`, that emits a VDAP based on variables being tracked. `f` is a language dependent function containing LSL commands for auralization. Thus, in LSL/C, `f` is a valid C function interspersed with LSL commands inside comments. `f` is executed after each assignment to the variable(s) being tracked because values of those variables may change as a result of the assignment.

The following example illustrates the use of `dtrack` with VDAP to track arbitrary functions of program variables. Suppose it is desired to track the dynamic relationship between two variables named `dist_remain` and `rock_remain`. We define a C function named `crit_dist_track()` which is passed the variables `dist_remain` and `rock_remain`. This function checks that the variables to be tracked are in a specific range and if so, it issues a `play` command whose parameters depend on values of the variables being tracked. As shown in Figure 3.2 any integer value may be specified for parameters `chan`, `volume`, and `note`. However, these parameters must be either an integer constant, or a variable name. Arbitrary expressions are not supported. Figure 3.3 shows the C file for this example.
begin aural spec
VDAP begin

crit_dist_track (int *distval, int *rockval)
{
    int crit_distance=55;
    int crit_rock=100;
    int noteval = *distval-15;

    if ((*distval > crit_distance)&&(rockval < crit_rock))
    {
        /* LSL: begin
        play flute1_end
          with inst = flute,
          mode = continuous,
          chan = 4,
          volume = 70,
          note = noteval;
        end */
    }
}
VDAP end;
specmodule vdap_illustration
begin vdap_illustration
    dtrack dist_remain and rock_remain
    when rule = pb
    until rule = pe
    using crit_dist_track(&dist_remain, &rock_remain);
end vdap_illustration;
end aural spec.

Figure 3.2: File rockets.lsl. Contains an LSL specification to track dist_remain and rock_remain using a VDAP in conjunction with a dtrack. VDAP function contains one LSL command play with several parameters which depend on the variables being tracked. Each such command is replaced by an appropriate function call. Syntax of the C code inside a VDAP function is not checked for correctness by lslCC.
```c
#include <stdio.h>

main()
{
    int i = 0;
    int dist_remain = 50;
    int rock_remain = 60;

    while (i<70) {
        i++;
        dist_remain++;
        printf("dist_remain = %d\n", dist_remain);
        if (i % 10 == 0) {
            rock_remain = rock_remain-2;
        }
    }
}
```

Figure 3.3: File `rockets.c`. Contains the main program which defines two variables `dist_remain` and `rock_remain` whose dynamic relationship is to be tracked using a VDAP in conjunction with a dtrack.
3.1.4 VDAP Implementation

As shown in Figure 3.4, the VDAP specification is parsed and the C source code inside the function extracted. Its syntax is not checked for correctness. The v2.0 implementation imposes a limit of a maximum of MAX_VDAP on the number of total VDAP functions in one LSL file. This parameter is set in a configuration file vdap.h. Each comment of the form shown in the example in Figure 3.2 is parsed to extract LSL commands. The v2.0 implementation recognizes only the play commands. Once the command is parsed and all parameters are found valid, a call to lsl_play4() replaces it. In case -syncaudio option was given, a call to direct_lsl_play4() is placed instead. The newly generated C function is written to a temporary file which is compiled and linked with the main source file. VDAP processor communicates with lslcc via a temporary file total_VDAP_files. This file contains the number of files containing C code extracted from VDAPs. lslcc generates the names of these files and compiles each of them.

Shown in Figure 3.5 are some of the data structures. Default values shown in the figure correspond to the parameters to a play command. Figure 3.6 shows the prototypes of some of the routines from file vdap.c.
Figure 3.4: VDAP implementation. In this example, two temporary files containing VDAP code get generated. The total number of temporary C files is communicated to *ls1cc* which constructs the names and includes them for compilation.
/* information required to process one VDAP */
struct VDAP_info {
    int from;     /* index at which comment starts */
    int to;       /* index at which comment ends */
    char *newcode; /* replace old code by */
};

/**
 *** structure used in extracting LSL commands from
 *** the comments appearing inside a VDAP
 ***/
struct pstatus {
    char *command;   /* command string */
    int x;           /* start position within VDAP code */
};

/* VDAP play command defaults */
#define DEFAULT_INST   "flute"
#define DEFAULT_CHAIN  "4"
#define DEFAULT_MODE   MODE_DISCRETE
#define DEFAULT_VOLUME "127"
#define DEFAULT_NOTE   "80"

Figure 3.5: Sample data structures from file vdap.h
/*
 * process_one_VDAP: Takes the VDAP code and a filename (absolute
 * pathname) and writes the C code corresponding to
 * the VDAP into this file.
 * returns:
 * nothing
 */
void process_one_VDAP(char *s, char *filename)

/*
 * find_LSL_comments: Detects all LSL comments from a given VDAP
 * and records their start and end positions in
 * VDAP_info
 * returns:
 * nothing
 */
void find_LSL_comments(char *s)

/*
 * replace_VDAP_code: Replace LSL commands with appropriate calls to
 * Listen library functions
 * returns:
 * modified command
 */
char *replace_VDAP_code(char *s)

/*
 * init_snd_pars_table:
 * Entries are filled from info given in play
 * Fill table with sound-parameters defaults
 * returns:
 * nothing
 */
void init_snd_pars_table()

Figure 3.6: Sample VDAP processor routines from file vdap.c
3.2 Listen Sound Library

As mentioned earlier, the instrumentation of source code consists of calls to library routines to send sound generation information to a MIDI device. This library provides two layers to provide a complete interface to MIDI. At the lower layer are routines which send a single command to MIDI. Examples of these routines are midiInit(), midiExit(), noteOn(), noteOff(), and playNote(). At the upper layer are routines to which calls are made from the instrumented code. Examples of these routines are _lsl_play_1(), _direct_lsl_play_1(), _lsl_play_2(), and _direct_lsl_play_2().

To provide a synchronized mode where audio output is synchronized with visual output, new routines were added which bypass the MIDI sequencer. These routines communicate directly with the MIDI device via the serial port of the workstation. Figure 3.7 shows the prototypes for these routines.

To support suspension and resumption of audio output from the run-time controller, the following routines were added.

1. void midiPause( void ) is called to suspend audio output. If audio is already suspended, this function returns without changing the state of audio. Otherwise, it records current time in a time variable and sets ispaused to TRUE.

2. void midiResume( void ) is called to resume suspended audio output. If audio was not in suspended state, this function returns without changing the state of audio. Otherwise, it computes the time for which audio was suspended by using the time when suspension began and the current time. It sets ispaused to FALSE.

3.2.1 Sequence Interrupt Handler

The task of Sequence Interrupt Handler is to wake up every 1000 micro seconds and send all those commands to MIDI which should have been sent by current time. It checks if there are any items in the MIDI queue. If none, it returns, else it compares
/*_lsl_say_3 : Speech Generation for dtrack
* (using "say")
*/

_lsl_say_3(value, line, col)

/*
*_lsl_play_4 : Play a note with all parameters
* specified
*/

_lsl_play_4(sndindex, instr, chan, mode, volume, note, line, col)

/*
*_direct_lsl_play_1 : Play a note directly (without going
* through the sequence interrupt handler
*/

_direct_lsl_play_1(sound,line,col)

Figure 3.7: Some functions from file lib.c. This file contains the upper layer routines for Listen library.
the time-stamp for the item at the head of queue with current time. It accounts for
the time program may have been paused by adding pause.sec and pause.microsec
to the time-stamp. If the time-stamp is less than or equal to the current time it
checks if this specification command is turned on. This is done by checking the
status fields in the midibox data structure. There are two separate status values for
the specification, and the class it belongs to (spec->status and class->status). If
status is ON it writes the command to MIDI device else it removes the item without
writing it to MIDI device. The process is repeated for subsequent items in the queue.

Sequence Interrupt Handler is not used when -syncaudio option is used; instead
library routines communicate directly with the MIDI device.

3.2.2 Porting Listen to Solaris

MIDI sequencer uses software signal handling which required rewriting in order
to port Listen from SunOS to Solaris. The code fragment from midi.c shown below
illustrates a difference in the software signal handler interface on Solaris and SunOS.

```
#define sparc5solaris_TARGET_MACHINE

static void handler(signum)
    int signum;
#else
    static void handler(sig,code, scp, addr)
        int sig;
        int code;
        struct sigcontext *scp;
        char *addr;
#endif
```

Several other modules required minor changes. All of those changes can be identified by
searching for the pattern #ifdef sparc5solaris_TARGET_MACHINE.

Figure 3.8 shows a typical arrangement of workstation and other sound-production
hardware used in Listen.
Figure 3.8: Workstation and sound-production hardware used in Listen. A MIDI synthesizer module is connected to a workstation via a serial port (/dev/ttya). A computer serial port to MIDI interface (MIDIATOR) is used. MIDI output is connected to a wireless speaker system and headphones.
3.3 The Graphical User Interface

To auralize a program using Listen one needs to create an LSL file which specifies the auralization. A graphical user interface (GUI) has been developed to ease the task of creating an LSL specification. The GUI provides an easy interface to LSL.

This section describes the functionality provided by the GUI and some implementation details. Figure 3.1 is an architecture diagram showing the interaction among the GUI, the Listen Preprocessor, and other parts of Listen.

3.3.1 Screens

Figure 3.9 shows the main screen. The menu-bar at the top contains File, Edit, Preferences, Sound, MIDI, and Help menus. File menu is similar to many other X applications with such options as, open a C file, create a new file, and save a file. In case a new file is to be created, it is given a default name lsl_noname.c. The names of the current C file and LSL file are displayed at the top right corner. From the Preferences menu, one can specify the command to use for compilation as well as the command to execute. This is shown in Figure 3.10.

The buttons at the bottom of the screen are provided as quick-clicks. Add Sounds button is to specify an LSL command such as notify, atrack, dtrack, syncto. Figure 3.11 comes up when user clicks on this button. This lists all the existing specification commands, if any. Each specification is identified by its name. A specification can be deleted by highlighting its name and then clicking on the Delete button. On choosing New the screen to provide the details of a command pops up. This screen is shown in Figure 3.12. As shown, each new specification is given a default name, for example, spec0. User may specify a different name. One may choose to track data, or track an activity, or notify an event. Figures 3.13, 3.14, and 3.15 illustrate the structure of the screen for each of the three options. In case of data tracking and activity tracking two events must be specified. These are the events to signal the start and end of tracking respectively. In case of notify only one event is to be
specified. The sound associated with a specification can be modified through the **Sound Pattern** dialog box shown in Figure 3.16.

Figure 3.17 shows the screen from which new classes may be defined or existing ones may be edited. Once a class is added, it can be referred to by its name. Different auditory domains may be created by grouping specifications into different classes. For example, when auralizing a compiler, one may create classes of specification commands for lexical analysis, symbol table creation, parsing, and error recovery.

Figure 3.18 shows the screen to select an event. An event can be one of several types such as a General Syntactic Entity, Special Syntactic Entity, Assertion, and Relative Timed Event. In Listen v2.0 Special Syntactic Entity and Assertion are implemented. For a new user Special Syntactic Entity is the easiest to use. In Figure 3.18 **Program** is chosen from the first column **Choose Category** and **prog_begin** is chosen from the second column **Choose Event Specifier**. Similarly there are predefined specific events under each category. Figure 3.19 shows the pre-defined events for **Functions** category. One or more function names (occurring in the current C file) can be chosen. The third column shows the names of all C functions defined. This list is extracted from the currently loaded C file. Figure 3.20 shows the pre-defined events for **Selection**. An exhaustive list of special syntactic entities defined in LSL appears in [BM93].

Figures 3.21 and 3.22 illustrate the **Preference** menu options. Figure 3.21 shows the screen from which the tempo of auralizations can be set. In case a program is to be synchronized with a metronome, a value for beats per minute (bpm) may be specified. As a result, a **syncto mm** command is generated. Figure 3.22 shows the screen from which the duration for the automatic saving of C and LSL files can be specified. The time is specified in minutes (default is 15 minutes).

Figure 3.23 shows the searching facility provided by the text editor of the GUI.

### 3.3.2 Environment Setup

The following commands may be placed inside `.Xdefaults` file to set up default resources:
The `buttoncolor` and `textwindowcolor` resources may be set to any valid Xwindow color. The resource `keyboardFocusPolicy` determines how a widget will be activated when running an X application. If it is set to `pointer`, then one need only point to a text widget in order to activate it. If this is not set, then one must click on the widget.

The `Xls1CC` script requires that two environment variables `LSL_HOME` and `DISPLAY` be set. The script uses the value of `LSL_HOME` to find the executable `Xls1` and to set appropriate fonts. Fonts used in the interface can be changed by editing the file `Xls1CC`.

Invocation syntax is:

```
Xls1CC [<C_file_name>] [<lsl_file_name>]
```

File name arguments are optional. They can be specified from the File menu. If only the C file name is specified, `Xls1CC` finds its base name by stripping off the trailing `.c` extension. It then loads the corresponding LSL file if present in the same directory as the C file, otherwise it starts with an empty LSL file. Only one C file may be specified, however, one can switch to a different file from within the interface. This allows for handling multiple C files in conjunction with one or more LSL files.

3.3.3 Implementation

The GUI is expected to be used in the X-Windows [You94, ON92] graphical environment for a C program which compiles without errors on a standard C compiler. The C program is then loaded into the GUI where the user specifies positions in the program at which sounds are to be added, choosing different sounds to “auralize”

---

1For more information on Xwindow colors, see file `rgb.txt` on your local system. On Purdue CS departmental machines this file is `/usr/local/X11/src/mit/rgb`.
2A particular font is required in order to display the special symbol for a note. The command used to install this font is `xset fp+ $LSL_HOME/font`. 
these positions. When the user is done specifying different program “events” to au-
ralize, the GUI creates a complete LSL specification for the program. The C program
and the LSL specification file are then ready to be compiled by ls1cc.

The LSL GUI can be used for two major tasks: editing C programs and creating
LSL specifications. The editor includes word wrap and indentation, and is more useful
to fix simple errors in a C program than to write one. The LSL specification editor
provides all of the commands in LSL with the exception of VDAP.

Terms and Definitions

Callback is the name for a link between a widget and a function. After a callback
is set, activating the widget calls the corresponding function. For instance, clicking
on a button calls a function set by the callback. Multiple callbacks may be defined
for the same widget.

Trip is an event which can be notified using a notify command of LSL. This
allows a user to add a sound to the program by adding trips to a specific place the
user wants hear sound. The name “trip” comes from the idea that the program
“trips” across that location in the source code setting off a sound.

Widget is the name for an object in the X-windows environment. Buttons, scroll
lists, text boxes, entire windows, and menus are examples of widgets. Widgets are
configured through a number of functions but do not appear on the screen until they
are realized by the window manager. Widget are generally placed on top of other
widgets.

Items on the Main Window

The main window has two ways to access different tools in the GUI, from the
menu at the top of the main window, and from the buttons at the bottom of the main
window. The menu and the buttons are all widgets that have callbacks to various
functions in the program. Pressing a button or selecting a menu option executes the
function written on the button. The only exceptions are the rightmost buttons on
the interface (Word Wrap and Midi Through) which toggle global flags to be true or false. A miniature button is placed inside these buttons so that when the flag is set to true, the mini-button appears depressed inside the larger button. When the flag is toggled to false, the mini-button appears to pop out to the same level as the larger button.

The button with a musical note on it in the upper right corner of the main window is used to place trip events in the main text window. Left-clicking on the note button changes the cursor to a note icon; middle-clicking on an area in the text window places a trip in the program at the place in the program that the cursor points to, and a note icon is placed in text window. The note that appears in the text is a modified text character that is added to the text.

The text window is used to edit a C program. It is implemented using a standard Motif widget. Every time a character is deleted in the text window, a callback calls a function to check if that character is a trip character. If it is, then the corresponding trip data structure is removed from the specification database.

The Add Sounds button and New Specification menu item both call the auralize list window. This window presents a list of all existing specifications and allows the user to organize them alphabetically, by class, and by type (dtrack, atrack, notify). Although only one of these lists is shown to the user at any time, a class list and an alphabetical specification list exist in memory. Both these lists must be updated when specifications are changed.

When the New button of the auralize list window is clicked, three functions are called; the first creates a window which gives the user a list of default sounds to add to their program, the second creates the specification creation window, and the third closes the auralize list window. When the OK button is clicked from the sound window, it uses specification database functions to change the current specification’s sound to the one selected. Control then passes to the specification creation window and the sound window is closed.
The specification creation window gives a way for the user to specify the type of a specification (atrack, dtrack, notify) and the events that it should monitor. Clicking any of the specification type buttons changes the form widget in the center of the creation window. When a new type button is pressed, the old set of options is removed and a new set is placed in the window. The specification window also allows for creation, deletion, and editing of events. When an event is created or edited, an empty template of an event is created. The template information is changed to old values if an event is being edited and the old event is deleted. Deletion of events involves removing it from the database and the event list window.

