A PORTABLE SYSTEM FOR DATA SONIFICATION

BY

TARA MAJA MADHYASTHA

B.A., Rutgers State University, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGMENTS

I wish to thank my research advisor, Dan Reed, for all of his guidance and assistance, and especially the infinite patient revisions and criticisms of all my text. This work would have been impossible without his unwavering encouragement and trust in my ideas.

I am also extremely grateful to Tom Birkett, who wrote the first version of the sound software for the SPARC audio device, for his help and inspiration. I also owe a great deal to Brian Totty for building a serial to MIDI interface and bringing in his own synthesizer, convincing everyone of the potential of MIDI for sonification and playing Christmas carols.

I am indebted to Joan Francioni and Jay Jackson for showing me their auralization work, which made me think about sound in a very different way.

Thanks to all the members of the Picasso group, past and present, for their assistance and friendship. In particular, thanks to Mustafizur Rahman for working so hard to provide traces from the Choices operating system. Thanks to Ruth Aydt, for (among other things) saving me from certain eviction from my first apartment. Thanks to Lawrence Rauchwerger and his army of guardian angels for a sympathetic ear and some of the best advice I have ever received. Thanks to Dave Jensen for constantly coming to my rescue with LaTeX help and thesis tips. Thanks to Chris Elford, Dave Kohr, Bradley Schwartz and Keith Shields for putting up with all my obsessions and complaints.

My friend David Zimmerman gets credit for the part of East Coast Voice, although it is unlikely to earn him his big break.

And finally, I thank my family for their constant love and support. And in particular, my parents, for believing in me unconditionally.

This work was funded in part by National Science Foundation grants NSF CCR86–57696, NSF CCR87–06653 and NSF CDA87–22836 (Tapestry), NASA ICLASS Contract No. NAG–1–613, DARPA Contract No. DABT63-91-K-0004, and by grants from the Digital Equipment Corporation External Research Program and Apple Computer.

TABLE OF CONTENTS

Chapter

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

Humans use incidental and ambient sounds to assimilate information about the world around them. Movies, television, and video games rely on the ability of sound to augment the information we derive from the visual dimension. For example, the sound of footsteps growing louder from beyond the edge of our field of vision implies that someone is approaching. The careful listener might be able to deduce a person's appearance or identity from the tapping of shoes or the rustling of clothing.

Although still underutilized, sound is not new to the computer interface. Gaver [10] describes a representation scheme using sound in the SonicFinder, an auditory interface to the Apple Macintosh that couples auditory icons with graphical feedback. Gaver suggests that sound should be used in the computer interface as it would be used in the real world. In the SonicFinder, commonplace sounds are naturally mapped to events in the user interface; selecting a window makes a clinking sound, dragging a file produces a scraping sound, and emptying the trash results in a crunch. The frequency, timbre, or other characteristics of the sound are used to convey additional information about the objects being manipulated (for example, the frequency of a scraping sound might indicate the size of the file being dragged).

This concrete representation is in contrast to the approach taken by Blattner, Sumikawa, and Greenberg [2], who describe a method for encoding information as auditory icons, or earcons, based on rhythmicised sequences of notes, called motives. Such motives bear little resemblance to "real world" sounds, but once learned, depict abstract concepts.

The abstract objects in current graphical user interfaces bear little resemblance to things in the "real world." To use sound effectively, either we must design with sounds that correspond to the new objects, or hope that interfaces become more realistic. The approaches taken by Gaver and Blattner *et al* both require learning what sounds mean in the context of the computer interface, but the rewards of the effort are potentially great. Even in the real world, the meaning of many commonplace audio signals, such as smoke alarms, police sirens, and human speech must also be learned.

Given the proven ability of sound to convey abstract concepts, it is interesting to consider it as a medium for presenting data. Sound can possibly highlight characteristics of data that cannot easily be seen, much in the same way that a movie soundtrack conveys information complementary to the imagery. The elements of sound (e.g., pitch, volume, duration, location and timbre) can be used in the same way that visual elements (such as color, form, and line) are manipulated to present and analyze data in visual displays. *Sonification* refers to the use of sound to present data, the auditory equivalent of visualization.

There is a natural relationship between numbers and music, made even more obvious by digital audio media. On a compact disk, for example, music is represented as a stream of numbers, or samples. One has only to write a program to generate the samples for a sine wave to see the mathematical connection between numbers and sound. It seems natural to seek patterns in data by the inverse process, using data from various sources to produce sounds and searching for musical patterns similar to those we know.

Sonification probably has its earliest beginnings in musical analog, a method of composition that maps measured data to musical parameters. This is the technique that was used in Heitor Villa-Lobos's *New York Skyline Melody* (1940), based on the 1940's New York skyline, and Charles Dodge's *Earth's Magnetic Field* (1970). Robert Keefe describes in detail the creation of such a musical analog based on planetary orbits [13]. Latitudinal coordinates of the planets are mapped onto a pitch letter name, an accidental, and an octave displacement, yielding a musical score. This mapping function is a product of the composer's intuition and creativity.

The point where a musical analog ceases, for a listener, to "represent" the data is not obviously demarcated. It depends on the complexity of the mapping, and the number of user–variable parameters. This is almost like saying that a photograph is more "realistic" than an abstract painting of the same object. The difference is that we have little idea about what constitutes a "sonic photograph." Most people can identify a guitar from a photograph of a guitar, an abstract painting of a guitar, or from a recording of the sound produced by a guitar; but it is not so easy to represent a rock with a sound. The problem of presenting abstract ideas with sound is even harder. When we consider the presentation of data sets with sound, it is apparent that the sonic equivalents of scatter plots and bar graphs do not exist. The algorithmically generated computer graphics used for visualizing data are drawn from centuries of artistic efforts to emphasize recognizable patterns through visual mappings using color, form, and line. Sonification, which is to musical analog what visualization is to art, is in a comparatively nascent state of development.

1.1.1 Previous Work

Systematic experimentation with the use of sound to present data has been underway since the early 1980's, when technology made applied research in this area economical. Bly's study [3] examined possible aural representations of multivariate, logarithmic, and time–varying data. Elements of data sets were mapped to pitch, volume, duration and waveshape, so that each data sample produced a single note. Pitch was varied over four octaves in a musical scale, volume was varied over twelve settings from soft to loud, duration ranged over 201 levels from 50 milliseconds to 1050 milliseconds, and waveshape ranged over 128 intervals from a pure sine wave to a random waveform.

In informal studies using these four parameters to encode four–dimensional data sets, Bly found that people were able to discriminate among the data sets with 94 percent accuracy, comparable to traditional visual methods. Six–dimensional data could be sonified by adding two additional parameters: six variations on the attack envelope and 128 variations of overtone waveshape.[1] In experiments with two sets of randomly generated six–dimensional data that were distinct only in six–space, Bly demonstrated that 64.5 percent of the samples could be correctly placed in a set based on sound alone. Sixty–two percent of the samples could be identified from two dimensional plots of each pair of variables, and 69 percent could be identified from a presentation combining sound and graphics. In summary, Bly's experiments showed that sound was at least as useful as traditional graphical techniques in categorizing data samples.

[1] See §1.3 for a review of sound characteristics.

This mapping technique was later used by Smith *et al* [22] and Rabenhorst *et al* [17] in concert with visual data exploration environments. Smith *et al* incorporate the ideas of basic data mapping with an iconographic display where each data sample is represented by an icon on the screen. Each sample may be mapped to sound attributes, including stereo balance, which are explored by mouse cursor movement over the iconographic display. Rabenhorst *et al* take a similar approach, mapping three–dimensional vector gradients in an exploratory environment to de–tuning and stereo balance of each of three notes in a musical chord.

Scaletti and Craig [20] developed a preliminary set of standard, high–level sonification tools that they used to add time–varying, data–driven sound to scientific visualizations. The fundamental tool, a Shifter, interprets a stream of data as samples of a waveform. The other tools use each value in the data stream to vary one or more parameters of another arbitrary sound, such as frequency, amplitude, and stereo position. This approach is extremely flexible, as the data steam can control any facet of a sound; indeed, it can be interpreted as the waveform itself. However, because the sound synthesis is software controlled, specialized digital signal processing hardware is required to achieve real–time output.

Francioni and Jackson [8, 12] have experimented with sound specifically to depict parallel program behavior, using relative timing information in event trace files to control timing and duration of sound events. Mappings of this nature reveal patterns in program execution (such as interprocessor communication patterns and processor busy/idle times) as recognizable rhythms and melodies. These mappings are used to generate MIDI [15] files that can be played on instruments conforming to the MIDI protocol. This protocol defines a standard format for representing the components of sound (e.g. pitch, duration, volume); many electronic instruments can play these files.

Two things are evident from this brief survey of approaches to the use of sound and data sonification. First, the essence of a sonification is the mapping of data to sound parameters. To "display" data with a sound, some characteristic of that sound must convey some aspect of the data. Second, the potential mappings are determined by the available sound hardware. In the next section we examine approaches to sound synthesis that affect the kinds of sonic mappings that can be created.

1.2 Sound Synthesis Approaches

To sonify data, some data dependent algorithm must be used to produce sounds. The approaches to sound synthesis, as they affect the design of a sonification system, may be classified in terms of control over the algorithms used to generate sounds, and whether sounds can be generated in real–time.

The most general technique relies on software to generate the samples of a waveform. These samples are then made audible using a digital to analog converter. Historically, software synthesis techniques have been the focus of computer music research. Hence, a plethora of software synthesis techniques exist [6, 16]. Given the rich history of software–controlled sound synthesis, anything more than a cursory description is beyond the scope of this thesis. The strength of software sound synthesis is its device independence; almost any algorithm can be used to produce sound samples. Although this makes software synthesis by far the most flexible means of creating sound, especially when creating sonifications, the time required to synthesize samples for complex sounds is very large, often requiring supercomputers to reduce the processing time to manageable levels.

The synthesis time can be reduced with special–purpose digital signal processing hardware. The Capybara digital signal processor (based on nine Motorola 56001 digital signal processors running in parallel) was used by Scaletti and Craig [20] to support software synthesis in real–time. The Capybara system is controlled by a functional, data stream language called Kyma, allowing the user to graphically combine sound synthesis building blocks and map data to sound. Although this is a flexible platform for building sonifications, the hardware is specialized and expensive.

Another approach to sound generation is real–time hardware sound synthesis. Although embedding sound synthesis algorithms in hardware potentially limits the space of possible synthesized sounds, the rich set of sound attributes practically permits synthesis of a wide variety of sounds. Moreover, the hardware specialization greatly reduces costs and promotes standardization. Indeed, almost all commercial, hardware sound synthesizers support the MIDI (Musical Instrument Digital Interface) protocol, a communication standard that allows sound synthesizers to be interconnected and computer controlled. MIDI commands support such options as selection of timbre, volume, and sustain. Furthermore, even low–end professional synthesizers offer a wide range of features and controls that can be used for data sonification.

In essence, a MIDI synthesizer represents a compromise; it sacrifices complete control of the sound waveform for an inexpensive, flexible, high–level interface to sound synthesis. Moreover, most synthesizers provide a number of voices (instruments), and parameters such as volume, location, and pitch can be controlled dynamically. Although a software synthesis toolkit can provide similar levels of abstraction and ease of use, taking full advantage of software synthesis' flexibility requires a comprehensive knowledge of software synthesis techniques and computer music.

Despite the advantages of MIDI synthesizers for sonification, they are not without problems. Most notably, identical MIDI commands are unlikely to produce the same results on two different synthesizers; some synthesizers support only a subset of the MIDI protocol. Hence, some MIDI synthesizers may ignore commands sent to them, or they may interpret the commands differently depending on context (e.g., two MIDI synthesizers might have non–intersecting sets of voices). Finally, many of the most interesting and powerful features of a particular synthesizer are often vendor and device–specific.

