```
        octave = octave;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,octave,25,60);
        }

}
```

```
_l_m()
{
        i = i;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,i,25,6);
        }

        second = second;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,second,25,14);
        }

        third = third;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,third,25,21);
        }

        fourth = fourth;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,fourth,25,29);
        }

        fifth = fifth;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,fifth,25,36);
        }

        sixth = sixth;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,sixth,25,43);
        }

        seventh = seventh;
        if (_dtrack_events[1]) {
        _lsl_heartbeat_nonote();
        _lsl_play_3(1,seventh,25,52);
        }
```

D.4.1 The LSL Specification

```
begin auralspec
specmodule test
begin test

        syncto mmabs q = 220;

        dtrack i and second and third and fourth and fifth and sixth
                 and seventh and octave and rest
                when  rule = while_statement_enter
                until rule = while_statement_exit
                using Wbb_snd;


end test;
end auralspec.
```

D.4.2 The Decorated Source File

```
int _lsl_events[1024];
int _dtrack_events[1024];
_lsl_initialize()
{
        setTempo(220);
}
_lsl_exit(i)
int i;
{
        midiExit();
        _exit(i);
}
int i = (60);
int rest = 0;
int second = (60) + 2;
int third = (60) + 4;
int fourth = (60) + 5;
int fifth = (60) + 7;
int sixth = (60) + 9;
int seventh = (60) + 11;
int octave = (60) + 12;
```

```
        _lsl_exit(1);

}
```

D.3 The Melody Maker

The melody maker defines directives which when tracked by LSL will create the corresponding note of the scale. This section contains the source code, the specification, and the instrumented source code.

D.4 The Source Code

```
#define LOW_VAL         (60)

#define ROOT    i = i
#define SECOND  second = second
#define THIRD   third = third
#define FOURTH  fourth = fourth
#define FIFTH   fifth = fifth
#define SIXTH   sixth = sixth
#define SEVENTH seventh = seventh
#define OCTAVE  octave = octave
#define REST    rest = rest

int i = LOW_VAL;
int rest = 0;
int second = LOW_VAL+2;
int third = LOW_VAL+4;
int fourth = LOW_VAL+5;
int fifth = LOW_VAL+7;
int sixth = LOW_VAL+9;
int seventh = LOW_VAL+11;
int octave  = LOW_VAL+12;

main()
{
        ROOT; SECOND; THIRD; FOURTH; FIFTH; SIXTH; SEVENTH; OCTAVE;
}
```

```
        int answer;

        printf("Your guess: ");
        scanf("%d",&answer);
        fgetc((&_iob[0]));
        return answer;

}

void
finale()
{
        fgetc((&_iob[0]));
        _lsl_events[2] = 1;
        if ( _lsl_events[2] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_2(38,0,75,18);
        }
        _lsl_events[2] = 0;

}

void
generate_key()
{
        time_t tloc;

        srand((int)time(&tloc));
        key = rand() % 200;

}

void
you_lose()
{
        _lsl_events[6] = 1;
        if ( _lsl_events[6] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_1(58,90,9);
        }
        _lsl_events[6] = 0;

        printf("Guess out of range. better luck next time\n");
```

```
        else
                _lsl_events[3] = 1;
        if (( quit != 1  ) )
                _lsl_events[1] = 0;
        else
                _lsl_events[1] = 1;
        if ( _lsl_events[3] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_1(20,45,17);
        }
        if ( _lsl_events[1] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_2(38,1,45,17);
        }

        printf("\n\nYou win\n Press return to stop applause.\n");
        finale();

        _lsl_events[5] = 1;
        if ( _lsl_events[5] )
                _dtrack_events[1] = 0;
        else
                _dtrack_events[1] = 1;
        _lsl_events[5] = 0;
}

void
welcome()
{
        system("clear");
        printf("Here's a guessing game for you...\n\n");
}

void
show_menu()
{
        printf("Guess a number between %d and %d\n",1,200);
        printf("To quit: guess a number outside this range.\n");
}

int
read_answer()
{
```

```
void generate_key();
static int key = 0;

void
_l_m()
{
        int guess;
        int quit = 0;
        int diff = 0;
        int musical_diff = 0;

        _lsl_events[4] = 1;
        if ( _lsl_events[4] )
                _dtrack_events[1] = 1;
        else _dtrack_events[1] = 0;
        _lsl_events[4] = 0;

        welcome();
        generate_key();

        while(1) {
                show_menu();
                guess = read_answer();

                if(guess < 1 || guess > 200) {
                        you_lose();
                }
                else {
                        diff = abs(key - guess);
                        if(diff == 0) {
                                break;
                        }
                        musical_diff = 90 - diff;
                        if (_dtrack_events[1]) {
                                _lsl_heartbeat_nonote();
                                _lsl_play_3(17,musical_diff,41,59);
                        }
                }
        }

        quit = 1;
        if (( quit != 1  ) )
                _lsl_events[3] = 0;
```

```
        syncto mm q=60;


        atrack when assertion = (quit != 1)
                until rule = function_return:"finale"
                using Applause_snd;

        notify assertion = (quit != 1) using Phone_snd;

        dtrack musical_diff when rule = function_entry:"main"
                until rule = function_return:"main"
                using Flute2_snd;

        notify rule = function_entry:"you_lose" using Bird_snd;

end temp;
end auralspec.
```

D.2.3 The Decorated Source File

```
int _lsl_events[1024];
int _dtrack_events[1024];

_lsl_initialize()
{
        setTempo(60);
}

_lsl_exit(i)
int i;
{
        midiExit();
        _exit(i);
}

void _l_m();
void welcome();
void show_menu();
int read_answer();
void finale();
void you_lose();
```

```
LOW_NUM, HIGH_NUM);
        printf("To quit: guess a number outside this range.\n");
}

int read_answer()
{
        int answer;

        printf("Your guess: ");
        scanf("%d", &answer);
        getchar();
        return answer;
}

void finale()
{
        getchar();
}

void generate_key()
{
        time_t tloc;

        srand((int)time(&tloc));
        key = rand()% HIGH_NUM;

}

void you_lose()
{

        printf("Guess out of range. better luck next time\n");
        exit(1);
}
```

D.2.2 LSL Specification

```
begin auralspec
specmodule temp
begin temp
```

```
void generate_key();

static int key=0;

void main()
{
        int guess;
        int quit=0;
        int diff=0;
        int musical_diff=0;

        welcome();
        generate_key();

        while (1) {
                show_menu();
                guess = read_answer();
                if (guess < LOW_NUM || guess > HIGH_NUM) {
                        you_lose();
                }
                else {
                        diff = abs(key - guess);
                        if (diff == 0) {
                                break;
                        }
                        musical_diff = HIGHEST_NOTE - diff;
                }
        }

        quit = 1;
        printf("\n\nYou win\n Press return to stop applause.\n");
        finale();
}

void welcome()
{
        system("clear");
        printf("Here's a guessing game for you...\n\n");
}

void show_menu()
{
        printf("Guess a number between %d and %d\n",
```

```
                _lsl_events[4] = 1;
                if ( _lsl_events[4] ) {
                        _lsl_heartbeat_nonote();
                        _lsl_play_1(2,9,12);
                }
                _lsl_events[4] = 0;
        }

        _lsl_events[2] = 1;
        if ( _lsl_events[2] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_1(6,10,2);
        }
        _lsl_events[2] = 0;
}
```

D.2 The Guessing Game

The guessing game prompts the user to guess a number within a certain range. The user feedback will consist of aural queues. Below are listed the source file, the specification, and the instrumented source file.

D.2.1 Original Source File

```
#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/time.h>

#define LOW_NUM         1
#define HIGH_NUM        200
#define HIGHEST_NOTE    90

void main();
void welcome();
void show_menu();
int read_answer();
void finale();
void you_lose();
```

```
end temp;
end auralspec.
```

D.1.3 The Instrumented Source File

```
int _lsl_events[1024];
int _dtrack_events[1024];

_lsl_initialize()
{
        setTempo(120);
}
_lsl_exit(i)
int i;
{
        midiExit();
        _exit(i);
}
_l_m()
{
        int i;

        i = 0;

        _lsl_events[1] = 1;
        if ( _lsl_events[1] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_1(5,7,2);
        }
        _lsl_events[1] = 0;

        while(i < 10) {
                _lsl_events[3] = 1;
                if ( _lsl_events[3] ) {
                        _lsl_heartbeat_nonote();
                        _lsl_play_1(1,8,3);
                }
                _lsl_events[3] = 0;

                printf("The value of i = %d\n",i);
                i = i + 1;
```

Appendix D: Sample Auralizations

Given in this appendix are the source files, the specification, and instrumented source files for various programs discussed in the thesis. The instrumented files have been reformatted to improve the readability. The instrumented source is not formatted for human reading. Here the include file expansions have been deleted and the instrumented code has been tabbed appropriately. Otherwise the content of the instrumented files is exactly as they were created by using lslCC.

D.1 The While Loop

D.1.1 Original Source File

```
main()
{
        int i;

        i = 0;

        while (i < 10) {
                printf("The value of i = %d\n",i);
                i = i + 1;
        }

}
```

D.1.2 LSL Specification

```
begin auralspec
specmodule temp
begin temp

        syncto mmabs q = 120;

        notify rule = while_statement_enter using Wse_snd;
        notify rule = while_statement_exit using Wsx_snd;
        notify rule = while_body_begin using Wbb_snd;
        notify rule = while_body_end using Wbe_snd;
```

```
/**
***     These structures are just for compilation purposes.
***     They will need to be completed as the time arises.
**/

struct global {
  int i;
};

struct play {
  int i;
};

struct assign {
  int i;
};

struct loop {
  int i;
};

struct if_command {
  int i;
};

struct turn {
  int i;
};

struct toggle {
  int i;
};
```

```
};

struct pars {
  int             key;  /* keywords to determine type of param */
  struct tag      *RHS; /* right hand side of the expression   */
};

struct  tag {
  char            *name;  /* id name of tag                    */
  int             val;    /* constant                          */
};


/******
******* Synchronization Related Data Structures
******/

struct mmspec   {
  int mode;                       /* absolute of relative         */
  struct mmvalue *mmvalue;        /* the metronome value          */
};

struct mmvalue {
  int duration;                   /* what note value gets tick    */
  int bpm;                        /* beats per minute             */
};


/******
******* Scope Related Data Structures
******/

struct filelist {
  char            *filename;      /* string of the instance list */
  struct filelist *next;          /* pointer to next entry       */
};

struct functionlist {
  char            *functionname;  /* string of the instance list */
  struct functionlist     *next;  /* pointer to next entry       */
};
```

```
struct instance {
  char              *string;        /* string of the instance list */
  struct instance   *next;          /* pointer to next entry       */
};

/**
***             Expression (from the rtime event)
**/

struct expression {
  int                unop;   /* TRUE or FALSE for-operator   */
  char               *name;  /* id name of var in expression */
  int                value;  /* const value                  */
  int                paren;  /* TRUE or FALSE for() id       */
  struct expression  *apl;   /* another parameter list       */
};

/**
***             assertion (from the assertion event)
**/

struct assertion {
  struct variable *varlist;       /* List of variables in the assr*/
  char *cond;                     /* the text rep of assertion    */
};

struct variable {
  char            *string;        /* string of the instance list */
  struct variable *next;          /* pointer to next entry       */
};

/******
******* Sound related data structures
******/

struct playlist {
  int                conj;   /* && / ||                           */
  int                code;   /* type of sound code                */
  struct expression *data;   /* all the forms data can come in    */
  struct soundspec  *PSP;    /* another soundspec in parenthesis */
  struct playlist   *next;   /* pointer to next list record       */
  struct pars       *with;   /* rtime with statement              */
  struct Event      *thread;/* thread back to event              */
```

```
};

/**
***     syncto parameter list
**/

struct syncparlist {
  int type;                         /* the type of syn par  */
  int bufsize;                      /* playback buffer size */
  int noslow;                       /* playback mode        */
  struct mmspec *mmspec;            /* mmspec               */
  struct syncparlist *next;         /* next sync par in list*/

};

/******
******* Event Related Data Structures
******/

struct eventkey {
  char                *label;         /* label                          */
  char                *trip_name;     /* if event a trip, this is name*/
  struct instance     *instance_list; /* strings in an instance list  */
  struct expression *expression;      /* rtime expression               */
  struct assertion    *assertion;     /* assertion event                */
  struct eventkey     *next;          /* next string in eventkey list */
};

struct Elink {
  int                 number;         /* event number                   */
  int                 code;           /* code for type of notify        */
  struct eventkey     *data;          /* equation data of the event     */
  struct Elink        *next;          /* pointer to next event record */
  struct Elink        *all_next;      /* next event master event list */
  struct Elink        *all_prev;      /* previous event master e list */
  char                *name;          /* the name of the event          */
};


/**
***             Instance List (from SSE function event)
**/
```

```
/******
******* Specification Command Attributes
******/

/**
***     Event Specifier
**/

struct Event {
  int                    count;  /* number of conjuncts */
  struct Elink           *link;  /* event in the occurrence list */
  struct Event           *next;  /* next event spec in ListOfEvents*/
};

/**
***     Scope Specifier
**/

struct scopespec {
  struct filelist         *files;/* linked list of file names    */
  struct functionlist  *functions;/* link list of function names  */
};

/**
***     Sound Specifier
**/

struct soundspec{
  int                    count;  /* number of sounds to auralize */
  struct playlist        *play;  /* pointer to playlist info     */
  struct soundspec       *next;  /* next in ListOfSounds         */
};

/**
***     Dtrack ID list
**/

struct didlist {
  char                   *variable;  /* dtrack identifier      */
  struct expression      *initval;   /* init expression        */
  char                   *capture;   /* id                     */
  struct scopespec       *scope;     /* func or filename scopes */
  struct didlist         *next;      /* next identifier        */
```

```
struct dtrack {
  int                     count;          /* the number of dtrack */
  struct didlist          *didlist;       /* dtrack id list       */
  struct Event            *start;         /* event list start     */
  struct scopespec        *start_scope;   /* scope for start event*/
  char                    *start_condition;/* the event condition */
  struct Event            *term;          /* event list term      */
  char                    *term_condition;/* the event condition  */
  struct scopespec        *term_scope;    /* scope for term event */
  struct soundspec        *sound;         /* sound spec list      */
  struct scopespec        *scope;         /* func or file scope   */
  struct dtrack           *next;          /* pointer next dtrack  */
};


/**
***     atrack
**/

struct atrack {
  struct Event            *start;         /* event list start     */
  char                    *start_condition;/* the event condition */
  struct scopespec        *start_scope;   /* scope for start event*/
  struct Event            *term;          /* event list term      */
  char                    *term_condition;/* the event condition  */
  struct scopespec        *term_scope;    /* scope for term event */
  struct soundspec        *sound;         /* sound to play        */
  struct scopespec        *scope;         /* func or file scope   */
  struct atrack           *next;          /* pointer next atrack  */
};


/**
***     syncto
**/

struct sync{
  int syncMode;                   /* syncto program or mm */
  struct syncparlist *syncParList;/* par list if sync mm  */
};
```

```
  struct toggle           *toggle;
  struct sync             * sync;
  struct classlist        * classlist;
  char                    * name;
  struct spec             * next;
};

/**
***     Specification Class Related Data Structures
**/

struct classlist {
  struct class *class;
  struct classlist *next;
};

struct class {
  char *name;
  struct speclist *speclist;
  struct class *next;
};

/******
******* Specification Commands
******/

/**
***     notify
**/

struct notify {
  int                     type;       /* all / selective      */
  char                    *label;     /* labels               */
  struct Event            *event;     /* event list           */
  struct soundspec        *sound;     /* sound spec list      */
  struct scopespec        *scope;     /* func or file scopes  */
  char                    *condition; /* the event condition  */
  struct notify           *next;      /* pointer next notify  */
};

/**
***     dtrack
**/
```

Appendix C: The Specification Database

C.1 Data Structures

```
/*******************************************************
*
*
*       Project         :       Project Listen
*       File            :       Occur.h
*       Date            :       January 12, 1994
*
*       Description     :
*
*       This file contains the data structure definition
*       for the lsl specification data base.
*
*
*
*
*******************************************************/

/******
******* The Specification List Data Structure
******/

struct speclist {
  struct spec *spec;
  struct speclist *next;
};


struct spec {
  int type;
  struct global         * global;
  struct play           * play;
  struct notify         * notify;
  struct dtrack         * dtrack;
  struct atrack         * atrack;
  struct assign         * assign;
  struct loop           * loop;
  struct if_command     * if_command;
  struct turn           *turn;
```

```
boardman> make public

creating /homes/boardman/lslpub
creating /homes/boardman/lslpub/lib
/homes/boardman/lslpub/lib/midilib.a created
/homes/boardman/lslpub/lib/spec_db.a created
/homes/boardman/lslpub/lib/lsl.proteus.snds created
/homes/boardman/lslpub/lib/lsl.roland.snds created
creating /homes/boardman/lslpub/include
/homes/boardman/lslpub/include/midi.h created
/homes/boardman/lslpub/include/database.h created
/homes/boardman/lslpub/include/lsl_values.h created
/homes/boardman/lslpub/include/spec_db.h created
/homes/boardman/lslpub/include/Occur.h created
/homes/boardman/lslpub/include/spec_spec.h created
/homes/boardman/lslpub/include/spec_notify.h created
/homes/boardman/lslpub/include/spec_atrack.h created
/homes/boardman/lslpub/include/spec_dtrack.h created
/homes/boardman/lslpub/include/spec_event.h created
/homes/boardman/lslpub/include/spec_class.h created
/homes/boardman/lslpub/include/spec_sound.h created
/homes/boardman/lslpub/include/spec_scope.h created
/homes/boardman/lslpub/include/spec_sync.h created
/homes/boardman/lslpub/include/spec_lib.h created
creating /homes/boardman/lslpub/bin
/homes/boardman/lslpub/bin/lslCC created
/homes/boardman/lslpub/bin/lsl created
/homes/boardman/lslpub/bin/lsl_cpp created
/homes/boardman/lslpub/bin/listen created
/homes/boardman/lslpub/bin/panic created

boardman> make private
Deleted /homes/boardman/lslpub
```

Figure B.4 Results of Executing `make` public and `make` private Commands

```
boardman> make install

/homes/boardman/lsl/software/version1/tools/lsl installed
/homes/boardman/lsl/software/version1/tools/lsl_cpp installed
/homes/boardman/lsl/software/version1/lib/midilib.a installed
/homes/boardman/lsl/software/version1/lib/spec_db.a installed
/homes/boardman/lsl/software/version1/tools/listen installed

boardman> make uninstall

listen software uninstalled
```

Figure B.3 Results of Executing `make` install and `make` uninstall

```
#
# Before building modify the following lines as appropriate.
#

LSLDIR=/homes/boardman/lsl/software/version1

TMP=/usr/tmp

PUBDIR=/homes/boardman/lslpub
PUBLIB=${PUBDIR}/lib
PUBINCLUDE=${PUBDIR}/include
PUBBIN=${PUBDIR}/bin

SPECDIR=${INSTALLDIR}/lsl_spec
MIDIDIR=${INSTALLDIR}/lsl_midi
INSTRUMENTDIR=${INSTALLDIR}/lsl_inst
TOOLDIR=${INSTALLDIR}/tools
SHAREDIR=${INSTALLDIR}/lsl_share
CPPDIR=${INSTALLDIR}/lsl_cpp
LIBDIR=${INSTALLDIR}/lib
```

Figure B.2 The Makefile Installation Variables

4in

Figure B.1 Directory Structure and Component location for the Listen Software

```
@cd ${SPECDIR}; $(MAKE) depend
@cd ${CPPDIR};  $(MAKE) depend
@cd ${SHAREDIR};$(MAKE) depend
@cd ${TOOLDIR}; $(MAKE) depend
@cd ${GUIDIR};  $(MAKE) depend
```

```
@cd ${MIDIDIR}; $(MAKE) checkin
@cd ${GUIDIR};  $(MAKE) checkin
@cd ${CPPDIR};  $(MAKE) checkin
@cd ${SPECDIR}; $(MAKE) checkin
@cd ${SHAREDIR};$(MAKE) checkin
@cd ${TOOLDIR}; $(MAKE) checkin
@cd ${LIBDIR};  $(MAKE) checkin
@ci -l ?akefile
```

- clean

  Removes all generated files from the LSLDIR directories. This is done by changing directories and issuing a local `make` clean.

```
clean:
                @cd ${INSTRUMENTDIR}; $(MAKE) clean
                @cd ${MIDIDIR}; $(MAKE) clean
                @cd ${CPPDIR};  $(MAKE) clean
                @cd ${SPECDIR}; $(MAKE) clean
                @cd ${SHAREDIR};$(MAKE) clean
                @cd ${TOOLDIR}; $(MAKE) clean
                @cd ${GUIDIR};  $(MAKE) clean
                @cd ${LIBDIR};  $(MAKE) clean
```

- depend

  Sets up the dependencies for the Listen environment makefiles. This is done by changing directories and issuing a local `make` depend.

```
depend:
                @cd ${INSTRUMENTDIR}; $(MAKE) depend
                @cd ${MIDIDIR}; $(MAKE) depend
```

in it's source. This change is done automatically upon a `make` public by the use of a sed script. The following script modifies the environment appropriately.

```
${PUBBIN}/lslCC : ${TOOLDIR}/lslCC ${PUBBIN}
  @sed '/^INSTALLDIR/,$$d' ${TOOLDIR}/lslCC > ${TMP}/lslCC
  @echo "INSTALLDIR=$(PUBDIR) #generated by makefile" >>${TMP}/lslCC
  @sed '1,/^INSTALLDIR/d' ${TOOLDIR}/lslCC >> ${TMP}/lslCC

  @sed '/^DIRNAME/,$$d' ${TMP}/lslCC > $@
  @echo "DIRNAME=/bin #generated by makefile" >> $@
  @sed '1,/^DIRNAME/d' ${TMP}/lslCC >> $@

---
changes   INSTALLDIR=/homes/boardman/lsl/software/version1
          DIRNAME=/tools

to        INSTALLDIR=/homes/boardman/lslpub #generated by makefile
          DIRNAME=/bin #generated by makefile
---
```

- installpublic

  Equivalent behavior as issuing a `make` install and `make` public. This is useful after making a change to Listen which needs to be propagated to the public.