The event creation window consists of three lists that user can choose from. The first list contains types of events (loops, functions, etc.). The second list contains specific events (for_loop.enter, function.return, while_body.begin etc.). The third list contains a list of functions. A button above the window labeled Press for Functions must be pressed before a function list is made. Extracting the list of functions involves parsing the source code which may take a significant amount of time, so the list is only computed when required. When a selection is made in the first window, all the entries are taken out of the second window and the appropriate set of events is popped up based on the selection made from the first window. Therefore, selecting functions in the first window would pop up function.begin, function.entry, and function.return in the second window. When the OK button is pressed, the event is stored using specification database functions.

Figure 3.24 points out the major components in the GUI and what they implement on the main screen. Figure 3.25 shows an architecture diagram of the GUI at source file level. Not all files are shown in this figure.

Data Structures

The data structures used to create the GUI are often used to pass and return X-windows related variables. As the GUI also parses LSL files, it uses the data
structures created by the LSL preprocessor. The following is a brief description of each of the data structures used by various components of the GUI.

`f_info` indicates if the current C file needs to be recompiled. Text editor uses this structure to keep track of the save status of each C file that is loaded in a session.

`event_info_type` is used when event lists are displayed. When it is passed to the event editing function, if edit is set to true, the create event window sets up the editing of the current event instead of creating a new event.

`event_info_type`

- **Widget list:** all the created events
- **Widget text:** text box to enter event name
- **Widget sourcelist:** list of events
- **int edit:** non-zero if the event should be edited

`aur_info_type` is used to pool together information about a specific specification until the specification is created.

`aur_info_type`

- **Widget relationship:** event information for a motif
- **Widget start_relationship:** event information for an atrack/dtrack
- **Widget end_relationship:** event information for an atrack/dtrack
- **Widget atrack:** whether this specification is an atrack
- **Widget dtrack:** whether this specification is an atrack
- **Widget notify:** whether this specification is an atrack
- **Widget name:** name of the auralization
- **Widget var_list:** list of variables to track for a dtrack
- **Widget discrete:** dtrack in discrete mode
- **Widget continuous:** continuous mode
- **Widget sustain:** sustain mode
- **Widget selective:** notify selective
- **Widget all:** notify all
Widget first: is it the first occurrence?
Widget class_list: list of possible member classes
Widget label_list: list of labels
char *init_value: initial value of the dtrack
int edit: is the an edited event or a new event?

`list_by_type` is used to pass information about how the `ListAuralizations` function should display the existing specifications to the user.

`list_by_type`

Widget by_alph_toggle: button to toggle to show specs in alphabetical order
Widget by_alph_list: list in alphabetical order
Widget by_class_list: button to toggle to show specs grouped by classes
Widget class_list: list according to classes
Widget notifies: notify toggle button
Widget dtracks: dtracks toggle button
Widget atracks: atracks toggle button

`three_widget_type` is used generically to pass three widgets through function calls and returns. However, the only usage of it in v2.0 is for passing events associated with dtrack, atrack, and notify.

`three_widget_type`

Widget first: event information for a notify
Widget second: event information for an atrack/dtrack
Widget third: event information for an atrack/dtrack

`file_data_type` is used to pass additional data about files.

`file_data_type`

Widget proj:
Widget list:
Widget lsl_file:

**file_box_info_type** is passed to the file window function to specify default path and title for a window.

**file_box_info_type**

char *filter: default file selection filter
char *path: default path
char *title: title of the save box
void (*func) (): function to call upon pressing OK on the file selection window

**file_data_type** *data: extra file data

**sure_info**

char *title: string that will be typed in the dialog string, "Are you sure you want to XXX?"
void *func(): function the box calls upon pressing of OK button
char *button_name: string to name the OK button to
void *data: extra data to pass through to dialog box

Algorithms

Events are stored by name in a binary search tree. Each event is stored only once even though it may be referred to in several specification commands. The events in the tree are stored alphabetically without regard to tree balance. Retrieval is done using binary search.

Function names are found by a simple parse of the program. A scanner either returns strings of alphanumeric characters or a non-alphanumeric symbol. Based on this return value, the parser identifies which of the scanned strings are function names.
Figure 3.9: Main Screen. The text editor shown here is the main screen of the Listen Graphical User Interface. The menu bar on top contains standard X application features such as File and Edit. Buttons along the bottom provide quick-click options. On the top right, names of the C and LSL files (Cal.c and Cal.lsl in this example) are displayed. A special note icon, located near the top right, is used to create an aural breakpoint during program execution.
Figure 3.10: Compiler Preferences Screen. Default compilation command is `islCC`. Options to `islCC` may be specified. The options are described in Section 3.1.2. A `make` command may be specified.
Figure 3.11: Add-Sounds Screen. All specification commands present in the current LSL file are listed. Selective commands may be listed by choosing one or more of the options Track Data, Track Activity, and Notify Event.
Figure 3.12: New Specification Screen. A new specification command may be created or an existing one may be modified. The type of command must be chosen from the list near the top right corner. The sound to be used must be specified using another screen.
Figure 3.13: Data Track Screen. A new specification command to track data values may be created or an existing one may be modified. Start and end events must be specified. Variable name (cp in this figure) whose value is to be tracked is specified. More than one variable names may be given by separating them with keyword **and**. The sound to be used must be specified using another screen.
Figure 3.14: Activity Track Screen. A new specification command to track an activity may be created or an existing one may be modified. Start and end events must be specified. The sound to be used must be specified using a screen not shown here.
Figure 3.15: Notify Screen. A new specification command to notify the occurrence of an event may be created or an existing one may be modified. The event to be notified must be specified. The sound to be used must be specified using a screen not shown here.
Figure 3.16: Sound Pattern Screen. This screen displays a list of predefined sounds loaded from the sound database. A sound can be played by highlighting it and clicking Play.

Figure 3.17: Edit Class Screen. Classes are used to group one or more specification commands together. Every specification command can be included in one or more classes. A new class may be added or an existing one may be modified using this screen.
Figure 3.18: Event Screen. Listen provides pre-defined special events based on programming language constructs. One such category of special events is Program which contains Special Syntactic Entities `prog_begin` and `prog_end`. These correspond to the start and end of program execution respectively. Here `prog_end` is selected by highlighting it.
Figure 3.19: Event Screen. Another category of special events is Functions which contains Special Syntactic Entities function_call, function_entry and function_return. Here function_entry is selected by highlighting it. There is a subtle difference between function_call and function_entry. The event function_call occurs before the call whereas function_entry occurs after control has reached inside the function.
Figure 3.20: Event Screen. Another category of special events is Selection which contains Special Syntactic Entities related to `if` and `switch` statements of C. Here `if_then_part` has been selected by highlighting it.
Figure 3.21: Synchronization Type Screen. An auralization may be synchronized to a metronome or to the program. In case of metronome, tempo must be specified. The default tempo is 120 beats per minute (bpm). **Heartbeat** can be turned on/off. When turned on, it generates a note using a default sound after each statement gets executed thus providing a relative sense of timing during execution.

Figure 3.22: Autosave Screen. The text editor saves the C and LSL files periodically to disk. User can specify the duration after which buffers should be saved. Default autosave duration is 15 minutes.
Figure 3.23: Search Screen. The text editor provides a facility to do simple string search inside the currently loaded C file.
Figure 3.24: Annotated Main Screen. Each arrow points to a component of the GUI screen and shows the name of the source file which implements that component. For example, menu.c contains the source to implement the menu bar on top.
Figure 3.25: Architecture of GUI shown at source file level.
3.4 Run-Time Audio Controller

The Listen run-time system allows a user to control the execution of auralized programs. It allows a user to change the attributes related to the audio output generated by a program. Such attributes include the name of the instrument, pitch, duration, velocity, and pan. It allows for turning on/off sounds for a specification command or classes of such commands. It allows for suspension and resumption of the audio output.

3.4.1 Screens

This section contains figures showing screens to illustrate the functionality of the run-time audio controller. Figure 3.26 shows the main screen with its File menu. Figure 3.27 shows the main screen after an executable and an LSL file have been loaded. A list of specifications sorted alphabetically on the name of the specification is displayed in the middle portion of the screen. Sound name corresponding to a specification is displayed in the second column. Buttons at the bottom, are used to control the execution.

Figure 3.28 shows the main screen after an executable and an LSL file have been loaded. A list of specifications is displayed in the middle portion of the screen. This list also shows the class to which a specification belongs. The list is sorted alphabetically on the class name, and within a class it is sorted on specification name.

Figure 3.29 shows the screen used to change the sound corresponding to a specification. A specification must be selected in order to get to this screen. One can turn on/off sound corresponding to a particular specification from this screen. Choosing Edit Existing Sound, or Add New Sound pops-up the screen shown in Figure 3.30. This screen allows one to specify the channel, note value, velocity, pan and duration by means of slider bars. Slider bars ensure that the input values are in valid ranges.
3.4.2 Architecture

The run-time system is comprised of three major components.

1. A sound server to maintain a map between each LSL specification and the sound associated with the specification. It also stores information on whether the specification is on or off. The server interacts with the MIDI device to play sounds when requested to do so. The sound server can handle specifications from only one LSL file.

2. The auralized program sends commands to the sound server to play the sound(s) associated with a given specification. This program is a client.

3. xlisten is an X-based graphical user-interface that provides the means for controlling the auralized program. Through it, the sound server can be sent commands to turn specifications or classes of specifications on/off, to re-map sound attributes, to play sounds for testing purposes, and to create new sounds to add to the sound database.

When a program is auralized using the -runtime option of ls1cc, the program can be executed through xlisten or as a stand-alone application. If run without the GUI, the auralized program starts the sound server, load the default sound database and LSL specification file and have the server play sounds as appropriate. If run with the GUI, the GUI starts the sound server and load the appropriate files. When the GUI executes the auralized program, the program (client) will connect to the server and play sounds as appropriate. As the server currently supports only one LSL specification file at a time, each GUI and auralized program interacts with a single invocation of the server.

The auralized program and the GUI communicate with the sound server through a UNIX socket. This socket, smdserver, is created in a temporary directory when the server starts and is destroyed when the server exits. The auralized program and the GUI receive responses from the server through their own sockets, aoutpid and
\texttt{xlisten}, respectively. The process which starts the server creates the temporary directory, the \texttt{sndserver} socket, and its own socket.

The protocol used between a client and the sound server is the following. Client writes null-terminated character strings to the socket and waits for a response. The string sent to the server is of the form

\begin{center}
\textit{command [arguments]}
\end{center}

A typical command might look like

\begin{center}
\texttt{pp3 spec0 70,}
\end{center}

which plays the sound for the \texttt{dtrack spec0} with pitch 70.

When the server receives a request from a client, the command string is passed to a dispatcher, which determines the command-type, extracts the arguments from the string and calls the appropriate function.

The response returned to the client is a string containing the status of the command (‘1’ if successful, ‘0’ if not) and some auxiliary data. For example, if the client requests the definition of a \texttt{dtrack spec0}, the server might return

\begin{center}
\texttt{13 1 spec0 flute2_snd 2 advance current}
\end{center}

which translates to: status 1 (success), type 3 (\texttt{dtrack}), spec status 1 (on), name \texttt{spec0}, sound \texttt{flute2_snd}, and 2 classes: \texttt{advance} and \texttt{current}.

3.4.3 Implementation

The sound server communicates with a client program according to a protocol described in Appendix B. The protocol includes commands to turn on or off output to a particular channel, turn on or off a specification or a class of specifications. It also includes commands to obtain information about specifications or classes of specifications, load or unload sound databases, and suspend or resume audio output.

\texttt{xlisten} and user executable file share the code for interacting with the sound server. This code exists in the client library, \texttt{libclient.a}. 
3.4.4 Limitations and Future Enhancements

The sound server communicates with only one client at a time. With each invocation of \texttt{xlisten} a copy of the sound server is started. Thus the sound server does not behave as a typical server. The protocol for communication between client and server is general enough for use in communicating with multiple clients simultaneously. In future versions, this extension may be done. The server will talk TCP/IP so that it can run on the machine connected with the MIDI device and clients can connect to it from across a network. Users can use wireless speakers to listen to this output even when they are not logged on at the console.

Using pause and resume causes the remaining audio output to deviate from its pre-specified tempo. This happens because interrupt sequence handler does not update the time-stamps in the MIDI queue correctly.

Edit/Add Sound dialog's Change Instrument button is stubbed out as there is currently no support for specifying instruments. This can be done by providing a file with generic instrument names which are available from any MIDI synthesizer. This file will also contain the mappings from a generic name to a specific name for each synthesizer that is supported. Changing the pan in the Edit/Add Sound dialog is not implemented. Saving LSL files and the sound database is not implemented.

3.5 Summary

Section 3.1 described the Listen preprocessor environment setup required to use v2.0, the sound database structure, invocation syntax and options, and VDAP implementation.

Section 3.2 described the enhancements to Listen sound library. In particular, support for a run-time audio controller, a primitive speech generation facility for use with \texttt{dtrack} command, sequence interrupt handler, and synchronization of audio with visual output were detailed.
Section 3.3 illustrates the functionality provided by the graphical user-interface through several figures. Implementation is described in terms of the data structures and algorithms used.

Section 3.4 described the run-time controller system, its architecture, and implementation details. The run-time system is comprised of three major components. A sound server maintains a map between each LSL specification and the sound associated with the specification. The server also interacts with the MIDI device to play sounds when requested to do so. Currently, the sound server can handle specifications from only one LSL file. A client embedded in the auralized program sends commands to the sound server to play the sound associated with a given specification. A graphical user-interface \texttt{xlisten} provides the means for controlling the auralized program. Through it, the sound server can be sent commands to turn specifications or classes of specifications on/off, to re-map sound attributes, to play sounds for testing purposes, and to create new sounds to add to the sound database.
Figure 3.26: File Menu available from the main screen. Options in this menu deal with the executable file, LSL file, and sound database. Names of the executable file, and LSL file may alternatively be specified on the command-line.
Figure 3.27: Specification commands listed according to their names. Each item in the list represents one specification command. There are five specifications in this figure, namely for_loop, go, stop, tmp, and while_loop. The sound name is displayed in the second column. The button with a + sign on it indicates that this specification is turned on. A specification can be turned off by clicking on its button.
Figure 3.28: Specification commands listed according to the classes they belong to. Each class may contain one or more specifications. Similarly, a specification may belong to one or more classes. In this figure, three classes are shown, namely, data, endpoint, and loop.
Figure 3.29: Change Sound screen. This screen is used to change a sound corresponding to a specification or add new ones.

Figure 3.30: Edit Existing Sound screen. This screen is used to edit a sound. This figure shows the parameter settings for a predefined sound flute2_snd. New sounds so defined can be saved to the sound database by selecting the option to do the same from File menu. User is prompted in case the sound database has changed during the current session.
4. EFFECTIVENESS OF AURAL, VISUAL, AND AURAL-VISUAL CUES: AN EXPERIMENT

Researchers have been involved in issues such as program and data auralization and their use in software testing and debugging [Jam92, DBO93]. In both these cases, special-purpose tools have been built which incorporate audio output during the testing and debugging process. In many instances, audio output has been felt to be useful. However, most of these experiences are anecdotal. There have been few scientific studies to obtain quantitative measures of the usefulness of audio. The purpose of this study is to investigate the effectiveness of audio output with that of the more common visual (text/graphics) output in tasks related to software development process.

In this chapter we discuss an experiment to compare the effectiveness of three different modes of output, namely aural, visual, and aural-visual to differentiate between “correct” and “incorrect” programs.

One question we wanted to answer is: Does aural output (with or without visual output) make it easier for users to understand the output of a program? In a study with a similar objective, Portigal [Por94] used abstract sounds as aural cues to display the structure of a hypertext document. He also used visual cues and a combination of aural and visual cues to convey the structure of the same document. His research showed that aural cues did not provide significant help to users to understand the document structure. These results give rise to another interesting question: Does the usefulness of aural output depend on the nature of the application?

Chapter 3 described Listen as a tool for program auralization. This experiment serves as an example of an application of Listen and demonstrates how this tool can ease the task of creating effective auralizations.
The remainder of this chapter is organized as follows. The experimental setup is described in Section 4.1. Evolution of the aural, visual, and aural-visual cues, and a pilot study are described in Section 4.2. Our results and statistical analysis are presented in Section 4.3. Finally in Section 4.4 we discuss our results, point to the limitations of this study, and suggest future work.

4.1 Method

A within-subject randomized complete block design [Mon91, CDS86] was used. Subjects were divided into three groups and each subject was given each of the three treatments in sequence. Group determined the order in which subjects received the treatment.