In summary, software synthesis provides the ultimate in flexibility; data can be mapped to sound using any user–specified algorithm. In contrast, MIDI synthesizers support a more limited repertoire of data–to–sound mappings. However, hardware to support real–time software sound synthesis is expensive and non–standard. MIDI synthesizers provide an excellent compromise between function at reasonable cost, but many of the most powerful features of a particular vendor's MIDI synthesizers are not portable to those of other vendors.

1.3 Sound Characteristics and Sonification

A sound is the sensation of pressure variations in the air caused by a vibrating source. The pattern of these variations over time determines the waveform of a sound; waveform determines timbre. Figure 1.1 shows two sample, sinusoidal waveforms; the first is represented by a a dark solid line, and the second by a dashed line. In both cases, the y–axis represents pressure, and the x–axis represents time. The measure of pressure is the amplitude of the sound; it is perceived as loudness. If the waveform is composed of a repeating pattern, as these waveforms

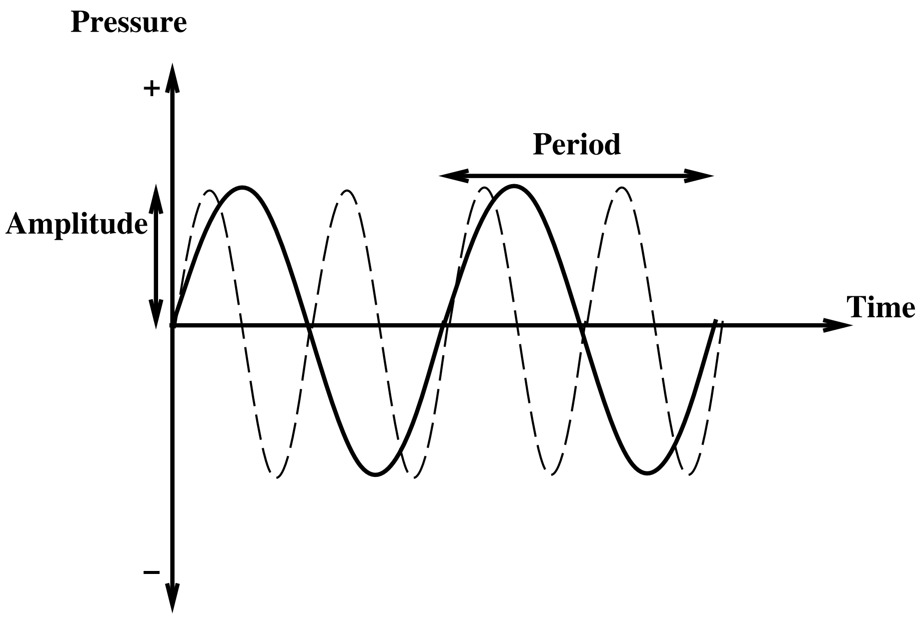

Figure 1.1: Periodic waveforms.

are, they are said to be periodic, and the smallest repeating pattern is a cycle. One period for the waveform shown in Figure 1.1 as a dark line is marked by the horizontal arrow. The number of periods per unit of time is called the frequency; it is discernible as the pitch of the sound. The waveform depicted by the dashed line has a frequency twice that of the other.

Although there are only a small number of sound characteristics, they can be manipulated to produce a rich set of sounds. One can change the amplitude, frequency, or shape of a sound wave, and each of these variations has profound psychoacoustical effects. Moreover, no single characteristic of sound can be considered alone; complex inter–relationships between sounds make sonification design interesting. Furthermore, computer–aided sound synthesis has expanded the sense of sound exploration beyond renditions of “classical” sounding instruments and compositions. For example, *Klangfarbenmelodie*, or a melody composed of different tone colors, is far easier to achieve with synthesized sounds under computer control than orchestrally. The same complexity that distinguishes the sound of a violin from a sine wave makes synthesis and sonification challenging.

1.3.1 Pitch

Logarithmic changes in the frequency of a waveform are perceived as linear changes in pitch. Despite the inherent linear to logarithmic translation, pitch is one of the most intuitive ways of expressing relative magnitude. Higher notes, or frequencies, can be used to represent larger data values, in the same way that the color red might display a larger magnitude than blue.

It is possible to map data to arbitrary frequencies, but mappings to notes in familiar musical scales are more likely to resolve themselves into common, recognizable patterns. Moreover, although the human ear can detect frequencies between 20 and 20,000 Hz, we are most sensitive to frequencies between 20 and 4000 Hz. This smaller range of frequencies corresponds approximately to the range of notes on a piano keyboard. Given the wide range of distinctly audible notes, pitch has a large bandwidth for information content.

Pitch is not without its disadvantages. A sound must last a certain number of cycles for a frequency to be perceived; lower notes, with longer periods, must be heard for longer durations than higher notes. Although large, the range of pitches in a traditional chromatic scale provides limited resolution; there are 12 notes in each of seven octaves within a reasonable audible range. As with gradually shaded colormaps, adjacent values can be difficult to distinguish.

1.3.2 Loudness

The perceived loudness of a sound is a response to its amplitude, but this response is influenced by many other sound characteristics. In particular, frequency and timbre have a large effect on how the loudness of a sound is perceived, and the minimum detectable change in amplitude. For example, lower frequencies in a "musical range" sound softer than higher frequencies. Furthermore, the frequency and amplitude relationships among multiple notes can affect their perceived loudness. A soft note played immediately after a loud note will be perceived differently than if heard alone.

For these reasons, loudness is not really a linear scale. Different musical instruments will have different perceived loudnesses, and if the volume is reduced overall, as by a global volume control, variations in loudness will be more difficult to detect. Hence, loudness should be used with care when it is a variable in a sonification incorporating different timbres. Because our ability to detect variations in loudness is small compared to the differentiability of pitch, loudness by itself is probably most effective for representing a parameter with at most two or three values, (e.g., small, medium, and large). Soft tones are naturally associated with smaller magnitudes than loud tones.

1.3.3 Timbre

Timbre is equivalent to the waveform of a sound. It is timbre that determines whether one hears a trumpet or a piano, even if they are playing the same note.

The capabilities of the sound synthesis mechanism determine the extent that timbre can be used to present data. Software sound synthesis can be used to interpret a data set as a waveform, which has its own characteristic timbre. For example, a purely random data set would be perceived as white noise. Because we can easily distinguish a transition from random noise to regular patterns, this facility alone might be useful in detecting regularity over large data sets. If using data as a waveform yields no insight, or if the number of samples is too small for it to be revealing, the data can be used to manipulate another waveform, thus creating a controlled, secondary effect on the timbre. It is impossible, however, to support this kind of software control in real–time without substantial computer processing power. Parallel digital signal processors have a substantial advantage over other sound hardware in this area. In contrast, most synthesizers today provide limited timbre–editing capabilities.

Even if one can only work with a fixed palette of voices, timbre can be useful for drawing distinctions between several concepts that do not have a numerical ordering. Most people can readily recognize instruments from different families (e.g., woodwind, strings). Associating a timbre with a concept is similar to associating a voice with a person. This idea is well known to musicians; consider Prokofiev's *Peter and the Wolf.*

1.3.4 Location

Location is the perception of the placement of a sound source. Unless one is in a room surrounded by speakers, the range of reasonably manipulated locations is limited to software synthesis techniques that can manipulate the virtual, or perceived, sound source. On synthesizer hardware, this is usually the stereo balance control. Implementation of a two–dimensional stereo display, which would allow horizontal and vertical positioning of a sound source, is difficult in real–time. One technique, developed at the NASA Ames Research Center, can position up to four sound sources in three–dimensional space in real–time over headphones [25]. This kind of device can greatly increase the potential applications of location in sonification. However, even the one–dimensional display provided by stereo balance (where the sound source can be positioned in discrete steps from far left to far right) can be useful. Stereo balance is inherently spatial, and can provide locational cues corresponding to visual charts, matrices, and other such displays.

It is difficult to discern fine differences in balance; however, balance can be effectively used to categorize two or three discrete values, especially if the relationship between the values is a non–ordered one. A sound source located to the right is not easily perceived as "less" than a source in the left.

1.3.5 Rhythm and Duration

Patterns of sounds are perceived in terms of notes and rhythms. Music is usually organized around a periodic event rate, or pulse. This usually varies, non–coincidentally, with the human heart rate. It is because of an organizing beat that listeners can anticipate future events and remember more easily what has been heard. Short–term memory limits the number of events that can be immediately remembered; in music it is generally accepted that this can extend upward to fifteen events per second [16]. Pulse is perhaps most important in the context of synchronizing human performance. In the context of sonification, it is interesting as a means to generate musical patterns.

Duration by itself could be used to provide insight to a data set where most numbers could be mapped to short, uniform lengths, with few exceptions that stand out as long, sustained notes. It is difficult to gain quantitative information from note durations unless they are sufficiently distinct. Duration can also be used to mark the lifetime of timestamped events; it is easy to detect overlapping notes, and to get a qualitative feel for the relative durations of different sounds.

1.3.6 Musical Considerations

Most people have a good memory for musical patterns [14]. Typically, the melody of a new song is the first thing remembered and the last thing forgotten. However, exactly what constitutes a melody has been the subject of a fair amount of research [16]. For example, a sequence of notes from a single instrument is more likely to be perceived as a melody than the same notes from different instruments. Certain patterns of notes are more "melodic" than others. For this reason, the choice of a scale or starting pitch can be particularly significant for a sonification designer, as one choice might generate less melodic, and thus less memorable, patterns than another [9].

1.4 Sonification Goals

Previous work in sonification suggests that a general–purpose mechanism to map parameters of data to characteristics of sounds on available sound hardware is essential for experimentation. Much of sonification's embryonic state is attributable to this lack of software standards and common hardware interfaces (e.g., there is no sonic equivalent of the X Window System protocol, nor do all workstations include similar sound hardware). Ideally, one would like to accompany visualizations with real–time output of high quality digital sound. However, general purpose digital signal processing hardware, while extremely flexible, is also expensive. At the other extreme, the audio chip in the Sun SPARCstation[2] is a nearly ubiquitous hardware platform; unfortunately, its sound fidelity is very low. The Sun SPARCstation audio device can play back digital audio data at a rate of 8000 samples per second with 12–bit precision, resulting in sound quality equivalent to standard telephone service.

A better compromise is instruments that conform to the Musical Interface Digital Interface (MIDI). Using a variety of built–in synthesis techniques, most MIDI instruments can play notes in many voices (timbres), ranging from traditional voices like a piano or trumpet to interesting sound effects, such as rush–hour traffic in the city. MIDI commands are just a few bytes long, so it is easy to control output in real–time.[3] A wide range of sound mappings can be efficiently produced by connecting one or more of these instruments to a workstation. However, MIDI is a very general protocol, and most manufacturers use "system exclusive," or device–dependent messages to control the unique features of their instruments (such as the ability to create new voices). This makes it extremely difficult to play a composition written for one MIDI instrument on another.

---

[2] Sun SPARCstation is a trademark of Sun Microsystems Incorporated.

[3] In contrast, the Sun SPARCstation plays sampled audio data, which can require a long time to synthesize and a lot of software coordination to correlate.

In light of these software and hardware constraints, we have identified the following criteria that we believe a portable sonification system should satisfy.

- Sound hardware should be manipulated by a network server, so that the interface to local and remote sound devices is the same.

- The interface to a particular sound device cannot be too abstract — it must reflect the actual hardware functions. Sound devices vary greatly in their functionality, and attempts to create an all–encompassing set of commands for all types of sonic hardware are bound to be either restrictive or vague.

- The core of a sonic display, analogous to a visual graph or chart, should be both configurable and able to utilize all available capabilities of a sound device.

- It should be easy to create a wide range of sonic displays that operate in real–time, and that can be synchronized with visual displays.