- checkin

  Checks in the current version of all source files. This is done by changing directories and issuing a local `make` checkin.

```
checkin:
                @cd ${INSTRUMENTDIR}; $(MAKE) checkin
```

- PUBLIB, PUBINCLUDE, PUBBIN

  Where the public versions of Ľsten software can be found and accessed by the public. These are children of the LSLDIR.

The Makefile Commands

The following commands are available using the LSLDIR makefile:

- all

  Brings all directory executables up to date. Each of the subdirectory makefiles is executed to accomplish this task.

- install,uninstall

  Make install copies directory executables into their local testing directories such as lib and tool. If these happen to be libraries the command ranlib is executed which converts each archive to a form that can be linked more rapidly. The results of the make install and make uninstall commands are shown in Figure B.3.

- public,private

  Takes the currently installed software and copies it into the public directories. This concept of public was designed to promote concurrent development of the Ľsten environment. By issuing a `make` private command the public installed files are deleted. Results of the `make` public and `make` private commands are shown in Figure B.4.

  There are two technical aspects to the `make` public command that should be recognized. First, directory creation is completely automated. Once the Ľsten environment variables have been set up appropriately, issuing a `make` public command will create the public directories. Second, the lslCC script must be modified so that it uses the public executables and not the private ones defined

- **lib**

  Link libraries for external development. These include midi and the spec_db libraries. The lib directory also contains the sound definition files for Listen.

B.2 The Make Environment

Each directory in the LSLDIR contains a makefile with a minimum of two commands: clean and checkin. Make clean removes all generated files from the directory. Make checkin checks in all modified sources using RCS. These make files can be run locally or controlled by the Listen makefile located in the LSLDIR. The LSLDIR makefile is very flexible and operates as described below.

B.2.1 The Installation Environment

Variables are defined which make the installation of the Listen environment software very flexible. These variables, as they appear in the current version, are shown in Figure B.2. Below is a description of the variables and how they are used.

- TMP

  Where the temporary files used by the makefile should be manipulated.

- LSLDIR

  Where the source for the Listen environment is located.

- SPECDIR, MIDIDIR, INSTRUMENTDIR, TOOLDIR, SHAREDIR, CPPDIR, LIBDIR, GUIDIR

  Where the source versions of Listen software can be found and accessed by the Listen software developer. These are children of the LSLDIR.

- PUBDIR

  Where the Listen environment should be installed for public usage.

- **tools**

  The only source in tools is lslCC. The lslCC script controls the compilation process. After installation this directory also contains the most recent versions of the decoration utility(lsl) as well as the C preprocessor(lsl_cpp).

- **lsl_cpp**

  Source files related to the C preprocessor.

- **lsl_midi**

  Source files related to the sound database, the MIDI sequencer, and the panic command.

- **lsl_spec**

  Source files related to the lsl specification database. This includes the lsl parser and data structure generation routines. The specification header files are included in this directory.

The public software and executables are available in the PUBDIR. The directory structure for PUBDIR is given below as well as in Figure B.1.

```
bin/             include/         lib/
```

- **bin**

  Executables which make up Listen. These include lsl_cpp, lsl, panic, listen, and the lslCC script.

- **include**

  Header files that are necessary for external development to interface with the midilib and the spec_db lib. These include function prototypes as well as data structure definitions.

B.1 Directory Layout

Software for project Listen is located in a directory called the LSLDIR. It is broken down into logical components which promote flexible maintenance and modification of Listen software. The directory structure and significant components are given in Figure B.1. Below is a list of the directories and a description of their contents.

`Makefile`	`lib/`	`lsl_gui/`	`lsl_midi/`	`lsl_spec/`
`RCS/`	`lsl_cpp/`	`lsl_inst/`	`lsl_share/`	`tools/`

- **lib**

  The MIDI sound definition files. There is a sound definition file for each MIDI device. At this writing there are two such files: lsl.proteus.snds and lsl.roland.snds.

  After the make install has been performed the lib contains the libraries midilib.a and spec_db.a. These are the versions to which all Listen software should be linked.

- **lsl_inst**

  Sources related to the instrumentation process. The instrumentation process includes creating the C parse tree, decorating the parse tree, and recreating instrumented source code.

- **lsl_gui**

  Contains the source files related to the development of the graphical user interface to lsl.

- **lsl_share**

  Header files which are likely to be used by software other than decoration and specification routines. These files include filenames.h, lsl_values.h, and midi.h.

Appendix B: The Listen Development Environment

The Listen environment was developed using the *C programming language* on the *UNIX operating system.* The tools *bison* and *flex* were used to derive parsing and scanning routines. The gcc compiler was used for compilation. Tools were used which aid in the management of the Listen software development environment. These tools are definitely crucial to the development of the product because of the large size of the software. Table B.1 gives the code size figures for the Project Listen. Note that these figures are not to give an estimate of the work involved only the size of the project as it relates to the complexity of maintenance. Some of the code was reused or developed by other individuals. For example the lsl_cpp C preprocessor is used from the ATAC project. The lsl_gui was written by a fellow student, Geoff Greene. The line count was generate using the UNIX utility `wc`.

Table B.1 The Project Source Size

Makefiles	713
Scripts	212
Header Files	4353
Bison Files	1329
Flex Files	310
C Source	46266
Total	53183

The tools include compiler construction, revision control, and compilation dependency tools. A directory structure was derived which promotes development by multiple developers. The `make` commands were developed to ease the installation and maintenance of the Listen software.

This appendix describes how these tools were used and how the directory structure was derived to develop a flexible environment to promote development of Listen products.

116.	*scope-tagid-list*	$\rightarrow$	*scope-tagid-list*, *scope-tagid*
			| *scope-tagid*
118.	*scope-tagid*	$\rightarrow$	**string**
119.	*dtrack-command*	$\rightarrow$	`dtrack` *dtrack-id-list start-event-spec term-event-spec*
			sound-specifier scope-specifier;
120.	*atrack-command*	$\rightarrow$	`atrack` *start-event-spec term-event-spec*
			sound-specifier scope-specifier;
121.	*start-event-spec*	$\rightarrow$	`when` *event-specifier scope-specifier*;
			| ϵ
123.	*term-event-spec*	$\rightarrow$	`until` *event-specifier scope-specifier*;
			| ϵ
125.	*dtrack-id-list*	$\rightarrow$	*dtrack-id-list* `and` *dtrack-id*
			| *dtrack-id*
127.	*dtrack-id*	$\rightarrow$	**l-** `id` *scope-specifier*
128.	*sync-command*	$\rightarrow$	`syncto` *sync-to-id*;
129.	*sync-to-id*	$\rightarrow$	`program` | *sync-par-list*
131.	*sync-par-list*	$\rightarrow$	*sync-par-list*, *sync-par*
			| *sync-par*
133.	*sync-par*	$\rightarrow$	*mmkeyword* | *mmkeyword mmspec*
135.	*mmkeyword*	$\rightarrow$	`mm` | `mmabs` | `mmrel`
138.	*mmspec*	$\rightarrow$	| `q` = `const`
140.	*id-list*	$\rightarrow$	**id** , *id-list*
			| **id**

			| **id** ::
91.	*unnamed-command*	→	*notify-command*
			| *dtrack-command*
			| *atrack-command*
			| *sync-command*
95.	*notify-command*	→	`notify` *event-specifier*
			sound-specifier scope-specifier;
96.	*event-specifier*	→	*event-specifier connector event*
			| *event*
98.	*connector*	→	`and` | `or`
100.	*event*	→	`rule` = **id**
			| `rule` = **id**:*instance-list*
			| `assertion` = **l-** `condition`
			| (*event-specifier*)
104.	*instance-list*	→	*instance-list* && *instance*
			| *instance*
106.	*instance*	→	**string**
107.	*sound-specifier*	→	`using` **constant**
			| ϵ
109.	*scope-specifier*	→	`in` *tagged-scope-list*
			| ϵ
111.	*tagged-scope-list*	→	*tagged-scope-list* `and` *tagged-scope*
			| *tagged-scope*
113.	*tagged-scope*	→	*scope-tag* = *scope-tagid-list*
114.	*scope-tag*	→	`filename` | `func`

A.4 The Implemented Subset of the Grammar

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.

76.	*lsl-spec*	→	`begin auralspec`
			spec-module-list
			`end auralspec`.
77.	*spec-module-list*	→	*spec-module-list spec-module*
78.			| *spec-module*
80.	*spec-module*	→	`specmodule` **id**
			`begin` **id**
			spec-def-body
			`end` **id**;
81.	*spec-def-list*	→	*spec-def spec-def-list*
			| ϵ
83.	*spec-def*	→	`specdef`
			`begin` **id**
			spec-def-body
			`end` **id**;
84.	*spec-def-body*	→	*spec-command spec-def-body*
			| *spec-command*
86.	*spec-command*	→	*named-command*
			| *unnamed-command*
88.	*named-command*	→	*name-tag-list unnamed-command*
89.	*name-tag-list*	→	**id** :: *name-tag-list*

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 commands 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.

However, they do not alter the global values. Global values of such parameters may be set using the `set` command within an LSL specification or in the program.

3. Identifiers declared within a `specmodule` M are global to M and may be used by all `specdef`s declared within M. Identifiers declared within a `specdef` S are local to S and may not be used by other `specdef`s or in any other `specmodule`. Identifiers may be exported by an `specmodule` for use by any other module by explicitly mentioning it in an `export` declaration. A module may use an identifier exported by another module by explicitly importing it using the `import` declaration. All program variables used in an `specdef` or a `specmodule` body must be specified as `external`s. 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 `begin`s and `end`s must match in the identifiers that follow the corresponding keyword. Thus, for example, a `begin` gear which matches with an `end` change will be flagged as a warning because *gear* and *change* do not match.

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

8. The *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.

Table A.1 Language Dependent Terminals in LSL Grammar.

Terminal	**Meaning**	**Example from C**
`l-condition`	Conditional expression which evaluates to *true* or *false*.	$(x < y$ && $p > q)$
`l-id`	Identifier	*drag_icon*
`l-expression`	An expression that evaluates to a value of type matching the type of the left side of the assignment in which it appears.	$(min - val * 2)$
`l-function`	A function invoked for tracking one or more variables.	Any C function definition.
`l-actual-parameter-list`	List of actual parameters.	`int` x, `int` * y

7. Interpretation of a string is context dependent. Thus, for example, when assigned to a variable of type `pattern`, the string ".cmajor C5" denotes a sequence of notes consisting of the value of the variable .cmajor followed by the note C5. The same string when used in the context `file` = ".cmajor C5" denotes a file name .cmajor 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 "cbb". 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 auralized program must appear as `externals`.

2. Local attribute values, such as metronome values, channels, etc. which are specified explicitly as attributes, take precedence over corresponding global values.

			| `ptime` = *integer*
45.	*duration-expression*	→	*duration-expression op simple-duration*
			| *simple-duration*
			| (*duration-expression*)
48.	*op*	→	`+` | ϵ
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` *mmspec*

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.	*constant*	→	*integer*
			| *string*
			|*time-sig*
4.	*integer*	→	*digit*$^+$
5.	*string*	→	"*char-sequence*"
6.	*char-sequence*	→	*note-sequence*
			| *key-sig*
			| *file-name*
			| *function-name*
10.	*note-sequence*	→	(*note* | .*id*)$^+$
			| (*note-sequence*: *attribute-sequence*)
			| (*note-sequence*)
14.	*note*	→	*note-generic note-modifier*
15.	*note-generic*	→	`c` | `d` | `e` | `f` | `g` | `a` | `b` | `r` | `C` | `D` | `E` | `F` | `G` | `A` | `B` | `R`
31.	*note-modifier*	→	*flat-sharp** *octave*
32.	*flat-sharp*	→	`b` | `#`
34.	*octave*	→	[0-8]
35.	*attribute-sequence*	→	*attribute*$^+$
36.	*attribute*	→	*duration tagged-value-list**
37.	*duration*	→	*simple-duration*
			| (*duration-expression*)
39.	*simple-duration*	→	`f` | `h` | `q` | `e` | `s`

			$\mid$ *cfactor*
224.	*cfactor*	$\rightarrow$	*expression*
			$\mid$ (*condition*)
			$\mid$ `not` *cfactor*
227.	*addop*	$\rightarrow$	$+ \mid -$
229.	*mulop*	$\rightarrow$	$* \mid /$
231.	*unop*	$\rightarrow$	$-$
232.	*relop*	$\rightarrow$	$< \mid > \mid <= \mid = \mid >= \mid <>$
238.	*logop*	$\rightarrow$	$\&\& \mid \|$

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 (_) can be used in an identifier. Upper and lower case letters are treated as being different in an **id**.

$$\begin{array}{lcl} id & \rightarrow & (\ \_\ )^{*} letter\ (\ letter \mid digit \mid \_\ )^{*} \\ letter & \rightarrow & [\text{a-zA-Z}] \\ digit & \rightarrow & [\ \text{0-9}\ ] \end{array}$$

			$\mid \epsilon$
193.	*var-decl-list*	$\rightarrow$	*var-type-list* ; *var-decl-list*
			$\mid$ *var-type-list*
195.	*var-type-list*	$\rightarrow$	*id-list* : *type*
196.	*id-list*	$\rightarrow$	**id** , *id-list*
			$\mid$ **id**
198.	*type*	$\rightarrow$	`int` $\mid$ `note` $\mid$ `pattern` $\mid$ `voice` $\mid$ `file` $\mid$ `ksig` $\mid$ `tsig`
			$\mid$ *array-declarator*
206.	*array-declarator*	$\rightarrow$	`array` [*range-list*] `of` *type*
207.	*range-list*	$\rightarrow$	*range-list* , *range*
			$\mid$ *range*
209.	*range*	$\rightarrow$	*expression* `..` *expression*
210.	*expression*	$\rightarrow$	*expression addop term*
			$\mid$ *term*
212.	*term*	$\rightarrow$	*term mulop factor*
			$\mid$ *factor*
214.	*factor*	$\rightarrow$	(*expression*)
			$\mid$ *unop factor*
			$\mid$ **id**
			$\mid$ **id**(*actual-par-list*)
			$\mid$ **id**( )
			$\mid$ `const`
220.	*condition*	$\rightarrow$	*condition relop cterm*
			$\mid$ *cterm*
222.	*cterm*	$\rightarrow$	*cterm logop cfactor*

162.	*sync-par*	→	`bufsize` = `const`
			| `noslow`
			| *mmkeyword*
			| *mmkeyword mmspec*
166.	*mmkeyword*	→	`mm` | `mmabs` | `mmrel`
169.	*mmspec*	→	*duration-expression* = `const`
170.	*duration-expression*	→	*duration-expression duration-factor*
			| *duration-expression* + *duration-factor*
			| *duration-factor*
173.	*duration-factor*	→	*duration-attribute*
			| (*duration-expression*)
175.	*duration-attribute*	→	`f` | `h` | `q` | `e` | `s`
180.	*declarations*	→	*applicability const-declaration var-declaration*
181.	*applicability*	→	*apply-list*
			| ϵ
183.	*apply-list*	→	*apply-list*; *apply-decl*
			| *apply-decl*
185.	*apply-decl*	→	`applyto` *tagged-scope-list*;
186.	*const-declaration*	→	`const` *const-list*;
			| ϵ
188.	*const-list*	→	*const-val-pair const-list*
			| *const-val-pair*
190.	*const-val-pair*	→	**id** = **constant** ;
191.	*var-declaration*	→	`var` *var-decl-list*;

137.	*statement-body*	$\rightarrow$	`begin` *spec-def-body* `end`
			| *spec-command*;
139.	*if-command*	$\rightarrow$	*if-then-command*
			| *if-then-else-command*
141.	*if-then-command*	$\rightarrow$	`if` *condition* `then` *statement-body*
142.	*if-then-else-command*	$\rightarrow$	`if` *condition* `then` *statement-body* `else` *statement-body*
143.	*specdef-use-command*	$\rightarrow$	**id** (*actual-par-list*);
			| **id** ( );
145.	*actual-par-list*	$\rightarrow$	*actual-par-list*, *actual-par*
			| *actual-par*
147.	*actual-par*	$\rightarrow$	*expression*
148.	*spec-par-list*	$\rightarrow$	*id-list* | ϵ
150.	*VDAP-call-command*	$\rightarrow$	**l**- id (**l**- actual-parameter-list);
151.	*turn-command*	$\rightarrow$	`turn` *on-off* *device-tag-list*;
152.	*on-off*	$\rightarrow$	`on`
			| `off`
154.	*toggle-command*	$\rightarrow$	`toggle` *toggle-source* = **constant**;
155.	*toggle-source*	$\rightarrow$	`keyboard`
			| `midi`
157.	*sync-command*	$\rightarrow$	`syncto` *sync-to-id*;
158.	*sync-to-id*	$\rightarrow$	`program`
			| *sync-par-list*
160.	*sync-par-list*	$\rightarrow$	*sync-par-list*, *sync-par*
			| *sync-par*

111. *term-event-spec* $\rightarrow$ `until` *event-specifier*
scope-specifier

$\mid \epsilon$

113. *ext-id-list* $\rightarrow$ *ext-id-list* , **l-** `id`

$\mid$ **l-** `id`

115. *dtrack-id-list* $\rightarrow$ *dtrack-id-list* `and` *dtrack-id*

$\mid$ *dtrack-id*

117. *dtrack-id* $\rightarrow$ **l-** `id` *init-value capture-specifier scope-specifier*

118. *init-value* $\rightarrow$ `init` = **l-** *expression*

$\mid \epsilon$

120. *capture-specifier* $\rightarrow$ `capture` = **id**

$\mid \epsilon$

122. *mode-specifier* $\rightarrow$ `mode` = `continuous` $\mid$ `mode` = `discrete` $\mid$ `mode` = `sustain`

125. *assign-command* $\rightarrow$ *selector* := *expression*;

126. *selector* $\rightarrow$ **id** $\mid$ **id**[*element-selector*]

128. *element-selector* $\rightarrow$ expression-list

129. *expression-list* $\rightarrow$ *expression-list* , *expression*

$\mid$ *expression*

131. *loop-command* $\rightarrow$ *for-loop* $\mid$ *while-loop*

133. *for-loop* $\rightarrow$ `for` **id** := *expression* `to` *expression step-expression*
statement-body

134. *step-expression* $\rightarrow$ `step` *expression*

$\mid \epsilon$

136. *while-loop* $\rightarrow$ `while` *condition* `do` *statement-body*

			| `instance` = *instance-list*
			| `assertion` = **l-** `condition`
			| `rtime` = *expression* `after` *event*
			| (*event-specifier*)
			| *event* (`first`)
91.	*instance-list*	$\rightarrow$	*instance-list* && *instance*
			| *instance*
93.	*instance*	$\rightarrow$	**string**
94.	*sound-specifier*	$\rightarrow$	`using` *play-list*
			| ϵ
96.	*scope-specifier*	$\rightarrow$	`in` *tagged-scope-list*
			| ϵ
98.	*tagged-scope-list*	$\rightarrow$	*tagged-scope-list* `and` *tagged-scope*
			| *tagged-scope*
100.	*tagged-scope*	$\rightarrow$	*scope-tag* = *scope-tagid-list*
101.	*scope-tag*	$\rightarrow$	`filename` | `func`
103.	*scope-tagid-list*	$\rightarrow$	*scope-tagid-list*, *scope-tagid*
			| *scope-tagid*
105.	*scope-tagid*	$\rightarrow$	*selector* | **string**
107.	*dtrack-command*	$\rightarrow$	`dtrack` *dtrack-id-list start-event-spec term-event-spec*
			sound-specifier scope-specifier;
108.	*atrack-command*	$\rightarrow$	`atrack` *start-event-spec term-event-spec sound-specifier*

scope-specifier;

109.	*start-event-spec*	$\rightarrow$	`when` *event-specifier scope-specifier*
			| ϵ