4.1.1 Subjects

Eighteen subjects, three groups of six each, from Purdue University Department of Computer Sciences participated in the experiment. Each subject had at least three years of programming experience. Each subject was also familiar with UNIX utilities used in the experiment (used at least two of the three utilities). Eight of them had formal training in music performance and/or theory. These eight could play an instrument and read music. Subjects were divided in three groups, namely, $G_1$, $G_2$, and $G_3$. They were randomly assigned to one of the groups to get an even mix of people with and without background training in music.

4.1.2 Input Variables

There is one independent variable: cue-type. Cue-type can be “visual”, “aural” or “aural-visual”. If cue type is “visual” it means that the program in question produces only textual output; if cue type is “aural” the program in question produces

---

1We have adopted the international standard format for reporting scientific experiments. This format is popularly known as IMRAD (Introduction, Method, Results, Analysis, and Discussion) in the scientific community.
only aural output and if \textbf{cue type} is “aural-visual” the program in question produces both textual and aural output.

4.1.3 Response Variables

There are two response variables: \textit{correctness} (C) and \textit{time-taken} (T). \textit{Correctness} represents the percentage of correct answers given by a subject, and \textit{time-taken} is the time he/she took in answering the questions. To denote C and T for each of the three cases, we use subscript \textit{v} for visual, \textit{a} for aural, and \textit{av} for aural-visual cues. Below we formally define the variables.

\begin{align*}
C_v &= \frac{\text{# of correct answers using visual cues}}{\text{total # of questions}} \\
C_a &= \frac{\text{# of correct answers using aural cues}}{\text{total # of questions}} \\
C_{av} &= \frac{\text{# of correct answers using aural-visual cues}}{\text{total # of questions}} \\
T_v &= \text{time taken (in minutes) using visual cues} \\
T_a &= \text{time taken (in minutes) using aural cues} \\
T_{av} &= \text{time taken (in minutes) using aural-visual cues}
\end{align*}

For each subject the response variable T is recorded and C is computed for each of the three cue types.

4.1.4 Hypotheses

The Null Hypothesis ($H_0$):

Cue type has no effect on response variables correctness and time-taken.

The Alternative Hypothesis ($H_1$):

Cue Type affects response variables correctness and time-taken.

4.1.5 Materials

A Sun workstation running SunOS 4.3 was used to execute all the programs. For sound generation, a \textit{Proteus/3 WORLD} MIDI synthesizer module was connected to the workstation through the serial port /dev/ttya of the workstation. A computer
serial port to MIDI port interface (Keytronics MIDIATOR) was used. To listen to audio output subjects used headphones which received the audio signal transmitted via a wireless speaker system [Rec]. The speakers created an intermittent noise due to static when the battery used in the headphones was not fully charged. The battery was kept fully charged to avoid the problem.

Each subject examined the three types of cues by executing different versions of a UNIX utility via a shell script. A sample fragment from one such script is shown in Figure 4.1.

Selection of Programs

Three UNIX utilities (cal, look, and sort) were used. All these programs produce some sort of textual output when executed. Our intention was to select programs that produced textual output because this output would serve as the primary input to each subject. Also we required that programs be written in C so they could be auralized using the current implementation of Listen. Subjects in G1 were given cal, those in G2 were given look and those in G3 were given sort.

Several incorrect versions of each of the three programs were created. Each such incorrect program is called a mutant [D+87, GJM91]. Mutants were chosen from Wong’s doctoral thesis [Won93]. Some mutants were generated using PROTEUM [DJC93], a tool for mutation testing of C programs. Appendix C lists all the mutants used.

4.1.6 Visual to Aural Output Mappings

Below we describe how the visual output was mapped to aural output.

In cal calls to functions pstr(), jani() were mapped to a note on flute and a different note on guitar. Variable i was tracked using piano sound. The LSL file to generate the above mappings is shown in Figure 4.2. The above mappings were chosen in order to create an aural signature for the correct cal program. It was conjectured that slight variations in the program would cause the aural signature to
cd /homes/vk/brhma/research/mutation/expt/executables/aural/Sort

while true
do

clear

  echo 'Correct Program'
  echo '------------------'
  echo; echo; echo; echo; echo; echo; echo; echo;
  echo 'UNIX sort utility'
  echo "In this section you will only hear the aural output"
  echo; echo; echo; echo; echo; echo; echo; echo;
  echo 'Press <return> to listen to the correct program...'

  read dummy

clear

Sort /homes/vk/brhma/research/mutation/expt/sources/Sort/inputs/in2

  2> /dev/null 1> /dev/null

echo; echo; echo; echo; echo; echo; echo; echo;

  echo -n "Again (y/n) ?"
  read choice

case $choice in
  y) ;;
  *) break ;;
esac
done

Figure 4.1: A fragment of the shell script run.sort used by subjects in group G3. The fragment shown here corresponds to the aural cues stage of the experiment. It executes the program Sort on a predetermined input and redirects stdout and stderr to /dev/null. This redirection disallows any visual output to appear on the screen.
change when both the correct and the incorrect program were executed on the same test case.

In look all the occurrences of while.body.begin were mapped to a note on the flute, i.e. each time control reached the first statement inside a while loop a particular note would be generated. Variable cnt was tracked using the dtrack command provided in LSL. The dtrack used a Value Dependent Aural Pattern (VDAP) to map the value of cnt to a sound. The LSL file to generate the above mappings is shown in Figure 4.3.

In sort the variable cp was tracked using dtrack only inside the function sort() using a VDAP. The LSL file to generate the above mappings is shown in Figure 4.4.

4.1.7 Procedure

Upon arrival each subject was escorted into the experiment room and was seated in front of the workstation. They were given the primary instructions. The primary instruction document is reproduced in Appendix C. The document described in brief, the purpose of the experiment, what each subject was expected to do, and what measurements would be taken. Each subject was provided with tables to record their classification of the mutants. Any questions a subject had regarding the instructions were answered at this point. Next they proceeded to the training phase.

4.1.8 Training

Each subject executed a sample script similar to the one they used in the main experiment. This was done in order to minimize any errors that might creep in due to the procedure. Most subjects were less familiar with aural mode of output as compared with the visual mode. In most cases, subjects took longer to go through the aural section of the training than the visual. Their questions about the experiment were answered before proceeding to the main experiment.
begin auralspec
specmodule temp
begin temp
/**
 *** Tempo
 **/
syncto mm q=360;
/**
 *** Corresponds to the dates printed
 **/
dtrack i
    when rule = function_entry:"cal"
    until rule = function_return:"cal"
    using piano_snd;

    notify rule = function_entry:"pstr"
    using flute_snd;

    notify rule = function_return:"jan1"
    using jump0;
end temp;
end auralspec.

Figure 4.2: LSL specification file for auralizing UNIX cal program. Command
syncto synchronizes the audio output to a metronome clicking 360 beats per
minute. Command dtrack i tracks the value of variable i. The tracking starts
after the function cal is entered and tracking ends just before control returns
from it. Command notify rule = function_entry:"pstr" and notify rule =
function_return:"jan1" notify the two said events by generating a simple note in
either case using flute_snd and jump0 respectively.
begin auralspec
VDAP begin
  track[cnt] (int cnt)
  {
    if (cnt > 4) {    /*SL:*/
      begin
        play iep_snd;
      end
      */
    }
    else {          /*SL: begin end */
      ...
    }
  }
VDAP end;
specmodule temp
begin temp
/**
***  Tempo
**/
    syncio mm q<360;

    notify rule = while_body_begin
      using flutel_snd;
    end temp;
end auralspec.

Figure 4.3: LSL specification file for auralizing UNIX look program. Command syncio synchronizes the audio output to a metronome clicking 360 beats per minute. Command dtrack cnt tracks the value of variable cnt. The tracking starts after the function canon is entered and ends just before control returns from it. Tracking is defined by a Value Dependent Aural Pattern (VDAP) specified by function track_cnt.
begin auralspec
VDAP begin
out_str_track(char ch)
{
    int value = ch - 'a';
    switch(value) {
    case 0: /*LSL: begin
            play flute1_snd;
            end */
            break;
    case 2: /*LSL: begin
            play flute2_snd;
            end */
            break;
    default: /*LSL: begin
            play iter2;
            end */
            break;
    }
}
VDAP end;
specmodule temp
begin temp
    syncto mm q=240;
    notify rule = function_entry:'main' using fna_snd;
    /* Map certain characters in the
       string to different sounds */
    dtrack cp
        when rule = function_entry:'sort'
        until rule = function_return:'sort'
        using out_str_track(\*cp);
    end temp;
end auralspec.

Figure 4.4: LSL specification file for auralizing UNIX sort program. Command syncto synchronizes the audio output to a metronome clicking 240 beats per minute. Command dtrack cp tracks the value of variable cp. The tracking starts after the function sort is entered and tracking ends just before control returns from it. Tracking is defined by a Value Dependent Aural Pattern (VDAP) specified by function out_str_track.
4.1.9 Measurements

The data in Table C.4 in Appendix C shows the mean values of correctness broken down by cue type (Visual, Aural, Aural-Visual). The data in Table C.5 in Appendix C shows the mean values of time-taken broken down by cue type (Visual, Aural, Aural-Visual).

4.2 Evolution of the Main Experiment

This section highlights some of the points in the evolution of the three different types of cues, and other materials.

4.2.1 Aural Output

As explained in the previous section, some aspect of the visual output was mapped to one or more notes creating different mappings by varying parameters such as pitch, and timbre. It seems intuitive and has been demonstrated in past research [Bre93] that the amount of information that can be presented using sound depends to a large extent on the “goodness” of the sounds employed. We are not aware of any quantitative methods to measure “goodness” of aural output. However, there are guidelines for designing good sounds [Bre93]. These guidelines address such issues as similarity and dissimilarity of attributes related to the sounds, proximity, good continuation, coherence etc. Several mappings were tried before choosing the final one. The feedback from the pilot study conducted on two subjects was used to modify the mappings. It is in this area that Listen was helpful is experimenting with various data-to-sound mappings.

4.2.2 Visual Output

During the initial stages of the experiment, it was decided to map run-time information such as control flow to sound. In order to have a balanced design, one would like to present the same information (control flow) via the three different modes of
output. Programs would then need to generate the control flow information which would require modifying the source programs. However, in order to simplify the design, it was decided to use the three different modes to display only the output and not the control flow information. This required no modification of the source programs.

4.2.3 Aural-Visual Output

This consisted of both the aural and visual output. It was left to the subjects to use the output as they wanted. No instructions were given as to whether they should concentrate on both the outputs or otherwise. It is to be noted that the audio and visual outputs were not synchronized in that all the visual output was produced much faster than the aural output. Aural output was generated at a slower pace so that a user could listen to it. ²

4.2.4 Pilot Study

The complete experiment was piloted on two subjects and this produced some modifications to the experiment. For example, some of the messages being displayed by the shell scripts were non-intuitive and they were improved. In the case of sort the tempo of the auralization was reduced because subjects in the pilot study indicated that they wanted to get an approximate count of the number of notes played in the three distinct phases of execution. Initially replay-tactic was thought of as an independent variable. According to this variable, subjects could be classified into two groups namely, replay and no-replay. Then the effect if any, of the replay tactic on the response variables could be studied. It was not clear whether there will be some subjects who will not need replay. However as both the subjects in the pilot study did replay sounds several times, it appeared that the classification based on replay tactic would be skewed in favor of the replay group. Replay tactic was therefore not included as a variable in the final experiment.

²In Listen v2.0 one can synchronize the two kinds of output by supplying the -syncaudio flag to the preprocessor. This blocks the audio output and returns control to the calling function only after audio output is over.
Table 4.1: Table of means showing effect of cue type on correctness

<table>
<thead>
<tr>
<th>cue type</th>
<th>N</th>
<th>Mean</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>aural</td>
<td>18</td>
<td>0.622</td>
<td>0.152</td>
</tr>
<tr>
<td>visual</td>
<td>18</td>
<td>0.867</td>
<td>0.137</td>
</tr>
<tr>
<td>aural-visual</td>
<td>18</td>
<td>0.978</td>
<td>0.065</td>
</tr>
</tbody>
</table>

4.2.5 Design Decisions

Since the aural-visual output was composed of both the aural and visual output the experiment design was a slightly compromised form of a completely balanced design. Subjects in groups $G_2$ and $G_3$ could have a transfer effect due to the aural-visual output affecting their response to the other cue types. The effect of this phenomenon was minimized by selecting different mutants for each cue type. Output produced by different mutants was different which reduced the transfer effect.

4.3 Results and Analysis

Based on the measurements as described in the measurement section and the value of correctness was calculated, time-taken was recorded. An ANOVA for a randomized complete block design [Mon91, Mot90] was performed to investigate the effect of cue type on correctness and time-taken.

The ANOVA revealed that cue type had a significant effect on correctness ($F_{2,4} = 19.87$, $p < 0.05$). To compare the respective effects of each of the three cue types, Paired T-Tests [Mon91] were performed. Comparison-wise T statistic values were computed using $\alpha = 0.0167$. They revealed a significant difference between aural and visual ($T = -4.89$, $p < 0.0167$), between aural and aural-visual ($T = -8.00$, $p < 0.0167$), and between visual and aural-visual ($T = -3.01$, $p < 0.0167$). Table 4.1 shows the mean and standard deviation values of correctness for each of the three cue types. Aural cues resulted in lowest scores, visual cues resulted in higher scores.
Table 4.2: Table of means showing effect of cue type on time-taken

<table>
<thead>
<tr>
<th>cue type</th>
<th>N</th>
<th>Mean</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>aural</td>
<td>18</td>
<td>4.06</td>
<td>1.21</td>
</tr>
<tr>
<td>visual</td>
<td>18</td>
<td>3.28</td>
<td>0.75</td>
</tr>
<tr>
<td>aural-visual</td>
<td>18</td>
<td>2.78</td>
<td>0.65</td>
</tr>
</tbody>
</table>

than aural cues, and aural-visual cues resulted in the highest scores. Effect of group was found non-significant ($F_{2,15} = 0.13$, ns). Effect of subject within group against the error term was also found non-significant ($F_{15,30} = 0.81$, ns). Group and cue type interaction effect was non-significant ($F_{4,30} = 2.08$, ns).

Figure 4.5 plots the mean values of correctness for each group and cue type. Appendix C contains the statistics and plots used to validate the normality assumption on the data.

cue type also had a significant effect on time-taken ($F_{2,4} = 21.21$, $p < 0.05$). To compare the respective effects of each of the three cue types, Paired T-Tests were performed. They revealed a significant difference between aural and visual cues ($T = 4.08$, $p < 0.0167$), between aural and aural-visual cues ($T = 6.06$, $p < 0.0167$), and between visual and aural-visual cues ($T = 3.43$, $p < 0.0167$). Table 4.2 shows the mean and standard deviation values of time-taken for each of the three cue types. Subjects took the longest when using aural cues, took less time with visual cues than aural cues, and took the least amount of time using aural-visual cues.

Effect of group was found non-significant ($F_{2,15} = 0.49$, ns). Effect of subject within group against the error term was found significant ($F_{15,30} = 6.52$, $p < 0.05$). Group and cue type interaction effect was non-significant ($F_{4,30} = 1.17$, ns).

Figure 4.6 plots the mean values of correctness for each group and cue type. Appendix C contains the statistics and plots used to validate the normality assumption on the data.
The SAS [SASb, SASa] program used for statistical analysis of the data appears in Appendix C. The output of the SAS program is also listed in this Appendix.

4.4 Discussion

The results indicate that cue type had a significant effect on both correctness and time taken. This implies that aural cues alone were not sufficient to differentiate between correct and incorrect outputs. However, with a combination of the aural and visual cues, subjects were able to detect the differences in output more accurately and faster. This result is not completely unexpected. Similar results have been reported in the past [Por94]. In particular, Portigal [Por94] reported similar results when different cue types were used to convey the structure of a hypertext document. Our results agree with those presented in that work except that we found a statistically significant difference between the effect of visual and aural-visual cues.

4.4.1 Limitations of Our Study

In our experiment, the quality of mappings to sound plays an important role in the effectiveness of aural cues. A particular mapping to sound may be unable to detect and thus convey the differences in output for a given test run. Or, it may produce an aural output which is only slightly different from the correct output. In both these situations it may not be possible for the subject to correctly classify the output. The quality of mappings to sound is another variable that may affect Correctness. We are not aware of any methods to quantify this “quality”.

In the particular task that was employed in this experiment, visual cues had an advantage over aural cues because the visual cues were more detailed than their aural counterparts. If we attempt to convey all the information provided in visual cues, the amount of audio cues generated will be very large. In order to play all of the audio cues one would either need to increase the speed at which notes are played or simply maintain the normal speed but take much longer to output. We decided to provide only a fraction of information to maintain the same speed at which to play notes,
so as to convey even minor differences in output. In a future study, more complex patterns of sounds such as earcons may be used to increase the amount of information in the aural cues.