The rest of this thesis describes Porsonify, an attempt to satisfy these goals. This thesis is organized into six chapters. Chapter 2 introduces the high–level design of Porsonify. It argues that Porsonify provides an environment where it is easy to create sophisticated sonic displays of data using available sound hardware. Chapter 3 describes the implementation of Porsonify and the mechanism to create sonifications. Portability is achieved by separating the sound hardware interface from the algorithm that creates sound to display the data. Chapter 4 illustrates the integration of Porsonify into a visualization system, creating an environment for coordinated sonification and visualization. Chapter 5 describes sonification experiments using Porsonify. We conclude in Chapter 6 with a summary and directions for future research.

Chapter 2

Porsonify Design

2.1 Hardware and Software Considerations

As we observed in Chapter 1, there are several possible approaches to sound synthesis that could be used as the basis for a sonification system. By far the most flexible approach is based on a powerful digital signal processor and associated software. As described in §1.1.1, Scaletti and Craig built a set of sonification tools on such a platform. This combination of special–purpose hardware and synthesis software (Capybara and Kyma) provides complete software control over sound in real–time. Its major drawback is expense and lack of generality. It is unlikely that those interested in casual experimentation with sonification will acquire such hardware. For a sonification system to be accessible, it should not depend on expensive, specialized hardware.

Music synthesis software can operate on other computing platforms without any digital signal processing hardware, but with loss of real–time control and possibly output quality. Audio samples for a sonification can be generated in advance and later replayed, but the software effort to do so might be very great compared to the rewards. For example, the audio hardware in the Sun SPARCstation, an AM79C30A Digital Subscriber Controller chip, is a common sound hardware platform. This chip contains analog to digital and digital to analog converters; however, the only functionality supported by the SunOS device driver is recording and playback of digital audio data at 8K samples per second with 12–bit precision on one output channel [24]. The resulting sound quality is equivalent to standard telephone service. In contrast, the Capybara sample rate is adjustable up to 28K samples per second [20], and CD quality sound is

44.1K samples per second at 16–bit precision per channel. Even at such a low sample rate, the Sun SPARCstation cannot synthesize and play complicated sonifications in real–time.

Real–time control of sound synthesis is a necessity. The advantage of real–time control is that sonifications can be generated at the same time as equivalent visualizations. Sonifications by themselves can be useful, but a consequence of their still–experimental nature is that they are best heard in connection with corresponding graphics. It is much harder to replay a previously synthesized sonification and synchronize it with the corresponding visualization than to generate both concurrently. Furthermore, if sonifications require a long time to synthesize, they are unlikely to be used frequently.

These considerations point to the use of synthesizers that support the MIDI protocol. MIDI instruments are relatively inexpensive and compromise some flexibility to provide real–time response. However, a system written for one MIDI synthesizer is not guaranteed work on others. For example, a sonification that uses balance to represent the most significant data dimension would be useless on a MIDI synthesizer without balance control. Thus, the MIDI protocol cannot be viewed as a uniform platform on which to build a sonification system, although MIDI instruments are clearly useful output devices.

The wide range of sound devices and their radical differences in function constrains the possible approaches to writing a sonification system. One can usually assume that every graphics device can draw a line. Unfortunately, no analogous assumption regarding sound device functionality is possible. The common denominator of sound devices is not a set of universally supported commands; the only guaranteed common attribute is that sound devices can make sounds. Although it would be possible to create a sound command language that can express all the functionality of a set of sound devices, lack of sound device standardization, even using the MIDI protocol, would force the revision of the language for every newly announced sound device. Letting the sound description language be the greatest common subset of the sound device functions suffers from another, deeper problem; in an area that demands experimentation, limiting the functions used to create sonifications is counterproductive. From this point of view, it is essential that underlying sound hardware functionality be accessible to the sonification designer while isolating device idiosyncrasies to the greatest extent possible. In Porsonify, sound devices are controlled by sound device servers that rely on a common interface for specifying what commands a particular device supports. This permits construction of a sonification interface that is independent of the underlying sound device; the interface need only understand the format of the sound commands and not their function.

2.2 Porsonify Overview

In Porsonify, access to sound devices is based on a server/client architecture. A *sound device server* specific to each sound device encapsulates all the device dependent code, and it accepts *sonic messages* that manipulate the device. Client applications interact with the sound device servers through a general–purpose network audio access daemon, or *naad*.

When an application program wants to use a sound device on a particular host, either local or remote, it first contacts the *naad* on its local host. The *naad* determines if the sound device is free, and if so, establishes a connection between the application and the sound device server. The sound device server then sends the application a description of all the commands the sound device can support. Thereafter, the application simply manipulates this description to interact with the sound device; the location of the sound device and its idiosyncrasies are hidden.

This design was chosen for several reasons. The first and most practical reason was to allow remote access to sound devices. In an office or laboratory with three workstations and a sound device connected to only one of them, the sound device should be equally accessible to all. Given the relative scarcity of sound hardware platforms, accessibility and the ability to experiment with multiple sound sources are important design goals. The second reason is portability — sound devices are manipulated exclusively through sonic messages, so that client application code needs only a textual representation of the sonic messages (saved in a *sound control file*) that it sends to the device to create a sonification. New sound hardware can be utilized by editing a sound control file to suit the new device. Finally, this scheme allows for concurrency. The *naad* allows multiple processes to access the same sound device server. Support for concurrency permits applications which create several processes to work in concert to create a sonification.

2.3 The Network Audio Access Daemon

Because most applications gain access to a sound device through the network audio access daemon, or *naad*, each workstation that supports the Porsonify interface must start a *naad* —

```
# Sound hardware configuration file for foobar.cs.uiuc.edu
#
#    Description                  port (from /etc/services)
"Sparc Audio Chip"                        sparcaudio
"Yamaha PortaSound MIDI synth"            midiaudio
```

Figure 2.1: A sound hardware configuration file for `foobar.cs.uiuc.edu`.

even if it supports no sound devices. Any application that wishes to access a local or remote sound device must first contact the *naad* on the local host. Specifically, there are two types of requests to which the *naad* can respond.

- Describe the sound devices that are available on the local workstation.
- Initiate a connection to a sound device on a local or remote workstation.

To determine what sound devices are available on remote workstations, an application can contact the *naad* on any remote workstation and ask what devices are available locally – much in the same way that one might call out–of–state directory assistance to get a phone number local to a remote city.

Locally available sound devices are described in a system configuration file that contains a textual description of the sound device and the TCP/IP port name listed in the file **/etc/services** that is associated with the specialized server; Figure 2.1 shows an example of a configuration file for a Sun SPARCstation, called **foobar.cs.uiuc.edu**, with a MIDI synthesizer connected to it. The configuration file specifies that that the SPARCstation audio device can be accessed through the **sparcaudio** port, and the MIDI synthesizer can be accessed through the **midiaudio** port.

As an example, a sound device server that accepts commands for the Yamaha PortaSound PSS–680 MusicStation[1] MIDI synthesizer would listen on port **midiaudio** on **foobar.cs.uiuc.edu**. In general, a host name and port name pair uniquely describes a particular sound device server. After a host name and a port name have been selected (e.g., by querying some host's *naad*) a connection can be initiated to that sound device server.

[1]Yamaha PortaSound PSS–680 MusicStation is a trademark of Yamaha Corporation.

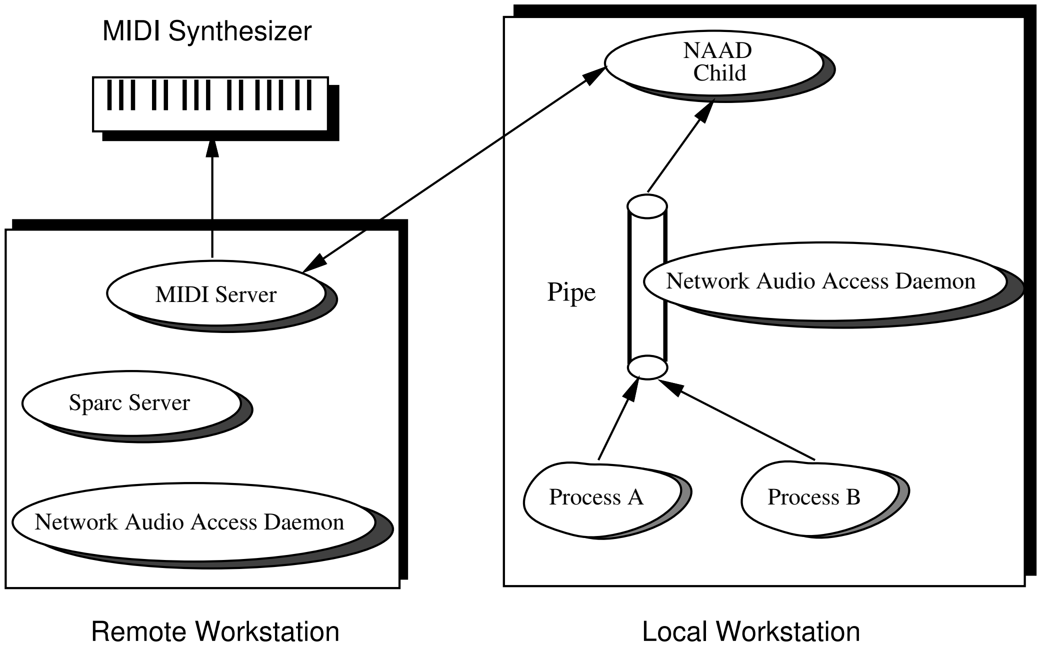

Figure 2.2: Communication between application processes and sound servers.

Clearly, there may be situations where *multiple* application processes on a particular host need to access a *single* sound device. For this reason, the *naad* accepts requests to initiate connections to sound devices. If a connection to the requested sound device is not already open, the *naad* creates a named FIFO pipe and forks. The child *naad* process then establishes a connection to the port associated with the specified sound device. Thereafter, the child process simply reads from the pipe and sends the commands to the remote device. Given the pipe's name, the requesting application process can then open the pipe, locking it before writing commands and unlocking when finished. The *naad* can answer future requests for this sound device from other application processes on the same host by simply returning the pipe name. The child *naad* process dies when all applications have closed the pipe.

Figure 2.2 illustrates a scenario where two application processes on the local workstation (A and B) are accessing a MIDI synthesizer on a remote workstation. The *naad* on the local workstation has forked a child process that manages the communication between the remote sound server for the MIDI synthesizer and the two application processes.

While these two application processes are connected to the MIDI server, no other processes on any host except the local workstation can access the associated sound device. Although not strictly necessary, this restriction simplifies the sound device servers; they simply receive and execute messages. The *naad* provides the necessary coordination between application processes on the same host.

2.4 Sound Device Servers

The network interface, as described above, is the logical level for the abstraction between the application and the software that drives the sound device. This layer of abstraction is designed so that sonifications can take advantage of the capabilities unique to specific hardware with no device–specific code.

Each instance of a sound device has its own server that listens for connections on a TCP/IP port. When a program initiates a connection to a sound device server, it receives a *sound device description* consisting of the following three parts. The *sonic message configuration table* defines the commands the server accepts, the *device defaults* table describes the default arguments to those commands, and the *device parameters* table specifies fixed, valid ranges for the arguments.

As an example, imagine a sound device that supports only one function: it can beep for a specified duration up to ten seconds. The textual representation of the command the server accepts, (i.e., the *sonic message configuration table*, that describes the command to make it beep) looks as follows.

```
"Beep"     doBeep       DURATION
```

The first field is the name of this command. The second field refers to the executable code that the sound device server calls to implement this command. The remaining field is an argument to the `Beep` command. Command arguments are described in the *device defaults* table, which for this simple device would appear as follows.

```
DURATION   INTEGER  "Duration in seconds"  1   MINDURATION  MAXDURATION
```

The first field above is the same key that appears in the sonic message configuration table. The second field is the type of the argument. This is followed by a textual description of this argument, its default value (1 second), and the key names of the device constants that bound the duration. These bounds are delineated in a *device parameters* table, shown below for the simple beep device.

```
MINDURATION   INTEGER   "Minimum Duration"   0
MAXDURATION   INTEGER   "Maximum Duration"   10
```

The first field in each line above is the same key that appears in the device defaults table. The second column is the type of these bounds, integer in this case. The third column is the description of the constant, and the fourth is the constant value.