59.	*score-const-id*	$\rightarrow$	*score-tag* = *const-id*
			| `mm` *mmspec*
			| *mode-specifier*
62.	*const-id*	$\rightarrow$	**constant**
			| *dotted-id*
64.	*dotted-id*	$\rightarrow$	**.id**
65.	*score-tag*	$\rightarrow$	`keysig`
			| `timesig`
67.	*device-tag-list*	$\rightarrow$	*device-const-id* , *device-tag-list*
			| *device-const-id*
69.	*device-const-id*	$\rightarrow$	*device-tag* = *const-id*
70.	*device-tag*	$\rightarrow$	`chan` | `inst`
72.	*notify-command*	$\rightarrow$	`notify` *all-selective label-parameter event-specifier sound-specifier scope-specifier*;
73.	*all-selective*	$\rightarrow$	`all` | `selective` | ϵ
76.	*label-parameter*	$\rightarrow$	`label` = *label-list*
			| ϵ
78.	*label-list*	$\rightarrow$	*label-list*, **id**
			| **id**
80.	*event-specifier*	$\rightarrow$	*event-specifier connector event*
			| *event*
82.	*connector*	$\rightarrow$	`and` | `or`
84.	*event*	$\rightarrow$	`rule` = **id**
			| `rule` = **id**:*instance-list*

			| *if-command*
			| *specdef-use-command*
			| *VDAP-call-command*
			| *turn-command*
			| *toggle-command*
			| *sync-command*
39.	*set-globals-command*	$\rightarrow$	`set` *global-par-list*;
40.	*global-par-list*	$\rightarrow$	*global-par-list*, *global-par*
			| *global-par*
42.	*global-par*	$\rightarrow$	*score-const-id*
			| *device-const-id*
44.	*play-command*	$\rightarrow$	`play` *play-list*;
45.	*play-list*	$\rightarrow$	*pattern-specifier* || *play-list*
			| *pattern-specifier* && *play-list*
			| *play-list*
48.	*pattern-specifier*	$\rightarrow$	**id**
			| **constant**
			| *specdef-use-command*
			| *VDAP-call-command*
			| *pattern-specifier play-pars*
			| (*play-list*)
54.	*play-pars*	$\rightarrow$	`with` *tagged-list*
55.	*tagged-list*	$\rightarrow$	*tagged-list*, *tags*
			| *tags*
57.	*tags*	$\rightarrow$	*score-const-id*
			| *device-const-id*

13.	*spec-def-list*	→	*spec-def spec-def-list*
			| ϵ
15.	*VDAP-list*	→	*VDAP-spec VDAP-list*
			| ϵ
17.	*VDAP-spec*	→	`VDAP begin`
			l- `function`
			`VDAP end`;
18.	*spec-def*	→	`specdef` **id** (*spec-par-list*)
			declarations
			`begin` **id**
			spec-def-body
			`end` **id**;
19.	*spec-def-body*	→	*spec-command spec-def-body*
			| *spec-command*
21.	*spec-command*	→	*named-command*
			| *unnamed-command*
23.	*named-command*	→	*name-tag-list unnamed-command*
24.	*name-tag-list*	→	**id** :: *name-tag-list*
			| **id** ::
26.	*unnamed-command*	→	*set-globals-command*
			| *play-command*
			| *notify-command*
			| *dtrack-command*
			| *atrack-command*
			| *assign-command*
			| *loop-command*

Appendix A: The LSL Language

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.	*lsl-spec*	$\rightarrow$	`begin auralspec`
			spec-module-list
			`end auralspec`.
2.	*spec-module-list*	$\rightarrow$	*spec-module-list spec-module*
3.			| *spec-module*
5.	*spec-module*	$\rightarrow$	`specmodule` **id**
			program-id-list
			global-interaction-list
			declarations
			spec-def-list
			VDAP-list
			`begin` **id**
			spec-def-body
			`end` **id**;
6.	*program-id-list*	$\rightarrow$	`external` *ext-id-list*;
			| ϵ
8.	*global-interaction-list*	$\rightarrow$	*global-interactions global-interaction-list*
			| ϵ
10.	*global-interactions*	$\rightarrow$	*interact-id id-list*;
11.	*interact-id*	$\rightarrow$	`import` | `export`

APPENDICES

[GJM91] C. Ghezzi, M. Jazayeri, and D Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[HL90] J. R. Horgan and S. A. London. Atac- automatic test analysis for C programs. internal memorandum TM-TSV-017980, Bell Communications Research, 1990.

[IEE94] IEE. Nonspeech audio at the interface. *Multimedia*, 1(1):33, Spring 1994.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Lan90] P. S. Langston. Little languages for music. *Computing Systems*, 3(2):193–282, Spring 1990.

[Set89] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[Tho90] T. Thompson. Keynote - a language and extensible graphic editor for music. *Computing Systems*, 3(2):331–358, Spring 1990.

BIBLIOGRAPHY

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[BH92] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, December 1992.

[Bla94] M. M. Blattner. In our image: Interface design in the 1990s. *IEEE Multimedia*, 1(1):25–36, 1994.

[BM93] D. B. Boardman and A. P. Mathur. Preliminary report on design rationale, syntax, and semantics of LSL: A specification language for program auralization. Technical Report SERC-TR-143-P, Software Engineering Research Center, Purdue University, W. Lafayette, IN, USA, 1993.

[BM294] *LSL: A SPECIFICATION LANGUAGE FOR PROGRAM AURALIZATION*, 1994.

[DBO93] ACM. *LogoMedia: A Sound Enhanced Programming Environment for Monitoring Program Behavior*, 1993.

[DJC93] *Proteum: Uma Ferramenta de Teste Baseada na Analise de MUtantes*, October 1993.

[DS88] S. Defuria and J. Scacciaferro. *MIDI Programming for the Macintosh*. M&T Publishing, Inc, Redwood City, CA, 1988.

[Edw89] A. D. N. Edwards. Soundtrack: An auditory interface for blind users. *Human-Computer Interaction*, 4(1):45–66, 1989.

[FJ92] J. M. Francioni and J. A. Jackson. Breaking the silence: Auralization of parallel program behavior. Technical Report TR 92-5-1, Computer Science Department, University of Southwestern Louisiana,, 1992.

[Gav86] W. W. Gaver. Using sound in computer interfaces. *Human-Computer Interaction*, 2:167–177, 1986.

[Gav89] W. W. Gaver. The sonicfinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4(1):67–94, 1989.

BIBLIOGRAPHY

may provide insight into program behavior. Based on initial experiences a developer may determine what part of the code is executing, how the program behaves on a given input, and narrow the search space when debugging.

and data. The ASPEC is parsed generating the auralization database which contains an internal representation of the ASPEC. A parse tree component is generated from the source C code during the Listen C parsing phase. Events from the auralization database are located in the parse tree and instrumentation code is inserted. Instrumented C source code is generated from the decorated parse tree by a process known as deparsing. The instrumented source code is compiled via a standard compiler and instrumented executable code is created. Running the instrumented code generates sound using MIDI (Musical Instrument Digital Interface) sound modules.

Several LSL specifications were created which provided initial insights into the application and the needs regarding Listen. Listen has been successfully integrated with an existing tool, **PROTEUM**, which is contributing to sound related research in software testing.

6.2 Conclusions

The development and application of Listen shows it is possible to produce a generic auralization tool. The following observations are a result of creating auralization specification with Listen.

1. The benefit of sound in a computing environment will be directly related to the quality of the ASPEC. One must choose sound patterns which portray appropriate information with respect to a given occurrence mapping.

2. With respect to debugging or analyzing behavior, it appears that a developer must spend significant time becoming familiar with the sound of a system or program. If this aural training period is not realized it appears a user gains less information from the sound. This suggests that there is a training time or sensitizing period that a developer must go through to gain the greatest benefit from sound in a computing environment.

3. It may be possible to establish a general ASPEC which creates a distinctive aural signature for a C program in an application domain. An aural signature

6. SUMMARY AND CONCLUSIONS

6.1 Summary

Project Listen was established in January 1994 to develop a program auralization environment applicable to sound in computing research. These research fields include, but are not restricted to, aural debugging, program auralization, auditory display, simulation, and sonification.

It appears that many current research efforts either provide specific tools for an experiment or require programmers to manually locate and instrument program events. This makes the experimentation process slow, tedious, and error prone. Listen separates sound specification from the source code and automatically locates and instruments a program events.

Listen Specification Language (LSL) is at the core of Listen. LSL is a general purpose mechanism to specify the auralization of programs. When the auralization specification and the program source are processed, the specification mappings are automatically located and instrumentation code is created to generate sound. LSL was designed considering two idealized requirements: generality and language independence. First, it should be possible to specify any auralization using LSL in terms of program data, position, and time. Second, it should be possible to use LSL with the commonly used programming languages such as C, C^{++}, Ada, Pascal, and Fortran.

To demonstrate that LSL meets the generality requirements, a minimal defined subset of LSL was defined and implemented for the C programming language. A summary of the implemented auralization process with respect to LSL/C follows.

Using any text editor, a programmer creates an auralization specification (ASPEC) using LSL. The ASPEC defines the mapping of specified sounds to program events. An event is located in the program occurrence space defined by time, position, graphic related information including additional windows on a display, points on a graph, or lines in space, could be displayed to the user.

5.4.4 A Teaching Tool

Sound may help in understanding algorithm behavior. Listen will be used in the upcoming year to experiment using sound in the teaching environment. A professor will create ASPECS for classic introductory algorithms. The auralizations will be used to an introduction to programming course. If the preliminary experience is positive this may provoke more formal experimentation with LSL.

5.4.5 Software Testing

The question has been posed by the developers of **PROTEUM** if sound can be used to aid in the task of classifying equivalent mutants. The equivalence problem is commonly known to be an undecidable problem. In this context a mutant is live and not know to be equivalent. If there is a significant difference between the mutant and original aural signature, it is proposed that this may suggest the mutant could be killed thus providing a classification.

Sound may also provide additional information in the data flow testing. Proposed tools for experimentation include ATAC and OTHERNAME. The tool source could be auralized in such a way that as a test case is executed, new coverage achieved by that test case generates sound. A test case designer could determine attributes of the test case using sound such as additional coverage obtained by the test case, at what point in time does the coverage occur, and the total running time to obtain the new coverage. This could be done visually but it is suggested the productivity of the test case designer may increase. A designer might make immediate decisions about the comparisons involving test case effectiveness.

5.4.6 Incorporating Graphic Mapping to Events

LSL is an event specification language. There is no inherent design that dictates sound must be the response to a given occurrence. LSL could be modified in such a way to support the graphic notification of events. Instead of sound being generated, To investigate these areas, a developer should create a general ASPEC that will apply throughout the life of a project. After significant development time, the developer should be interviewed regarding their experience with sound.

5.4.3 System Monitoring

The concept of a program aural signature can be extended to that of a system. In an environment requiring the monitoring of a complex system, sound may provide additional information not obvious visually or statistically.

A plant manager from a manufacturing facility gave an interesting analogy to the system monitoring problem. Often production line workers inform engineers that there is a problem with a complex system before the computers recognize the problem. The experienced workers have become subconsciously conditioned to the aural signature of the system. When the subconscious expectation regarding the aural signature of the system is not met, workers often have the ability to approximate the location of specific problems.

An interesting experiment would be to auralize certain daemon activities such as news, talk, and mail. An observer could listen to how these utilities are used. It is obvious that this work could be done using scripts and statistical feedback, however, monitoring could take place second hand while working on other projects. The observer might only pay attention during certain aurally busy times. The monitoring would not require their total attention because of the subconscious feedback. This is a toy example but may provoke future experimentation.

Another interesting example would be to auralize system routines that usually affect system performance. These routines may handle page faults, process swapping, disk seeks, or similar processes. Possibly as subjects develop expectations of the environment, they could detect and locate possible problem areas just as the production line worker.

5.3.4 Training Time

With respect to debugging or analyzing behavior, it appears that a developer must spend significant time becoming familiar with the sound of a system or program. If this aural training period is not realized it appears less information is gained from the sound. For example, a developer on Project Listen had been working with a given specification which contained a bug. Clearly when listening to the program execute the trained developer was having expectations met or broken. When another developer was asked to enter the debugging process they could make very little sense of the sound being produced because they had not developed the specification and the source.

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

5.4 Future Directions

5.4.1 Instrumenting Large Programs

The largest program that LSL has been applied to is the the UNIX sort routine. The number of source lines computed using the UNIX utility `wc` was approximately 921. It is desired to test LSL on programs exceeding 10,000 lines to determine the overhead of the instrumentation and the reality of using LSL in real world software development environments.

5.4.2 Program Signatures

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

5.3.1 Appropriate Sound Selection

The usefulness of sound in a computing environment will be directly related to the quality of the ASPEC. One must choose sound patterns which portray appropriate information with respect to a given the occurrence mapping. For example when instrumenting the body of a loop it is more appropriate to use short percussive sounds as they tend to be executed frequently. These percussive sounds seem less distracting than a recurring instrument sound. Data tracking is best associated to sounds that have distinct pitch. The relative difference between notes high in pitch is hard to distinguish.

5.3.2 Aural Expectations

When working with an auralized program one begins to generate expectations of the sound produced before program execution. Based on previous experience, one subconsciously forms an expectation of the generated sound before throwing an object into a garbage. The same appears to be true with respect to program auralization. As the source code is changed, an internal expectation of the sound to be produced is generated subconsciously. When the program is executed the expectations are either met or broken.

5.3.3 Code Replay

As experience is gained with the auralization, a mental pointer is envisioned traversing the code. After attending a musical, if a recording of the performance is played, one tends to visualize the performance associated with the sounds. Experience has yielded a common result when listening to auralized programs. As a program generates sound, a mental visualization of the code is generated.

5.2 Integration with Existing Tools

Listen has been integrated with **PROTEUM**, (PROG)ram for (TE)sting using (M)utation. **PROTEUM** [DJC93] is a mutation testing tool for C programs developed at ICMSC-SC of the University of São Paulo in Brazil. The authors have recently collaborated with Project Listen to determine if sound can be used by test case developers to aid in the recognition of equivalence mutants. The experiment is discussed in Section 5.4.5.

The integration of LSL and **PROTEUM** required no tool source code modifications. **PROTEUM** contains an option to specify the compiler and command line arguments to use during mutant compilation. An example usage appears below:

```
Compiler Preference:  lslCC
Compiler Options   :  proteum.lsl -o a.out
```

The test case developer has the ability to modify the **PROTEUM** ASPEC and experiment creating an effective ASPEC for classifying live or equivalent mutants. The collaborative work also hopes to auralize the actual **PROTEUM** source in order to provide additional test related feedback to the test case developer.

This is an example of how LSL is (a) general enough to promote experimentation of sound in computing environments and (b) separates the auralization process from the coding.

5.3 Lessons Learned

In developing LSL, many positive and negative aspects of using sound in computing environments have been experienced. This section discusses the major observations of Project Listen. These observations are likely to provoke future research with LSL.

A sample program and ASPEC were derived to play a musical scale. The code exploited the fact that the `dtrack` command generates sound when the value of a variable could potentially change. Therefor when the `dtrack` specification command is written to track the variable *second*, the occurrence of the source code *second* = *second* generates a sound. The complete program and instrumented program are given in Appendix D.

By creating the specification in figure 5.1.3 and initializing the variables in the source appropriately, notes of the major scale are generated by referencing SECOND, THIRD, and other preprocessor definitions. There is no immediate benefit observed however new ideas for future auralization applications may be generated.

```
begin auralspec
specmodule test
begin test

        syncto mmabs q = 220;

        dtrack i and second and third and fourth and fifth and sixth
                 and seventh and octave and rest
                 using Flute_snd;


end test;
end auralspec.
```

Figure 5.3 An LSL Specification for the Music Maker.

```
begin auralspec
specmodule temp
begin temp

        syncto mm q=60;


        atrack when assertion = (quit != 1)
                until rule = function_return:"finale"
                using Applause_snd;

        notify assertion = (quit != 1) using Phone_snd;

        dtrack musical_diff when rule = function_entry:"main"
                until rule = function_return:"main"
                using Flute2_snd;

        notify rule = function_entry:"you_lose" using Bird_snd;

end temp;
end auralspec.
```

Figure 5.2 The LSL Specification for the Guessing Game

value difference of the guess and the random number. The greater the result, the lower the pitch. As pitch increases the guess is closer to the generated random number.

The specification is given in Figure 5.2. The source code and the decorated source code are given in Appendix D. This example demonstrates all possible decorations as the defined by the minimal working subset of LSL specification commands given in Section 2.4.

5.1.3 Making Music

As the specifications were being tested it was found that constructing a program could generate a desired aural output. In other words, writing a program to generate melodies.

```
begin auralspec
specmodule temp
begin temp

        syncto mmabs q = 120;

        notify rule = while_statement_enter using Wse_snd;
        notify rule = while_statement_exit using Wsx_snd;
        notify rule = while_body_begin using Wbb_snd;
        notify rule = while_body_end using Wbe_snd;

end temp;
end auralspec.
```

Figure 5.1 An LSL Specification for the Simple While

5. APPLICATION AND EXPERIENCE WITH PROJECT LISTEN

Over the course of the project several LSL specifications were written and tested. This section describes complete sample specifications, lessons learned, and future directions for Project Listen.

5.1 Implementing Auralization Specifications

Experience was gained in the development and testing of LSL auralization specifications (ASPECS). The experience is discussed in Section 5.3. The sample auralizations are given in this section. A complete listing of the original source and instrumented source is given in Appendix D.

5.1.1 A Simple While Loop

Small ASPECS were initially used to test Listen. A simple test case is that of a while loop. This ASPEC, see Figure 5.1.1, demonstrates the basic functionality of LSL. The source code consists of a simple while loop that performs 10 iterations. The instrumented code generates a sound when the while loop is entered, at the beginning of the iteration body, at the end of the iteration body, and at the end of the while loop. It also serves as a test case for experimenting with different synchronization modes. A complete listing of the source and the instrumentation are given in Appendix D.

5.1.2 The Guessing Game

It was desired to construct a simple specification for demonstrating the complete set of LSL specification commands. The program chosen was the classic guessing game. The program generates a random number between a range and a guess is supplied as input. The program then generates a note depending on the absolute

```
main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{
        midiInit();
        initMidiSoundDatabase();
        initMidiPatches();

        _lsl_initialize();

        /**
        ***     Always call _l_m() with the standard arguments.
        ***     It is okay even if user's main() function
        ***     does not have any formal parameters.
        **/

        _l_m(argc, argv, envp);

        /**
        ***     Dummy loop to wait for midi device buffer
        ***     to be flushed.  Also resets midi before exiting.
        **/

        _lsl_exit(0);
}
```

Figure 4.19 The LSL Driver Routine

```
extern int _lsl_events[1024];
extern int _dtrack_events[1024];
```

Figure 4.17 Initialization Routines Generated by Deparse without Main

Figure 4.18 An Example `TNODE`

```
dtrack i                                 _lsl_events[1] = 1;
   when  rule = while_statement_enter  if ( _lsl_events[1] )
   until rule = while_statement_exit        _dtrack_events[1] = 1;
                                         else
                                              _dtrack_events[1] = 0;
                                         _lsl_events[1] = 0;

                                         while(i < 10) {
                                              ++i;
                                              if (_dtrack_events[1])
                                                   _lsl_play_3(10,i);
                                         }

                                         _lsl_events[2] = 1;
                                         if ( _lsl_events[2] )
                                              _dtrack_events[1] = 0;
                                         else
                                              _dtrack_events[1] = 1;
                                         _lsl_events[2] = 0;
```

Figure 4.15 A Dtrack Specification Command and Related Intrumentation Code

```
int _lsl_events[1024];
int _dtrack_events[1024];

_lsl_initialize()
{
        setTempo(80);
}

_lsl_exit(i)
int i;
{
        midiExit();
        _exit(i);
}
```

Figure 4.16 Initialization Code Generated by Deparse with Main

```
atrack
   when  rule = while_statement_enter
   until rule = while_statement_exit
   using Flute2_snd;
```

```
_lsl_events[0] = 1;
if (_lsl_events[0])
       _lsl_play_2(10,1);
_lsl_events[0] = 0;


while ( i < 10 ) {
        i = i + 1;
}

_lsl_events[1] = 1;
if (_lsl_events[1])
        _lsl_play_2(10,0);
_lsl_events[1] = 0;
```

Figure 4.14 An Atrack Specification Command and Related Intrumentation Code the main program. The user defined main routine (_l_m()) is called. The _lsl_exit() routine cleans up the MIDI environment and calls exit().

When a node is visited the before decoration data fields defined in the `TNODE` structure are written to the source file.[3] The children are recursively traversed writing related text to the file. When the node is visited after the recursion has bottomed out, decoration fields are written to the source file.

Consider the parse tree node given in Figure 4.18. Upon visiting the node the before_on_flags, the before_condition, and the before_off_flags are written to the file. If the text field were non null then the corresponding text would be written to the file. When tree traversal recursion bottoms out, the after_on_flags, the after_condition, and the after_off_flags are written to the file. For the `TNODE` given in Figure 4.18 the following code would be produced.

```
lsl\_events[1] = 1;
lsl\_events[2] = 1;
if (lsl_events[1]) lsl_play_1(23);
lsl_events[1] = 0;
```

4.5 Compilation

The LSL library routines are linked to the instrumented code. These provide access to the sound database, the runtime environment, and the MIDI routines. The _lsl_driver.o is also linked which initializes the environment, calls the instrumented source main program, and then waits for the MIDI environment to terminate.