The number of “incorrect” programs examined was kept small so as to limit the total time per subject. This caused the response time data to get skewed in that many subjects were clustered around the same values for time-taken.

4.5 Summary

An experiment was conducted to compare the effectiveness of aural, visual, and aural-visual cues in conveying the differences in output of a program. A randomized complete block design was used. ANOVA revealed that cue type had a significant effect on both correctness and time-taken. Subjects performed best with aural-visual cues and worst with aural cues. Aural cues alone were not able to convey the differences in output in all cases. Same was the case with visual cues. A combination of both, however, revealed minute differences in some cases. The results should encourage the use of aural cues in applications where minute changes in program output need to be monitored.
Figure 4.5: Mean values of correctness plotted by group and cue type.
Figure 4.6: Mean values of time taken plotted by group and cue type.
5. CONCLUSIONS

5.1 Conclusions

The work reported here is divided into two distinct parts. In the first part, we describe the architecture and implementation of Listen v2.0. We believe that in its current form this tool will be usable by software developers and testers to add audio output to their programs for various purposes. In the second part, we describe an experiment to compare the effectiveness of aural, visual, and aural-visual cues to convey differences in program output. Three UNIX utilities cal, look, and sort were used as correct programs. Each program’s visual output was mapped to sound. Data to sound mappings were designed using the data tracking facility available in Listen v2.0. It was found that subjects could distinguish correct program outputs from incorrect ones in the case of aural-visual cues more often than in the remaining two cases. Statistical analysis showed the aural-visual cues to be significantly better than visual cues for this particular task.

It has been demonstrated by several researchers that audio can be used to convey the output generated by a program. However, there have been few studies to obtain quantitative measures of the relative effectiveness of aural, visual, and aural-visual cues in various programming-related tasks. This experiment provides quantitative results to suggest that aural cues could be a useful supplement to visual cues to convey the output of a program.

5.2 Future Work

Using Listen v2.0 one can create a variety of auralizations for C programs. Commands to notify an event, track an activity, track data, define classes of auralizations,
and synchronize the audio output in different ways combine to give a powerful tool to map program events and states to sound. The run-time audio controller allows the user to control the output in several useful ways. However, the implementation of LSL in v2.0 is not complete. Some of the enhancements that we recommend for future versions are listed below.

- To support other methods of audio production. This includes using the built-in audio capabilities of the workstation. It would involve providing a database of sounds for each method of sound-production. This database would contain a rich set of default sounds as well as facility to add new sounds making the tool more accessible to those users who do not have access to a MIDI synthesizer.

- To produce audio output in MIDI file format. This would enable the storing of audio output for playing at a later time without having to re-execute the program. Also it would allow for easy exporting of audio output to other tools.

- To provide more features in the text editor of the GUI to aid in producing quick auralizations. Examples of such features are notes, bells, and whistles similar to the already existing trip note feature.

- To improve the run-time audio controller. To provide support for manipulating instruments. This will involve providing an extra layer of abstraction in sound database. This layer will contain a generic set of instruments which will be mapped to specific instruments for different MIDI devices. To provide support for communication between a single sound server and multiple clients over a network.

We would like to conduct some variations of the study reported here. It should be useful to conduct a similar experiment using subjects who are visually-handicapped to compare the results with sighted users. Another possibility is to use aural cues which vary in parameters such as pan and spatial location of sound.

The size of the programs employed is another factor in the results obtained from an experiment of this kind. The programs used in this experiment were small, ranging
from 170 LOC (Look) to 913 LOC (Sort). Do similar results hold when the same experiment is conducted on large size programs? Since the motivation for using sound is to supplement the information provided through the visual channel, we expect aural cues to be useful for large programs as well.
BIBLIOGRAPHY


APPENDICES
Appendix A: LSL Grammar

The grammar below is reproduced from [BM93].

A.1 LSL Syntax Conventions

The syntax of LSL is described below using a modified form of BNF[ASU86]. Nonterminals are in *italics*, keywords in *teletype* font, and lexical symbols in *bold* font. Alternates of a nonterminal are separated by the | symbol.

1. \textit{ls}l-\textit{spec} \rightarrow \textit{begin} aural\textit{spec}
   \textit{spec-module-list}
   \textit{end aural\textit{spec}}.

2. \textit{spec-module-list} \rightarrow \textit{spec-module-list spec-module}

3. \textit{spec-module-list} \rightarrow \textit{spec-module}

5. \textit{spec-module} \rightarrow \textit{specmodule id}
   \textit{program-id-list}
   \textit{global-interaction-list}
   \textit{declarations}
   \textit{spec-def-list}
   \textit{VDAP-list}
   \textit{begin id}
   \textit{spec-def-body}
   \textit{end id};

6. \textit{program-id-list} \rightarrow \textit{external ext-id-list;}
   \textit{\epsilon}

8. \textit{global-interaction-list} \rightarrow \textit{global-interactions global-interaction-list}
   \textit{\epsilon}

10. \textit{global-interactions} \rightarrow \textit{interact-id id-list;}

11. \textit{interact-id} \rightarrow \text{import | export}

13. \textit{spec-def-list} \rightarrow \text{spec-def spec-def-list}
\mid \epsilon

15. \textit{VDAP-list} \rightarrow \text{VDAP-spec VDAP-list}
\mid \epsilon

17. \textit{VDAP-spec} \rightarrow \text{VDAP begin}
\text{l-function}
\text{VDAP end;}

18. \textit{spec-def} \rightarrow \text{specdef id \texttt{(spec-par-list)}}
\text{declarations}
\text{begin id}
\text{spec-def-body}
\text{end id;}

19. \textit{spec-def-body} \rightarrow \text{spec-command spec-def-body}
\mid \text{spec-command}

21. \textit{spec-command} \rightarrow \text{named-command}
\mid \text{unnamed-command}

23. \textit{named-command} \rightarrow \text{name-tag-list unnamed-command}

24. \textit{name-tag-list} \rightarrow \text{id :: name-tag-list}
\mid \text{id ::}

26. \textit{unnamed-command} \rightarrow \text{set-globals-command}
\mid \text{play-command}
\mid \text{notify-command}
\mid \text{dtrack-command}
\mid \text{atrack-command}
\mid \text{assign-command}
| loop-command |
| if-command |
| specdef-use-command |
| VDAP-call-command |
| turn-command |
| toggle-command |
| sync-command |

39. \( \text{set-globals-command} \rightarrow \text{set global-par-list;} \)
40. \( \text{global-par-list} \rightarrow \text{global-par-list, global-par} \)
| global-par |
42. \( \text{global-par} \rightarrow \text{score-const-id} \)
| device-const-id |
44. \( \text{play-command} \rightarrow \text{play play-list;} \)
45. \( \text{play-list} \rightarrow \text{pattern-specifier || play-list} \)
| pattern-specifier & play-list |
| play-list |
48. \( \text{pattern-specifier} \rightarrow \text{id} \)
| constant |
| specdef-use-command |
| VDAP-call-command |
| pattern-specifier play-pars |
| ( play-list ) |
54. \( \text{play-pars} \rightarrow \text{with tagged-list} \)
55. \( \text{tagged-list} \rightarrow \text{tagged-list, tags} \)
| tags |
57. \( \text{tags} \rightarrow \text{score-const-id} \)
59.  score-const-id  →  score-tag = const-id
|  mm mmspec
|  mode-specifier

62.  const-id  →  constant
|  dotted-id

64.  dotted-id  →  .id

65.  score-tag  →  keysig
|  timesig

67.  device-tag-list  →  device-const-id , device-tag-list
|  device-const-id

69.  device-const-id  →  device-tag = const-id

70.  device-tag  →  chan | inst

72.  notify-command  →  notify all-selective label-parameter event-specifier
|  sound-specifier scope-specifier;

73.  all-selective  →  all | selective | ε

76.  label-parameter  →  label = label-list
|  ε

78.  label-list  →  label-list, id
|  id

80.  event-specifier  →  event-specifier connector event
|  event

82.  connector  →  and | or

84.  event  →  rule = id
\begin{verbatim}
| rule = id:instance-list \\
| instance = instance-list \\
| assertion = l-condition \\
| rtime = expression after event \\
| (event-specifier) \\
| event (first) \\

91. instance-list → instance-list & & instance \\
| instance \\

93. instance → string \\

94. sound-specifier → using play-list \\
| e \\

96. scope-specifier → in tagged-scope-list \\
| e \\

98. tagged-scope-list → tagged-scope-list and tagged-scope \\
| tagged-scope \\

100. tagged-scope → scope-tag = scope-tagid-list \\

101. scope-tag → filename | func \\

103. scope-tagid-list → scope-tagid-list, scope-tagid \\
| scope-tagid \\

105. scope-tagid → selector | string \\

107. dtrack-command → dtrack dtrack-id-list start-event-spec term-event-spec sound-specifier scope-specifier; \\

108. atrack-command → atrack start-event-spec term-event-spec sound-specifier scope-specifier; \\

109. start-event-spec → when event-specifier scope-specifier \\
\end{verbatim}
111. \( \text{term-event-spec} \rightarrow \text{until event-specifier} \)
\( \hspace{1em} \text{scope-specifier} \)
\( \mid \epsilon \)

113. \( \text{ext-id-list} \rightarrow \text{ext-id-list}, l\text{-id} \)
\( \hspace{1em} l\text{-id} \)

115. \( \text{dtrack-id-list} \rightarrow \text{dtrack-id-list and dtrack-id} \)
\( \hspace{1em} \text{dtrack-id} \)

117. \( \text{dtrack-id} \rightarrow l\text{-id init-value capture-specifier scope-specifier} \)

118. \( \text{init-value} \rightarrow \text{init} = l\text{-expression} \)
\( \hspace{1em} \epsilon \)

120. \( \text{capture-specifier} \rightarrow \text{capture} = \text{id} \)
\( \hspace{1em} \epsilon \)

122. \( \text{mode-specifier} \rightarrow \text{mode} = \text{continuous} \mid \text{mode} = \text{discrete} \mid \text{mode} = \text{sustain} \)

125. \( \text{assign-command} \rightarrow \text{selector} := \text{expression;} \)

126. \( \text{selector} \rightarrow \text{id} \mid \text{id[element-selector]} \)

128. \( \text{element-selector} \rightarrow \text{expression-list} \)

129. \( \text{expression-list} \rightarrow \text{expression-list}, \text{expression} \)
\( \hspace{1em} \text{expression} \)

131. \( \text{loop-command} \rightarrow \text{for-loop} \mid \text{while-loop} \)

133. \( \text{for-loop} \rightarrow \text{for id} := \text{expression} \text{ to expression} \text{ step-expression} \)
\( \hspace{1em} \text{statement-body} \)

134. \( \text{step-expression} \rightarrow \text{step expression} \)
\( \hspace{1em} \epsilon \)
136. while-loop → while condition do statement-body

137. statement-body → begin spec-def-body end
   | spec-command;

139. if-command → if-then-command
   | if-then-else-command

141. if-then-command → if condition then statement-body

142. if-then-else-command → if condition then statement-body else statement-body

143. specdef-use-command → id (actual-par-list);
   | id ( );

145. actual-par-list → actual-par-list, actual-par
   | actual-par

147. actual-par → expression

148. spec-par-list → id-list | ε

150. VDAP-call-command → l-id (l-actual-parameter-list);

151. turn-command → turn on-off device-tag-list;

152. on-off → on
   | off

154. toggle-command → toggle toggle-source = constant;

155. toggle-source → keyboard
   | midi

157. sync-command → syncto sync-to-id;

158. sync-to-id → program
   | sync-par-list
160. sync-par-list &rarr; sync-par-list, sync-par
    | sync-par

162. sync-par &rarr; bufsize = const
    | noslow
    | mmkeyword
    | mmkeyword mmspec

166. mmkeyword &rarr; mm | mmabs | mmrel

169. mmspec &rarr; duration-expression = const

170. duration-expression &rarr; duration-expression duration-factor
    | duration-expression + duration-factor
    | duration-factor

173. duration-factor &rarr; duration-attribute
    | (duration-expression)

175. duration-attribute &rarr; f | h | q | e | s

180. declarations &rarr; applicability const-declaration var-declaration

181. applicability &rarr; apply-list
    | e

183. apply-list &rarr; apply-list; apply-decl
    | apply-decl

185. apply-decl &rarr; applyto tagged-scope-list;

186. const-declaration &rarr; const const-list;
    | e

188. const-list &rarr; const-val-pair const-list
    | const-val-pair
190. const-val-pair → id = constant ;

191. var-declaration → var var-decl-list;

      | ε

193. var-decl-list → var-type-list ; var-decl-list

      | var-type-list

195. var-type-list → id-list : type

196. id-list → id , id-list

      | id

198. type → int | note | pattern | voice | file | ksig | tsig

      | array-declarator

206. array-declarator → array [ range-list ] of type

207. range-list → range-list , range

      | range

209. range → expression .. expression

210. expression → expression addop term

      | term

212. term → term mulop factor

      | factor

214. factor → ( expression )

      | unop factor

      | id

      | id(actual-par-list)

      | id( )

      | const

220. condition → condition relop cterm
222. \( ct\)erm \( \rightarrow \) \( ct\)erm \( logop\) \( cfactor\)
    | \( cfactor\)

224. \( cfactor\) \( \rightarrow \) \( expression\)
    | \( (condition)\)
    | \( not\) \( cfactor\)

227. \( addop\) \( \rightarrow \) \(+\) | \(-\)

229. \( mulop\) \( \rightarrow \) \(*\) | \(/\)

231. \( unop\) \( \rightarrow \) \(-\)

232. \( relop\) \( \rightarrow \) \(<\) | \(>\) | \(<=\) | \(=\) | \(>=\) | \(<>\)

238. \( logop\) \( \rightarrow \) \&\& | \(|||

A.2 Lexical Conventions

Using regular expressions[ASU86] we define the lexical elements of LSL.

1. Comments are enclosed inside /* and */. Comments may not appear within a token. A comment within another comment is not allowed.

2. char denotes any ASCII character.

3. One or more spaces separates tokens. Spaces may not appear within tokens.

4. An id is a sequence of letters or digits with the first character being a letter. The underscore (_2) can be used in an identifier. Upper and lower case letters are treated as being different in an id.

\[
\begin{align*}
id & \rightarrow ( _2)^* letter ( letter | digit | _2)^* \\
letter & \rightarrow [\alpha-zA-Z]
\end{align*}
\]
\[ \text{digit} \rightarrow [0-9] \]

5. A keyword may not be used as an id. Upper and lower case are treated differently.

6. A constant can be an integer or a string. An integer is a sequence of digits. A string is a sequence of characters enclosed within double quotes. As a constant can be interpreted in a variety of ways in LSL, we provide below a complete grammar for constants.

1. \textit{constant} \rightarrow \textit{integer}
   \[ \mid \text{string} \]
   \[ \mid \text{time-sig} \]

4. \textit{integer} \rightarrow \text{digit}^+

5. \textit{string} \rightarrow "\text{char-sequence}" 

6. \textit{char-sequence} \rightarrow \textit{note-sequence}
   \[ \mid \text{key-sig} \]
   \[ \mid \text{file-name} \]
   \[ \mid \text{function-name} \]

10. \textit{note-sequence} \rightarrow (\textit{note} \mid .\textit{id})^+ 
    \[ \mid (\textit{note-sequence} \ddot{\text{attribute-sequence}}) \]
    \[ \mid (\textit{note-sequence}) \]

14. \textit{note} \rightarrow \textit{note-generic} \ddot{\text{note-modifier}}

15. \textit{note-generic} \rightarrow c \mid d \mid e \mid f \mid g \mid a \mid b \mid r \mid C \mid D \mid E \mid F \mid G \mid A \mid B \mid R

31. \textit{note-modifier} \rightarrow \text{flat-sharp}^* \ddot{\text{octave}}

32. \textit{flat-sharp} \rightarrow b \mid \# 

34. \textit{octave} \rightarrow [0-8]

35. \textit{attribute-sequence} \rightarrow \text{attribute}^+

36. \textit{attribute} \rightarrow \text{duration} \ddot{\text{tagged-value-list}}^*

37. \textit{duration} \rightarrow \text{simple-duration}
| ( duration-expression )

39. `simple-duration` → f | h | q | e | s
   | ptime = integer

45. `duration-expression` → `duration-expression` op `simple-duration`
   | `simple-duration`
   | ( `duration-expression` )

48. `op` → + | e

50. `key-sig` → `pre-defined`
   | `user-defined`

52. `pre-defined` → `note-mode`

53. `mode` → `major`
   | `minor`
   | `lydian`
   | `ionian`
   | `mixolydian`
   | `dorian`
   | `aeolian`
   | `phrygian`
   | `locrian`

62. `user-defined` → ( `note-sequence` )

63. `time-sig` → ( `beat-structure` : int )

64. `beat-structure` → `beat-structure` + int
   | int

66. `filename` → char+

67. `function-name` → char+

68. `tagged-value-list` → `tagged-value-list` tagged-value

69. `tagged-value` → `play-attribute-tag` = constant

70. `play-attribute-tag` → chan | play | inst | mm | mm | mm spec
7. Interpretation of a string is context dependent. Thus, for example, when
assigned to a variable of type pattern, the string "c major C5" denotes a sequence
of notes consisting of the value of the variable c major followed by the note C5.
The same string when used in the context file = "c major C5" denotes a file
name c major C5. Notes enclosed in parentheses such as in "G3 (C4E4G4) C5"
are treated as forming a blocked chord. The string "hello" results in an invalid
assignment command when it appears on the right side of an assignment to a
variable of type pattern.