When an application has received these tables, it knows what kinds of commands the sound device supports and the range of arguments for these commands. In this example, an application can determine that the beeping sound device supports only one command, `Beep`, which accepts one integer argument, `Duration in seconds`, that must lie between a `Minimum Duration` of 0 and a `Maximum Duration` of 10. An application can cause the device to beep for two seconds by sending it a *sonic message* of the form:

```
"Beep" 2
```

Sonic messages, synthesized using these tables, cause commands to be executed on a sound device and are the basic units of activity. Sonic messages may either cause a sound to occur (e.g., play a note), or change some sound parameter (such as volume) that affects current or future sounds.

Examples of some sonic messages for more sophisticated devices might include the following.

```
"Set Volume"       65
"Set Instrument"   piano
"Play Note"        60  1.0
```

These messages are not very descriptive without understanding the associated configuration tables, which specify the range of valid volumes for the `Set Volume` command, a list of instruments for the `Set Instrument` command, and the range of the pitch and duration arguments to the `Play Note` command.[2]

A sound device server listens on a network port unique to the interface for that device. If a host supports two sound devices of exactly the same type, their servers must run on different ports (e.g., `midi1` and `midi2`). However, the software assumes that all ports with

[2] In reality, once the configuration tables have been communicated, messages are identified by number, not name (as shown above).

the same name accept the same (or compatible) sets of commands. Two different port names might map to the same physical device, but manipulate it with completely different interfaces. For example, a workstation might be connected to a MIDI synthesizer via a serial to MIDI converter. The synthesizer might be accessed through the workstation in two ways: by writing MIDI control messages directly to the serial port, and by a UNIX kernel device driver, which can accurately play timestamped MIDI messages. In the first case, commands are executed as they are received. In the second, each command has an extra argument: a timestamp. Commands are executed only when the timestamp matches the time on the MIDI device driver clock. These two interfaces to the same physical hardware would be managed by two different sound device servers. They would synchronize to avoid conflicts, such as ensuring that there is only one active connection to the MIDI device and that the synthesizer is placed in a consistent state after each interaction.

This layer of abstraction can be used to create a general interface to any sound device, without requiring that each device support the same set of messages. Sound devices are so varied in their capabilities that making them conform to one interface would be impossible. Instead, each device advertises only the commands it can process.

2.5 Sonifications

A *sonification* is anything that aurally presents data. It is analogous to a visual display, except that when the display is updated, instead of changing without regard to time in a fixed two–dimensional screen space, it changes over a variable amount of time as a rhythmicized sequence of sounds is played.

Updating a sonification involves sending a stream of sonic messages to one or more sound devices for each data value that the sonification should "display." Data values may be scalars, or a sequence of scalars may be grouped as vectors. Each scalar data value, or each element of a vector can be used as a sonic message argument. For example, a simple sonification of points in a two–dimensional space might accept a stream of vectors of length two that contain the x–coordinate (index 0) and the y–coordinate (index 1). For each input vector, a short note is played whose pitch is determined by the y–coordinate, and whose source location (balance) is determined by the x–coordinate. This kind of sonic display requires no memory of previous data and no knowledge of future inputs.

The purpose of a sonification is to present data. Therefore, the sonification must produce sounds or affect the sounds that are played every time it receives a new data value. The constructs provided in Porsonify for sonification creation are stateless; we did not want to assume any lookahead in the data stream. For example, suppose one wanted to construct a display that would accept an ordered set of data values and play a note corresponding to each value in reverse order. This would require the sonification to record and then process every value it has received. This functionality should be external to the sonification; reversing the order of the data set is unrelated to sound. Assuming we can preprocess the data so that values to be presented can trigger temporally ordered sonic events, writing a sonification can be reduced to determining what sonic messages are sent to what sound devices for each data value.

Creating sonifications is simplified by four major abstractions: *sound control files*, *transformation functions*, *sonic widgets* and *widget control files.* Sound control files are scripts written for a particular sound device that list sonic messages to be sent to that device's server. Transformation functions, which provide simple functional modifications of input data values, are used to choose between sound control files or to vary the parameters of individual sonic messages. Sonic widgets manipulate one or more sound control files and an optional transformation function to create a specific sonification. For example, a sound control file for the SPARCstation audio device might include a command to play a sampled chime. A time display called "Big Ben," which accepts hours and minutes as input, might be constructed using a sonic widget which executes the sound control file once for every hour in the current time if the minute field is zero. Finally, sonic widgets can save their complete configuration information in widget control files. In this way, sonifications can be saved, edited, and reused.

2.6 Creating a Sonification

Figure 2.3 illustrates how the major components of the Porsonify system interact. The sonic widget represented encapsulates two sound control files. The first sound control file contains sonic messages for the server `midiaudiod` listening on port `midiaudio` on the local host. The second sound control file contains sonic messages for the server `sparcaudiod` on port `sparcaudio` on the remote host `somewhere.cs.uiuc.edu`.

In the figure, the application has established connections via the *naad* on the local host to these servers, and has received copies of the sound device descriptions. The *naad* has spawned two children, each of which reads messages that the application writes to designated named FIFO pipes (not shown).

The transformation function in the sonic widget performs some operation in the input data to select one of the two sound control files for processing. Different types of sonic widgets use constituent sound control files in different ways; for example, one widget might use a transformation function to determine how many times to send a single sound control file's sonic messages to a sound server, and another might use the transformed data to determine whether to send the sound control file's messages at all.

The next chapter examines the implementation of Porsonify. The behavior of the *naad*, the construction of sound device servers, and the software components used to create sonifications are described in greater detail.

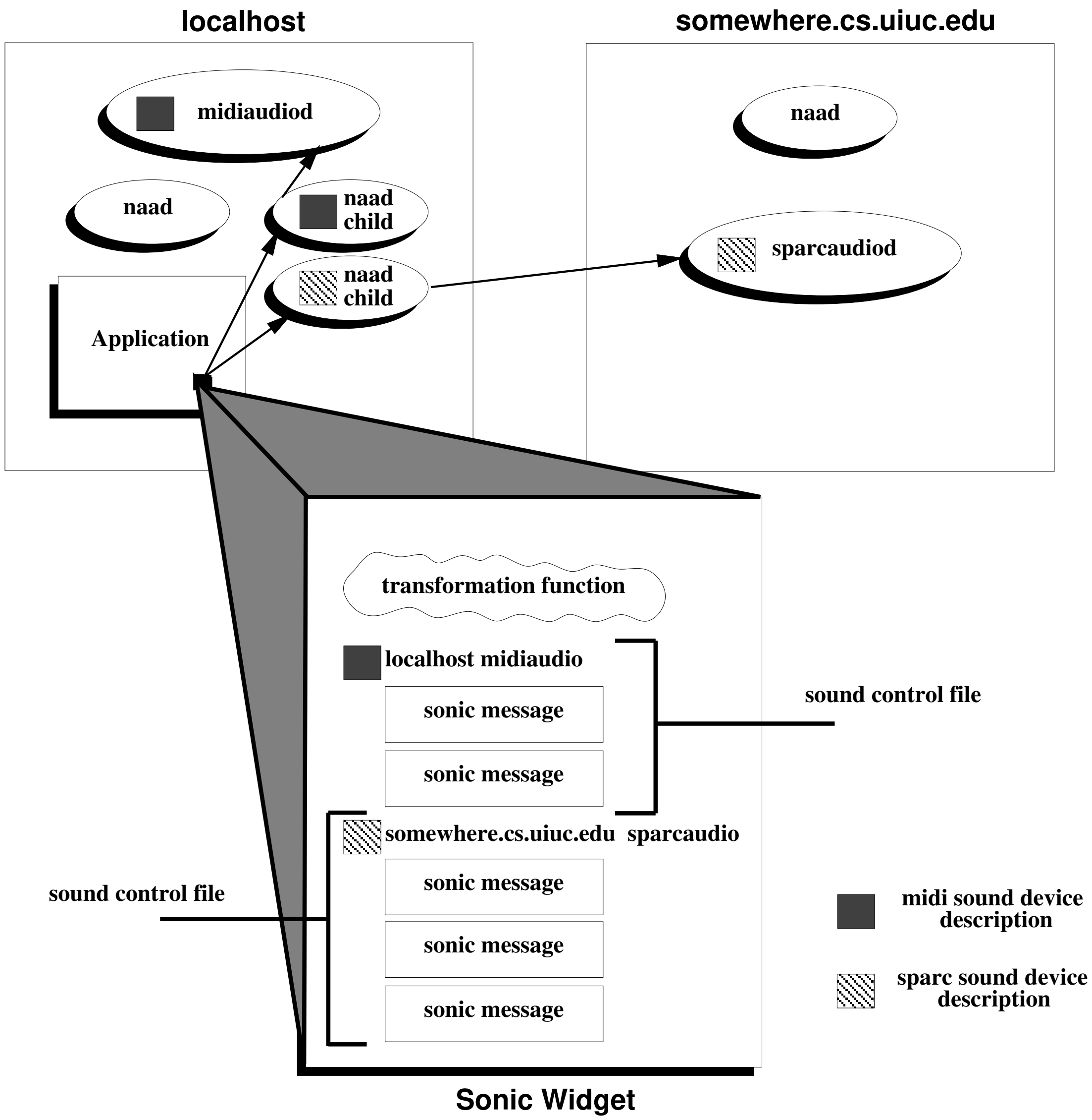

Figure 2.3: Porsonify system components.

Chapter 3

Implementation

As described earlier, portability, software reuse, and information hiding were key concerns in the design and implementation of Porsonify. For this reason, all code except for hardware–specific functions was written in C++, a superset of the C programming language that provides support for object–oriented programming. Similarly, the user interface is built using the X Window System and Motif widget set. This software environment can be found on a large platform of workstations. Sound device servers were written for the Sun SPARCstation audio device and two MIDI synthesizers, the Yamaha TG33 Tone Generator[1] and Yamaha PortaSound PSS–680 MusicStation. These particular synthesizers were chosen because they are relatively inexpensive and provide a wide range of functionality.

Porsonify's interprocess communication is built upon a socket–based network communication model, which makes interprocess communication similar to file access, again for portability; most variants of BSD UNIX support sockets.

To simplify programming the X user interface, the Widget Creation Library (WCL) was used. This library allows one to specify a hierarchy of X widgets and appropriate callbacks that describe an interface in the form of an X resource file. The resource file is completely customizable; it can be edited to rearrange the layout of the X widgets and the resulting look and feel of the interface, or to select different widgets. The ability to generate an X resource file to be interpreted by WCL, instead of constructing the widget hierarchy directly via application source code, was an invaluable aid to user interface development.

[1] Yamaha TG33 Tone Generator is a trademark of Yamaha Corporation.

This chapter describes the implementation of Porsonify. Porsonify can be viewed at two levels. At the highest level it consists of an application and a collection of network servers that communicate via sonic messages. Within the application and the network servers, there is a deeper layer of organization, consisting of interacting software objects. Each of these two levels will be examined in turn.

3.1 Implementation Overview

To support a client/server protocol, there must be clients and servers. In Porsonify, there are actually two layers of servers. Each sound device has its own server, which manipulates the sound hardware. Each workstation has a network audio access daemon (*naad*) which is the intermediary between clients and the sound device servers.

When a client opens a connection to a sound device server, it receives a *sound device description* with information about the commands the sound device supports. This description consists of a set of valid *sonic messages* and constraints on their arguments. It defines the communication protocol thereafter; the sound device only recognizes the sonic messages it has advertized.

Certain abstractions help in the implementation of sonifications, built using these device–dependent sets of sonic messages. Sonic messages can be grouped, along with the description of the sound device for which they are intended, into *sound control file* objects. These sound control file objects can instantiate sonic message arguments from input data using *transformation functions*, which perform simple mathematical operations on data values. Higher–level constructs, called *sonic widgets*, manipulate groups of sound control file objects and tranformation functions in a device–independent and data–independent way to produce sonifications.