4.6 Execution

When a program is executed the LSL driver program initializes the environment. The source for the driver routine is given in figure 4.19. The driver initializes the MIDI environment and calls the LSL initialize routine. The initialization routine is generated by the deparse phase and is located in the decorated source which contains

[3]For a description of the `TNODE` data fields refer to Section 3.3.1.

When a start or term event is located the appropriate code is inserted to turn the `dtrack` variable on or off. The variables in the `dtrack` identifier list are located using the assertion variable location algorithm described in Section 4.3.2.2. Auralization code is inserted to evaluate the event specification condition and play the corresponding sound if necessary. An example atrack specification and the generated instrumented code are given in Figure 4.15.

Decoration refers to instrumenting the parse tree with code that will generate sound. This sections describes preparing the parse tree for decoration, locating the event specifications, and the decorating of event specifications.

4.4 Deparsing

The deparsing phase reconstructs source code from the decorated parse tree. Deparsing constructs initialization code and reconstructs instrumented source code. This section details the deparsing process.

4.4.1 Constructing Initialization Code

If the parse tree for the source code contains the main program, data structures are defined, an initialization routine is generated, and the Listen specific exit routine is generated. Figure 4.16 gives example source code for a parse tree which contains a main program.

If the parse tree does not contain a main program the data structures are declared as extern. Figure 4.17 gives example constructed code that does not contain a main program.

4.4.2 Reconstructing Decorated Source Code

After initialization code has been output to the file for reconstruction, the decorated source code is generated by a recursive deparse routine which walks the C parse tree.

4.3.3.4 The `atrack` Specification Instrumentation

An activity is a sequence of actions which begins at the occurrence of an event and ends at the occurrence of a later event. Recall that LSL allows specification of tracking activities using the `atrack` command. The appropriate play code for the `atrack` is the _lsl_play_2(int *sound*, int *mode*) procedure call. The *sound* is either turned on or off depending on the value of *mode*. The *mode* is on if the event specification condition occurs as part of a start event. The *mode* is off if the event specification condition occurs as part of a terminating event. An example `atrack` specification and the generated instrumented code are given in Figure 4.14.

4.3.3.5 The `dtrack` Specification Instrumentation

Recall the `dtrack` allows data dependent auralizations. The tracking of the data occurs between the start event specifier and the end event specifier. The appropriate play code that corresponds to the `dtrack` is the _lsl_play_3(int *sound*, int *val*) procedure call.

The instrumentation of the `dtrack` introduces the `_dtrack_events` data structure which controls the tracking state depending on the starting and terminating event specification conditions. One `_dtrack_events` entry is allocated for each `dtrack` specification.

```
notify                                 _lsl_events[0] = 1;
   rule = while_statement_enter        if (_lsl_events[0])
   using Wse_snd;                              _lsl_play_1(23);
                                       _lsl_events[0] = 0;

                                       while ( i < 10 ) {
                                       i = i + 1;
                                       }
```

Figure 4.13 A Notify Specification Command and Related Intrumentation Code

The target statement for the while_body_begin is the printf located in the while loop body. The instrumentation corresponding to the specific syntactic entity includes:

1. The event is activated.
2. The specification event condition is evaluated and sound is generated appropriately.
3. The event is deactivated.

The target statement for the assertion is the assignment to i within the body of the loop. For assertions, the decoration always occurs after the target statement. An assertion event is active from the initial time the assertion is violated until the assertion becomes valid again.

The instrumentation corresponding to the assertion includes:

1. The assertion is evaluated activating or deactivating the event.
2. The event specification condition is evaluated and sound is generated appropriately.
3. Deactivating the event is accounted for in step 1.

4.3.3.3 The `notify` Specification Instrumentation

Recall that `notify` generates sound at the occurence of the event specifier. The appropriate play code is the _lsl_play_1(int sound) procedure call. The sound is played for a system defined duration and terminated. An example `notify` specification and the generated instrumented code are given in Figure 4.13. Note that the event specification condition always evaluates to true. An LSL optimizer could be written to remove this statement.

4.3.3.2 Decorating the Target Code Related to an Event

Once the event target code has been located, instrumentation takes place in a three step process.

1. The event is activated by setting the appropriate `_lsl_events` entry to ON(1).
2. The event specification condition is evaluated to determine sound generation. This step supports the boolean combination of events.
3. The event is deactivated by setting the appropriate `_lsl_events` entry to OFF(0).

The instrumentation related to the while_body_begin and assertion for a simple while loop is shown in Figure 4.12 [2].

```
while ( i < 10 ) {
        _lsl_events[0] = 1;
        if (_lsl_events[0] && _lsl_events[1])
                /** appropriate play code **/
        printf("In the body of the loop\n");
        _lsl_events[1] = 0;

        i = i + j;

        if (i < 5)
                _lsl_events[1] = 0;
        else
                _lsl_events[1] = 1;
        if (_lsl_events[0] && _lsl_events[1])
                /** appropriate play code **/
}
```

Figure 4.12 Instrumentation Related to the Event Specification Condition

[2] The appropriate play code is detailed in Section 4.3.3.2.

4.3.2.2 Locating the Assertion

An assertion is checked after the execution of a statement that can possible change the value of one of the assertion variables. Currently the following locations are decorated.

1. On the left hand side of an assignment.

2. An operand of the increment operator.

3. An operand of the decrement operator.

The statement that contains the expression variable becomes the target statement.

4.3.3 Decorating the Target Statement

Decoration refers to inserting instrumented source code to generate sound. Once a target statement has been located it must be decorated. This section describes the instrumented code associated with the decoration process. How instrumented code is generated is described Section 4.4.

4.3.3.1 The Event Specification Condition

Each event defined in an event specifier is assigned to an entry in the `_lsl_events` data structure. As an event specifier is parsed an event specification condition is built with the corresponding event data structure. For example, given the event specifier

```
rule = while_body_begin and assertion = i < 5
```

the specific syntactic entity while_body_begin corresponds to `_lsl_events`[0] and the assertion event corresponds to `_lsl_events`[1]. After parsing the event list, the event specification condition contains "(`_lsl_events`[0] && `_lsl_events`[1])". The event specifier and the specification condition are stored in the specification list within the spec database. The specification condition will be used by the decoration routines.

Figure 4.11 Parse Tree Target Statements Related to the Function Entities

```
foo()
{
        int i;

        /** function_entry sound **/
>>>     printf("Here in function foo\n");
>>>     printf("I printed this statement\n");
        /** function_exit_sound **/

}

main()
{
        int i;

        /** function_call sound **/
>>>     foo();

}
```

Figure 4.10 Source Code Target Statements Related to the Function Entities

5. function_call,function_entry,function_return

   The function related specific syntactic enties may include an instance list. This instance list consists of a list of function names to which the rule applies. If no instance list is supplied then the rule is applied to all functions within the given command scope. An example of the instance list is shown below.

```
rule = function_entry:"foo" using Fne_snd;
```

   For the function_call ,locate the statement node of type GEN_EXPR-EXPR_LFCALL1 or GEN_EXPR-EXPR_LFCALL0. If an instance list was provided check to see if the name of this function matches the one in question. The containing statement then becomes the target statement. Note that if two function calls exist on the same line the auralization will occur at the same time. For instance if the code reads:

```
i = foo() + bar();
```

   The statement given will be the target statement for both function_call:"foo" and "bar". This is the specified behavior for this implementation of LSL.

   For the function_entry and function_return the node of type GEN_FUNC_SPEC is located in the parse tree. The instance list is processed to determine if the function located is a candidate for auralization. If so the GEN_COMPSTMT within the function is located. The first statment in the STMT_LIST is the target for entry and the last statement is the target for exit.

   If flow control encounters a return statement of exit call during the execution of a procedure, the function_return may not be realized.

   Figure 4.10 shows the text position of the target statements. A parse tree with highlighted target nodes is given in Figure 4.11.

Figure 4.9 Parse Tree Target Statements Related to the If Statement Entities

4. `if_then_part`,`if_else_part`

   Locate the statement node of type GEN_STMT-STMT_IF or GEN_STMT-STMT_IF_ELSE. For `if_then_part` the statement that corresponds to the true evaluation becomes the target statement. If this is a COMP_STMT then the first statement in the statement list becomes the target.

   For the `if_else_part` there are two different target statements depending on the species. If the STMT_IF is processed then no corresponding else auralization is created in this implementation. If the STMT_IF_ELSE is encountered then the stmt that corresponds to the false evaluation becomes the target statement. If this is a COMP_STMT then the first statement in the statement list becomes the target.

   Figure 4.8 shows the text position of the target statements. A sample parse tree with highlighted target statements is given in Figure 4.9.

```
main()
{
        int i;

        i = 0;

        if (i == 0) {
                /** if_then_part instrumentation **/
>>              printf("The value of i = 0\n");
        }
        else {
                /** if_else_part instrumentation **/
>>              printf(The value of i != 0\n");
        }
}
```

Figure 4.8 Source Code Target Statements Related to the If Statement Entities

Figure 4.7 Parse Tree Target Statements Related to the For Loop Entities

```
main()
{
        int i;

        i = 0;

        /** for_loop_enter instrumentation **/
>>>     for (i = 0; i < 10; ++i) {
                /* for_body_begin instrumentation **/
>>>             printf("The value of i = %d\n",i);
                /* for_body_end instrumentation **/
        }
        /** for_loop_exit instrumentation **/
}
```

Figure 4.6 Source Code Target Statements Related to the For Loop Entities

3. for_statement_enter,for_statement_exit,for_body_begin,for_body_end

   Locate the statement node of type GEN_STMT-STMT_FOR_* where * matches any of the species values from the list:

   - STMT_FOR_EEES, STMT_FOR_EEE_, STMT_FOR_EE_S
   - STMT_FOR_EE__, STMT_FOR_E_ES, STMT_FOR_E_E_
   - STMT_FOR_E__S, STMT_FOR_E___, STMT_FOR__EES
   - STMT_FOR__EE_, STMT_FOR__E_S, STMT_FOR__E__
   - STMT_FOR___ES, STMT_FOR___E_, STMT_FOR____S

   The located node is the target statement for the for_statement_enter and the for_statement_exit. For the for_body_begin and for_body_end the enclosed statement is located. If GEN_STMT then the node is the target node in both instances. If GEN_STMT-COMP_STMT then the first statement in the statement list is the target statement for the for_body_begin and the last statement in the statement list is the target for the while_body_end.

   Figure 4.6 shows the text position of the target statements. A sample parse tree is given in Figure 4.7.

Figure 4.5 Parse Tree Target Statements Related to the While Entities

Locate the statement node of type GEN_STMT-STMT_WHILE. For the while_statment_enter, and the while_statement_exit the located node is the target statement. For the while_body_begin and while_body_end the enclosed statement is located. If the genus is GEN_STMT then the node is the target in both instances. If the node is of type GEN_STMT-COMP_STMT the first statement in the statement list is the target statement for the while_body_begin. The last statement in the statement list is the target for the while_body_end.

Figure 4.4 shows the text position of the target statements. A sample parse tree with highlighted target statements is given in Figure 4.5.

```
main()
{
        int i;

        i = 0;

>>>     while (i < 10) {
                /* while_body_begin instrumentation **/
>>>             printf("The value of i = %d\n",i);
>>>             i = i + 1;
                /* while_body_end instrumentation **/
        }
}
```

Figure 4.4 Source Code Target Statements Related to the While Entities

```
main()
{
        int i;

        /** prog_begin instrumentation **/
>>>     i = 0;

>>>     while (i < 10) {
                printf("The value of i = %d\n",i);
                i = i + 1;
        }
        /** prog_end instrumentation **/

}
```

Figure 4.3 Target Statements Related to the Program Entities

4.3.2.1 Locating the Specific Syntactic Entity

A specific syntactic entity is a syntax related code segment of a block structured language. For example, while_statment_enter corresponds to the while construct in C. A traversal of the parse tree locates the specific syntactic entity the appropriate GENUS-SPECIES combination. When looking for while_statment_enter, the target statement is located with the GENUS-SPECIES combination GEN_STMT-STMT_WHILE.

1. `prog_begin`, `prog_end`

   Locate the function declaration node with `text` corresponding to the main function. For `prog_begin` the target statement is determined by the first node corresponding to the statement within the program body. The `prog_end` target statement is determined by the node corresponding to the last statement in the program body. Figure 4.3 shows the text position of the target statements.

2. while_statment_enter,while_statement_exit, while_body_begin, while_body_end

4.2 C Parsing

The C parser, reused from ATAC, generates a C parse tree. After completion of parsing a parse tree with nodes of type `TNODE`, which represent the parsing rules, is generated. This parse tree component is used as input to the decoration phase.

4.3 Decoration

4.3.1 Preparing the Parse Tree

Two parse tree manipulations must be performed before decoration can begin. The main node is renamed so that it may be called by the LSL driver program at runtime. This is accomplished by traversing the parse tree searching for the `TNODE` which relates to the main function name. It is renamed to _l_m.

To guarantee that the all MIDI notes are played before the program exits, all programmer calls to exit() or _exit() must be changed to _lsl_exit. This is accomplished by traversing the parse tree searching all function call `TNODE` which relate to the exit calls. The node text is renamed to _lsl_exit(). The LSL exit routine is part of the LSL library.

4.3.2 Locating Events

When an auralization is parsed the specifications are stored in the specification database. It is neccessary to instrument event specifiers provided by the specification database in the related code. An event list is obtained from the specification database relating to a given specification. Each of the events is located in the parse tree and decorated appropriately. The located source statement position in the parse tree is defined as the target statement. This section describes how to locate the target statement for the specific syntactic entities and the assertions.

```
unnamed_command          :       notify_command
                                 {
                                         $$=saveNotifySpec($1);
                                 }
                         |       dtrack_command
                                 {
                                         $$=saveDtrackSpec($1);
                                 }
                         |       atrack_command
                                 {
                                         $$=saveAtrackSpec($1);
                                 }
                         |       sync_command
                                 {
                                         $$=saveSyncSpec($1);
                                 }
```

Figure 4.1 Example Specification Parsing of the unnamed_command

```
mmkeyword                :       MM
                                 {
                                         $$ = saveMmAbsolute();
                                 }
                         |       MMREL
                                 {
                                         $$ = saveMmRelative();
                                 }
                         |       MMABS
                                 {
                                         $$ = saveMmAbsolute();
                                 }
                         ;
```

Figure 4.2 Example Specification Parsing of the mmkeyword

4. DETAILED PROCESSING PHASE ARCHITECTURE

4.1 LSL Specification Parsing

The auralization specification (ASPEC) is parsed by the LSL parser. The parser, developed using bison and flex[1], interfaces with the specification database component to build the spec_db.

A routine LSLzzparse(*filename*) is provided in lsl_spec/main.c which will parse the ASPEC stored in *filename*. When parsing is complete the specification data structure is built and ready for use. The parse routine is called by the decoration routines as well as the GUI. The integration process is given below. For a complete description of the grammar refer to Appendix A. For a complete description of the specification database refer to Section 3.2.

Figure 4.1 gives the parsing rules related to the *unnamed_command* non-terminal. The specification database routines are driven by the parsing rules. For example, when processing a `notify`, a specification data structure is generated by the reduction of the *notify_command* nonterminal. When the reduction is complete the routine saveNotifySpec() inserts the constructed `notify` structure into the specification list. Similar specification database routines are constructed for each parsing rule related to a specification command.

Figure 4.2 shows the parsing rules related to the reduction of the *mmkeyword* nonterminal. The reduction of the rule calls the specification database routines generating the related data structures. In this instance, a routine is called to save the metronome mode into the synchronization command structure.

[1] Bison and flex are parser construction tools.

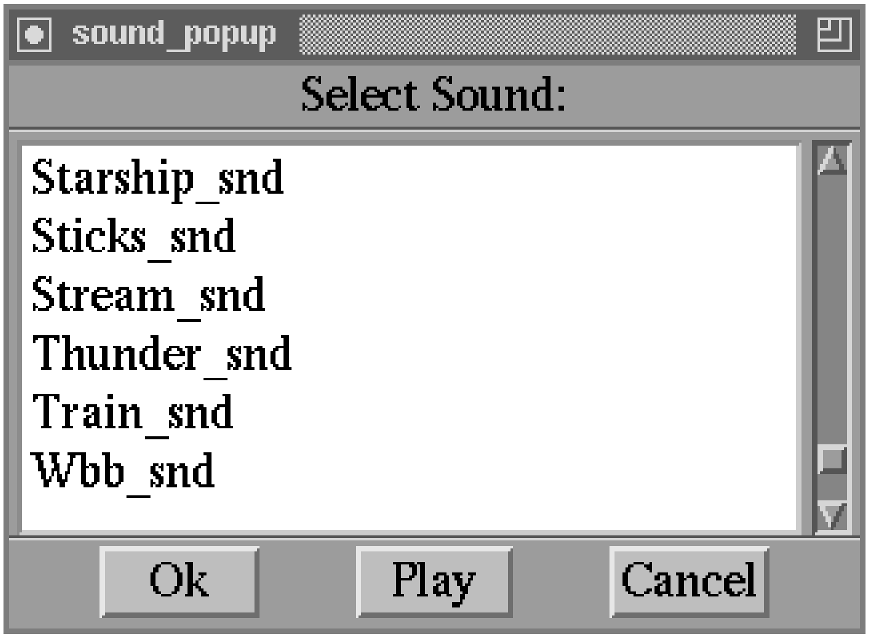

sound_popup
Select Sound:
Starship_snd
Sticks_snd
Stream_snd
Thunder_snd
Train_snd
Wbb_snd
Ok
Play
Cancel

Figure 3.30 The Sound Selection Window

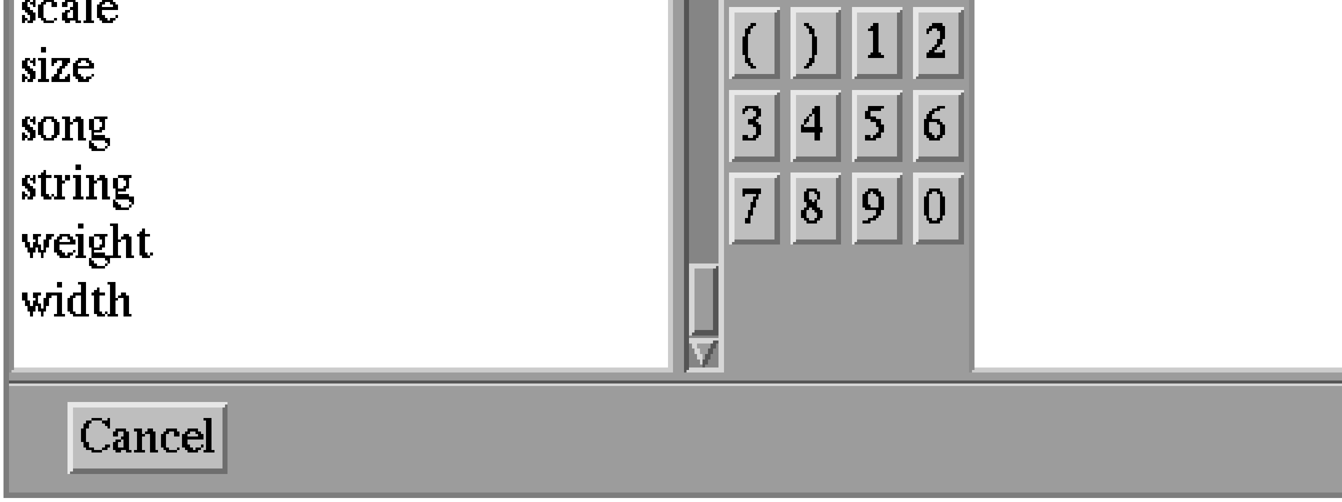

scale
size
song
string
weight
width
( ) 1 2
3 4 5 6
7 8 9 0
Cancel

Figure 3.29 The Relative Timed Event Interface Screen

Sound specification currently supports the selection of a sound from a list of given sounds. This occurs when the drag and drop is placed or the sound pattern specification is chosen by the sound pattern specification button from the specification command window. Figure 3.30 depicts the selection process. In the example a list of sounds is provided for the user to choose from. The user can listen to any of these sounds by clicking the play button. When the desired sound is located the user clicks ok to associate the sound with the related specification command.

3.5.3 Future Releases

Additional features can be added to the GUI. These include, but are not limited to:

1. A full featured music editor.
2. A sound library generation tool.
3. A library of VDAP's and sounds
4. An integrated graphic runtime environment which supports the dynamic modification of ASPECS. These attributes include the state of an specification and the given sound attributes such as sound type, level, and pan.


width
Cancel

Figure 3.28 The Assertion Interface Screen

Figure 3.28 gives and example of the assertion event window. An event named *size_spec_met* has been created. The assertion is located in the text window. Whenever this assertion fails the event becomes true. Tools are available to assist in the creation of the assertion including the variable list and the graphical expression keypad.

Figure 3.29 gives an example of the relative timed event. The user has created an event named time_expired. Thirty seconds after the activation of the event named event0 the event time_expired becomes true. A list of variables and expression keypad is available to assist in the creation of relative time expressions.