8. Ambiguity may arise while defining a note sequence such as in "cb b". To avoid
this, the notes may be separated by at least one space character such as in "cb
b".

9. The grammar above contains some terminals prefixed by l*. Such terminals
denote language specific constructs. A complete list of such terminals appears
in Table A.1. These terminal symbols may be nonterminals or terminals in the
grammar of the language L of the auralized program. The LSL preprocessor
attempts to parse over the strings corresponding to such symbols. These strings
are parsed by the compiler for L.

A.3 Static Semantics

The following constraints apply to LSL specifications. These are not indicated by
the syntax.

1. All identifiers must be declared before use. Identifiers that belong to the aural-
ized program must appear as externals.

2. Local attribute values, such as metronome values, channels, etc. which are spec-
ified explicitly as attributes, take precedence over corresponding global values.
Table A.1: Language Dependent Terminals in LSL Grammar.

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Meaning</th>
<th>Example from C</th>
</tr>
</thead>
<tbody>
<tr>
<td>l-condition</td>
<td>Conditional expression which evaluates to \textit{true} or \textit{false}.</td>
<td>(x &lt; y &amp;&amp; p &gt; q)</td>
</tr>
<tr>
<td>l-id</td>
<td>Identifier</td>
<td>\textit{drag_icon}</td>
</tr>
<tr>
<td>l-expression</td>
<td>An expression that evaluates to a value of type matching the type of the left side of the assignment in which it appears.</td>
<td>(min - val*2)</td>
</tr>
<tr>
<td>l-function</td>
<td>A function invoked for tracking one or more variables.</td>
<td>Any C function definition.</td>
</tr>
<tr>
<td>l-actual-parameter-list</td>
<td>List of actual parameters.</td>
<td>\textit{int x, int * y}</td>
</tr>
</tbody>
</table>
However, they do not alter the global values. Global values of such parameters may be set using the \texttt{set} command within an LSL specification or in the program.

3. Identifiers declared within a \texttt{specmodule} $M$ are global to $M$ and may be used by all \texttt{specdefs} declared within $M$. Identifiers declared within a \texttt{specdef} $S$ are local to $S$ and may not be used by other \texttt{specdefs} or in any other \texttt{specmodule}. Identifiers may be exported by an \texttt{specmodule} for use by any other module by explicitly mentioning it in an \texttt{export} declaration. A module may use an identifier exported by another module by explicitly importing it using the \texttt{import} declaration. All program variables used in an \texttt{specdef} or a \texttt{specmodule} body must be specified as \texttt{externals}. Program identifiers, global to a VDAP definition, need not be declared. However, all such identifiers must be declared in the context wherein VDAP will be placed and compiled by the C compiler.

4. A VDAP specification must be a valid C function when using LSL/C.

5. The formal and actual parameters must match in number and type between a specification definition and its use.

6. All matching \texttt{begins} and \texttt{ends} must match in the identifiers that follow the corresponding keyword. Thus, for example, a \texttt{begin} \texttt{gear} which matches with an \texttt{end} \texttt{change} will be flagged as a warning because \texttt{gear} and \texttt{change} do not match.

7. LSL has default values for various parameters such as metronome, channel, and instrument.

8. The \texttt{expression} in a relative timed event must evaluate to a positive integer or else a run time warning is issued. A relative timed event is ignored if it occurs after program execution terminates.

9. A file or function specified in a scope tag must exist for the program to be auralized.
10. While monitoring an activity or data, tracking will terminate upon program
termination if the start event occurs after the terminating event.

11. An expression in a range-list must evaluate to an integer and must not contain
any variable names. Subscript expressions that evaluate to a value outside the
specified range are not allowed.

12. If both the initial value and the capture location are specified for a variable to
be tracked, LSL will attempt to satisfy both requirements. Thus, the variable
will be initialized at an appropriate point during program execution. Its value
will also be captured as specified. The value captured will override any previous
value of the variable.

13. The syntax of LSL allows for the naming of any command. However, only names
of notify, dtrack, and atrack correspond to classes. Naming of other com-
mands is permitted to allow referencing of commands while editing or reading
an LSL specification.

14. Use of toggle may give rise to ambiguities at run time. For example, if the
space key on the computer keyboard has been specified as a toggle source and the
executing program requests for input data, it is not clear if the space character
should be treated as a toggle request or input to the program. The user may
avoid such ambiguities by selecting a toggle source that will not be required as
input to the program. Alternately, the user may rely on the run time window
based monitor to input toggle requests.
A.4 Grammar Rules Sorted Alphabetically

The syntax of LSL is described below using a modified form of BNF[ASU86]. Nonterminals are in *italics*, keywords in *teletype* font, and lexical symbols in *bold* font. Alternates of a nonterminal are separated by the $|$ symbol.

1. $\text{actual-par-list} \rightarrow \text{actual-par-list}, \text{actual-par}$
   $\quad | \text{actual-par}$

3. $\text{actual-par} \rightarrow \text{expression}$

4. $\text{addop} \rightarrow + | -$  

6. $\text{all-selective} \rightarrow \text{all} | \text{selective} | \epsilon$

9. $\text{applicability} \rightarrow \text{apply-list}$
   $\quad | \epsilon$

11. $\text{apply-decl} \rightarrow \text{applyto tagged-scope-list;}$

12. $\text{apply-list} \rightarrow \text{apply-list}, \text{apply-decl}$
   $\quad | \text{apply-decl}$

14. $\text{array-declarator} \rightarrow \text{array} [\text{range-list}] \text{of type}$

15. $\text{assign-command} \rightarrow \text{selector} := \text{expression};$

16. $\text{atrack-command} \rightarrow \text{atrack start-event-spec term-event-spec sound-specifier scope-specifier;}$

17. $\text{capture-specifier} \rightarrow \text{capture} = \text{id}$
   $\quad | \epsilon$

19. $\text{cfactor} \rightarrow \text{expression}$
   $\quad | (\text{condition})$
   $\quad | \text{not cfactor}$

22. $\text{condition} \rightarrow \text{condition relop eterm}$
24. \( \text{const-declaration} \rightarrow \text{const \ const-list;} \)  
   \| \text{\(e\)}

26. \( \text{const-id} \rightarrow \text{constant} \)  
   \| \text{dotted-id}

28. \( \text{const-list} \rightarrow \text{const-val-pair \ const-list} \)  
   \| \text{const-val-pair}

30. \( \text{const-val-pair} \rightarrow \text{id = constant ;} \)

31. \( \text{cterm} \rightarrow \text{cterm \ logop \ cfactor} \)  
   \| \text{cfactor}

33. \( \text{declarations} \rightarrow \text{applicability \ const-declaration \ var-declaration} \)

34. \( \text{device-const-id} \rightarrow \text{device-tag} = \text{const-id} \)

35. \( \text{device-tag-list} \rightarrow \text{device-const-id, device-tag-list} \)  
   \| \text{device-const-id}

37. \( \text{device-tag} \rightarrow \text{chan | inst} \)

39. \( \text{dotted-id} \rightarrow \text{id} \)

40. \( \text{dtrack-command} \rightarrow \text{dtrack \ dtrack-id-list \ start-event-spec \ term-event-spec} \)  
   \text{sound-specifier \ scope-specifier;}

41. \( \text{dtrack-id-list} \rightarrow \text{dtrack-id-list \ and \ dtrack-id} \)  
   \| \text{dtrack-id}

43. \( \text{dtrack-id} \rightarrow \text{l-id \ init-value \ capture-specifier \ scope-specifier} \)

44. \( \text{duration-attribute} \rightarrow \text{f | h | q | e | s} \)

49. \( \text{duration-expression} \rightarrow \text{duration-expression \ duration-factor} \)  
   \| \text{duration-expression + duration-factor} \)
52. \( \text{duration-factor} \rightarrow \text{duration-attribute} \)
    | (duration-expression)

54. \( \text{element-selector} \rightarrow \text{expression-list} \)

55. \( \text{event-specifier} \rightarrow \text{event-specifier} \text{ connector} \text{ event} \)
    | \( \text{event} \)

57. \( \text{event} \rightarrow \text{rule} = \text{id} \)
    | \( \text{rule} = \text{id} : \text{instance-list} \)
    | \( \text{instance} = \text{instance-list} \)
    | \( \text{assertion} = \text{l-condition} \)
    | \( \text{rtime} = \text{expression} \text{ after} \text{ event} \)
    | (event-specifier)
    | \( \text{event} \text{ (first)} \)

64. \( \text{expression-list} \rightarrow \text{expression-list} , \text{expression} \)
    | \( \text{expression} \)

66. \( \text{expression} \rightarrow \text{expression addop term} \)
    | \( \text{term} \)

68. \( \text{ext-id-list} \rightarrow \text{ext-id-list} , \text{l-id} \)
    | \( \text{l-id} \)

70. \( \text{factor} \rightarrow ( \text{expression} ) \)
    | \( \text{unop factor} \)
    | \( \text{id} \)
    | \( \text{id( actual-par-list) } \)
    | \( \text{id( ) } \)
    | \( \text{const} \)

76. \( \text{for-loop} \rightarrow \text{for id} := \text{expression to expression step-expression} \)
77. \( \text{global-interaction-list} \rightarrow \text{global-interactions} \text{global-interaction-list} \)
    | \( \epsilon \)

79. \( \text{global-interactions} \rightarrow \text{interact-id} \text{id-list;} \)

80. \( \text{global-par-list} \rightarrow \text{global-par-list}, \text{global-par} \)
    | \( \text{global-par} \)

82. \( \text{global-par} \rightarrow \text{score-const-id} \)
    | \( \text{device-const-id} \)

84. \( \text{id-list} \rightarrow \text{id}, \text{id-list} \)
    | \( \text{id} \)

86. \( \text{if-command} \rightarrow \text{if-then-command} \)
    | \( \text{if-then-else-command} \)

88. \( \text{if-then-command} \rightarrow \text{if condition then statement-body} \)

89. \( \text{if-then-else-command} \rightarrow \text{if condition then statement-body else statement-body} \)

90. \( \text{init-value} \rightarrow \text{init = l-expression} \)
    | \( \epsilon \)

92. \( \text{instance-list} \rightarrow \text{instance-list} \&\& \text{instance} \)
    | \( \text{instance} \)

94. \( \text{instance} \rightarrow \text{string} \)

95. \( \text{interact-id} \rightarrow \text{import} | \text{export} \)

97. \( \text{label-list} \rightarrow \text{label-list}, \text{id} \)
    | \( \text{id} \)

99. \( \text{label-parameter} \rightarrow \text{label = label-list} \)
    | \( \epsilon \)
101. \textit{logop} \quad \rightarrow \quad \&\& \mid ||

103. \textit{loop-command} \quad \rightarrow \quad \textit{for-loop} \mid \textit{while-loop}

105. \textit{isl-spec} \quad \rightarrow \quad \textit{begin} \texttt{auralspec} \textit{spec-module-list} \textit{end} \texttt{auralspec}.

106. \textit{mmkeyword} \quad \rightarrow \quad \texttt{mm} \mid \texttt{mmabs} \mid \texttt{mmrel}

109. \textit{mmspec} \quad \rightarrow \quad \textit{duration-expression} = \texttt{const}

110. \textit{mode-specifier} \quad \rightarrow \quad \texttt{mode} = \texttt{continuous} \mid \texttt{mode} = \texttt{discrete} \mid \texttt{mode} = \texttt{sustain}

113. \textit{mulop} \quad \rightarrow \quad \ast \mid /

115. \textit{name-tag-list} \quad \rightarrow \quad \texttt{id} :: \textit{name-tag-list}

\hspace{1cm} \mid \texttt{id} ::

117. \textit{named-command} \quad \rightarrow \quad \textit{name-tag-list} \textit{unnamed-command}

118. \textit{notify-command} \quad \rightarrow \quad \texttt{notify all-selective label-parameter event-specifier sound-specifier scope-specifier;}

119. \textit{on-off} \quad \rightarrow \quad \texttt{on}

\hspace{1cm} \mid \texttt{off}

121. \textit{pattern-specifier} \quad \rightarrow \quad \texttt{id}

\hspace{1cm} \mid \texttt{constant}

\hspace{2cm} \mid \texttt{specdef-use-command}

\hspace{2cm} \mid \texttt{VDAP-call-command}

\hspace{2cm} \mid \texttt{pattern-specifier play-pars}

\hspace{3cm} \mid ( \texttt{play-list} )

127. \textit{play-command} \quad \rightarrow \quad \texttt{play play-list;}

128. \textit{play-list} \quad \rightarrow \quad \textit{pattern-specifier} \mid \textit{play-list}
| pattern-specifier & & play-list
| play-list

131. play-pars \rightarrow with tagged-list

132. program-id-list \rightarrow external ext-id-list;
| \epsilon

134. range-list \rightarrow range-list , range
| range

136. range \rightarrow expression . . expression

137. relop \rightarrow < | > | <= | = | >= | <>

143. scope-specifier \rightarrow in tagged-scope-list
| \epsilon

145. scope-tagid-list \rightarrow scope-tagid-list , scope-tagid
| scope-tagid

147. scope-tag \rightarrow filename | func

149. score-const-id \rightarrow score-tag = const-id
| mm mmspec
| mode-specifier

152. score-tag \rightarrow keysig
| timesig

154. selector \rightarrow id | id[element-selector]

156. set-globals-command \rightarrow set global-par-list;

157. sound-specifier \rightarrow using play-list
| \epsilon

159. spec-command \rightarrow named-command
161. \textit{spec-def-body} \rightarrow \textit{spec-command spec-def-body}
| \textit{spec-command}

163. \textit{spec-def-list} \rightarrow \textit{spec-def spec-def-list}
| \epsilon

165. \textit{spec-def} \rightarrow \textit{specdef id (spec-par-list)}
\textit{declarations}
\textit{begin id}
\textit{spec-def-body}
\textit{end id};

166. \textit{spec-module-list} \rightarrow \textit{spec-module-list spec-module}

167. \rightarrow \textit{spec-module}

169. \textit{spec-module} \rightarrow \textit{specmodule id}
\textit{program-id-list}
\textit{global-interaction-list}
\textit{declarations}
\textit{spec-def-list}
\textit{VDAP-list}
\textit{begin id}
\textit{spec-def-body}
\textit{end id};

170. \rightarrow \textit{spec-module}

172. \textit{spec-module} \rightarrow \textit{specmodule id}
\textit{program-id-list}
\textit{global-interaction-list}
\textit{declarations}
\[ \text{spec-def-list} \]
\[ \text{VDAP-list} \]
\[ \text{begin id} \]
\[ \text{spec-def-body} \]
\[ \text{end id;} \]

173. \[ \text{spec-par-list} \rightarrow \text{id-list} | \epsilon \]

175. \[ \text{specdef-use-command} \rightarrow \text{id (actual-par-list);} \]
\[ \text{\hspace{1em}} | \text{id ( );} \]

177. \[ \text{start-event-spec} \rightarrow \text{when \ event-specifier scope-specifier} \]
\[ \text{\hspace{1em}} | \epsilon \]

179. \[ \text{statement-body} \rightarrow \text{begin spec-def-body end} \]
\[ \text{\hspace{1em}} | \text{spec-command;} \]

181. \[ \text{step-expression} \rightarrow \text{step expression} \]
\[ \text{\hspace{1em}} | \epsilon \]

183. \[ \text{sync-command} \rightarrow \text{sync to sync-to-id;} \]

184. \[ \text{sync-par-list} \rightarrow \text{sync-par-list, sync-par} \]
\[ \text{\hspace{1em}} | \text{sync-par} \]

186. \[ \text{sync-par} \rightarrow \text{bufsize = const} \]
\[ \text{\hspace{1em}} | \text{noslow} \]
\[ \text{\hspace{1em}} | \text{mmkeyword} \]
\[ \text{\hspace{1em}} | \text{mmkeyword mmspec} \]

190. \[ \text{sync-to-id} \rightarrow \text{program} \]
\[ \text{\hspace{1em}} | \text{sync-par-list} \]

192. \[ \text{tagged-list} \rightarrow \text{tagged-list, tags} \]
\[ \text{\hspace{1em}} | \text{tags} \]
194. tagged-scope-list → tagged-scope-list and tagged-scope
| tagged-scope

196. tagged-scope → scope-tag = scope-tagid-list

197. tags → score-const-id
| device-const-id

199. term-event-spec → until event-specifier

scope-specifier
| ε

201. term → term mulop factor
| factor

203. toggle-command → toggle toggle-source = constant;

204. toggle-source → keyboard
| midi

206. turn-command → turn on-off device-tag-list;

207. type → int | note | pattern | voice | file | ksig | tsig
| array-declarator

215. unnamed-command → set-globals-command
| play-command
| notify-command
| dtrack-command
| atrack-command
| assign-command
| loop-command
| if-command
| specdef-use-command
| VDAP-call-command
A.5 Subset of LSL Grammar Implemented in version 2.0

The syntax of LSL is described below using a modified form of BNF[ASU86]. Nonterminals are in *italics*, keywords in *teletype* font, and lexical symbols in *bold* font. Alternates of a nonterminal are separated by the | symbol.