3.2 Interprocess Communication in Porsonify

By analogy with the X window protocol, a sound client, or some application that produces a sonification, must establish a connection with the server for each sound device it wants to use. Sound device servers encapsulate information about the messages they accept in sound device descriptions. Each sound client must first retrieve this description from the appropriate sound device server to determine the device's functionality and the commands it accepts.

The intermediary in most communication between client and sound device servers is the *network audio access daemon*, or *naad*. This daemon is executed on all hosts that support the Porsonify protocol. The purpose of the *naad* is to permit multiple processes on the same workstation to concurrently access a single sound device.

This section describes the protocol followed by the client and servers to open communication with sound devices.

3.2.1 Locating Sound Devices

Before establishing a connection to a sound device server, a client knows a physical sound device only by the name of its associated host and the port on which its server listens for connections. Once a connection is established to the sound device through the *naad*, the client receives from the sound device server a *sound device description* describing the sonic messages the server can accept. It also receives the name of a FIFO pipe, a special file used for interprocess communication, on which it can write sonic messages that will be communicated to and executed by the sound device server. Thereafter, the client knows the physical sound device by the sound device description and the name of the FIFO pipe. To a client, the interface to a sound device is simply the collection of sonic messages the sound device server accepts. Therefore, a single, physical sound hardware device can potentially have several logical sound devices, each with an associated sound device description, corresponding to different interfaces it supports.

To discover the port name of a sound device server, a client must know the host on which the server resides. It can then ask the *naad* on that host for a list of all local sound devices. Upon startup, the *naad* reads a configuration file with descriptions of any local sound hardware and the associated ports. The contents of that configuration file are returned to the client in response to the query. Through this mechanism, applications can probe remote hosts to determine what sound hardware they support.

3.2.2 Establishing a Connection to a Sound Device

The most important function of the *naad* is to establish connections to sound device servers for clients. It performs this function to allow multiple clients on the same host to access the same sound device concurrently. A sound device server cannot process sonic messages promptly if it also has to accept multiple connections; each connection request would be an interruption.

A client that wishes to use a sound device first contacts the *naad* on its local host, and asks it to open a connection to the sound device, using the previously obtained a host and port. In response, the client expects the name of a FIFO pipe that it can use to communicate with the sound device server, and the sound device description. The *naad* has an internal table of open connections. If it does not have an open connection to this device, it must attempt to open one. Assuming this attempt succeeds, the *naad* reads the sound device description from the sound device, adds information about this open connection to its table, creates a FIFO pipe, and returns the sound device description and pipe name to the client. If another process on the same host has already opened a connection to the same sound device, the *naad* can simply return the cached information to the client.

Some process on the local host must read messages from the FIFO pipe and write them to the sound device server. This is the responsibility of the *naad*. When the *naad* establishes a new connection to a sound device server, it forks, producing two processes. The child process passes messages through the pipe between client and sound device server, and the parent continues to handle requests.

In C++, user–defined types may be described by classes, and the operations on these types are member functions of the class. The details of contacting the *naad* and receiving the name of a FIFO and a sound device description are managed for the client by the `openConnection` member function of the `SoundDevice` class. This function takes a host name and a port name as arguments. A `SoundDevice` object is the logical representation of a connected sound device; it encapsulates the sound device description and hides all details of network communication from the client. A different protocol for establishing a connection could be implemented by replacing this class.

3.2.3 Sonic Messages

After a client has established a connection to a sound device server, it communicates with that server by sending sonic messages. The information sent in a sonic message consists of a message number (by which the sound device server identifies the sound command) and a list of command arguments. The `executeMessage` member function of the `SoundDevice` object can execute sonic messages singly or in groups.

3.3 Network Audio Access Daemon

This section describes the network audio access daemon, or *naad*, in more detail. The *naad* runs on every host that supports the Porsonify interface, even no sound hardware is physically connected to the local host.

As previously described, the main task of the *naad* is to open connections to sound devices for client applications. This allows multiple processes on the same host to connect to a single sound device with a single associated server. The client provides the name of a host and port; in response it expects the name of a pipe through which it can communicate with the sound device server.

The *naad* maintains a table of open connections. Each entry in this table contains a host name, a port name, the name of a FIFO pipe used to communicate with the application, an object representing the connection to the sound device server, the UNIX process identifier of the *naad* child managing the connection, and the sound device description (consisting of the device defaults, device parameters, and message configuration information) from the sound device server. If the host and port pair do not match any entries in the open connection table, the *naad* attempts to establish a connection with the sound server. If this attempt fails for any reason, an error flag is returned to the querent. If it succeeds, the *naad* immediately reads the sound device description from the sound device server. It then stores a copy of this description in its table of open devices, creates two named FIFO pipes for future communication, sends the application the name of one of the pipes and the sound device description, and forks. If the connection was already opened, this information is sent immediately to the application, and the *naad* does not fork.

The parent *naad* then continues to accept new requests. The client application reads the name of the FIFO pipe and the sound device description, and breaks the connection with the *naad*. Thereafter, the client communicates with the sound device server solely through the two pipes. The pipe whose name it receives is used for writing sonic messages; the second pipe (which has the same base name) is used for reading acknowledgements. Acknowledgements are needed only when a client operates in blocking, or synchronous mode. In synchronous mode, a client will only send additional commands when it can read an acknowledgement indicating that the previously executed synchronous batch of commands has completed. Any client may execute synchronously on the same device; therefore, the acknowledgement might have been left by a different client. Meanwhile, a non–synchronous client may execute messages whenever it wants, without checking for acknowledgements.

Finally, when a child *naad* process dies due to an error, or because it has detected that all applications have closed the FIFO pipe from which it was reading (i.e., they have all exited), the parent *naad* closes the connection with the sound device server and updates its internal tables.

3.4 Sound Device Servers

By design, sound device servers are extremely simple table–driven command interpreters. When a connection to a sound device server is first established, it sends the device defaults, device parameters, and message configuration table to the *naad*, which gives them to the client. Then the server simply accepts sonic messages, executes them, and if necessary, returns an acknowledgement.

Fundamentally, the only thing that differs between sound device servers is the set of sonic messages they accept. Each sonic message, of class `Message`, encapsulates a list of arguments and a pointer to a member function of a *sound device description class.* The sound device description class is a collection of methods that accepts lists of sonic message arguments and calls the appropriate device–specific functions that actually manipulate the sound hardware. The `execute` method of the `Message` class executes the object's associated member function of the sound device class with its message argument list.

Therefore, three layers of software describe the sonic messages for each sound device server.

1. Basic functions to access the physical hardware.

2. Interface functions that accept sonic message arguments and call the appropriate device–specific functions (the *sound device description class*).

3. A sound device description, consisting of sonic message configuration, device default, and device parameter files. These files describe sonic messages and their arguments, and they map each message to the corresponding interface functions from the second layer.

The key factor in sound server implementation is partitioning the functions of the sound device into sonic messages that do not mask its capabilities. For example, MIDI supports both note on and note off commands. Although it is convenient to encapsulate this functionality as a single "play note" sonic message that accepts a duration as an argument, it is vital to also support individual note on and note off commands. Otherwise, it would be impossible to gradually change the volume of a note, slowly change its pitch, or play chords.

The interface functions of the second layer of the sound server belong to their own C++ class. In the current implementation, `MIDISoundDevice` and `SPARCSoundDevice` are examples of such classes with methods that each accept a list of sonic message arguments (of type `ValList`), cast each argument into a basic type (e.g., integer), and call hardware–specific routines. Figure 3.1 illustrates the current hierarchy of sound device description classes.

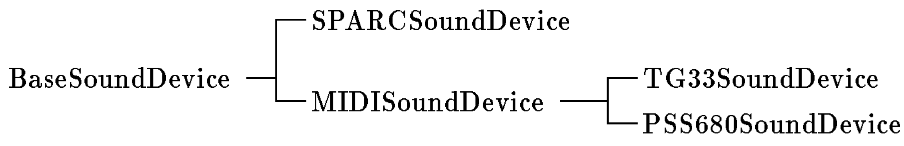

Figure 3.1: Class hierarchy for sound device description classes.

Because no two subclasses of `BaseSoundDevice` are guaranteed to have any common functions, `BaseSoundDevice` does nothing. `SPARCSoundDevice` defines methods to manipulate the SPARC audio chip, such as playing and recording sample files. `MIDISoundDevice` defines basic MIDI commands such as turning notes on and off, sending system exclusive (device dependent) commands and initializing the device. `TG33SoundDevice` and `PSS680SoundDevice` are subclasses of `MIDISoundDevice` that define new methods particular to the Yamaha TG33 Tone Generator and Yamaha PortaSound PSS–680 MusicStation, respectively. `PSS680SoundDevice` also redefines some of the methods defined in `MIDISoundDevice`; it does not feature a complete MIDI implementation.

The remainder of the server details are specified in the message configuration file, device default file, and device parameter file. These files are parsed by a server creation script that generates C++ code to initialize internal, device–specific tables used by the sound device server. Specifically, the script matches names of member functions specified in the message configuration file to actual member functions in the sound device description class. It builds a lookup table of sonic message objects. Each sonic message can call the C++ member function that implements it with the arguments it receives from a client. A new sound hardware server is built from a skeleton hardware server, the sound device description class and base hardware routines, and initialization functions generated from the sound device description by the server creation script. This eliminates much of the tedium of writing a new sound server, and minimizes the probability of errors.

Figure 3.2 illustrates how the sound device description, sound device description class, and base hardware functions interact. The light gray layer is completely table driven, and each of the dark gray layers is either its own C++ class or a collection of functions. The dark arrows represent calls to methods or functions between layers, and the light arrows represent dependencies between definitions in the message configuration, device defaults, and device parameters files. The format and meaning of each file is described below. The examples shown are the actual files used to build the server for the SPARCstation audio device.

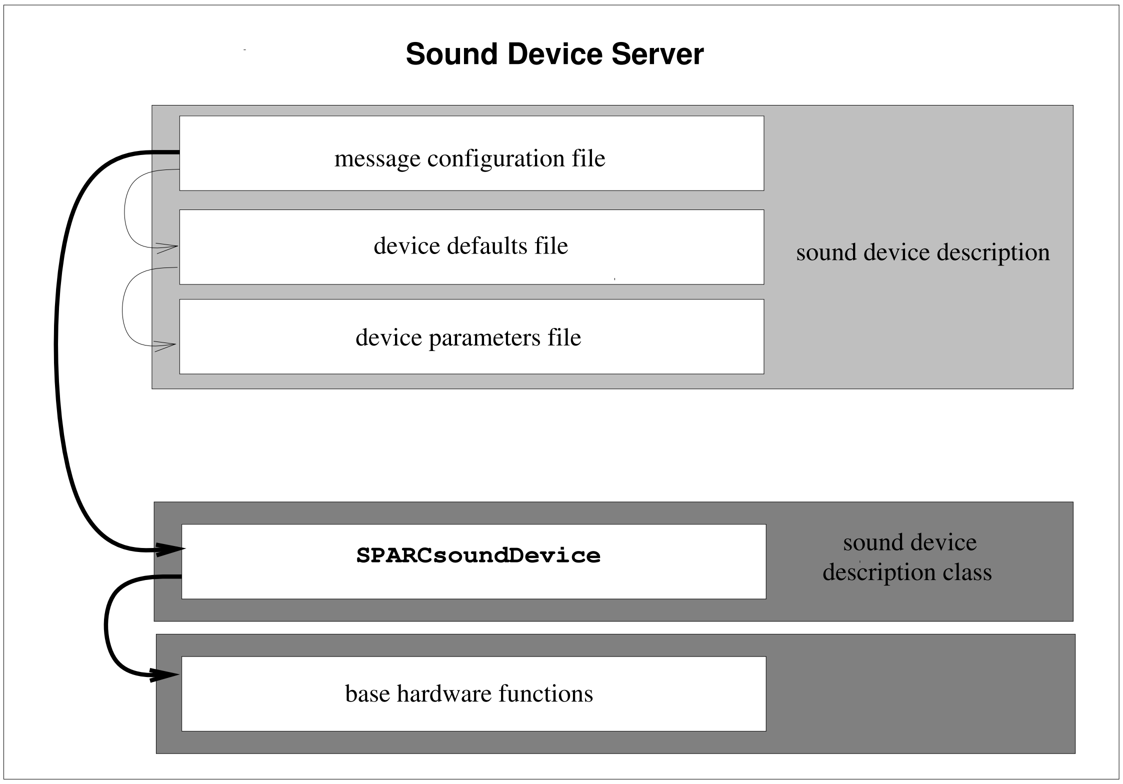

Figure 3.2: Sound device server structure.