UserInfo
Cancel

Figure 3.27 The Specific Syntactic Entity Interface Screen

Figure 3.27 gives an example of specific syntactic entity event window. An event named *event*0 which corresponds to a function entry has been created. The `category` `category` column lists the entities available. Here the category *functions*, which provides the event specifier list shown, has been selected. Choosing the function category also generates the `choose function` scrollable window. All of the functions available in the source code are listed. If a function is not chosen from the list all functions are applicable.


Cancel

Figure 3.26 The General Syntactic Entity Interface Screen

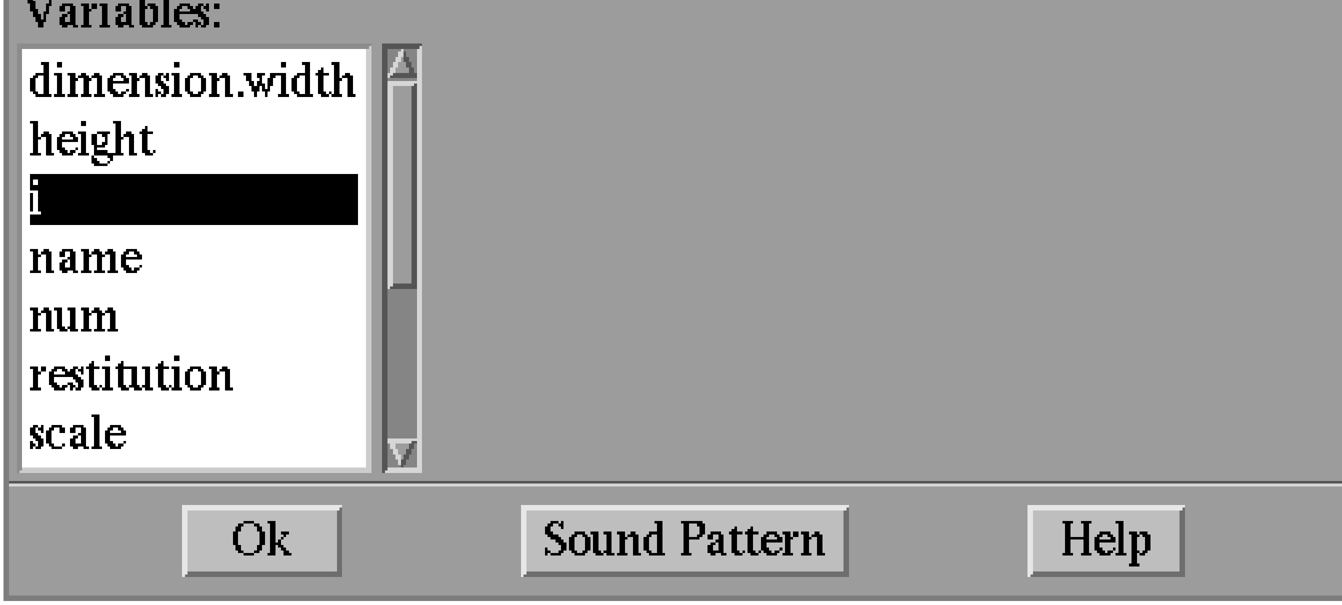

Variables:
dimension.width
height
i
name
num
restitution
scale
Ok
Sound Pattern
Help

Figure 3.25 The `dtrack` Related Graphical User Interface Screen

Event specification takes place graphically by naming and specifying the attributes corresponding to the four event types supported by LSL. These events include the

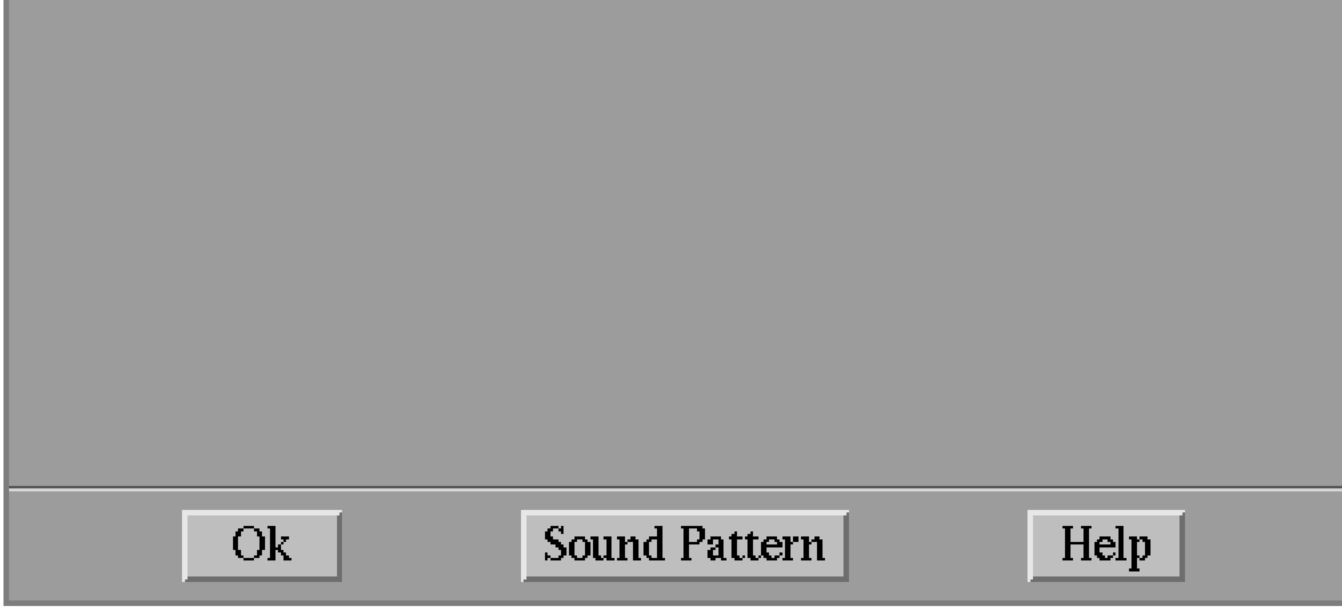

Ok
Sound Pattern
Help

Figure 3.24 The `atrack` Related Graphical User Interface Screen

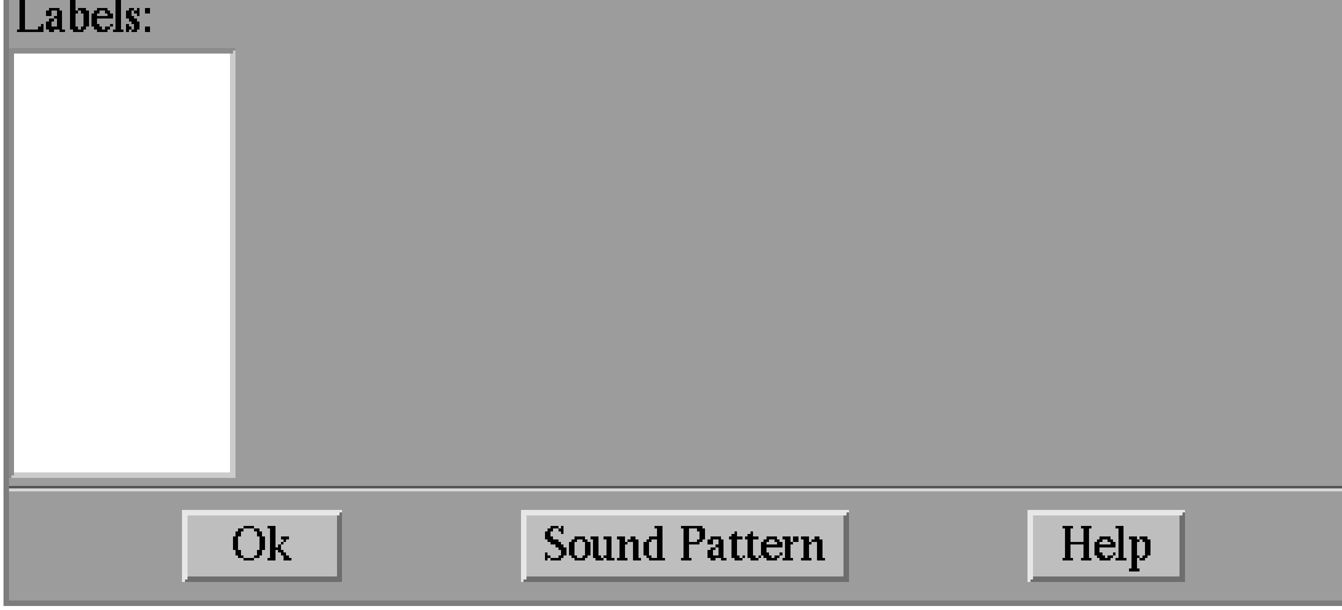

Labels:
Ok
Sound Pattern
Help

Figure 3.23 The `notify` Related Graphical User Interface Screen

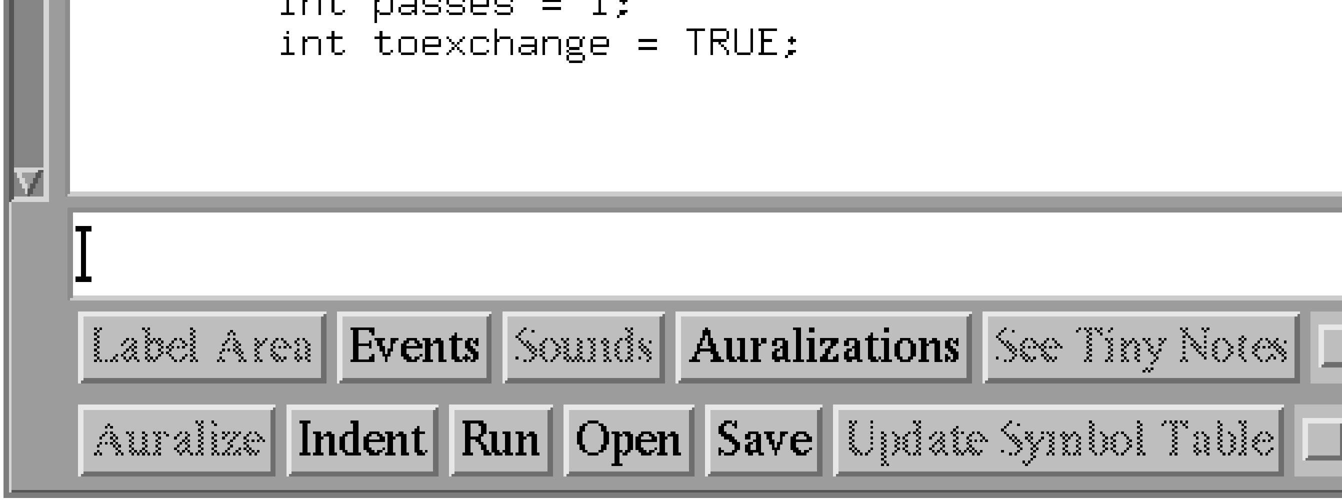

int passes = 1;
int toexchange = TRUE;
Label Area
Events
Sounds
Auralizations
See Tiny Notes
Auralize
Indent
Run
Open
Save
Update Symbol Table

Figure 3.22 The Main Screen of the Graphical User Interface the user to configure the environment. These attributes include the compiler preferences, the execution preferences, and the definition of path names to the required LSL routines. The project menu allows the user to set up which source files are currently being auralized. The sound menu contains sound related manipulation routines. The MIDI menu contains options which set up the MIDI device environment. The VDAP menu is currently not implemented but will eventually support value dependent auralization pattern creation and modification. Across the bottom of the screen are buttons which relate to commonly issued commands in the GUI.

The text area shown in the figure is a fully functional text editor. This editor is different in that it allows the dropping of graphic icons into the text for auralization purposes. The icons which can be dragged and dropped are located to the right of the text entry area. The icons include the note, metronome, time signature, key signature, and the lightbulbs which modify the auralization state of specification classes. Currently the note is the only drag and drop icon available.

The drag and drop note allows the mapping of the location to sound Users may find immediate benefit by using Listen as an application providing aural breakpoints.

The auralization window depicted in Figures 3.23, 3.24, and 3.25 allows the user to generate LSL specification commands such as the `notify`,`dtrack`,and `atrack`. Each of the three screens provides the ability to name the specification and designate class membership.

3.5.1 Interface Design Goals

In order to make the LSL language accessible to a large audience, a graphical user interface has been provided which allows the user to create auralizations by point and click methods.

When designing the user interface, the following criteria was specified:

1. The GUI shall use interface application standards

   The GUI should use an interface similar to existing programs when possible. This includes attributes such as opening a file, quitting the application, and cut and paste functionality.

2. The GUI shall be graphically informative

   LSL is a specification language only, it should be possible to provide the user with the information regarding the ASPEC. By graphically looking at the program text, it should be possible to quickly understand what auralization has been specified for that section of text.

3. The GUI shall provide an intelligent assistant

   The program should be able to assist the user in creating a specification. For example if the user highlights the word while, the interface should present the user with specification choices available to that section of text.

4. The GUI shall support both musical and non musical interfaces

   Not all LSL users are going to have the same degree of familiarity with either reading music or with MIDI. Design should facilitate numerical oriented sound specification as well as note oriented.

3.5.2 The Interface

Figure 3.22 shows the main screen of the Listen graphical user interface. The menubar consists of the familiar file and edit menus. The preferences menu allows _lsl_play_3(int sound, int value) is called to generate sound related to the `dtrack`. It controls sound production by calling the lsl MIDI routine playNote() with the appropriate MIDI note and MIDI channel. The value of the note is determined by the play routine argument *value*.

As of this writing, data tracking supports type integer with value between 0 and 127. The default mapping is to map value to the MIDI note. If the bounds are exceeded the boundary values are used.

After the channel and note have been determined the function call to playNote(chan,value,velocity,duration) is made. The velocity is always 127 and the duration is always a quarternote.

3.4.6 Summary

Musical Instrument Digital Interface, or MIDI, is a communication standard developed by the manufacturers of electronic musical instruments. It was determined to use external MIDI compatable sound devices for Listen because it provides the flexability, documentation, performance, and tools required. In order to use MIDI driver routines were developed which act as a music sequencer. An interface to developed MIDI routines hides details for future Listen developers. In the future it may be desirable to use on board DSP technologies so that no additional hardware is required.

3.5 The Graphical User Interface

Even though the graphical user interface(GUI) is not depicted in the component-phase diagram it is discussed in this section of the component chapter [8]. This section describes the design and initial implementation results of the Listen GUI.

[8] The GUI design was implemented by Geoff Greene as an independent study course at Purdue University.

have been sent, it is necessary to send an all notes off message to the on line MIDI devices. A command is available in the Listen environment which will turn all MIDI notes off. The command is panic.

The source consists of calling the midiInit() routine described in Section 3.4.3.4 above and then immediately exiting. Upon midiExit() the all_notes_off() command is sent to the MIDI device.

3.4.5 The LSL Library

The LSL library routines are inserted into the instrumented source code to generate the sound associated to a specification. Routines are related to the `notify`, `dtrack`, and the `atrack` commands.

The routines are enumerated below. Each routine determines the correct channel and note associated to the given sound by querying the sound database.

1. The `notify` Related Play Routine

   _lsl_play_1(int *sound*) is called to generate the sound related to activation of a `notify`. The routine generates sound by calling the LSL MIDI routine playnote(int *chan*,int *note*,int *velocity*,int *duration*). The *velocity* is always passed as the maximum 127 and the *duration* is always passed as a quarter-note.

2. The `atrack` Related Play Routine

   _lsl_play_2(int *sound*, int *mode*) is called to generate sound related to the atrack. It controls sound production by calling the LSL MIDI routines noteOn() and noteOff() depending on the value of *mode*. If the `atrack` starting event specifier occurs, _lsl_play_2() is called with *mode* ON which triggers the activation of the sound. Upon the terminating event specifier, _lsl_play_2() is called with *mode* OFF which terminates the production of sound.

3. The `dtrack` Related Play Routine

- playNote

```
playNote(channel, note, velocity, duration)
int channel, note, velocity, duration;
```

Playnote takes the information regarding a note and construct the appropriate MIDI command. This allows the programmer to have no prior knowledge of the details of MIDI byte formats. The validity of the fields are checked and passed on to the MIDI driver with the appropriate duration.

- noteOn and noteOff

```
noteOn(channel, note, velocity)
int channel, note, velocity;

int noteOff(channel, note, velocity)
int channel, note, velocity;
```

These routines interface with the MIDI driver. No duration is specified so only the note on or note off message is passed to the MIDI driver.

3.4.4 The Panic Command

Due to possible abnormal termination, a program may not terminate all notes generated by a given instrumentation. Since many of the note off messages may not

Figure 3.21 The MIDI Queue

3.4.3.3 The MIDI Queue

Figure 3.21 depicts the MIDI queue. The queue is a linked list of mdbox structures in sorted order using the `tv_sec` timestamp as the primary key. The `tv_usec` or microseconds is used to break collisions. The list is maintained dynamically.

3.4.3.4 An Interface to the MIDI Driver

The following interface allows Listen developers to interface to the MIDI queue without understanding the complexity of the data structures and algorithms. The routines promote information hiding and abstraction.

- midiInit

  `usage: void midiInit(void)`

  Allows a programmer to initialize the MIDI environment. This includes initializing the interval timer and opening the device.

- midiExit

  `usage: void midiExit(void)`

  When a programmer calls midiExit() the routine waits for the MIDI queue to empty. Then the corresponding device is closed.

```
struct mdbox
{
        struct timeval ts;
        char comm[MAX_COMMAND_LENGTH];
        struct mdbox *next;
};
```

Figure 3.20 The MIDI Queue Data Structure Definition

Upon program termination, the program waits for the MIDI event queue to process all of the pending data. When this occurs the serial port is closed and the MIDI related portion of LSL is complete.

The following sections describe the components which make up the MIDI driver.

3.4.3.1 The Sequencer Interrupt Handler

An interval timer is set up when the user program is executed. This is accomplished by using the UNIX interval timer. The setitimer() call is used to set the interval timer to the appropriate number of seconds and microseconds. When the timer expires it is reloaded or reset to the appropriate timer value.

Every 1000 microseconds the handler is called. If the MIDI event queue is empty, the handler immediately returns. If the handler continues, the current time of day is stored into a current time variable. The handler then traverses the MIDI event queue writing out all MIDI commands which have a time stamp less than or equal to the current timestamp.

Since the handler is called so frequently it is crucial that minimal time be spent in processing the list. Several routines were unrolled directly into the handler code and a modification to the data structure was made. The MidiOut routine was unrolled so that the device was accessed directly eliminating the overhead of several procedure calls. By placing the number of midibytes in the first byte of the MIDI command the need to call $strlen()$ to calculate the number of bytes was eliminated which improved processing time significantly.

3.4.3.2 The MIDI Box Data Structure

The data structure contains a timestamp, MIDI commands, and a next pointer. The command array contains a series of MIDI bytes to be written at the given timestamp. The first byte, comm[0], is the number of midibytes in the command. This was created so that strlen would not need to be called thus eliminating crucial overhead time related to the handler. The structure is given in figure 3.20.

cheaper and easier to use, better solutions may become viable. This would allow Listen distribution without the costly addition of sound producing devices.

Figure 3.19 shows the Listen hardware environment consisting of a Sun workstation connected to the Emu Proteus III World and the Roland SC-55 via a Mediator MS-124 interface.

Figure 3.19 The Listen Hardware Environment

3.4.3 The MIDI Driver

When an instrumented program is executed, the executable calls to midiInit() which opens the serial port at the appropriate baudrate (38400) and initializes the interval timer.

During execution of instrumented code, calls to the LSL library are encountered which generate MIDI data. This data is passed to the LSL MIDI sequencer. The sequencer affixes a time stamp to the MIDI data and inserts it appropriately into the MIDI event queue. This queue is ordered by increasing timestamp value.

As the instrumented program continues execution signals are generated every 1000 microseconds. A handler is entered which determines if any MIDI events in the queue need to be written to the corresponding device.

3.4 MIDI Data

Sounds are determined in Listen by MIDI parameters. MIDI messages are generated during the execution of an instrumented program. These events are sequenced in a MIDI queue and output to the MIDI device. A programmer interface to the MIDI sequencer that hides details regarding MIDI data structures. This section describes the use in Listen.

3.4.1 What is MIDI

Musical Instrument Digital Interface, or MIDI, is a communication standard, developed and adopted by the manufacturers of electronic musical instruments. MIDI makes it possible to connect various musical instruments and sound processing devices to a computer.

MIDI is a serial communication bus similar to the RS 232 or SCSI bus. The MIDI signal is a serial voltage transmission, standardized at the rate of 31,250 bits per second. A MIDI cable is a shielded, twisted pair cable with a 5 pin DIN plug at either end.

MIDI messages are comprised of bytes encoded to define the type of message being sent and the related data. The information can express a range of information including which key on a synthesizer was pressed to the simulation of an entire multitrack recording studio[DS88].