1. \textit{lsl}\text{-}spec \quad \rightarrow \quad \texttt{begin auralspec} \\
   \texttt{spec-module-list} \\
   \texttt{end auralspec}. \\

228. \textit{unop} \quad \rightarrow \quad - \\
229. \textit{var-decl-list} \quad \rightarrow \quad \texttt{var}\text{-}type-list; \\
   | \texttt{var}\text{-}type-list \\
231. \textit{var-declaration} \quad \rightarrow \quad \texttt{var}\texttt{ var-decl-list;} \\
   | \epsilon \\
233. \textit{var-type-list} \quad \rightarrow \quad \texttt{id}\text{-}list:\texttt{ type} \\
234. \textit{VDAP-call-command} \quad \rightarrow \quad \texttt{l}\text{-}id (\texttt{l}\text{-}actual-parameter-list); \\
235. \textit{VDAP-list} \quad \rightarrow \quad \texttt{VDAP}\text{-}spec \texttt{VDAP-list} \\
   | \epsilon \\
237. \textit{VDAP-spec} \quad \rightarrow \quad \texttt{VDAP begin} \\
   \texttt{l-function} \\
   \texttt{VDAP end;} \\
238. \textit{while-loop} \quad \rightarrow \quad \texttt{while condition do statement-body}
2. \[\text{spec-module-list} \rightarrow \text{spec-module-list} \text{ spec-module}\]

3. 

5. \[\text{spec-module} \rightarrow \text{specmodule id} \]
   \[\text{begin id} \]
   \[\text{VDAP-list} \]
   \[\text{spec-def-body} \]
   \[\text{end id;}\]

6. \[\text{spec-def-list} \rightarrow \text{spec-def spec-def-list} \]
   \[\mid \epsilon\]

8. \[\text{spec-def} \rightarrow \text{specdef} \]
   \[\text{begin id} \]
   \[\text{spec-def-body} \]
   \[\text{end id;}\]

9. \[\text{spec-def-body} \rightarrow \text{spec-command spec-def-body} \]
   \[\mid \text{spec-command}\]

11. \[\text{spec-command} \rightarrow \text{named-command} \]
    \[\mid \text{unnamed-command}\]

13. \[\text{named-command} \rightarrow \text{name-tag-list} \text{ unnamed-command}\]

14. \[\text{name-tag-list} \rightarrow \text{id :: name-tag-list} \]
    \[\mid \text{id ::}\]

16. \[\text{unnamed-command} \rightarrow \text{notify-command} \]
    \[\mid \text{dtrack-command}\]
    \[\mid \text{attrack-command}\]
    \[\mid \text{sync-command}\]

20. \[\text{notify-command} \rightarrow \text{notify event-specifier} \]
    \[\text{sound-specifier scope-specifier},\]
21. \textit{event-specifier} \rightarrow \textit{event-specifier} \textit{connector} \textit{event} \\
\hspace{1cm} | \textit{event} \\
23. \textit{connector} \rightarrow \textit{and} \mid \textit{or} \\
25. \textit{event} \rightarrow \textit{rule} = \textit{id} \\
\hspace{1cm} | \textit{rule} = \textit{id} : \textit{instance-list} \\
\hspace{1cm} | \textit{assertion} = \textit{l-condition} \\
\hspace{1cm} | (\textit{event-specifier}) \\
29. \textit{instance-list} \rightarrow \textit{instance-list} \&\& \textit{instance} \\
\hspace{1cm} | \textit{instance} \\
31. \textit{instance} \rightarrow \textit{string} \\
32. \textit{sound-specifier} \rightarrow \textit{using constant} \\
\hspace{1cm} | \epsilon \\
34. \textit{scope-specifier} \rightarrow \textit{in} \textit{tagged-scope-list} \\
\hspace{1cm} | \epsilon \\
36. \textit{tagged-scope-list} \rightarrow \textit{tagged-scope-list} \textit{and} \textit{tagged-scope} \\
\hspace{1cm} | \textit{tagged-scope} \\
38. \textit{tagged-scope} \rightarrow \textit{scope-tag} = \textit{scope-tagid-list} \\
39. \textit{scope-tag} \rightarrow \textit{filename} \mid \textit{func} \\
41. \textit{scope-tagid-list} \rightarrow \textit{scope-tagid-list}, \textit{scope-tagid} \\
\hspace{1cm} | \textit{scope-tagid} \\
43. \textit{scope-tagid} \rightarrow \textit{string} \\
44. \textit{dtrack-command} \rightarrow \textit{dtrack dtrack-id-list start-event-spec term-event-spec} \\
\hspace{1cm} \textit{sound-specifier} \textit{scope-specifier}; \\
45. \textit{attract-command} \rightarrow \textit{attract start-event-spec term-event-spec}
sound-specifier scope-specifier;

46. \( \text{start-event-spec} \rightarrow \text{when event-specifier scope-specifier}; \)
   \[\varepsilon\]

48. \( \text{term-event-spec} \rightarrow \text{until event-specifier scope-specifier}; \)
   \[\varepsilon\]

50. \( \text{dtrack-id-list} \rightarrow \text{dtrack-id-list and dtrack-id} \)
    \[\varepsilon\]

52. \( \text{dtrack-id} \rightarrow \text{l-id scope-specifier} \)

53. \( \text{sync-command} \rightarrow \text{sync to sync-to-id;} \)

54. \( \text{sync-to-id} \rightarrow \text{program} | \text{sync-par-list} \)

56. \( \text{sync-par-list} \rightarrow \text{sync-par-list, sync-par} \)
    \[\varepsilon\]

58. \( \text{sync-par} \rightarrow \text{mmkeyword} \mid \text{mmkeyword mmspec} \)

60. \( \text{mmkeyword} \rightarrow \text{mm} | \text{mmabs} | \text{mmrel} \)

63. \( \text{mmspec} \rightarrow \varepsilon | q = \text{const} \)

65. \( \text{id-list} \rightarrow \text{id, id-list} \)
    \[\varepsilon\]

67. \( \text{VDAP-call-command} \rightarrow \text{l-id (l-actual-parameter-list);} \)

68. \( \text{VDAP-list} \rightarrow \text{VDAP-spec VDAP-list} \)
    \[\varepsilon\]

70. \( \text{VDAP-spec} \rightarrow \text{VDAP begin} \)
    \[\text{l-function} \]
    \[\text{VDAP end;}\]
Appendix B: Run-Time Audio Controller Protocols

In Section B.1, we describe all the commands that may be issued to the sound server. Sections B.2, B.3, and B.4 provide the purpose and a brief description of the functions in the client library, sound server, and run-time GUI, respectively.

B.1 The Sound Server Protocol

- **CHANNEL_OFF channel**
  
  Turn off a single MIDI channel.

- **CHANNEL_ON channel**
  
  Turn on a single MIDI channel.

- **CLASS_OFF class_name**
  
  Turn off a class of specifications.

- **CLASS_ON class_name**
  
  Turn on a class of specifications.

- **DEFINE_NEW_SOUND name channel note patch velocity duration**
  
  Add a new sound to the sound database.

- **DEFINE_SOUND name channel note velocity duration**
  
  Modify an existing sound in the database.

- **EXIT**
  
  Finish playing all notes and kills the server.

- **GET_CLASS_DEF class_name**
  
  Get the definition of a class. Class definitions are of the form: class_name status, where status is 0 when the class is off and 1 when on.
• GET_FIRST_CLASS
  Reset the iterator to the first class in the class list and return its definition.

• GET_NEXT_CLASS
  Increment the iterator to the next class in the class list and return its definition.

• GET_FIRST_SOUND
  Reset the iterator to the first sound in the sound list and return its definition.

• GET_NEXT_SOUND
  Increment the sound iterator to the next sound in the sound list and return its definition.

• GET_FIRST_SPEC
  Reset the iterator to the first specification in the specification list and return its definition.

• GET_NEXT_SPEC
  Increment the iterator to the next specification in the specification list and return its definition.

• GET_NUMBER_OF_SOUNDS
  Return the number of sounds in the sound list.

• GET_NUMBER_OF_SPECS
  Return the number of sounds in the sound list.

• GET_SOUND_DEF sound_name
  Return the definition of a sound. Sound definitions are of the form: name channel note velocity duration patch.

• GET_SPEC_DEF spec_name
Return the definition of a specification. Specification definitions are of the form:

\textit{type status name sound number of classes classes.}

- **HEARTBEAT**
  
  Do a heartbeat.

- **HEARTBEAT\_NO\_NOTE**
  
  Do a heartbeat without sound.

- **INIT**
  
  No-op. Test the connection from a client.

- **LOAD\_SOUND\_DB file name**
  
  Load a sound database file.

- **LOAD\_SPEC\_DB file name**
  
  Load an LSL specification file.

- **MAP\_SOUND spec name sound name**
  
  Remap a specification’s sound to another sound in the sound list.

- **MERGE\_SOUND\_DB file name**
  
  Load a sound database file and merge it with the sound list.

- **PAUSE**
  
  Pause the MIDI driver.

- **PLAY\_RAW channel note velocity duration**
  
  Add a raw sound to the MIDI queue.

- **PLAY\_SOUND\_1 sound name**
  
  Add a sound to the MIDI queue.
- **PLAY\_SOUND\_2 sound name note**
  Add a sound to the MIDI queue and change the note.

- **PLAY\_SOUND\_3 sound name mode**
  Add a sound to the MIDI queue. Turn the note off if mode = 0, on if mode = 1.

- **PLAY\_SOUND\_DIRECT sound name**
  Write the sound directly to the MIDI device, bypassing the queue.

- **PLAY\_SPEC\_1 spec name**
  Add the sound for the given specification to the MIDI queue.

- **PLAY\_SPEC\_2 spec name note**
  Add the sound for the given specification to the MIDI queue and change the note.

- **PLAY\_SPEC\_3 spec name mode**
  Add the sound for the given specification to the MIDI queue. Turn the note off if mode = 0, on if mode = 1.

- **RESUME**
  Restart the MIDI driver after a pause.

- **SAVE\_SOUND\_DB file name**
  Save the sound list to a sound database file.

- **SAVE\_SPEC\_DB file name**
  Save the specification list to an LSL specification file.

- **SET\_TEMPO tempo value**
  Set the tempo.
• **SOUND_INDEX** *sound_name*
  Get the index of the sound in the sound list. Used to check that a sound is in the list.

• **SPEC_OFF** *spec_name*
  Turn off a specification.

• **SPEC_ON** *spec_name*
  Turn on a specification.

• **UNLOAD_SOUND_DB**
  Free the sound list.

• **UNLOAD_SPEC_DB**
  Free the specification and class lists.

B.2  The Client Library

*xlisten* and user executable file share the code for interacting with the sound server. This code exists in the client library, *libclient.a*.

• **_lsl_send[.ch]** contains the code for starting the sound server, opening connections to the server, and sending and receiving messages from the server.

  - Public Functions:
    
    /* Opens a connection to the server, starting it if reqd */
    int _lsl_open( const char *serv_name, const char *tmp_path,
                   const char *serv_socket, const char *cli_socket );

    /* Sends a command to the server and returns the response*/
    int _lsl_send( int servfd, const char *wbuf, char *rbuf );

  - Private Functions:
/* Gets a response from the server. */
int _ls1_recv( int servfd, char *rbuf );

/* Initiate a connection to the server. */
int _ls1_conn(const char *cli_socket,
              const char *serv_socket);

/* Creates a UNIX domain socket and bind it to a name */
int _ls1_create_socket( const char* name );

/* Bind a socket to a path name. */
int _ls1_bind_socket( int fd, const char *name );

- _ls1_proto.h contains the list of all command strings in the protocol.
- _rt_driver.c contains the main() function. It initializes LSL and calls the real main(), which is renamed by ls1oc to _l.m.

B.3 The Sound Server

The sound server is responsible for maintaining a map between the LSL specifications in an LSL file and the sound associated with the specification, for monitoring the status of the LSL specifications, for playing sounds, and for pausing and resuming the MIDI device.

- main.[ch] checks command-line arguments and initializes the program.
  - Public Functions:
    /* Initialize command-line arguments and enter server loop*/
    int main( int argc, char *argv[] );

    /* Shuts down the server. */
void quit( int sig );

• loop.[ch] contains the main loop of the server.
  
  - Public Functions:
    
    /* The main loop of the server. */
    int loop( const char *name, int semid );
  
  - Private Functions:
    
    /* Cleans up after a client. */
    void cli_done( int clifd, fd_set *allset );

• serv.[ch] contains utility functions for the server.
  
  - Public Functions:
    
    /* Announce server’s willingness to listen for connections */
    int serv_listen( const char *name );

    /* Wait for a client connection to arrive. */
    int serv_accept( int listenfd, uid_t *uidptr );
  
  - Private Functions:
    
    /* Bind a socket to a path name. */
    int bind_socket( int fd, const char *name );

• cli_list.[ch] maintains the server’s list of clients.
  
  - Public Data:
    
    /* The client list. */
    extern ClientStruct *AppClientList;
  
  - Public Functions:
/* Add a new client to the list. */
int client_add( int fd, uid_t uid );

/* Delete a client from the list. */
void client_del( int fd );

/* Delete all clients from the list. */
void client_empty( void );

/* Get the client with the given descriptor. */
ClientStruct *client_get( int fd );

- Private Functions:

 /* Allocate space for new client structures. */
 void client_alloc( void );

- dispatch.[ch] dispatches commands to the appropriate function.

- Public Functions:

 /* Dispatches commands to the appropriate function and sends
  * results back to the client. */
 void dispatch( ClientStruct *cliptr, char *buf );

- Private Functions:

 /* Looks up a keyword in a table to get the case label for
  * dispatch(). */
 int which_keyword( char *token );

- table.h contains an array of all the command strings in the protocol. This list
  must be in the same order as the list in commands.h.
• **commands.h** contains an enumerated type with entries corresponding to the commands in the protocol. This type is used to map the command strings in the protocol to integers. This list must be in the same order as the list in **table.h**.

• **serv_send[ch]** sends responses back to the client.

  - Public Functions:
    
    ```c
    /* Send a reply back to a client. */
    int serv_send( int clifd, int status, char *buf );
    ```

• **sound[ch]** sends responses back to the client.