3.4.1 Sonic Message Configuration File

The sonic message configuration file, shown in Figure 3.3, describes all the messages that can manipulate the state of a sound device. The quoted text in the first column is a description of the message's function. The second column is the name of a C++ member function in the sound device description class for this particular device. This information is used by a script to initialize the sound server code that executes the member function corresponding to each message received. The remaining, optional columns are the expected arguments to the method. They are indexes into an associated device defaults file.

```
"Set Volume"                setVolume         VOLUME
"Set Recording Volume"      setRecordVolume   RECORDVOLUME
"Reinitialize Device"       reinitialize
"Select Instrument"         selectInstrument  INSTRUMENT
"Set Output Port"           setOutputPort     OUTPUTPORT
"Play Note"                 playNote          NOTE DURATION
"Play Sample File"          playSampleFile    FILENAME DURATION
"Say A Number"              playNumber        NUMBER VOICE
```

Figure 3.3: Sonic message configuration file for SPARCstation audio device.

The messages in Figure 3.3 are the ones understood by the SPARCstation sound device server. The `SPARCSoundDevice` class was written to hide the details of sound synthesis; for example, the `Select Instrument` command offers a palette of instruments and either synthesizes the samples for the voice or reads them from saved sample files. Nothing prevents adding methods to the `SPARCSoundDevice` to perform any operation the Sun SPARCstation audio chip is capable of executing.

Internally, messages are numbered according to their order in this file. However, the message number is only significant when writing sound control files (see §3.5.2).

3.4.2 Device Defaults File

A device defaults file, shown in Figure 3.4, describes the valid arguments to sonic messages for the device. The first column is simply a tag to identify the default. It corresponds to a

```
VOLUME          CHAR       "Volume"               75    MINVOLUME   MAXVOLUME
RECORDVOLUME    CHAR       "Recording Volume"     50    MINVOLUME   MAXVOLUME
INSTRUMENT      ENUMVAL    "Instrument"            0    INSTRUMENTS
OUTPUTPORT      ENUMVAL    "Output Port"           0    OUTPUTPORTS
NOTE            CHAR       "Pitch"                60    MINNOTE MAXNOTE
DURATION        FLOAT      "Duration"            1.0
FILENAME        REMOTEFILE "Sample File"           0    REMOTESAMPLEFILES
NUMBER          ARRAY      "Number"              "0"
VOICE           ENUMVAL    "Voice for speech"      1    VOICES
```

Figure 3.4: Device defaults file for the SPARCstation audio device.

sonic message argument in the message configuration file. In the above example, `VOLUME` is an argument of the `"Set Volume"` message in Figure 3.3. The second column indicates the type of the default. The permitted types are shown below.

`INTEGER` integer

`BOOLEAN` 1 (true) or 0 (false)

`CHAR` an ASCII character, represented as a number in the range 0–255

`FLOAT` single precision floating point

`DOUBLE` double precision floating point

`ARRAY` character array, in double quotes

`REMOTEFILE` the name of a remote file, which must be a valid UNIX path name

`ENUMVAL` an enumerator of `ENUMTYPE`, defined in the device parameters file below

This set of types would be made complete by the addition of a `FILE` type, representing an arbitrary stream of bytes. This has not been included in the current implementation because all operations we wished to support on the available sound devices could be performed using these simple argument types, and because delays incurred by sending large amounts of data over the network would impede "real–time" responses.

```
MINVOLUME          CHAR      "Minimum Volume"     0
MAXVOLUME          CHAR      "Maximum Volume"     100
INSTRUMENTS        ENUMTYPE  "Instrument List"    "SparcInstruments"
OUTPUTPORTS        ENUMTYPE  "Output Ports"       "SparcOutputs"
MINNOTE            CHAR      "Minimum Note"       0
MAXNOTE            CHAR      "Maximum Note"       127
REMOTESAMPLEFILES  ENUMTYPE  "Sample File Menu"   "SampleFileMenu"
VOICES             ENUMTYPE  "Spoken Voice List"  "Voices"
```

Figure 3.5: Device parameters file for SPARCstation audio device.

The character string in the third column is a description of the default. This is followed by the default value in the fourth column. If there are remaining columns, they are indexes into the device parameters file, and describe upper and lower bounds or an enumerated type (a collection of named integer constants).

3.4.3 Device Parameters File

The device parameters file, an example of which is shown in Figure 3.5, describes static sound device parameters that cannot be changed under software control. This file has a format similar to the device defaults file, except that no bounds are specified, and only types `INTEGER`, `BOOLEAN`, `CHAR`, `FLOAT`, `DOUBLE`, and `ENUMTYPE` are permitted.

The first column is the name of the device parameter. It corresponds to some parameter that was listed as a bound or as an enumerated type in the device defaults file. The second column is the type of the parameter. The string in the third column is a description of the default. Finally, the value is given in the last column. The "value" of an `ENUMTYPE` is actually the name of a file that contains the names of the enumerators, one per line.

3.4.4 SPARCstation Sound Server Functionality

Together, these files form the sound device description of the Sun SPARCstation audio device. A client receiving this description can select an instrument from among those listed in the file `SparcInstruments` and play a note with a pitch in the range 0 to 127 and some specified duration, or play a sample file, reinitialize the device, and so forth.

A sonic message sent between processes is just an identifying message number (corresponding to the position of the message in the sound device server's message configuration file) and a list of arguments. Thus, a command to play a note with a pitch of 60 and duration of 1.5 seconds would be received by the SPARCstation sound server in the following format:

```
5 60 1.5
```

The SPARCstation sound server first reads the command identifier (`5`). This identifier is an index into the table of sonic messages that the sound server accepts; an object of class `Message` is created to receive the arguments, performing the appropriate bounds and error checking. Then the sound server calls the `execute` method of the `Message` class, which executes the member function associated with the sonic message (as specified in the message configuration file) with the list of sonic message arguments. For this particular message, the member function `SPARCSoundDevice::playNote(ValList&)` would be executed.

3.5 Software Support for Sonifications

With support for executing sonic messages in place, as described above, creating a sonification is reduced to determining how the data stream controls the generation of sonic messages for a particular sound device. Recall that Porsonify does not assume any lookahead on the input data stream. Each individual data value must cause at least one sonic message to be executed if it contributes in some way to the sonification. There are only two fundamental ways each data value can be used to control sonic messages. The data value can be used to vary the parameters of some message or list of messages that are executed for each value (e.g., the pitch of a "Play Note" command), or it can be used to select which message or list of messages should be sent to the sound server for execution (e.g., a note is played only if the input data is greater than zero).

This observation provides some insight into the kinds of objects which are necessary to support the creation of sonifications. We have noted that sound devices have radically different functionality; these differences are reflected in the sonic messages each sound server supports. Ideally, the instantiation of sonic message arguments or selection of particular messages for each data value should be independent of the messages themselves and associated sound device. This would allow one to approximate a sonification on different hardware by replacing the sound device and sonic messages. To support this, there must be a way to group sonic messages and the sound device for which they are intended. There must also be a mechanism for manipulating these groups of messages to present each data value.

One could assume that if a data value is used to instantiate an argument of a sonic message, it could be mapped into a valid range by some external means before reaching the infrastructure of the sonification. Instantiating a sonic message argument would then be as simple as using the data value, unchanged, as the argument. This is inappropriate for two reasons. First, if the underlying sound device is replaced, its arguments might lie a different range. The data would then have to be changed. Second, it would make experimentation difficult. Suppose one first decides to use data values to control the pitch of notes, and then decides to use them to vary note volume instead. Unless pitch and volume have the same ranges, the data must be edited. For these reasons, the system should provide some general way to transform data values, at least to compensate for differences in acceptable ranges, and to use them as sonic message arguments.

Finally, for convenience, there should be a uniform way of graphically configuring the sonification that requires no knowledge of the underlying sound device. Sound device descriptions provide enough structure and information to automatically create a simple graphical user interface.

In Porsonify, sonic messages and the description of the sound device for which they are intended are grouped to form *sound control files*. *Transformation functions*, which provide simple mathematical modifications of input data values, may be used to instantiate sonic message arguments. *Sonic widgets* may use a transformed data value to select and conditionally exeecute one or more sound control files, creating sonifications. Sonic widgets can save their configuration information in *widget control files*, allowing sonifications to be saved, edited, and reused. Graphical user interfaces for configuration of sonic widgets can be automatically created from the widget control file.

Using these abstractions, a new sound device with an associated sound device server can be integrated with an application based on the Porsonify software simply by writing a widget control file to describe a sonification. The rest of this section describes sound control files, transformation functions, and sonic widgets in greater detail.

3.5.1 Sound Control Files

A sound control file is a list of sonic messages and arguments specific to a particular sound device server. Arguments may be fixed (cannot be changed), configurable (can be graphically edited), or variable (a function of values in an external data stream). Each variable argument may have an associated transformation function, used to instantiate its value from the input data.

An application uses a sound control file to open a connection to the sound device server which can accept the sonic messages. This creates a `SoundControlFile` object, which is capable of executing its sonic messages (sending them to the sound device server) when it receives a new data value. When the sound control file executes, it recalculates all the variable arguments of all its sonic messages using the new scalar or vector data. This is simplified because of some bookkeeping that is done during initialization. Variable arguments may be calculated from the element in any position in a vector, and more than one variable argument can be dependent upon the same element. Therefore, a list of variable arguments is kept for each vector index encountered in the initialization information for the sound control file. Essentially, scalar input data is treated as a vector of length one. For each index less than the size of the input vector, the variable arguments dependent upon that index are recalculated with the vector element at that index. The sound control file then sends its list of sonic messages, with the newly instantiated arguments, to the sound device server.

After executing its sonic messages, the sound control file object can either wait for acknowledgement from the sound device server that all pending messages have been processed, or it can continue to execute messages without waiting for acknowledgements. Acknowledgements are sent by the sound device server only if the sound control file has enabled blocking mode with the `setBlocking` member function. This primitive function allows sound control files to be synchronized with other sonic and graphic displays.

3.5.2 Sound Control File Syntax

Sound control file objects can write their configurations to files. Typically this configuration information is not actually written to individual files, but it is included as part of the configuration file of a sonic widget, which can encapsulate one or more sound control file objects.

```
1  # hostname           port name
2  foobar.cs.uiuc.edu   midiaudio

3  # sonic message name      msg number arguments
4  /* Select Instrument   */      3      ?9
5  /* Set Balance         */     13       2
6  /* Set Sustain (Decay) */     11      ?0
7  /* Play Note           */      0      $0 ( /* Linear */ 1
8                                             /* A =    */ 30.00
9                                             /* B =    */ 60.00 )
10                                       ?0.50
11                                       ?127
12 .
```

Figure 3.6: Sound control file to play a note with variable pitch on MIDI sound device.

The configuration information is everything an application needs to reconstruct a sound control file object. This includes:

- the host/port pair needed to connect to the server for an audio device.
- a list of sonic messages and default message arguments (which are fixed, variable, or configurable), and
- optional default transformation functions, and initial configuration information, for variable message arguments.