3.4.2 MIDI and the Listen

It was determined to use external MIDI compatable sound devices for the sound production in Listen. Even though on chip signal processing is becoming widely used and less expensive, it does not currently provide the flexability required by the Project Listen. The sound production capabilities of MIDI devices have been proven in years of use in the professional music field. In the future as DSP technology becomes

Figure 3.18 A Parse Tree Example for a Simple Program.

```
#define GEN_COMPSTMT    33
#define   COMPSTMT_DCL_STMTS 0 /* { INDATA_DCLS STMT_LIST }  */
#define   COMPSTMT_STMTS     1 /* { STMT_LIST }              */
#define   COMPSTMT_DCL       2 /* { INDATA_DCLS }            */
#define   COMPSTMT_EMPTY     3 /* { }                        */
#define GEN_STMT        34
#define   STMT_EXPR          0 /* EXPR ;                     */
#define   STMT_EMPTY         1 /* ;                          */
#define   STMT_COMPSTMT      2 /* COMPSTMT                   */
#define   STMT_IF_ELSE       3 /* if ( EXPR ) STMT else STMT */
#define   STMT_IF            4 /* if ( EXPR ) STMT           */
#define   STMT_WHILE         5 /* while ( EXPR ) STMT        */
#define   STMT_DO            6 /* do STMT  while ( EXPR ) ;  */
#define   STMT_FOR_EEES      7 /* for(EXPR;EXPR;EXPR) STMT   */
...
...
#define GEN_EXPR        36
#define   EXPR_QCOLON        0 /* EXPR ? EXPR : EXPR         */
#define   EXPR_COMMA         1 /* EXPR , EXPR                */
#define   EXPR_BINOP         2 /* EXPR BINOP EXPR            */
#define   EXPR_UNOP          3 /* UNOP EXPR                  */
#define   EXPR_INCOP         4 /* EXPR INCOP                 */
...
...
#define GEN_BINOP       40
#define   BINOP_PLUS         0 /* +                          */
#define   BINOP_MINUS        1 /* -                          */
#define   BINOP_MUL          2 /* *                          */
#define   BINOP_DIV          3 /* /                          */
#define   BINOP_MOD          4 /* %                          */
```

Figure 3.17 Sample `genus` and `species` values for the `TNODE`.

Statement nodes within a statement list are ordered similarly to function nodes within a function list. A statement list consists of statments which are children to the node of `genus` type GEN_STMT_LIST. The node of `genus` type GEN_STMT_LIST has a `down` pointer which points to the last statement in the statement list. The `over` pointer from the this last statement node points to the first statement. By accessing the `over` pointer of the `down` node from the GEN_STMT_LIST one gets the first statement of the statement list. One can traverse the statement list by traversing the `over` pointer until the parent `down` pointer equals the current node address. This is shown in figure 3.16

Figure 3.16 The `TNODE` Ordering in a Statement List

3.3.2 Summary

A parse tree, generated by ATAC consisting of nodes of type `TNODE`, is used by Listen. The `TNODE` has a `genus` and `species` which represents the parse rule used to generate that node. The nodes are linked into the tree by `up`, `down`, and `over` pointers. There are also pointers added by Project Listen that facilitate decoration strings. The tree is manipulated by the decoration phase and is used to reconstruct the source C code by the deparse process.

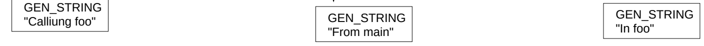


Figure 3.14 Multiple Function Parse Tree Structure

Figure 3.15 Sample Parse Tree of a Multiple Function Program.

The `genus` and `species` fields contain values that describe which parser rule generated the node. All of the possible values for the `genus` and `species` are given in a file lsl_inst/tree.h. Sample values are given in figure 3.17.

The `up`, `down`, and `over` pointers are used for connecting the nodes of the parse tree. The `up` pointer points to the node corresponding to the production that created the node. The `down` pointer points to the first resolution of the production. The `srcpos` field is used to store the position in the source of the related production which is used when reporting errors. The `text` field is non NULL if the production relates to a rule which produces source code. Figure 3.18 gives the parse tree for a simple C program.

The parse tree diagrams should be read according to the following rules with respect to the fields of the `TNODE` data structure.

1. If a `down` down pointer is not given it is NULL.
2. If an `over` pointer is not given then:
   - if there is only one node at the current level it points to itself.
   - if there are multiple nodes at the same level it points to the first (or leftmost) node at that level.
3. The `up` pointer points to the node above the leftmost child at that level. For example in figure 3.18 the node of `genus` type GEN_FUNC_SPEC has an `up` pointer to the node of `genus` type GEN_FUNCTION.
4. If no text value is given between <> then it is NULL.

The parse tree is rooted by the GEN_MODULE `TNODE`. Functions of the program are represented by GEN_MODULE_ITEMS. A node is created for each functions in the source. An example of the function list is given in Figure 3.14. A sample parse tree for a multiple function program is shown in Figure 3.15.

```
typedef struct tnode {
        int             genus;
        int             species;
        int             error;
        SRCPOS          srcpos[2]; /* LEFT_SRCPOS, RIGHT_SRCPOS */
        struct tnode    *up;
        struct tnode    *down;
        struct tnode    *over;
        char            *text;
        union {
                struct symlist  *symtab;
                struct sym      *sym;
                struct {
                                        short blkno;
                                        short tempno;
                                        struct valtype *type;
                }                hook;
        } sym;
} TNODE;
```

Figure 3.13 The `TNODE` Data Structure

3.3 The Parse Tree

The parse tree is built during the C parsing phase which was adapted from the ATAC project developed at Bellcore[HL90]. The parse tree consists of nodes which represent the grammar rule productions of the C source code. This parse tree is decorated by the LSL decoration routines to created the instrumented program. This section provides the background and understanding of the parse tree required to develop the Listen decoration utilities.

3.3.1 The Data Structure

Each node of the parse tree is of structure type `TNODE`. The fields of interest to Listen include `genus`, `species`, `srcpos`, `up`, `down`, `over`, and `text`. Figure 3.12 gives a graphic representation of the `TNODE` data structure. Figure 3.13 gives the `TNODE` structure definition.

Figure 3.12 A Sample `TNODE` from the Parse Tree.

```
/*******************************************************
*
*
*       struct Elink *saveEvent(int type,
*                               char *syn_entity,
*                               struct instance *instance_list)
*
*       Takes the information and creates an event structure.
*       This event structure is then returned to the caller.
*
*       Valid Input Values
*               type
*                       RULE_ID_EVENT
*                       RULE_ID_INST_EVENT
*                       TRIP_EVENT
*
*               syn_entity
*                       non null character string
*
*               instance_list
*                       list created by
*
*       returns
*               pointer to Elink structure
*
*******************************************************/
/*******************************************************
*
*
*       struct Event *saveEventSpec(struct Elink *e1)
*
*       saves an event specification
*
*       returns
*               struct event
*
*******************************************************/
```

Figure 3.11 Sample Specification Database Modification Routines

```
/*******************************************************
*
*       int getEventType()
*
*       Gives the type of the event that is currently
*       being processed
*
*       returns
*               RULE_ID_EVENT
*               RULE_ID_INST_EVENT
*               TRIP_EVENT
********************************************************/
/*******************************************************
*
*       int getSpecSynEntityType()
*
*       Gives the type of the specific syntactic entity.
*
*       returns
*               SSE_EMPTY       SSE_BEGIN       SSE_END
*               SSE_VAR         SSE_AEX         SSE_CEX
*               SSE_IST         SSE_IBB         SSE_IBE
*               SSE_WSE         SSE_WSX         SSE_DOW
*               SSE_FRE         SSE_FRX         SSE_WBB
*               SSE_WBE         SSE_FBB         SSE_FBE
*               SSE_DBB         SSE_DBE         SSE_JMP
*               SSE_CST         SSE_BST         SSE_RST
*               SSE_GST         SSE_SST         SSE_IFS
*               SSE_ITP         SSE_IEP         SSE_SWS
*               SSE_SBB         SSE_SBE         SSE_FNC
*               SSE_FNE         SSE_FNR
********************************************************/
```

Figure 3.10 Sample Specification Database Value Query Routines

```
/*******************************************************
*
*
*       int getFirstSpec()
*
*       Position at the first specification in the speclist.
*
*       returns
*         SPEC_OK   if possible
*         SPEC_END  upon failure
*
********************************************************/
/*******************************************************
*
*       int getNextSpec()
*
*       Position at the next specification
*
*       returns
*         SPEC_OK  if another spec was available
*         SPEC_END if the last one of the list
********************************************************/
/*******************************************************
*
*       int getSpecType()
*
*       Find out what kind of spec is at the current position
*
*       returns
*         GLOBAL_SPECTYPE PLAY_SPECTYPE    NOTIFY_SPECTYPE
*         ATRACK_SPECTYPE DTRACK_SPECTYPE  ASSIGN_SPECTYPE
*         LOOP_SPECTYPE   IF_SPECTYPE      TURN_SPECTYPE
*         TOGGLE_SPECTYPE SYNC_SPECTYPE    ERROR_SPECTYPE
*
********************************************************/
```

Figure 3.9 Sample Specification Database Traversal Routines

Similar data structures are defined for all of the implemented LSL parsing non-terminals. The discussion of each structure is to detailed for this document. The complete data structure is given in Appendix C.

3.2.2 The Database Related Routines

In order to promote the Listen object oriented design methodology, routines were developed which manipulate, traverse, and access the specification data structure. This hides the implementation details from the Listen developer. The routines are located in the lsl_spec directory[7] and are linked to developed code using the spec_db.a library.

Routines which traverse the speclist are given in Figure 3.2.2. To traverse the list a developer first calls getFirstSpec() which initializes to the begining of the list. The routine getSpecType() is called to determine the type of the specification command. The process is repeated calling getNextSpec() until it returns the value SPEC_END which signifies the termination of the list. Similar traversal routines are provided for all specification database list structures.

Routines which obtain information from the specification database are available. The routine getSpecType() was an example of a routine which returns a data value. Figure 3.2.2 shows some example data retrieving routines. The return values are defined in the file lsl_share/lsl_values.h.

Example routines which save and build the specification database structure are given in Figure 3.2.2 gives an example of such routines.

3.2.3 Summary

The specification database is an internal representation of the ASPEC. A data structure is defined for each non-terminal in LSL. Routines are provided which support object oriented design methodologies. These routines save, build, modify, and retrieve information regarding the specification database.

[7] See Appendix B for a directory structure layout of Project Listen.

Figure 3.8 gives a graphical representation of the structure related to the `dtrack` specification command. The *count* field contains an increasing number which assigns the `dtrack` to an entry in the `_dtrack_events`. This is used for activating and deactivating tracking. [6]. The *start* and *term* fields point to the event lists which determine when the tracking of the variables given in the *didlist* is to occur. The *start_condition*, *start_scope*, *term_condition*, and *term_scope* are defined as in the `atrack` *start* and *term* related fields. The *sound* field points to the corresponding sound. The scope field has the value of the related scope specifier and the next pointer points to the next `dtrack` specification command in the speclist.

Figure 3.8 A Graphical Representation of the `dtrack` Related Data Structure

[6] Information related to the `_dtrack_events` construction can be found in Section 4.3.3.5

Figure 3.7 A Graphical Representation of the `atrack` Related Data Structure

```
notify rule = while_statement_enter && assertion = i < 10
        using Drum_snd
```

The event specification condition would contain

```
"_lsl_events[0] && _lsl_events[1]"
```

The scope field has the value of the related scope specifier and the next pointer points to the next `notify` specification command in the speclist.

Figure 3.6 A Graphical Representation of the `notify` Related Data Structure

Figure 3.7 gives a graphical representation of the structure related to the `atrack` specification command. The *start* field points to the eventlist which corresponds to the event specifier related to the given `atrack`. The *start_condition* is constructed by the LSL parser in a similar manner to the `notify` event specification condition discussed previously. *start_scope* defines the scope in which the starting events should be located. The *term* field points to the eventlist which corresponds to the terminating event specifier realated to the given `atrack`. The *term_condition* is constructed similar to the *start_condition*. The *term_scope* defines the scope in which the terminating events should be located. The *sound* field points to the corresponding sound. The scope field has the value of the related scope specifier and the next pointer points to the next `atrack` specification command in the speclist.

3.2 The Specification Database

The specification database is an internal representation of an LSL specification. The data structure promotes the object oriented design principles of Listen. This section describes the data structure and the developed interface to access the data structures.

3.2.1 The Data Structure

The main structure of the specification data base is the speclist. The speclist is a list of the specification commands related to an auralization specification (ASPEC). Each node of the speclist is of a type corresponding to the specification command. Figure 3.5 gives an example specification list which consists of `notify`, `dtrack`, `atrack`, and `syncto` specification commands.

Figure 3.5 The Specification List Data Structure

The data items referenced by the speclist are of the related specification command structure type. Figure 3.6 gives a graphical representation of the structure related to the `notify`. The *type* field represents the all or selective attribute of the `notify`. The *label* field is a non null character string label when the `notify` is of type selective. The *event* field points to the list of events associated with the `notify` event specifier. The *sound* field points to the corresponding sound specifier. The condition field is a text string built by the LSL parser which contains the event specification condition. [5] For example, given the specification

[5] Construction of the boolean event condition is described in Section 4.3.3.1.

It is possible to identify specific constructs of a C program by labeling. A label is placed inside a comment by using the keyword `label` as the first keyword starting with a letter immediately following the comment start delimiter. Thus, for example, /* `label`=here, onemore */ provides two labels *here* and *onemore* for possible use by the LSL preprocessor. The following example shows how to label the beginning and end of a loop.

```
  ⋮
while (c = getchar()!=eof)
  {
  /*label=special_loop This is an LSL label for the beginning of loop body. */
    ++nc;
  ⋮
  /*label=special_loop This is an LSL label for the end of loop body. */
  }
```

3.1.3 Definition of the Minimal Working Subset

A subset of LSL was chosen which would provide the flexability desired by Project Listen but limit the amount of implementation details. This subset was defined in section 2.4. The implemented grammar is given in Appendix A.4.

3.1.4 Summary

The syntax and semantics of a language named LSL provides a notation to specify a variety of program auralizations. LSL is generic and needs to be adapted to the programming language of an environment in which programs are expected to be auralized. A language specific implementation of LSL serves as a tool to auralize programs. This project uses LSL/C, a C adaptation of LSL.

class named *data_related* consists of data items *a, b,c, p*, and *q*. Yet another class named *special* consists of data items *p* and *q*.

The notion of a class can be used to model abstraction during program auralization. For example, consider the auralization of tractor control software. The programmer may like to group all the events into two classes. One class consists of events that correspond to engine control. Another class consists of events that correspond to the control of paraphernalia attached to the tractor, e.g. a seeding device. By simply using the event specification mechanism of LSL there is no way to explicitly incorporate these classes into an LSL specification. The mechanism of naming a command, as described above, however, does provide a convenient means for defining classes.

Once defined, classes of events can be accessed at an abstract level using their names. For example, during the execution of an auralized program, it is possible to interact with the LSL run-time system and turn off the auralization of all events within a class. It is also possible to request LSL a comprehensive list of classes and their individual elements. Thus the use of classes enables a user to interact with an auralized program in terms of "high level" occurrences, e.g. events, instead of dealing with syntax based definitions.

3.1.2.11 Embedding LSL Commands

LSL commands can be embedded in C programs inside comments. The LSL preprocessor recognizes an LSL command embedding if the first token beginning with a letter immediately following the comment begin delimiter (/*) is `LSL:`. Immediately following the delimiter, a sequence of LSL commands can be placed enclosed within the `begin` and `end` delimiters. The LSL commands so embedded are translated to C code by the LSL preprocessor. LSL commands such as `play` and `notify` get translated into calls to library functions. Other LSL commands, such as assignments and `dtrack` commands get translated into more complex C code.

3.1.2.10 Event, Data, and Activity classes

An event class[4] consists of one or more events. A **notify** command specifies one or more events which may occur at several positions inside a program and several times during program execution. Events specified in one or more **notify** commands constitute an event class. Similarly, a *data class* is a collection of one or more variables. A **dtrack** command specifies one or more variables to be tracked. Variables specified in one or more **dtrack** commands constitute a data class. An *activity class* is defined similarly with respect to activities specified in one or more **atrack** commands. A class that consists of at least two elements of different types, e.g. event and activity, or event and data, or data and activity, is known as a *mixed class*.

It is possible for a user to define each of the above classes in an LSL specification. This is done by naming one or more **notify**, **dtrack**, and **atrack** commands. Any of these three commands can be named using the following syntax:

id_1::id_2::...::id_n::command

where each subscripted *id* above denotes a name and *command* denotes any event, data, or activity specification command. Multiple commands can share a name. Each *id*, when used as the name of a command, is treated as the name of a class. The class so named consists of events, data, or activities specified in the commands named by *id*. One command can be assigned multiple names. This makes it easy to define classes that are not disjoint. Consider the following example.

function_related::**notify** **rule**=function_call;

function_related::**notify** **rule**=function_return;

data_related::**dtrack** a **and** b **and** c;

special::data_related::**dtrack** p **and** q;

The above three commands have been named to identify three classes. Class *function_related* consists of events that correspond to function calls and return. Another

[4]Classes defined in this section have no intentional relationship with the notion of classes in C++ and object oriented programming literature.

toggle note from the MIDI keyboard and the toggle key from the computer keyboard. When specified, `id` denotes the name of a class (defined below) of events, activities, and data items to be affected by this command.

During program execution, the auralization state can be toggled using the source specified in the command. For example, if the middle C on a MIDI keyboard is the toggle source, tapping the middle C once, after program execution begins, turns the sound off. Tapping it again turns it on. Input from the toggle source is processed only when an auralized event occurs. When such an event occurs, an LSL library routine is invoked to check for a pending toggle request. If a request is pending, the auralization state is switched to OFF if it is ON, or to ON if it is OFF.

A program may contain both `turn` and `toggle` commands. A `turn` might change the auralization state to off only to be switched back to on by a toggle. This is certainly one useful scenario. Note that whereas `turn` commands are placed into the code prior to compilation and do not provide the user any control *after* compilation, the `toggle` command permits dynamic changes to the auralization state. The toggle default in LSL is the space bar on the computer keyboard. Thus, even when no `toggle` is specified in a program, auralization state may be toggled using the space bar.

Regardless of the auralization state, note values are generated and sent to the library routine responsible for playback. It is this library routine that decides, based on the current auralization state, if the received notes are to be played or not. In the metronome sync mode, all notes emitted are buffered in a special playback buffer maintained by the library routine. The buffered notes are removed from the buffer when their turn comes for playback. This is determined by the current metronome setting. When playback resumes due to a `toggle` or a `turn` changing the auralization state to `on`, the notes are played back in accordance with the metronome setting. In program sync mode, notes received by the library routine are discarded if playback is turned off.

`while` <condition> `do` <spec_sequence>;

The semantics of each of the above commands are similar to that of the `for` and `while` statements in Pascal. All expressions in a `for` command must evaluate to integers. A <spec_sequence> is a sequence of zero or more LSL specification commands.

Conditional commands are provided in LSL for selectively specifying an auralization. The syntax of a conditional command appears below. Its semantics are similar to that of the `if` statement in Pascal.

`if` <condition> `then` <spec_sequence> {`else` <spec_sequence>}

3.1.2.9 Controlling Auralization State

During execution, an auralized program can be in one of two auralization states: ON or OFF. In the ON state any sound data resulting from the occurrence of an auralized event is sent to the sound processor. In the OFF state any such sound data is suppressed. LSL provides two commands to dynamically alter the auralization state. These are the `turn` and the `toggle` commands. These commands have no effect when placed inside an LSL specification. They may affect the auralization state when placed inside the auralized program.

Using `turn` is one way to switch sounds on or off. `turn on` switches the sound on and `turn off` switches it off. The command may be placed anywhere inside the auralized program. Upon the start of program execution, the auralization state is ON. The `turn` command takes effect immediately after it is executed. Sound channels can be switched off selectively by specifying the channel number as in `turn off chan`=4; switches off any sound on channel 4.

Another way to turn the sound on or off is with the `toggle` command. The syntax of `toggle` is given below.

`toggle` {id} <toggle-source> = constant

where <toggle-source> could be the MIDI or computer keyboard indicated, respectively, by the keywords `midi` and `keysig`. The constant is a string containing the metronome. The `syncto` command is used for setting the synchronization mode. The syntax of `syncto` is:

`syncto` <sync-to>

The <sync-to> parameter can be `program` or `mm` for synchronization with, respectively, program execution or a global metronome. Multiple `syncto` commands may be placed in an LSL specification to alter the synchronization mode.

In the metronome mode, a buffer holds the notes generated by the executing program. When this buffer is full and the program attempts to send a note for playback, the playback routine does not return control to the program until the received note can be buffered. This may slow down program execution. To avoid this situation in metronome mode, one may use the `noslow` parameter such as in the command `syncto mm q`=120, `noslow`. When the `noslow` parameter has been specified, playback routine discards notes that are received when the buffer is full. This could cause some events or data tracking to pass by unauralized. The size of the playback buffer can be controlled by setting the `bufsize` parameter such as in `syncto mm`=120, `bufsize`=1000 which specifies a buffer size that will hold at least 1000 notes.[3]

3.1.2.8 Assignments, Loops, and Conditionals

An assignment command has the general syntax shown below.

identifier {<subscript_list>}:= <expression>;

where identifier is the name of a variable. Expression is any valid expression that evaluates to the type of the identifier on the left of the assignment. <subscript_list> is a list of subscripts used for selecting array elements if the identifier denotes an array. Loops can be formulated in an LSL specification using the `for` and `while` constructs. Syntax of these two constructs is given below.