  - Public Functions:
    
    ```c
    /* Turn off/on a MIDI channel. */
    int channelOff( int channel );
    int channelOn( int channel );
    
    /* Turn off/on a class. */
    int classOff( char *class );
    int classOn( char *class );
    
    /* Turn off/on a spec. */
    int specOff( char *spec );
    int specOn( char *spec );
    
    /* Change the sound of a spec. */
    int mapSpecToSound( char *spec, char *sound );
    
    /* Modify an existing sound. */
    int modifySound( char *sound, int channel, int note,
```
int velocity, int duration);

/* Add a new sound. */
int newSound( char *sound, int channel, int note, 
             int velocity, int duration, int patch);

/* Play the sound with the given parameters. */
int playRaw( int channel, int note, int velocity, 
            int duration);

/* Play the sound for the spec. */
int playSpec1( char *spec);
int playSpec2( char *spec, int mode);
int playSpec3( char *spec, int value);

/* Play the sound. */
int playSound1( char *sound);
int playSound2( char *sound, int mode);
int playSound3( char *sound, int value);

/* Play the sound, bypassing the MIDI queue. */
int playSoundDirect( char *sound);

/* Get the number of specs. */
int getNumberSpecs( void);

/* Reset/increment speclist iterator, returning the spec. */
int resetSpecIter( char *reply);
int incrSpecIter( char *reply);
/* Reset/increment classlist iterator, returning the class. */
int resetClassIter( char *reply );
int incrClassIter( char *reply );

/* Reset/increment soundlist iterator, returning the sound. */
int resetSoundIter( char *reply );
int incrSoundIter( char *reply );

/* Get the spec’s definition. */
int getSpecDef( char *name, char *reply );

/* Get the class’s definition. */
int getClassDef( char *name, char *reply );

/* Get the sound’s definition. */
int getSoundDef( char *name, char *reply );

- Private Functions:

/* Play the sound with the given number. */
int playSoundByNum1(int sound, struct spec *spec);
int playSoundByNum2(int sound, int mode, struct spec *spec);
int playSoundByNum3(int sound, int value, struct spec *spec);

- files.[ch] handles file operations such as loading and unloading ISL specification files and sound databases.

- Public Functions:

/* Load another sound file; merge it with the one loaded */
int mergeSoundFile( char *file );
/* Load a sound file. */
int loadSoundFile( char *file );

/* Unload a sound file. */
int unloadSoundFile( void );

/* Load a LSL specification file. */
int loadSpecFile( char *file );

/* Unload a LSL specification file. */
int unloadSpecFile( void );

B.4 The Run-Time Controller GUI

xlisten is a X/Motif GUI which uses the sound server to control the behavior of an auralized program. xlisten connects to the sound server through the client library and has controls for loading and unloading files, remapping LSL specification sounds, turning LSL specifications and classes of specifications on and off, editing existing sounds, and adding new sounds. The GUI can fork off a single executable program which can then connect to the sound server and play sounds in a way that the GUI defines. During the execution of the auralized program, a user can use the GUI to suspend and resume the program execution.

- main.c contains the top-level code for xlisten.
  - Public Data:
  - Public Functions:

/* The top-level function. Parses command-line arguments,
  * starts the sound server, creates the main window, loads
  * files, and enters the X event loop. */
int main( int argc, char *argv[], char *envp[] );

/* Signal handler/exit function. Cleans up the mess. */
void quit( int sig );

- mainWindow.[ch] contains the code for the main window of the GUI, including the status lines and the sort and display toggle button groups. It also contains the error, warning and question dialog code.

- Public Data:

- Public Functions:

  /* Creates the main window and all its subwindows. */
  Widget createMainWindow( Widget parent );

  /* Displays a question dialog. */
  int questionDB( char *message );

  /* Displays a warning dialog. */
  int warningDB( char *message );

  /* Displays an error dialog. */
  int errorDB( char *message );

- Private Functions:

  /* Creates the lines in the main window displaying the * names of the files currently loaded. */
  Widget createStatusLine( Widget parent );

  /* Creates two toggle button groups for sorting and display * selection for the specification list. */
Widget createToggleButton( Widget parent );

/* Creates a single toggle button inside a toggle button group. */
Widget createRadioBox( Widget parent, char* labelName,
    char *frameName, char *boxName, int behavior );

/* Creates Change Sound (Edit Auralization) button. */
Widget createControlButtons( Widget parent );

/* Callback for Change Sound (Edit Auralization) button*/
void editAuralCB( Widget w, XtPointer clientData,
    XtPointer callData );

/* Callback for the sort toggle button group. */
void sortBoxCB( Widget w, XtPointer clientData,
    XtPointer callData );

/* Callback for the display toggle button group. */
void displayBoxCB( Widget w, XtPointer clientData,
    XtPointer callData );

/* Displays a question, warning, or error dialog depending
 * on the mode and returns the response. */
int messageDB( char *message, int mode );

/* Callback for the message dialog buttons. */
void messageDBCBI( Widget w, int *answer,
    XmAnyCallbackStruct *cbs );
/* From Feb 1995 Motif FAQ #116. */
* Creates an XmString to handle embedded \n characters.
* It’s mostly equivalent to XmStringCreateLtoR(). */
XmString multilineXmString( char *s );

- mainMenu.[ch] contains the code for the main menu inside the main window.

- Public Functions:

  /* Create the main menu inside the main window. */
  Widget createMainMenu( Widget parent );

- Private Functions:

  /* Make the pulldown menu for each top-level menu item */
  Widget makePulldown( Widget parent, char *name, int mnemonic,
                       MenuItem *items );

  /* Callback for Open Default Sound Database menu item. */
  void fileOpenDefaultSDBCB( Widget w, XtPointer clientData,
                              XtPointer callData );

  /* Callback for Open Executable menu item. */
  void fileOpenExecCB( Widget w, XtPointer clientData,
                       XtPointer callData );

  /* Callback for Open Executable File Selection dialog. */
  void loadExecFileCB( Widget w, XtPointer clientData,
                       XmFileSelectionBoxCallbackStruct *callData );
/* Callback for Open LSL Specification File menu item. */
void fileOpenLSLCB( Widget w, XtPointer clientData,
                    XtPointer callData );

/* Callback for Open LSL Specification File Selection dialog. */
void loadLSLFileCB( Widget w, XtPointer clientData,
                   XmFileSelectionBoxCallbackStruct *callData );

/* Callback for Import Sound Database File Selection dialog. */
void importSDBFileCB( Widget w, XtPointer clientData,
                      XmFileSelectionBoxCallbackStruct *callData );

/* Callback for Import Sound Database menu item. */
void fileImportSDBC( Widget w, XtPointer clientData,
                     XtPointer callData );

/* Callback for Open Sound Database menu item. */
void fileOpenSDBC( Widget w, XtPointer clientData,
                   XtPointer callData );

/* Callback for Open Sound Database File Selection dialog. */
void loadSDBFileCB( Widget w, XtPointer clientData,
                    XmFileSelectionBoxCallbackStruct *callData );

/* Callback for Save LSL Specification menu item. */
void fileSaveLSLC( Widget w, XtPointer clientData,
                   XtPointer callData );

/* Callback for Save Sound Database menu item. */
void fileSaveSDBDB( Widget w, XtPointer clientData,
                      XtPointer callData );

/* Callback for Quit menu item. */
void fileQuitCB( Widget w, XtPointer clientData,
                        XtPointer callData );

/* Callback for Global Sync item. */
void prefGlobalSyncCB( Widget w, XtPointer clientData,
                        XtPointer callData );

• fileBox.[ch] contains the code for the file selection dialogs.

  – Public Functions:

  /* Create the file selection dialog. */
  Widget createFileBox( void );

  /* Open the file selection dialog. */
  Widget openFileBox( char *title, char *filter,
                           void (*okCallback)(), XtPointer okClientData );

  – Private Functions:

  /* Callback for Cancel button in file selection dialog. */
  void fileBoxCancelCB( Widget w, XtPointer clientData,
                              XtPointer callData );

  /* Closes file selection dialog. */
  void closeFileBox( XmFileSelectionBoxCallbackStruct *callData );

• fileOps.[ch] contains the code for interacting with the sound server to perform file operations.
- Public Functions:

  /* Load an executable file. */
  int loadExecutable( char *name );

  /* Load an LSL specification file. */
  int loadSpecFile( char *name );

  /* Load a sound database. */
  int loadSoundDatabase( char *name );

  /* Load the default sound database. */
  int loadDefaultSoundDatabase();

  /* Import a sound database into one already loaded. */
  int importSoundDatabase( char *name );

- Private Functions:

  /* Return the base name of the file name. */
  char *basename( char *name );

  /* Return the absolute path of the file name, assuming
   * it is relative to the current working directory. */
  char *absolutePath( char *base );

- runButtons.[ch] contains the code for executing, pausing, and resuming the
  auralized program.

- Public Functions:

  /* Create the main window buttons for running, pausing, and
   * resuming execution of the auralized program. */
Widget createRunButtons( Widget parent );

/* Callback for Run with Arguments menu item. */
void runWithArgsCB( Widget w, XtPointer clientData,
                     XtPointer callData );

/* Callback for Run with Arguments dialog box. */
void runCommandCB( Widget w, XtPointer clientData,
                   XmSelectionBoxCallbackStruct *callData );

/* Callback for Run menu item and Run button. */
void runCB( Widget w, XtPointer clientData,
            XtPointer callData );

/* Callback for Pause menu item and Pause button. */
void pauseCB( Widget w, XtPointer clientData,
              XtPointer callData );

/* Callback for Continue menu item and Continue button. */
void continueCB( Widget w, XtPointer clientData,
                 XtPointer callData );

- Private Functions:

/* Executes a process. */
void execProcess( const char *cmd, const char *args,
                   char *envp[] );

/* Sets button sensitivity depending on whether the mode is
 * STOPPED, RUNNING, or PAUSED. */
void setButtonSensitivity( int mode );
• `specWindow[ch]` contains the code for the specification selection list in the main window.

  - Public Functions:

    /* Create the specification list window. */
    Widget createSpecWindow( Widget parent );

    /* Destroy the window. */
    void destroySpecWindow( void );

    /* Initialize the lists of specifications and classes after
    * a new file has been loaded. */
    void initializeInfoLists( void );

    /* Update the structure describing the given specification*/
    void updateSpec( char *name );

    /* Update the structure describing the given class. */
    void updateClass( char *name );

    /* Update the list after a change is made. */
    void updateSpecWindow( void );

    /* Callback for a list element window’s on/off button. */
    void specToggleCB( Widget w, XtPointer clientData,
                       XtPointer callData );

    /* Callback for a click on a list element window.
    * Selects the element. */
void selectSpecActionCB( Widget w, XEvent *event,
        String *params, Cardinal *numParams );

/* Callback for a button-3 click on a list element window. */
/* Selects the element and opens the Change Sound dialog*/
void editAuralActionCB( Widget w, XEvent *event,
        String *params, Cardinal *numParams );

- Private Functions:

/* Create a generic list element window. */
LineStruct *createBaseLine( Widget parent, int index );

/* Create a class list element window. */
LineStruct *createClassLine( Widget parent, int index,
        ClassInfoStruct *class );

/* Create a specification list element window. */
LineStruct *createSpecLine( Widget parent, int index,
        SpecInfoStruct *spec );

/* Destroy the lists of specifications and classes. */
void destroyInfoLists( void );

/* Remove all the list element windows from specification
   * list window. */
void clearSpecWindow( void );

/* Compare two specifications for sorting. */
int compareSpec( void *a, void *b );
/* Compare two classes for sorting. */
int compareClass( void *a, void *b );

- editAural.[ch] contains the code for the Change Sound (aka Edit Auralization) dialog that is invoked by clicking the Change Sound button or by clicking a specification with the third mouse button.

  - Public Functions:

    /* Create the Change Sound Dialog. */
    void createEditAuralDialog( Widget parent );

    /* Open the Change Sound Dialog. */
    void openEditAuralDialog( LineStruct* line );

    /* Callback for the OK button. */
    void specOkCB( Widget w, XtPointer clientData, 
                   XtPointer callData );

    /* Callback for the Cancel button. */
    void specCancelCB( Widget w, XtPointer clientData, 
                       XtPointer callData );

    /* Callback for the Edit Existing Sound button. */
    void editSoundCB( Widget w, XtPointer clientData, 
                      XtPointer callData );

    /* Callback for the Add New Sound button. */
    void addSoundCB( Widget w, XtPointer clientData, 
                     XtPointer callData );
/* Callback for the Play Sound button. */
void testSoundCB( Widget w, XtPointer clientData,
    XtPointer callData );

/* Callback for item selection in the sound list. */
void changeSoundCB( Widget w, XtPointer clientData,
    XmListCallbackStruct *callData );

/* Callback for the on/off radiobuttons. */
void specToggleBtnCB( Widget w, XtPointer clientData,
    XtPointer callData );

- Private Functions:

    /* Fills the list of classes for the specification. */
    void fillClassList( SpecInfoStruct *spec );

    /* Inserts a string into a list widget in alphabetical order*/
    int insertIntoList( char *string, Widget list );

    /* Makes a radio button box. */
    Widget makeRadioBoxWithSeparators( Widget parent );

    /* Fills the sound list. */
    void fillSoundList( char *selectedSound );

    /* Empties the sound list. */
    void clearSoundList( void );

*editSound*.ch contains the code for the Edit Existing Sound and the Add New Sound dialogs.
- Public Functions:

  /* Creates the dialog. */
  void createEditSoundDialog( Widget parent );

  /* Opens the dialog. If sound is NULL, its Add New Sound
   * dialog, otherwise, its the Edit Existing Sound dialog*/
  void openEditSoundDialog( char *sound );

  /* Destroys the dialog. */
  void destroyEditSoundDialog( void );

  /* Callback for the OK button. */
  void editOkCB( Widget w, XtPointer clientData,
                 XtPointer callData );

  /* Callback for the Cancel button. */
  void editCancelCB( Widget w, XtPointer clientData,
                     XtPointer callData );

  /* Callback for the Play button. */
  void testRawSoundCB( Widget w, XtPointer clientData,
                       XtPointer callData );

  /* Callback for the Change Instrument button. */
  void instrumentChangeCB( Widget w,
                           XtPointer clientData, XtPointer callData );

- Private Functions:

  /* Makes an option menu widget. */
Widget makeOptionMenu( Widget parent, char *label,
        char *choices[], XtCallbackProc callback );

/* Callback to destroy the option menu's container. */
void deleteOptionMenuContainerCB( Widget w, XtPointer clientData,
        XtPointer callData );

/* Callback to check Name field when the user leaves it*/
void textFocusOutCB( Widget w, XtPointer clientData,
        XtPointer callData );

/* Makes a scale widget. */
ScaleBox *makeScaleBox( Widget parent, char *name,
        char *lowLabel, char *highLabel, int low, int high,
        int initial );

/* Gets the duration selected in the duration option menu. */
int getDuration( void );
Appendix C: Experiment Materials and Results

C.1 Instructions to Subjects

Thank you for your participation in this experiment. The purpose of this experiment is to compare the effectiveness of aural cues, visual cues, and a combination of aural and visual cues in understanding program behavior.

- You will be given a C program and some erroneous versions of the same program. Each erroneous version contains one bug e.g. it may differ from the correct program in exactly one relational operator (e.g. a ‘<’ changed to ‘>’ or ‘<=’). Each erroneous program is called a mutant.

- Execute the correct program on a given test input. Examine the output.