Figure 3.6 shows a sound control file that plays a note with a variable pitch on a MIDI sound device. The comment syntax is taken from the Bourne shell and C; a `#` indicates a comment that continues to the end of the line; comments may also be delineated by `/*` and `*/`.

The first two strings (on line 2) specify the Internet hostname (`foobar.cs.uiuc.edu`) and the port name (`midiaudio`) for the target audio device. Because more than one host might support the server that monitors port `midiaudio`, if the hostname is given as `localhost`, the software will attempt to establish a connection to the server on the local host. This makes sound control files for a particular device portable across many hosts. The port name should not be changed, however, because the sonic messages in the sound control file are specific to that logical interface to the hardware (e.g., sonic messages intended for a MIDI synthesizer will generate errors if sent to a SPARCstation audio device).

Next come a sequence of one or more sonic messages (lines 4–11 in Figure 3.6). Each `Message` consists of its identifying number, or "message identifier", and the arguments. Recall that the meaning, range, and default values for command arguments are sent to the sound control file by the server upon connection. It is not only redundant, but error prone, to save this information between sound control file initializations; the sound device server might augment the list of messages it supports, rendering old copies of these tables obsolete.

The fixed arguments of sonic messages require no special syntax; they are simply specified in the sound control file. For example, on line 5 in Figure 3.6 there are two numbers. The first (`13`) is the message identifier for the `Set Balance` command. The second number, `2`, is the selected balance setting. In this sound control file, the stereo balance cannot be changed by user configuration. Fixed parameters are especially useful for situations where changing a particular parameter would render that sonification less aesthetically pleasing, or ineffective. For example, consider a sound device with both a master volume and a note volume control. If the note volume is a variable argument (i.e., controlled by the input data), it might be desirable to fix the master volume at some reasonable value so that the changes in note volume specified by the data can be heard.

Configurable arguments are indicated by a question mark (`?`) preceding the default value. For example, on line 6, the `Set Sustain` command accepts one argument: sustain (the rate of decay of a note, or how slowly it trails off). Here, sustain is a configurable argument, given a default value of `0` in the file.

Variable arguments do not have default values in the sound control file. They are indicated by a dollar sign (`$`) followed by an index. This index corresponds to an element of a vector. Sonic widgets may "display" scalar or vector data values. If the sonic widget that manages this sound control file maps a vector of data values to sound parameters, then the sonic widget's transformation function will operate on the scalar data value corresponding to the specified vector element. If the index is out of bounds, (i.e., the index is greater than the length of the vector) no transformation is performed and the value of that message argument will be the device default, as provided by the sound device server (see §3.4.2). If the sonic widget displays a scalar data value, the scalar is treated as a single element vector, and all variable arguments except for those of the form `$0` are ignored. The remainder of the variable arguments default to the device default.

A transformation function and its initial configuration may be specified in parentheses directly after a variable argument. The first number in parentheses is the identifying number of the transformation function; see §3.5.3 for the current list of these. Some transformation functions may also have parameters that can be configured in the sound control file; if so, they are read from the sound control file after the number of the transformation function. Therefore, on line 7 in Figure 3.6, the first argument of the `Play Note` sonic message is controlled by transform function 1, or `Linear`. This function performs the transformation $a * x + b$ on the input data x. The values for a and b specified in this sound control file are `30.0` and `60.0`, respectively. If no transformation function configuration information is specified in parenthesis after a variable argument, the device default for that argument is used until a transformation function is selected.

3.5.3 Transformation Functions

As described, transformation functions accept a scalar input value and transform it with a simple mathematical operation to produce a scalar output value. Transformed data can either be substituted for variable parameters in sound control files managed by a sonic widget or used to select a particular sound control file for execution.

Transformation functions are not intended to compute complicated mathematical operations on a single data value. They are necessary merely to promote interchangeability of different sound devices and data files in a particular sonification. For example, consider two data files that each contains information about eight different events. In the first file, these events are numbered 0–7. In the second file, they are numbered 8–15. If the event identifier is mapped to the pitch of a note with the `Linear` transformation function (which calculates $a * x + b$, where x is the event number), the same sonification can be used for both files by adjusting the constant b for each case.

Thus, the transformation functions are designed for simple scaling and range mapping. Like visual displays, sonifications include only those transformations necessary to display the data, not transformations that change or combine elements of the data stream. Manipulations of the second kind are best realized by a general purpose data analysis program.

All transformation functions are classes derived from the base class `DataTransform`. Figure 3.7 shows the hierarchy of transformation functions. The `DataTransform` class contains

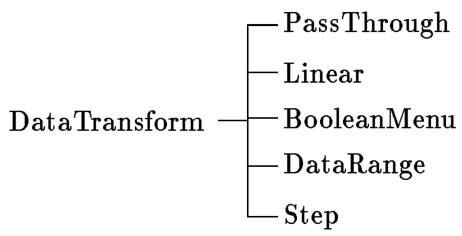

Figure 3.7: Class hierarchy for transformation functions.

member functions to calculate the new value of a parameter for a sonic message, read its configuration information from a file, print its configuration information, and set the value of its internal variables. New transformation functions are derived from `DataTransform` and should redefine these member functions as appropriate.

In Porsonify, each transformation function has a unique internal identifying number. A transformation function manager, which knows the numbers of all transformation functions, can create a transformation using this number; it is used by sound control file and widget control file readers to create transformation functions where necessary. Once created, the transformation function reads and writes its own configuration information.

There are currently five transformation functions, described in Figure 3.8. Except for `PassThrough`, which simply returns the input data unaltered, all perform simple scaling and mapping operations; typically the sonification designer would like to map data in a certain range to the range of some sound device parameter. The `Linear` transform is probably the most obvious way to do this. The `DataRange` transform is useful for mapping ranges of data values to particular sound effects. For example, on the Sun SPARCstation audio device, one might want to play a particular file of audio samples for values that fall below a certain threshold, and another for those that fall above. One way to specify which files should be played for which ranges is to use the `DataRange` transformation. The `Step` transform can be used to provide greater flexibility than the `DataRange` transformation when mapping data to a sequence of non–regular ranges. Because actual parameter values need not be written in the data range file, the same file can be used to control variable parameters with different ranges (e.g. volume might range from 0–100 and pitch from 60–90). The `BooleanMenu` transformation can be viewed as a mapping operation as well; it turns an input data value into a 1 or a 0. However, `BooleanMenu` is most often used to choose between sound control files in a sonic widget, not to instantiate a sonic message argument.

3.5.4 Sonic Widgets

A *sonic widget* manipulates one or more sound control file objects and an optional transformation function without knowledge of either the underlying sounds produced by the manipulation or the characteristics of a sound device. If a transformation function is present, it can be used to select a sound control file or determine how many times to execute a single sound control file. As described above, the sound control files themselves may have variable parameters that are instantiated from the data stream. In short, the sonic widget provides a high–level means of manipulating groups of sonic messages with data.

Suppose one wishes to map incoming scalar data values to some characteristic of a note, such as pitch, timbre, duration, or volume. This would be a suitable task for the simplest sort of sonic widget: `SimpleSonicWidget`. This widget manages a single sound control file, which contains the commands to play a note on a particular sound device. The sound attribute of interest would be a variable argument in the sound control file, and a transformation function would determine how the argument's value is calculated from the values in the data stream.

Another common type of sonification is an alarm, where a sound is produced when certain conditions in the input data are satisfied. The `BooleanSonicWidget`, which manages one transformation function and one sound control file, can be used to implement an alarm. If the value produced by the transformation function is greater than zero, the commands in the sound control file are executed. As always, the sound control file may in turn contain variable arguments, so that like `SimpleSonicWidget`, it can map the input data to different sound characteristics.

Alternatively, the data value can be used to select one of several sound control files for execution. The `MultiSonicWidget` manages a transformation function and several sound control files to do just this. For example, suppose one wishes to sonify a stream of positive and negative integers. If the number is positive, a high note is played, and if it is negative, a low note is played. This sonification can be created with two sound control files, one to play each pitch, and a boolean transformation function that compares each input data value to zero. The result

Name	ID	Description
`PassThrough`	0	Does not modify the scalar data value, and does not take any arguments.
`Linear`	1	Computes $(a * x) + b$, where x is the data value, and a and b are user–specified constants. This is an easy way to scale a range of data values to an equivalent range of device defaults (e.g., to map a stream of numbers into audible musical notes). This transform can also be used to produce a constant value from a variable data stream, by setting $a = 0$ and b to the desired constant.
`DataRange`	2	Returns a value based on the range in which the data value lies. Ranges are specified in a file that contains one or more lines of the format `lower upper v`. The values `lower` and `upper` may be integer or floating point, and `v` can be any sequence of non–whitespace characters. The value v is returned if the input data value (x) satisfies the conditions $x >$ `lower` and $x <=$ `upper`. If x does not satisfy these conditions for any line in the file, -1 is returned.
`Step`	3	Like DataRange, the Step transformation returns a value based on a data range given in a file. However, any values (`v`) present in the file are ignored. Instead, the returned value is calculated by multiplying the number of the line in the file that satisfies the conditions $x >$ `lower` and $x <=$ `upper` by *stepWidth*, and adding to that an *offset*. The *stepWidth* and *offset* can be changed to provide greater flexibility than the DataRange transformation when mapping data to a sequence of non–regular ranges. Since actual parameter values do not need to be written in the data range file, the same file can be used to control variable parameters with different ranges (e.g. volume might range from 0–100 and pitch from 60–90).
`BooleanMenu`	4	This uses the data value x and a constant to evaluate a simple boolean expression (x `op` *constant*) and return either true (1) or false (0). The supported boolean operations are: $=$, $<$, $>$, $\leq$, $\geq$ and $\neq$. These are numbered consecutively from 0 to 5.

Figure 3.8: List of transformation functions.

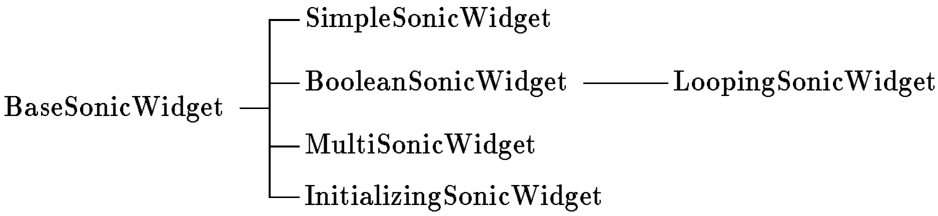

Figure 3.9: Class hierarchy for sonic widgets.

of the transformation function, true (1) or false (0) is used to select which sound control file is executed.

The input data can also be used to determine how many times a particular sound control file should be executed. This kind of widget, a `LoopingSonicWidget`, is what we might use to implement a "Big Ben" sonic display, described in §2.5. The `LoopingSonicWidget` contains a transformation function and a single sound control file; the transformation function operates on the input data to produce an iteration count, and the sound control file is executed that number of times. In the "Big Ben" sonification, the input data is the number of hours in the current time, the transformation function passes the hour value through unchanged, and the sound control file plays a single chime.

The last kind of widget provided in Porsonify, `InitializationSonicWidget`, manages two sound control files. One sound control file is used once to initialize the sound device, and the other is used in the same way as with the `SimpleSonicWidget`. This kind of widget is invaluable for creating sonifications with MIDI synthesizers. Typically one selects a certain voice (and other characteristics) once, and reuses those settings thereafter.

Each sonic widget is represented by a class derived from the base class `BaseSonicWidget`. The class hierarchy for sonic widgets is shown in Figure 3.9. `BaseSonicWidget` defines public member functions to do the following.

- set and return information about the transformation function and sound control files it manages,
- write a widget control file that represents the current configuration of the sonic widget,
- write a WCL file that controls graphical configuration of the parameters of the transformation function and individual sound control files,

- read a widget control file and initialize itself, and

- execute the appropriate sound control file(s) with the original input vector or scalar data.