`for` <for_index> := <init_expression> `to` <final_expression>
{`step` <step_expression>} <spec_sequence>

[3] Each note belonging to a chord counts as one note.

Time is measured in system dependent ticks; each tick being the smallest unit by which `time` could be incremented. Thus, any expression using `time` can be used as a timed event. As an example, suppose that the em gear_change function must be invoked in a program in less than 60 seconds after the program execution begins. It is desired to playback variable *bad_program* if this condition is not satisfied. The following `notify` illustrates how to write this specification in LSL.

`notify rule` = function_call: gear_change `and assertion=time` $\leq$ sectotick(60)
`using` bad_program `mode` = `discrete`;

In the above example, *sectotick* is an LSL predefined function to convert seconds to ticks. Notice that the expression `time` > sectotick(60) is a valid way to specify an event as described earlier while discussing the syntax of `notify`.

It is often required to specify time relative to the occurrence of some event. This can be done in LSL using relative timed events as shown below.

`rtime` = <expression> `after` <event-specifier>

Consider the use of this mechanism in the following example for tracking an event.

`dtrack when` (`rtime`=sectotick(30)) `after rule` = function_call: missile_launch
`until rule`= function_return: target_hit `using` missile_in_motion;

The above `dtrack` can be read as "Begin tracking 30 seconds after the function *missile_launch* has been called and terminate tracking when the function *target_hit* returns. The tracking sound is defined by the LSL variable *missile_in_motion*. Thus, using a combination of `time` and `rtime`, one may specify a variety of timed events for auralization.

3.1.2.7 Playback Synchronization

Synchronization mode controls the playback of notes during program execution. There are two such modes: *program* or *metronome*. In the program mode, playback is synchronized to the program. In the metronome mode it is synchronized to a global

Figure 3.4 Sample Activity Patterns Specifiable in LSL.

is omitted. If both the start and terminating events are omitted then the entire program execution is tracked. In `continuous` mode, an activity begins whenever the starting event occurs and terminates at the terminating event. In the `discrete` mode, an activity occurs as above but does not resume. Using the start and terminating events one may specify a variety of activity tracking patterns as shown in Figure 3.4.

3.1.2.6 Timed Events

LSL provides a powerful mechanism to auralize timed events. `time` is a special variable in LSL which denotes the time spent from the start of program execution.

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

2. `dtrack` *crash* `init=false`; will track *crash* assuming an initial value of `false`.

3. `dtrack` x `capture=`x_reset; will track x after capturing its initial value at the assignment labeled by the LSL label *x_reset*

4. `dtrack` *mouse* `and` *color* `using` *color_mouse_melody* (&mouseval, &colorval) ; will track variables *mouse* and *color* using a user defined function named *color_mouse_melody* with two parameters.

5. `dtrack` *speed* `when` speed>65 `until` x≤65 `mode=continuous`; will begin tracking *speed* whenever its value exceeds 65 and will stop tracking it immediately after its value becomes equal to or less than 65. Tracking will resume if the start event occurs again. The discreet mode can be used to avoid resumption of tracking of *speed*.

3.1.2.5 Activity Monitoring

An activity is a sequence of actions between two events. An activity begins at the occurrence of an event and ends at occurrence of a later event. As mentioned earlier, start and termination of program execution are considered as events. LSL allows specification of tracking arbitrary activities using the `atrack` command given below.

`atrack` { `when` <event-specifier> } { `until` <event-specifier> }
<sound-specifier> {<mode-specifier>}

<event-specifier>, <sound-specifier>, and <mode-specifier> have the same meaning as in the `dtrack` command. Tracking begins when the event specified immediately following `when` occurs (start event) and stops when the event specified following `until` occurs (terminating event). If the start event is omitted, tracking begins at the start of program execution. Tracking ends at program termination if the terminating event

dependent auralization, LSL provides the `dtrack` command. The syntax of `dtrack` appears below.

> `dtrack` <track-id-list> <sound-specifier> {<mode-specifier>}
> {<start-event-spec>} {<term-event-spec>}

Using `dtrack`, one or more variables can be tracked. For the variable to be tracked, an initial value can optionally be specified using the `init` keyword. The type of the initial value must match that of the variable to be tracked. The initial value may also be captured immediately after the execution of an assignment labeled using an LSL label.

As in `notify`, a <sound-specifier> specifies the sound to be used while tracking the variables. Here we introduce another method for specifying sounds which is particularly useful in conjunction with the `dtrack` command. A sound pattern whose characteristics depend on program generated data will be referred to as a *Value Dependent Aural Pattern* and abbreviated as VDAP. The `using` clause in the <sound-specifier> specifies the name of the 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. f is executed after each assignment to the variable being tracked.

Tracking may be carried out in continuous or discrete mode. In continuous mode, tracking begins at the start of program execution, unless specified otherwise. A note pattern is emitted continuously until there is a change in the value of the variable being monitored. When the value changes, a newly computed note pattern is emitted continuously. In discrete mode, a note pattern is emitted once whenever the tracked variable changes its value. In discrete mode tracking begins the first time the tracked variable changes its value after program execution.

Tracking can also be controlled using <start-event> and <term-event>. Start and terminating events are specified, respectively, using the `when` and `until` clauses. A few examples of `dtrack` use appear below.

Table 3.5 Keywords and Codes for LSL event specifiers in C.

Category	**Event specifier**	**Code**†	**Event specifier**	**Code**†
Program	start	start	end	end
Expression	variable	var	assignment_expression	aex
	conditional_expression	cex		
Iteration	iteration_statement	ist	iteration_body_begin	ibb
	iteration_body_end	ibe	while_statement_enter	wse
	while_statement_exit	wsx	do_while	dow
	for_statement_enter	fre	for_statement_exit	frx
	while_body_begin	wbb	while_body_end	wbe
	for_body_begin	fbb	for_body_end	fbe
	do_while_body_begin	dbb	do_while_body_end	dbe
Jump	jump_statement	jmp	continue_statement	cst
	break_statement	bst	return_statement	rst
	goto_statement	gst		
Selection	selection_statement	sst	if_statement	ist
	if_then_part	itp	if_else_part	iep
	switch_statement	sst	switch_body_begin	sbb
	switch_body_end	sbe		
Functions	function_call	fnc	function_entry	fne
	function_return	fnr		

† Event specifiers and their abbreviated codes can be used interchangeably to specify a rule in a `notify` statement.

Example 2 is the same as Example 1 except that the event selection is selective. Thus, any loop body labeled by *special_loop* will be auralized. Any syntactic entity can be labeled in the program being auralized by placing an LSL `label` command in front of that entity as described in Section 3.1.2.11.

Example 3 specifies the execution of the statements *++count* and *search(x)* as the events. When any of these two events occur, *count_or_search* is played. However, these events are to be recognized only inside functions *search* and *report*.

Example 4 above specifies an event which occurs whenever the condition (x<y || p≥q) is not satisfied. Note that this condition is based on variables in the program being auralized. When this condition is not satisfied, *assertion_failed* is to be played. Example 5 shows how to specify the auralization of all conditional expressions that occur in file *myfile.c* only when condition *odd(x)* is not satisfied.

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

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

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

The above `notify` commands specify the same type of events as in Example 2 except that loop body begins and ends that contain any one of the two labels *loop_1* and *loop_2* will be selected for auralization.

3.1.2.4 Data Tracking

Event notification consists of specifying one or more events and reporting them aurally during program execution. There are applications wherein changes to values of variables need to be monitored. It is certainly possible to specify assignments to such variables as events and then report the execution of these assignments aurally. Such reporting is, however, independent of the data being assigned. To obtain data

The scope of a `notify` may be restricted using the <scope-specifier>. In LSL/C, the scope can be restricted to one or more functions or files. For example, if an assertion is to be checked only inside function *sort*, one may suitably restrict the scope to that function. Labels can be used in conjunction with scope restrictions to specify arbitrarily small regions in a program.

The sound specifier is a variable name, constant, or a function call that specifies the intended auralization of the selected events. Sample `notify` commands appear below.

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

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

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

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

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

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

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

Example 1 above specifies two event types, namely the beginning and end of a while-loop body using two general purpose syntactic specifiers. It also indicates that all positions in the program where such events could occur are to be auralized. Thus, a C program auralized using the above `notify` will generate the sound corresponding to the variables body_begin and body_end, respectively, whenever the beginning and end of a while-loop body are executed.

time as described earlier. For example, in an automobile simulator, events such as *gear change, speed set, resume cruise*, and *oil check* may be candidates for auralization. Suppose that the occurrence of these events is indicated by calls to procedures that correspond to the simulation of an activity such as *gear change*. It is these procedure calls that serve as event indicators to LSL. Thus, for example, such a call to the *gear_change* procedure could be mapped to sound using an LSL specification.

Event specification is achieved by the `notify` command. `notify` is a generic command and can be adapted to a variety of procedural languages. In examples below we assume that programs being auralized have been coded in C. The syntax of `notify` appears below:

`notify` {<all-selective>} {<label-parameter>} <event-specifier>
{<sound_specifier>} <scope-specifier>}

<all-selective> specifies which subset of events selected by a `notify` are to be auralized. Possible event codes are `all` and `selective`. If `selective` is used, one or more labels must be specified to indicate which events are to be selected. <event-specifier> specifies one or more events to be notified aurally.

There are five ways to specify an event. One may specify a general syntactic entity, a special syntactic entity, an assertion, a relative timed event, and any combination of the above four. Relative timed events are discussed in Section 3.1.2.6; other methods are described below. Table 3.5 lists all event codes in LSL/C. For example, *while-statement-enter* is an event specifier; the corresponding event occurs once each time a while statement is executed. The start and termination of program execution serve as events.

The expression $(x < y)$ serves as a special syntactic entity. The associated event occurs whenever the expression (x<y) is executed. An assertion such as $(x + y) > (p+q)$ also specifies an event which occurs whenever the assertion evaluates to false. If e_1 and e_2 are two events specified using any of the above approaches, then (e_1 `and` e_2) and (e_1 `or` e_2) are also events.

an array can be accessed by subscripting. Thus tclef_staff [k+1] refers to the (k+1)th element of tclef_staff which is of type pattern.

```
const
  scoresize = 25;

var
  tclef_staff: array [1..scoresize] of pattern;
```

3.1.2.2 Sound Pattern Specification

The **play** command is used to specify what sounds are to be generated when some part of a program is executed. The general syntax[2] of **play** is:

play <playlist>

where <playlist> is a list consisting of one or more notes and patterns specified using constants, variables, and function calls. Key and time signatures are some of the parameters that may be specified. Elements of <playlist> can be separated by a comma (,) or a parallel (||) sign. An example of **play** command appears below.

play (loop_background || (func_call, no_parameters)) **with mm q** =120, **inst** = "piano";

The above **play** when executed will play the sound associated with the variable loop_background together with a sequence of sounds denoted by the variables func_call and no_parameters. Default key and time signatures will be used. The metronome will be set to play 120 quarter notes per minute and the notes will be played using a piano sound.

3.1.2.3 Event Notification

A useful characteristic of LSL is its ability to specify events to be auralized. A programmer may formulate an event to be auralized in terms of the application. However, such a specification is translated in terms of program position, data, and

[2]Syntactic entities are enclosed in < and >. Optional entities are enclosed in { and }. For a complete syntax of LSL see Appendix.

Table 3.4 Default Values of Run Time Parameters.

Item	Default value
Metronome	q=120
Key signature	C major
Time signature	(4:4)
Channel	1
Instrument code	1
Note duration	`q`
Play mode	`discrete` for `notify`
	`discrete` for `dtrack`
	`continuous` for `atrack`
Pitch	"C4"

Table 3.3 Sample Note Values using LSL duration attributes.

Note value	**Attribute combinations**
Quarter note	`q` or `hh`
Eight note	`hq`
Sixteenth note	`hhq` or `qq`
Thirty second note	`hhhq`
Sixtyfourth note	`hhhhq` or `ss`
Dotted half note	`h+q`
Dotted quarter note	`q+hq`
Dotted eihgth note	`hq+hhq`

"E4:q" denotes a quarter note whose duration will be determined by the time signature and the metronome value. The duration attributes can be multiplied or added to get dotted quarter note, and other fractions of note values. For example, (hq) read as *half of quarter* denotes an eight note, (hhq) read as *half of half of a quarter* denotes a sixteenth note. Table 3.3 lists sample note values and the corresponding attribute combinations. Various rests could be obtained using the attribute combinations shown in Table 3.3 with the letter `R`. For example, "R:(hq+hhq)" denotes a dotted eight rest.

Duration can be specified for a chord by a single duration attribute. For example, "(C4E4G4):q" denotes a chord consisting of three quarter notes. Notes and chords for which the duration is not specified explicitly, as in "E4", are played for a duration determined by implementation dependent default durations (See Table 3.4 for various defaults.).

Type Constructor

Values of primitive types can be combined together into an array. The following sequence declares an array of measures, each measure being a pattern. Elements of

Table 3.2 Attributes in LSL.

Code	Applicability	Description
`f`	Note	Indicates a full note .
`h`	Note	Indicates a half note.
`q`	Note	Indicates quarter note.
`e`	Note	Indicates eight note.
`s`	Note	Indicates sixteenth note.
`chan`	Note, pattern	Specifies the MIDI† channel on which to play.
`play`	Note	Indicates one or more play styles.
`inst`	Note, pattern	Specifies which instrument is to play.
`mm`	Pattern	Metronome setting. This is applicable only to patterns. Notes not part of a pattern are played for a duration determined by global metronome setting. A metronome setting specified for a pattern takes priority over any global setting only while this pattern is played.
`ptime`	Note, pattern	Specifies the exact time in seconds to play the note or a pattern.

† MIDI is an acronym for Musical Instrument Digital Interface.

constant and can be assigned to a variable of type `voice`. Voice can be used in note patterns by specifying variables of type `voice`.

Variables must be declared before use. The following declaration declares body_begin and body_end to be of type `note`, loop_begin, loop_end, and measure to be of type `pattern`.

```
var
  body_begin, body_end: note;
  loop_begin, loop_end, measure: pattern;
```

Note and Rest Values

Attributes aid in specifying various properties of notes and patterns. Perhaps the most common attribute of a note or a chord sequence is its duration. For example,

Table 3.1 Primitive Types in LSL.

Keyword	**Sample values**	**Description**
`int`	−20 or 76	Set of integers.
`note`	"E4b"	Set of notes; not all of these may be played back in a particular implementation. A subset of the notes is labeled starting at A0 and going up to C8 as found on an 88-key piano keyboard. These 88 notes correspond to integer values of 0 to 87. &R&A rest is treated as a silent note with duration specified by a duration attribute.
`tsig`	(3:8) or (3+2+2:4)	Set of pairs of values denoting a time signature. The first element in the pair specifies the beat structure i.e. the number of beats per measure. The second element is the note value that corresponds to one beat. The beat structure could be complex as explained in the text.
`ksig`	"Eb:minor" "(C D E F# G A B)"	Set of k-tuples of pitch values. The set may be specified using abbreviations such as Eb:minor to indicate the key of Eb minor or by enumerating all pitches regardless of their specific position on a keyboard as in the example.
`pattern`	"G3E3C4"	Set of note and/or chord patterns consisting of zero or more notes or chords.
`voice`	†	Set of digitized voice patterns. A variable of this type can be set to point to a memory or disk file containing a digitized voice pattern.
`file`	"done-voice.v"	Set of file names. File extensions are interpreted. .c is for C program files, .v for digitized voice files.

† Any digitized sound in a suitable format.

in computer programs, values of type `note` and `pattern` could be *played back* during the execution of an auralized program.

The set of key signatures constitutes the type `ksig`. Pre- or user-defined functions are used to manipulate values of type `ksig`. Constants of type `ksig` are enclosed inside double quotes and can be assigned to variables of the same type. A key signature could be predefined or user defined. A predefined key signature consists of two parts: a key name and a modifier. Examples of key names are Eb (denoting E flat) and C# (denoting C sharp). Modifiers could be major, minor (same as harmonic minor), lydian, ionian (same as major), mixolydian, dorian, aeolian, phrygian, and locrian. Thus, for example, "C#:minor" and "E:phrygian" are valid key signatures. A user defined key signature is any enumeration of notes. For example, "C D Eb G A" is a key signature of a pentatonic scale.

The set of time signatures constitutes the type `tsig`. Constants of type `tsig` are enclosed within parentheses. A time signature consists of two parts: the beat structure and the note that takes one beat. For example, (4:4) is a simple time signature indicating 4 beats to a measure with a quarter note of one beat in duration. A more complex time signature is (3+2+2:8) which indicates a beat structure of 3+2+2 with an eighth note taking one beat. A beat structure such as 3+2+2 indicates that the first measure is of 3 beats in duration, followed by two measures each of 2 beats duration, followed by a measure of 3 beats and so on. Time signatures can be assigned to variables of the same type and manipulated by functions.

Type `file` is the set of file names. A filename is specified by enclosing the name within double quotes. Thus, "your_name_please.v" can serve as a file name. The use of file names is illustrated through LSL examples below. Note that we use a string of characters enclosed within double quotes in a variety of contexts. It is the context that unambiguously determines the type of a string.

A special type `voice` has been included to play digitized voice during program execution. Voice will be stored as a sample in a file. It is this sample that becomes a

```
begin auralspec
    specmodule myprog_auralize

    /* This module contains specifications to auralize myprog procedure.      */

    /* Applicability constraints, if any, come here.                           */

    /* Declarations for variables global and external to this module .         */

    specdef specdef_1 (parameters);
     /* Declarations of parameters, local variables, and functions. */
     begin specdef_1
     ⋮
     end spec specdef_1;

    spec-def spec_def_2 (parameters);
     /* Declarations of parameters, local variables, and functions. */
     begin specdef_2
     ⋮
     end specdef_2;

     ⋮
    specdef specdef_n (parameters);
     /* Declarations of parameters, local variables, and functions. */
     begin specdef_n
     ⋮
     end specdef_n;

    begin myprog_auralize;
    /* Specifications for module myprog_auralize. */
     ⋮
    end myprog_auralize;

    /* Other module specifications. */
    ⋮
end auralspec.
```

Figure 3.3 Structure of an LSL Specification Containing One Module..

by the module name such as spec_module_1, spec_module_2, and so on in this example. A module header is followed by applicability constraints which specify parts of the program to which the specifications are to be applied. Then come declarations of variables used in this module followed by zero or more specification definitions such as spec_def_1, spec_def_2, and so on. Global variables are shared between various modules. Variables and specification definitions to be exported (imported) are listed in the `export` (`import`) declaration. Variables declared in the program being auralized can also be used inside LSL specifications. These are known as *external* variables.

3.1.2.1 Constants, Variables, and Types

LSL is a typed language. It contains constants, variables, and types just as several other languages do. An identifier name is a sequence of one or more characters consisting of upper or lower case letters, digits, and the underscore (_). The first character in an identifier must be a letter or an underscore. Upper and lower case letters are treated as being different. Variables and constants can be assigned arbitrary names. Values likely to arise during program auralization are grouped together into primitive types. Table 3.1 lists the primitive types available in LSL. Values of type `note` and `pattern` are enclosed in quotes to distinguish them from variable names. A note is specified by indicating its pitch e.g. "E4b" indicates E-flat above the middle C on a piano keyboard. Attributes listed in Table 3.2 can be added to a note separated by a colon (:). A pattern is a sequence of notes and voices[1] played in the specified sequence. A sequence of notes within a pattern can be enclosed in parentheses to indicate a blocked chord also referred to as a chord pattern. A variable name can be used within a pattern by preceding it with a dot. For example, if the identifier *cmajor* denotes a chord pattern, then p:= ".cmajor E5" denotes a pattern consisting of the value of *cmajor* followed by the note E5. Just as values could be printed or displayed

[1]Data of type "voice" refers to digitized sound. Thus, for example, both digitized voice and digitized guitar sound are characterized as voice data.

3.1.1.3 Sound Space Characterization

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

3.1.1.4 Programming Language Independence

The second requirement stated above is significant as LSL should be usable by programmers regardless of their preference for one or the other programming language. Adherence to this requirement has produced a language which in the strict sense should be considered as a meta-language. One can therefore adapt LSL to specific programming languages. However, in the implementation for this research the C language is implemented[KR88].

3.1.2 Features and Syntax of LSL

The features of LSL are reviewed next. Details of LSL syntax and semantics appear in Appendix A. An LSL program is known as a specification. Each specification is composed of one or more specification modules. Each specification module is composed of zero or more specification definitions and one main specification. A specification module, a specification definition, and a main specification are analogous to, respectively, a module, a procedure, and a module body in a Modula-2 [Set89] program. As an example of LSL specification structure consider the specification listed in Figure 3.3. It begins with `begin auralspec` and ends with `end auralspec`. Each module begins with a header identified by the `specmodule` keyword followed

Figure 3.2 Occurrence Space Characterization in LSL.

are realized for all executions of P. An implementation of L for programs in a given programming language PL is said to be correct if each ASPEC, written in L, for any program P, written in PL is realized.