- Repeat the following three steps for each mutant.

- Execute a mutant on the same input. Examine the output.

- Classify the mutant into one of the two categories, namely, LIKELY-KILLED or LIKELY-EQUIVALENT.

You will use the following criterion for classification. If the output produced by a mutant is same as that produced by the correct program, then that mutant should be classified as LIKELY-EQUIVALENT else it should be classified as LIKELY-KILLED.

C.1.1 Examining visual-only cues

Look at the output produced by the correct program. Look at the output produced by a mutant. If outputs are exactly the same, classify as LIKELY-EQUIVALENT else as LIKELY-KILLED.

You may execute the programs as many times as you like.
C.1.2 Examining aural-only cues

Listen to the output produced by the correct program. Listen to the output produced by a mutant. If outputs are exactly the same, classify as LIKELY-EQUIVALENT else as LIKELY-KILLED.

You may execute the programs as many times as you like.
### Table C.1: Mutants for Cal program

<table>
<thead>
<tr>
<th>Line #</th>
<th>Original Code</th>
<th>Modified Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td><code>y = tm-&gt;tm_year + 1900</code></td>
<td><code>y = tm-&gt;tm_year + 1900</code></td>
</tr>
<tr>
<td>44</td>
<td>`if(y&lt;1</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td><code>for(i=0; i&lt;624; i=24)</code></td>
<td><code>for(i=0; i&lt;624; i=24)</code></td>
</tr>
<tr>
<td>52</td>
<td>`if(y&lt;1</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td><code>for(i=0; i&lt;12; i+=3)</code></td>
<td><code>for(i=0; i&lt;12; i+=4)</code></td>
</tr>
<tr>
<td>98</td>
<td><code>if(s++ == '\0')</code></td>
<td><code>if(s++ == 'O')</code></td>
</tr>
<tr>
<td>98</td>
<td><code>if(s++ == 'E')</code></td>
<td><code>if(s++ == 'O')</code></td>
</tr>
<tr>
<td>108</td>
<td><code>30, 31, 30, 31,</code></td>
<td><code>30, 31, 30, 30,</code></td>
</tr>
<tr>
<td>120</td>
<td><code>mon[0] = 30</code></td>
<td><code>mon[0] = 31</code></td>
</tr>
<tr>
<td>127</td>
<td><code>case 1: mon[2]=28; break;</code></td>
<td>delete this case</td>
</tr>
<tr>
<td>149</td>
<td><code>if(i=3 &amp;&amp; mon[m]=19)</code></td>
<td>`if(i=3</td>
</tr>
<tr>
<td>155</td>
<td><code>s++</code></td>
<td>delete this statement</td>
</tr>
<tr>
<td>146</td>
<td><code>d %= 7</code></td>
<td><code>i %= 7</code></td>
</tr>
<tr>
<td>160</td>
<td><code>s = p+w</code></td>
<td><code>s = p</code></td>
</tr>
</tbody>
</table>

### Table C.2: Mutants for Look program

<table>
<thead>
<tr>
<th>Line #</th>
<th>Original Code</th>
<th>Modified Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>118</td>
<td><code>puts(entry)</code></td>
<td><code>puts(copy)</code></td>
</tr>
<tr>
<td>158</td>
<td><code>*copy = BES</code></td>
<td><code>*src = BES</code></td>
</tr>
<tr>
<td>95</td>
<td><code>top = mid</code></td>
<td><code>bot = mid</code></td>
</tr>
<tr>
<td>97</td>
<td><code>else bot = mid</code></td>
<td><code>bot = mid</code></td>
</tr>
<tr>
<td>155</td>
<td><code>for(cnt=len+1; (*src++) &amp;&amp; cnt--;cnt)</code></td>
<td><code>for (cnt=len;(*src++) &amp;&amp; cnt--;cnt)</code></td>
</tr>
<tr>
<td>138</td>
<td><code>*buf++ = c</code></td>
<td><code>*buf = c</code></td>
</tr>
<tr>
<td>155</td>
<td><code>for (cnt=len+1; (*src++) &amp;&amp; cnt--;cnt)</code></td>
<td><code>for (cnt=len+1; (*src++) &amp;&amp; cnt--;cnt)</code></td>
</tr>
<tr>
<td>157</td>
<td><code>*copy++=fold &amp;&amp; isupper(c)? tolower(c):c</code></td>
<td><code>*copy++=fold &amp;&amp; (isupper(c)? tolower(c):c)</code></td>
</tr>
<tr>
<td>47</td>
<td><code>fold = YES</code></td>
<td><code>len = YES</code></td>
</tr>
<tr>
<td>106</td>
<td><code>canon(entry, copy)</code></td>
<td><code>canon(copy, entry)</code></td>
</tr>
<tr>
<td>Line #</td>
<td>Original Code</td>
<td>Modified Code</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>783</td>
<td><code>cflg = 1</code></td>
<td><code>cflg = 0</code></td>
</tr>
<tr>
<td>772</td>
<td><code>p-&gt;ignore = dict+128</code></td>
<td><code>p-&gt;ignore = dict+290</code></td>
</tr>
<tr>
<td>211</td>
<td><code>break</code></td>
<td><code>continue</code></td>
</tr>
<tr>
<td>442</td>
<td><code>if(rline(ibuf[i-1])))</code></td>
<td><code>if(rline(ibuf[i]))</code></td>
</tr>
<tr>
<td>631</td>
<td><code>if(b == *-ipb - *-ipa)</code></td>
<td><code>if(b == *-ipb - *-ipa)</code></td>
</tr>
<tr>
<td>439</td>
<td><code>while(*dp++ = *cp++) != '\n'</code>)`</td>
<td><code>while(*dp++ = *cp++) != '\n'</code></td>
</tr>
<tr>
<td>634</td>
<td><code>(*-ipa != ')')</code></td>
<td><code>(--ipa != ')')</code></td>
</tr>
<tr>
<td>235</td>
<td><code>p-&gt;nflg = q-&gt;nflg</code></td>
<td><code>q-&gt;nflg = p-&gt;nflg</code></td>
</tr>
<tr>
<td>689</td>
<td><code>*pb = '\n' ? fields[0].rflg:</code></td>
<td><code>*pb = '\n' ? fields[0].rflg:</code></td>
</tr>
<tr>
<td>910</td>
<td><code>j = --hp</code></td>
<td><code>delete this statement</code></td>
</tr>
<tr>
<td>691</td>
<td><code>fields[0].rflg</code></td>
<td><code>fields[0].rflg</code></td>
</tr>
<tr>
<td>336</td>
<td><code>if(cp[1en - 2] != '\n')</code></td>
<td><code>if(cp[1en - 2] != '\t')</code></td>
</tr>
</tbody>
</table>
Table C.4: Raw Data on correctness by cue type. Each entry is the value of correctness for a given subject using a given cue type.

<table>
<thead>
<tr>
<th>Subject #</th>
<th>Visual</th>
<th>Aural</th>
<th>Aural-Visual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>0.8</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>1.0</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>7</td>
<td>1.0</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>8</td>
<td>0.6</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>9</td>
<td>0.6</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>1.0</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>11</td>
<td>0.8</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>12</td>
<td>1.0</td>
<td>0.8</td>
<td>1.0</td>
</tr>
<tr>
<td>13</td>
<td>0.8</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>14</td>
<td>0.8</td>
<td>0.8</td>
<td>1.0</td>
</tr>
<tr>
<td>15</td>
<td>0.8</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>16</td>
<td>1.0</td>
<td>0.8</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Table C.5: Raw Data on time-taken by cue type. Each entry is the value of time-taken for a given subject using a given cue type.

<table>
<thead>
<tr>
<th>Subject #</th>
<th>Visual</th>
<th>Aural</th>
<th>Aural-Visual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
options ls=70;
%include plot1w;

data first;
  input subject cuetype $ correct Time @;
  if _n_<= 18 then group=1;
  else if _n_<= 36 then group=2;
  else group=3;
cards;
  1 visual 1.0 4
  1 aural 0.8 6
  1 av 0.8 3
  2 visual 0.8 5
  2 aural 0.8 6
  2 av 0.8 3
  3 visual 1.0 3
  3 aural 0.4 2
  3 av 1.0 2
  4 visual 0.8 3
  4 aural 0.4 2
  4 av 1.0 2
  5 visual 1.0 3
  5 aural 0.4 4
  5 av 1.0 2
  6 visual 1.0 2
  6 aural 0.4 2
  6 av 1.0 2
  7 visual 1.0 2
  7 aural 0.6 3
  7 av 1.0 2
  8 visual 0.6 3
  8 aural 0.6 4
  8 av 1.0 2
  9 visual 0.6 3
  9 aural 0.6 4
  9 av 1.0 3
  10 visual 1.0 4
  10 aural 0.6 5
  10 av 1.0 4
  11 visual 0.8 3
  11 aural 0.6 4
  11 av 1.0 3
12 visual 1.0  4
12 aural  0.8  5
12 av     1.0  4
13 visual 0.8  4
13 aural  0.6  4
13 av     1.0  3
14 visual 0.8  4
14 aural  0.8  5
14 av     1.0  3
15 visual 0.8  3
15 aural  0.6  4
15 av     1.0  3
16 visual 1.0  3
16 aural  0.8  4
16 av     1.0  3
17 visual 0.8  3
17 aural  0.8  4
17 av     1.0  3
18 visual 0.8  3
18 aural  0.6  5
18 av     1.0  3
;
run;

********************************************************************************
/** Tables of data **/
********************************************************************************

Titlel 'Frequency Table of % Correct';
proc tabulate;
   class cuetype correct group;
   table group<cuetype,correct;
run;

Titlel 'Frequency Table of Time';
proc tabulate;
   class cuetype time group;
   table group<cuetype,time;
run;

********************************************************************************
/** Transform % correct with an arcsine to stabilize residuals *****/
********************************************************************************
data one;
   set first;
correct=arsin(correct);
run;

/* *****************************************/
/** Nested Factorial ***/
/* *****************************************/
/** % Correct ***/
/* *****************************************/
Title1 'Nested Factorial';
Title2 '% Correct?';
proc glm data=one;
   class group subject cuetype;
   model correct = group subject(group) cuetype group*cuetype;
   test h=group e=subject(group);
   test h=cuetype e=group*cuetype;
   output out=resid p=pred r=resid;
run;

/* *****************************************/
/** normality assumption ***/
/* *****************************************/
proc rank normal=blom out=nresid;
   ranks nresid;
   var resid;
run;

Title4 'Q-Q Plot?';
proc gplot;
   plot resid*nresid / frame;
run;

Title4 'Shapiro-Wilk Test of Normality?';
proc univariate normal;
   var resid;
run;

/* *****************************************/
/** variance assumption ***/
/* *****************************************/
Title4 'Plot to Review Residual Variance';
symbol1 value=square;
symbol2 value=dot;
symbol3 value=star;
run;

proc gplot;
   plot resid*pred=cuetype / frame;
run;

proc gplot;
   plot resid*cuetype / vref=0 frame;
run;

proc gplot;
   plot resid*group / vref=0 frame;
run;

proc gplot;
   plot resid*subject / vref=0 frame;
run;

>Title2 'Time';
proc glm data=one;
   class group subject cuetype;
   model time = group subject(group) cuetype group*cuetype;
   test h=group e=subject(group);
   test h=cuetype e=group*cuetype;
   output out=resid p=pred r=resid;
run;

>Title3 'Normality assumption';
proc rank normal=blom out=nresid;
   ranks nresid;
   var resid;
run;

SetTitle4 'Q-Q Plot';
proc gplot;
   plot resid*nresid / frame;
run;
Title4 'Shapiro-Wilk Test of Normality';
proc univariate normal;
   var resid;
run;

/******************************/
/******** variance assumption  **/
/******************************/
Title4 'Plot to Review Residual Variance';
proc gplot;
   plot resid*pred=ctype / frame;
run;

proc gplot;
   plot resid*ctype / vref=0 frame;
run;

proc gplot;
   plot resid*group / vref=0 frame;
run;

proc gplot;
   plot resid*subject / vref=0 frame;
run;

/******************************/
/******** Plot of data  **/
/******************************/
proc sort data=first;
   by group ctype;
run;

proc means data=first noprint;
   by group ctype;
   var time correct;
   output out=mean mean=time correct;
run;

symbol1 valuesquare i=join;
symbol2 values=dot i=join;
symbol3 value=star i=join;

title1 'Mean Correctness by Group and Cue Type';
proc gplot;
plot correct*group=ctype / frame;
run;

title1 'Mean Time Taken by Group and Cue Type';
proc gplot;
   plot time*groups=ctype / frame;
run;

/**********************/
/***  Paired t-tests  **/
/**********************/
transpose data=first out=trans;
   id cuertype;
   by subject;
   var time correct;
run;

proc sort; by _name_; run;
data two;
   set trans;
      a_v=aural-visual;
      a_av=aural-av;
      v_av=visual-av;
run;

title1 'Paired t-test';
proc means n mean stderr t prt;
   by _name_;
   var a_v a_av v_av;
run;
C.3 Results of SAS Program

### Nested Factorial

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Sum of Squares</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>23</td>
<td>7.55507032</td>
<td>4.49</td>
<td>0.0001</td>
</tr>
<tr>
<td>Error</td>
<td>30</td>
<td>2.19436000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corrected Total</td>
<td>53</td>
<td>9.74933632</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-Square C.V. CORRECT Mean

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Type I SS</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP</td>
<td>2</td>
<td>0.01600772</td>
<td>0.11</td>
<td>0.8967</td>
</tr>
<tr>
<td>SUBJECT(GROUP)</td>
<td>15</td>
<td>0.8245916</td>
<td>0.81</td>
<td>0.6566</td>
</tr>
<tr>
<td>CUETYPE</td>
<td>2</td>
<td>0.03976365</td>
<td>41.29</td>
<td>0.0001</td>
</tr>
<tr>
<td>GROUP*CUETYPE</td>
<td>4</td>
<td>0.60783380</td>
<td>2.06</td>
<td>0.1086</td>
</tr>
</tbody>
</table>

### Tests of Hypotheses using the Type III MS for SUBJECT(GROUP) as an error term

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Type III SS</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP</td>
<td>2</td>
<td>0.01600772</td>
<td>0.13</td>
<td>0.8750</td>
</tr>
</tbody>
</table>
Tests of Hypotheses using the Type III MS for
GROUP*CUETYPE as an error term

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Type III SS</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUETYPE</td>
<td>2</td>
<td>6.03976365</td>
<td>19.87</td>
<td>0.0094</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nest Factor 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>% Correct 16:07 Saturday, May 6, 1995</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shapiro-Wilk Test of Normality

Univariate Procedure

Variable=RISID

Moments

<table>
<thead>
<tr>
<th></th>
<th>54</th>
<th>Sum Wgt.s</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0</td>
<td>Sum</td>
<td>0</td>
</tr>
<tr>
<td>Std Dev</td>
<td>0.203473</td>
<td>Variance</td>
<td>0.041401</td>
</tr>
<tr>
<td>Skewness</td>
<td>0.015648</td>
<td>Kurtosis</td>
<td>-0.28624</td>
</tr>
<tr>
<td>USS</td>
<td>2.194066</td>
<td>CSS</td>
<td>2.194066</td>
</tr>
<tr>
<td>CV</td>
<td>.</td>
<td>Std Mean</td>
<td>0.027689</td>
</tr>
<tr>
<td>T:Mean=0</td>
<td>0</td>
<td>Pr&gt;</td>
<td>T</td>
</tr>
<tr>
<td>Num = 0</td>
<td>54</td>
<td>Num &gt; 0</td>
<td>26</td>
</tr>
<tr>
<td>M(Sign)</td>
<td>~1</td>
<td>Pr&gt;</td>
<td></td>
</tr>
<tr>
<td>Sgn Rank</td>
<td>16.5</td>
<td>Pr&gt;</td>
<td></td>
</tr>
<tr>
<td>W:Normal</td>
<td>0.979962</td>
<td>Pr&gt;</td>
<td>W</td>
</tr>
</tbody>
</table>

Nested Factorial 8

Time 16:07 Saturday, May 6, 1995

General Linear Models Procedure

Dependent Variable: TIME

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Sum of Squares</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>23</td>
<td>47.592505959</td>
<td>6.90</td>
<td>0.0001</td>
</tr>
<tr>
<td>Error</td>
<td>30</td>
<td>9.000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corrected Total</td>
<td>53</td>
<td>56.592505959</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>DF</td>
<td>Type I SS</td>
<td>F Value</td>
<td>Pr &gt; F</td>
</tr>
<tr>
<td>-----------------</td>
<td>----</td>
<td>-----------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>GROUP</td>
<td>2</td>
<td>1.92592593</td>
<td>3.21</td>
<td>0.0545</td>
</tr>
<tr>
<td>SUBJECT(GROUP)</td>
<td>15</td>
<td>29.333333333</td>
<td>6.52</td>
<td>0.0001</td>
</tr>
<tr>
<td>CUE_TYPE</td>
<td>2</td>
<td>14.92592593</td>
<td>24.88</td>
<td>0.0001</td>
</tr>
<tr>
<td>GROUP*CUE_TYPE</td>
<td>4</td>
<td>1.40740741</td>
<td>1.17</td>
<td>0.3425</td>
</tr>
</tbody>
</table>

Tests of Hypotheses using the Type III SS for

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Type III SS</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBJECT(GROUP)</td>
<td>2</td>
<td>1.92592593</td>
<td>3.21</td>
<td>0.0545</td>
</tr>
<tr>
<td>GROUP</td>
<td>2</td>
<td>1.92592593</td>
<td>0.49</td>
<td>0.6307</td>
</tr>
</tbody>
</table>

Tests of Hypotheses using the Type III SS for

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Type III SS</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUE_TYPE</td>
<td>2</td>
<td>14.92592593</td>
<td>21.21</td>
<td>0.0074</td>
</tr>
</tbody>
</table>

Shapiro-Wilk Test of Normality

Univariate Procedure

Variable=RESID

Moments
The content of the page includes statistical analyses and paired t-tests with variables named `A_V`, `A_AV`, and `V_AV`. The results are presented in two tables, one for each variable, showing the number of observations (`N`), mean, standard error, t-statistic, and p-value for the paired t-test.

### Variable A_V

| Variable | N | Mean  | Std Error | T   | Prob>|T| |
|----------|---|-------|-----------|-----|------|
| A_V      | 18| -0.2444444 | 0.0496818 | -4.8906651 | 0.0001 |

### Variable A_AV

| Variable | N | Mean  | Std Error | T   | Prob>|T| |
|----------|---|-------|-----------|-----|------|
| A_AV     | 18| 0.3555556 | 0.0444444 | 6.0000000 | 0.0001 |

### Variable V_AV

| Variable | N | Mean  | Std Error | T   | Prob>|T| |
|----------|---|-------|-----------|-----|------|
| V_AV     | 18| 0.1111111 | 0.0369458 | -3.0070383 | 0.0079 |

The tests were conducted on May 6, 1995, at 16:07.
Figure C.1: Nested Factorial Q-Q Plot for correctness to test if residuals are normally distributed. A Shapiro-Wilk test confirms the normal distribution.
Figure C.2: Nested Factorial Q-Q Plot for time\textsuperscript{taken} to test if residuals are normally distributed. A Shapiro-Wilk test confirms the normal distribution.