3.1.1.2 Occurrence Space Characterization

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

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

As shown in Figure 3.2, a three dimensional space is used for specifying occurrences in LSL. Two kinds of occurrences are distinguished: events and activities. LSL allows an arbitrary combination of data relationships, positions, and time to specify an event or an activity associated with program execution.

Figure 3.1 A Domain Based View of Program Auralization

3.1.1.1 ASPECs and Realizations

To be able to design a language that can specify all possible auralizations, a quantification of two domains is established. Let E be the domain of all those occurrences during the execution of any program that one may wish to auralize. The nature of such occurrences is discussed below. Let S be the domain of all possible sound patterns that may be associated with each element of E. A mapping from E to S is an association of sound patterns in S to occurrences in E. Such a mapping is specified as a set of pairs (e, s) where $e \in E$ and $s \in S$. The term *program auralization* for a given program P refers to the set $\{(e_1, s_1), (e_2, s_2), \ldots, (e_n, s_n)\}$, where each $(e_i, s_i), 1 \leq i \leq n$ is an association of an occurrence to a sound pattern. A *language* L for program auralization is a notation to specify any such mapping for any program. A mapping specified using L is referred to as *auralization specification* abbreviated as ASPEC. Specifications are always written with reference to a given, though arbitrary, program in some programming language. Figure 3.1 illustrates this view of program auralization. Note that an ASPEC is a many-to-many mapping.

Let (e, s) be an element of an ASPEC for program P. During the execution of P if each occurrence e is identified by a sound pattern s, it is said that the pair (e, s) has been *realized*. An ASPEC for program P is considered realized if all its elements

3. DETAILED COMPONENT ARCHITECTURE

3.1 The Listen Specification Language

A language has been designed that simplifies the task of specifying which occurrences during program execution are to be auralized and how. The language is named Listen Specification Language, abbreviated as LSL.

LSL fulfills the need for a general purpose mechanism to specify the auralization of programs. In the absence of such a mechanism, auralization is done by editing the source code and adding calls to library procedures that generate sound.

Music specification languages have been developed before. Note the pioneering work in the design of languages for music [Lan90, Tho90]. The main purpose of these languages was to specify music. They do not fill the mapping criteria required.

3.1.1 Basic Definitions and LSL Requirements

Based on the perceived need for a specification language, the following idealized requirements for LSL were established.

1. *Generality*: It should be possible to specify any auralization using LSL.

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

Below the basic terms are defined and concepts that help formalize the above goals are introduced. The formalization brings reality to the above requirements. LSL satisfies the requirements with respect to this formalization.

```
_l_m()
{
        int i = 0;

        _lsl_events[1] = 1;
        if ( _lsl_events[1] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_1(5);
        }
        _lsl_events[1] = 0;

        while(i < 10) {
                _lsl_events[3] = 1;
                if ( _lsl_events[3] ) {
                        _lsl_heartbeat_nonote();
                        _lsl_play_1(1);
                }
                _lsl_events[3] = 0;

                printf("The value of i = %d\n",i);
                i = i + 1;

                _lsl_events[4] = 1;
                if ( _lsl_events[4] ) {
                        _lsl_heartbeat_nonote();
                        _lsl_play_1(2);
                }
                _lsl_events[4] = 0;
        }

        _lsl_events[2] = 1;
        if ( _lsl_events[2] ) {
                _lsl_heartbeat_nonote();
                _lsl_play_1(6);
        }
        _lsl_events[2] = 0;
}
```

Figure 2.9 Sample Decorated Source File

3. The specific syntactic entity and the assertion event types.

4. Boolean event evaluation such as $(event1 \&\& event2)$ and $(event1 \&\& (event2 || event3))$ for the purpose of event specification.

5. Predefined sounds for specification mapping.

6. Specification command scopes to limit the decoration with respect to file or function.

This minimal working set was implemented by Project Listen to begin investigation into the use of sound in computing environments. The complete grammar associated with the minimal working set is given in Appendix A.4.

Figure 2.8 The Project Listen Hardware Environment

2.3.8 Summary

During the transformation process many components are generated and used to create the final instrumented executable. The auralization specification (ASPEC) is constructed using the LSL language. The auralization database is an internal representation of the ASPEC. A parse tree is generated from the C source code and after decoration becomes a decorated parse tree. Deparsing of the decorated parse tree results in the decorated source file. The source file is compiled generating an instrumented executable which generates MIDI data upon execution.

2.4 Establishing a Minimal Working Subset for Implementation

Given the generality of LSL, it is necessary to chose a working subset will (1) minimize the amount of implementation and (2) provide enough power to demonstrate the feasability of the method. The following aspects are implemented in the minimal working subset:

1. All three auralization commands `notify`, `dtrack`, and `atrack`
2. The synchronization command (`syncto`) to synchronize the playback to the program or a metronome.

3. LSL Initialization routines.

A sample of decorated C source code is given in Figure 2.9. The complete example including the original source and specification are given in Appendix D.

2.3.6 The Executable

The instrumented executable component contains the compiled decorated sources linked to Listen libraries. The Listen libraries handle interaction with the specification database, setting up the MIDI environment, and initializing the program event state. Figure 2.7 depicts the components of the executable.

Figure 2.7 Architecture of the Executable Component

2.3.7 MIDI Data

Musical Instrument Digital Interface, or MIDI, is a communication standard, developed and adopted by the manufacturers of electronic musical instruments which makes it possible to connect various musical instruments and sound processing devices to a computer[DS88]. MIDI data is generated and sent to a MIDI device to produce sound. The MIDI devices currently used for Listen include the Roland SC-55 and the Emu Proteus III World. Figure 2.8 depicts the Project Listen MIDI environment.

2.3.4 The Decorated Parse Tree

Each `TNODE` of genus type GEN_STMT has the potential of being decorated. To facilitate decoration the `TNODE` data structure has additional fields that contain event conditions and flags. After completion of the decoration phase the parse tree is considered decorated. This decorated parse tree has the additional fields constructed according to the mapping specification. Figure 2.6 depicts a decorated `TNODE` structure. A detailed description of the decorated parse tree is given in section 3.3.

Figure 2.6 A Sample `TNODE` from the Parse Tree.

2.3.5 The Decorated Source Code

A component of the deparse phase is the decorated source file. The file contains instrumented source which performs the following:

1. Playback of specified sounds.

2. Triggering and tracking of events.

Figure 2.5 Parse Tree Structure for a Simple Program.

```
extern int              getEventType();
extern int              getNextEvent();
extern int              getFirstSpec();
extern int              getNextSpec();
extern int              getSpecType();

extern struct Elink     *saveEvent();
extern struct spec      *saveSpec();
extern struct spec      *saveNotifySpec();
extern struct spec      *saveAtrackSpec();
extern struct spec      *saveDtrackSpec();
```

Figure 2.4 Sample Interface Routines to the Specification Data Structure.

2.3.2 The Auralization Database

The auralization database is created by the LSL specification parsing routines. It is interrogated by the Listen decoration routines, Listen graphical user interface, and the Listen runtime environment. The auralization database consists of a data structure and routines to manipulate that data structure. A detailed explanation of the data structure is given in section 3.2.

The database consists of specifications linked together where each specification in the list is of type `notify`,`dtrack`, `atrack`, or any specification command. Depending on the type of command, the appropriate information is stored specifying the scope, the sound, and the related event specification information.

An interface to this data structure has been developed using object oriented design principles [GJM91] promoting data abstraction, information hiding, and code modularity. By providing this interface developers need not concern themselves with the details of the data structure. Routines are provided which get the first specification, get the specification type, get the next specification, save an event, save a specification, etc. Figure 2.4 shows some of operations provided for the data structure.

The data structure and it's interface routines are located in a specification library that can be linked to LSL related software.

2.3.3 The Parse Tree

The parse tree is generated by the C parsing phase which was reused and modified from the ATAC project at Bellcore??. Nodes of the parse tree, `TNODE`s, are of a given `genus` and `species` which describe the parser rule generated at this node. A sample parse tree depicting the `TNODE` structure for a simple program is given in Figure 2.5. The sample parse tree is rooted by a node with `genus` GEN_MODULE. The child of this node is of `genus` type GEN_MODULE_ITEM and `species` DECL_ITEM. A detailed description of the parse tree is given in section 3.3.

```
/************************************************************
*
* File  : guessing.lsl
* Description  :
*
* This specification defines a sound mapping which
* gives the user an aural response when trying
* to guess a number.
*
*************************************************************/
begin auralspec
specmodule guessing_game
begin guessing_game

        /**  When they get the right answer play a bell **/

        notify assertion = (quit != 1) using Phone_snd;

        /**  After the user guesses correct play applause **/

        atrack when assertion = (quit != 1)
                until rule = function_return:"finale"
                using Applause_snd;
        /*
        ** play the sound corresponding to the difference
        ** after a guess has been made
        */

        dtrack musical_diff
                when  rule = prog_begin
                until rule = prog_end
                using Flute2_snd;

end guessing_game;
end auralspec.
```

Figure 2.3 A Sample LSL Specification.

1. `atrack when assertion` = (x<y || p≥q)
   `until rule`=while_statement_exit `using` Fill_snd;

2. `atrack when rule`=while_statment_enter
   `until rule`= while_statement_exit `using` Flute_snd;

In (1) the Fill_snd is emitted when the assertion is violated until a while statement is exited. Example (2) results in a Flute_snd being played for the duration of all while loops.

Playback synchronization controls the playback of sounds during program execution. The `syncto` command is used for setting the synchronization mode. There are two synchronization modes: program or metronome. In the program mode, playback is synchronized to the program. In the metronome mode playback is synchronized to a global metronome.

A complete sample specification is shown in Figure 2.3.

timed event, and any combination of the above four. Events are described in detail in Section 4.3.

Mapping of an event to sound is achieved by the use of the `notify` command. The `notify` command specifies the event, the scope of the event, and the sound to which this event is mapped. Examples of the `notify` are given below:

1. `notify` `rule`=while_body_begin `using` Wbb_snd;

2. `notify` `assertion` = (x<y || p≥q) `using` Failure_snd;

In (1) while_body_begin denotes a syntactic entity and Wbb_snd denotes a predefined sound for the while_body_begin. In (2) a predefined failure sound is emitted when the assertion is violated.

There are applications wherein changes to values of variables need to be monitored. To obtain data dependent auralization, LSL provides the `dtrack` command. The `dtrack` command specifies the variables to be tracked, when to start tracking, when to terminate tracking, and the sound to be used during tracking. Examples of the `dtrack` command are given below:

1. `dtrack` *speed* `using` Speed_snd;

2. `dtrack` *speed* `when` `rule`=while_statment_enter
   `until` `rule`= while_statement_exit `using` Speed_snd;

In (1) the source code variable *speed* is tracked using the predefined Speed_sound. In (2) the variable *speed* is tracked only within a while loop construct.

An activity is a sequence of actions between two events. An activity begins at the occurrence of an event and ends at occurrence of a later event. LSL allows specification of tracking arbitrary activities using the `atrack` command. The `atrack` command specifies when to start generating sound, when to terminate generating sound, and what sound to generate. Examples of the `atrack` command are given below:

6. Compilation

   The instrumented source code is compiled via a standard complier and an instrumented executable is created.

7. Execution

   The instrumented executable is run and MIDI [1] information is output to the appropriate device to generate the sound. Section 3.4.1 `midi` describes midi in detail.

2.3 High Level Component Descriptions

During the transformation process components are manipulated to obtain an instrumented executable. This sections describes each of the generated components. Detailed explanations of each component can be found in Chapter 3.

2.3.1 The LSL Specification

ASPECS are written in the Listen Specification Language (LSL) described in Section 3.1. For a complete and detailed description of the LSL language refer to the the technical report [BM93]. Additional literature is also available regarding the LSL language [BM294].

Each specification is composed of one or more specification modules. LSL is a typed language which consists of constants, variables, and types. LSL attributes aid in specifying various properties of notes and patterns which make up sound specifications. A useful characteristic of LSL is its ability to specify events to be auralized. A programmer may formulate an event in terms of program position, data, and time as described earlier. Events can be specified in five ways. One may specify events in terms of a general syntactic entity, a special syntactic entity, an assertion, a relative

[1] MIDI is an acronym for Musical Instrument Digital Interface. It is a serial interface to connect computers and musical instruments

source code. Compilation compiles the decorated source code and links to Listen libraries. Execution generates the MIDI data which generates sound.

Enumerated below are summary descriptions of and components produced by the Listen processing phases. These are presented in order of execution. Details regarding each of the process phases appear in Chapter 4.

1. Editing

   Using any text editor, the source code and the LSL specification components are generated.

2. LSL parsing

   The ASPEC is parsed generating the auralization and sound database components. The auralization database contains an internal representation of the specification. The sound database contains all of the sounds available to Listen.

3. C parsing

   A parse tree component is generated from the source C code. Code reuse was utilized from the ATAC project at Bellcore. A part of ATAC [HL90] was modified to serve as a preprocessor.

4. Decoration

   Given the parse tree and the auralization database, decoration occurs. Events from the auralization database are located in the parse tree and the tree decorated appropriately. Decoration involves inserting C code to generate sound into the parse tree.

5. Deparse

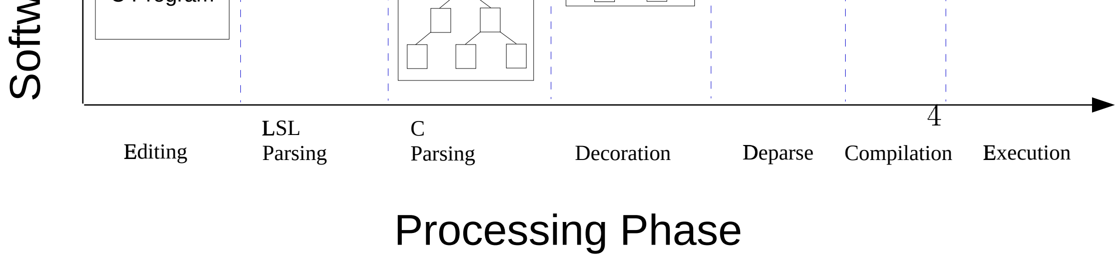
   The instrumented C source code is produced by a process known as deparsing. Deparsing involves traversing the parse tree thereby reconstructing the instrumented C source code.

event is located in the program occurrence space defined by time, position, and data. Figure 2.2 depicts this relationship. A detailed explanation of the language and occurrence space is given in Section 3.1.

Figure 2.2 Occurrence Space Characterization of LSL Events

2.2 High Level Process Phase Descriptions

The Processing phases control the transformation of source code and related specification to instrumented executable. The program source and LSL specification components are provided as input to the process which is controlled by the lslCC script. These files are editing using any text editor. LSL preprocessing parses a specification and generates the auralization and sound databases. C preprocessing generates a source code parse tree. Decoration instruments the specification events within the parse tree. Deparse takes as input the decorated parse tree and generates decorated



2. LISTEN: A HIGH LEVEL SYSTEM DESCRIPTION

2.1 The Listen Environment

Developed as a general purpose environment, Listen provides automated code instrumentation to investigate the use of sound in computing environments. The Listen software components and processes are shown in Figure 2.1. Certain data structures, files, and data interfaces are defined as software components. The processing phase performs transformations on the given components. The source file and specification components are provided to lslCC which drives the transformation process.

Figure 2.1 A High Level Component-Phase View of Listen Architecture

A programmer creates an auralization specification (ASPEC) using LSL (Listen Specification Language). The ASPEC maps specified program events to sound. An to provide a general purpose tool that is applicable in several fields of research related to sound in computing environments.

Many of the tools developed so far require programmers to manually locate and decorate program events. This makes the experimentation process slow and tedious. It is a goal of Project Listen to separate the sound specification from the source code and automatically perform the location and decoration of program events.

1.3 Organization

The architecture of Listen is discussed in Chapters 2,3, and 4. This architecture discussion is presented in terms of software components and processing phases which manipulate these components. A high level description of Listen is given in Chapter 2. A detailed description of Listen components appear in Chapter 3. The processing phases are detailed in Chapter 4. Application and experiences with Listen are presented in Chapter 5. Finally Chapter 6 presents the summary and conclusions of Project Listen.

Francioni and Jackson propose that sound offers an alternative form of investigation to simply using multiple graphical and textual views for studying the behavior or a program. They found general sound to be effective in depicting certain patterns and timing information related to the behavior of programs [FJ92].

Edwards suggests that audio can be used to develop computer human interfaces for blind users. The introduction of complex displays have rendered useless the speech generated interfaces. He suggests representing a mouse based interface by musical tones and synthetic speech to assist visually impaired computer users [Edw89].

Brown and Hershberger developed a tool Zeus for algorithm animation. They added sound capabilities into the application and had positive preliminary experiences using audio in algorithm animations for reinforcing visuals, conveying patterns, replacing visuals, and signaling exceptional conditions. They believe people can hear relations in data that are never seen or displayed. Because sound is intrinsically time dependent, it is very effective for displaying dynamic phenomena, such as running algorithms [BH92].

A common statement in much of the research regarding aural computing environments is the need for additional research and toolkits. Applications that realize the full potential of sound will require the ability to make fairly complex manipulations of sound and an imaginative use of sound effects [Gav89]. This will require the use of several new tools or environments. Blattner [Bla94] states that the increased capability to reproduce sounds and the development of toolkits to generate sound have stimulated new research efforts in nonspeech audio.

The tools developed have been specifically tailored for a special area of investigation. For example, Infosound was developed to create and store musical sequences and special sound effects and to associate these sounds to events in an application program. However it was not developed as a general purpose tool and thus the sound mappings had to be customized for each program [FJ92]. Zeus supports sound in algorithm animations only [BH92]. LogoMedia [DBO93] supports research into program behavior and aural debugging for Logo programs. It is a goal of Project Listen

1. INTRODUCTION

1.1 Definition

It is desired to develop an environment that is applicable in several fields of research related to sound computing. The environment should provide an automated mechanism for mapping program events to sound. The Listen environment has been developed to meet these requirements. This thesis describes the underlying rationale, architecture of, and experience with the Listen environment [1].

1.2 Related Work

Research into computing environments which use sound has been increasing in the last several years. Some areas of interest include auditory display, aural debugging, program auralization, and data sonificiation. Forums for discussion are being organized as well. In 1993 an Association of Computing Machinary Special Interest Group on Sound (or ACM SIGSOUND) was established to discuss issues related to sound in the computing[IEE94]. The International Conference on Auditory Display was established to address related issues.

As the sound production software and hardware becomes increasingly available, developers are begining to use complex sounds to convey more information than just an interrupt beep[Gav86]. William Gaver proposes the use of Auditory Icons to provide a natural way of representing dimensional data as well as conceptual objects. Sounds might provide information about the status of background processes, the number of links in a networked environment, and other factors of the computer environment [Gav89].

[1] In this thesis the term Listen refers to a complete environment for program auralization.

ABSTRACT

Boardman, David Bradley. MS, Purdue University, August 1994. Listen: An Environment for Program Auralization. Major Professor: Aditya P. Mathur.

The use of sound in computing environments is a growing field of research. Project Listen contributes to the field by providing a generic programming environment (Listen) that is applicable in several areas of research. These include aural debugging, program auralization, auditory display, simulation, and sonification. Listen separates the sound specification from the source code and automatically locoates and instruments program events. The language LSL ((L)isten (S)pecification (L)anguage) has been designed and implemented to support the desired generality and automation criteria. Specifications written in LSL and included in the program to be auralized are preprocessed by an LSL preprocessor. The preprocessed program when compiled and executed generates MIDI or voice data sent through a MIDI interface to a synthesizer module, or via audio channels, to an audio processor, which transforms the notes or voice into audible sound. LSL has the generality to specify auralization of a variety of occurrences during program execution. It derives its broad applicability from its few generic elements that when adapted to any procedural programming language, such as C, C^{++}, or Ada, enable the writing and use of LSL specifications for auralizing sequential, parallel, or object oriented programs in that language. This thesis details the design of the Project Listen environment and demonstrates the feasibility through the implementation of a minimal working subset of the environment.

Figure Page

Figure | Page

LIST OF FIGURES

LIST OF TABLES

Page

TABLE OF CONTENTS

DISCARD THIS PAGE

ACKNOWLEDGMENTS

A life is unique in the way it is guided and formed by experiences. Without many diverse peeple and influences in my life, the listen project would have never been realized. I owe special thanks to my musical, technical, and creative influences.

I thank Will Montgomery and Neil Herzinger for helping me explore my musical interests growing up. I would especially like to remember Mr. William Montgomery who exposed so many young individuals to the world of high tech computer music production.

Many thanks go to Geoff Peters, Mark Crosbie, Dan Schikore, Taimur Aslam, Geoff Greene and Vivek Khandelwal who provided essential feedback during the development of Listen. Additional thanks go to my committee members Dr. Jorg Peters and Dr. Vernon Rego for their assistance.

Without the unconditional support of Dr. Aditya Mathur I would not even be here. He has provided the opportunity and support for which I am forever grateful. Most important I thank my mother and father for giving me the support and desire to accomplish my dreams.

In dedication to all those who believe in the unique.

LISTEN: AN ENVIRONMENT FOR PROGRAM AURALIZATION

A Thesis
Submitted to the Faculty

of

Purdue University

by

David Bradley Boardman

In Partial Fulfillment of the
Requirements for the Degree

of

Master of Science

August 1994