Usually, only the last two of these items need to be redefined when creating a new class of sonic widget. The initialization function reads from a widget control file a transformation function (if the widget control file specifies one) and the correct number of sound control files. Since `BooleanSonicWidget` and `LoopingSonicWidget` both manage exactly one transformation function and one sound control file, `LoopingSonicWidget` can inherit the `initialize` member function from `BooleanSonicWidget`. The `execute` member functions (for scalars and vectors) execute the appropriate sound control files based on the algorithm the sonic widget represents. The procedure for executing sonic messages is independent of the type of the controlling transformation function, if one is present. For example, a `BooleanSonicWidget`, which executes one sound control file if the transformed input data is greater than zero, does not have to test the input data with a `BooleanTransform` function. The sonification designer might wish that a sound be produced for each non–zero value in an input data stream, in which case it is sufficient to select the `PassThrough` transformation. The `BooleanSonicWidget` does not know what specific transformation controls execution of the sound control file it manages; it calls the transformation function on the input data, produces a value, and performs an action based on that value.

Sonic widgets can create a file containing a textual representation of the objects they manage; this is called a *widget control file*. Sonic widgets can be recreated from these files. Thus, sonifications can be created that use new sound hardware simply by replacing the sonic messages in an existing widget control file with messages for the new sound device.

For example, Figure 3.10 illustrates a `BooleanSonicWidget`. This sonic widget encapsulates a sound control file object that maps data to notes in a scale if the input data value satisfies the condition specified by the transformation function. The sound control file used to initialize the sound control file object in this widget is shown in Figure 3.12. The messages in this file are specific to the Sun SPARCstation; for example, "Set Output Port" determines whether the sound will be output to headphones or speakers. The sonic message most critical to the sonification is the one with the variable argument, "Play Note." It is this command that maps the input data to pitch. The sonification produced by this sonic widget can be made to run

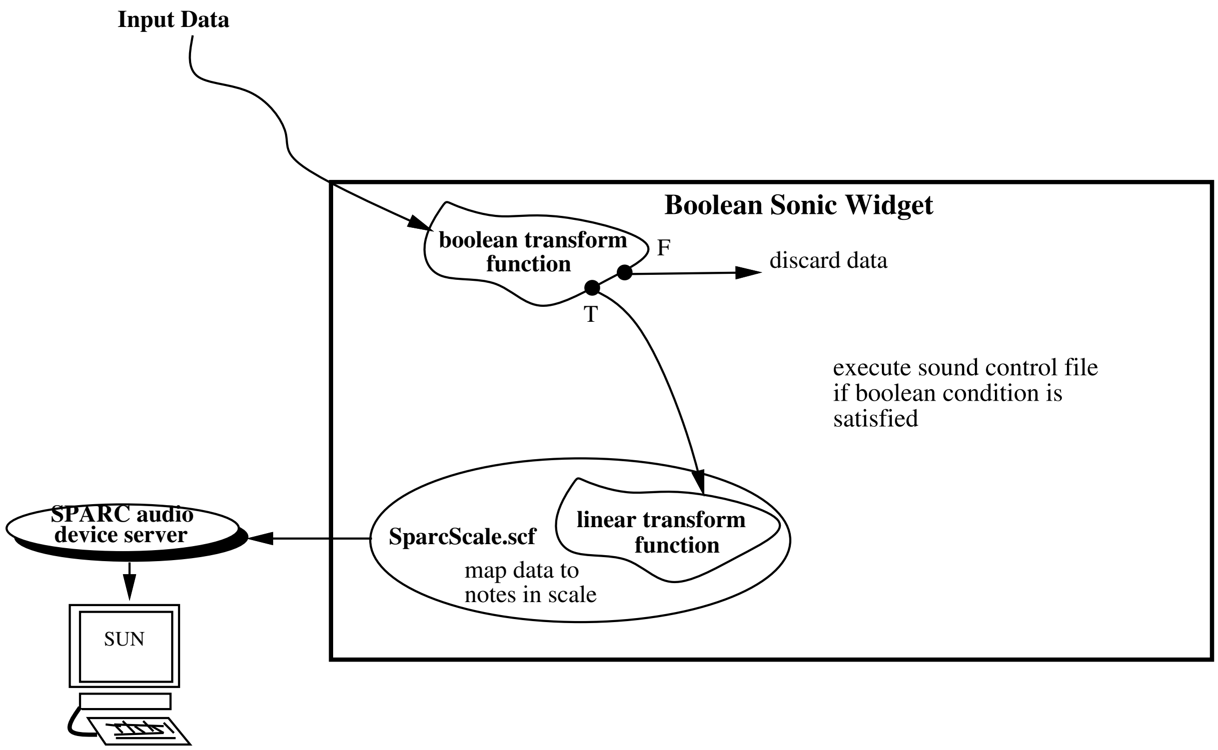

Figure 3.10: A `BooleanSonicWidget` on the SPARCstation audio device.

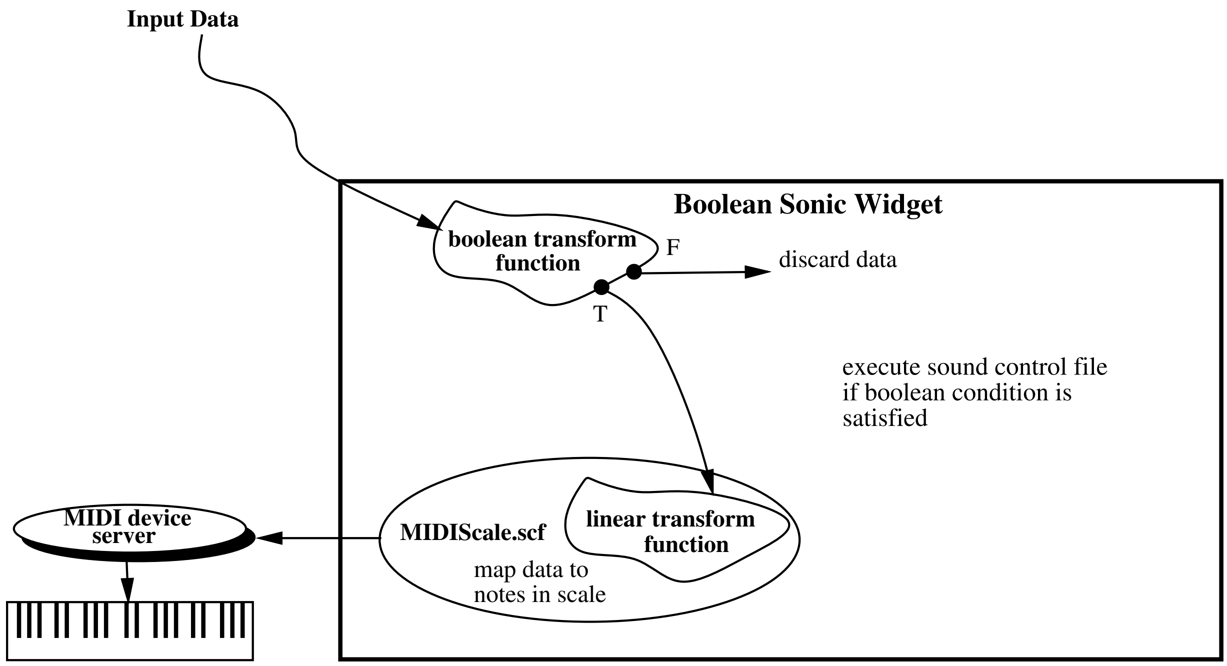

Figure 3.11: A `BooleanSonicWidget` on the MIDI audio device.

```
# Map input data to notes in a scale

# hostname    port name
localhost    sparcaudio

# sonic message name     message number      arguments
/* Set Output Port   */        4             ?1
/* Set Volume        */        0             ?60
/* Select Instrument */        3             ?3
/* Play Note         */        5             $0  ?1.0
```

Figure 3.12: Sound control file to map data to pitches on the SPARCstation audio device.

```
# Map input data to notes in a scale

# hostname    port name
localhost   midiaudio

# sonic message name        message number      arguments
/* Play Note on Channel*/         18            $0  ?1.0   ?127  ?0
```

Figure 3.13: Sound control file to map data to pitches on a MIDI device.

on different sound hardware, such as a MIDI synthesizer, by replacing the sonic messages for the SPARCstation sound device with sonic messages to imitate similar functions on the MIDI sound device. Figure 3.13 is an example of an sound control file that would map data to pitches on a MIDI device. Note that there is only one sonic message, "Play Note on Channel," but it has a variable pitch argument as well. Any assortment of messages that makes sense could go here, as long as the basic mapping function is the same. Selection of sonic messages is left to the sonification designer. Figure 3.11 illustrates how the software hierarchy changes when the sound control file data in the `BooleanSonicWidget` is replaced with the sound control file data for the MIDI device.

The next section will describe widget control file syntax in some detail.

3.5.5 Widget Control File Syntax

As described in 3.5.4, new sonifications can be created by writing widget control files which are later used to create sonic widgets.

Figure 3.14 shows a widget control file for a `MultiSonicWidget` that incorporates the MIDI sound control file of Figure 3.6 and another sound control file for the Sun SPARCstation audio device. This widget control file represents a `MultiSonicWidget` that tests its input data value x in the expression $x == 0.00$ to determine whether to execute the first or second sound control file. Recall that the first sound control file maps input data to pitches. The second sound control file plays a file of audio samples. This sound control file has no variable arguments, so the value of the input data does not affect the sonic messages that are sent when it is executed. Therefore, this widget control file represents a sonification that plays a note on a MIDI synthesizer for each data value not equal to zero. The pitch of the is controlled by the input data. A file of audio samples is played on a Sun SPARCstation audio device each time a zero is seen in the input stream.

As with the sound control file syntax, a `#` indicates a comment that contines to the end of the line; comments may also be delineated by `/*` and `*/`. A period (`.`) denotes the end of transformation function configuration information or sound control information. Sonic widgets that support multiple sound control files may list them one after another in the widget control file. Each sound control file listing must be separated from the next listing by a period, as shown. The first non–comment string of characters is the name of the class, or type, of sonic

```
MultiSonicWidget
4    #Boolean Transformation, to compute (x == 0.00)
0
0.00
.
# Sound Control File 0

# hostname  port name
localhost   midiaudio

# sonic message name      msg number arguments
/* Select Instrument   */     3      ?9
/* Set Balance         */    13       2
/* Set Sustain (Decay) */    11      ?0
/* Play Note           */     0      $0 ( /* Linear */ 1
                                  /* A =    */ 30.00
                                          /* B =    */ 60.00 )
                                     ?0.50
                                     ?127
.
# Sound Control File 1

# hostname          port name
flute.cs.uiuc.edu   sparcaudio

# sonic message name      msg number arguments
/* Set Output Port     */     4      ?0
/* Set Volume          */     0     ?47
/* Play Sample File    */     6      ?9      -1.00
.
```

Figure 3.14: `Demo.wcf`: A demonstration widget control file for a `MultiSonicWidget`.

<SonicWidgetClassName>
[*<transformation function information>* .]
[*<sound control data>* .]+

Figure 3.15: Outline of widget control file syntax.

widget that the widget control file describes. The rest of the information in the file depends on the type of sonic widget. In this example, the widget is of type `MultiSonicWidget`.

If the sonic widget manages a transformation function, the transformation function configuration information must appear following the name of the widget class in the widget control file, and it must be delineated by a period. As described in §3.5.4, a `MultiSonicWidget` manages one transformation function and an arbitrary number of sound control files. In Figure 3.14, the transformation configuration information is listed on lines 2–4. The transformation function number, on line 2, is `4`. This Boolean transformation function accepts, as user–supplied parameters, the identifier of an operation and a constant value. Lines 3–4 are the user–supplied parameters for the Boolean transformation function: the numerical identifier of an operation (`0`) and a constant value (`0.00`). As explained in §3.5.3, the operation identifier 0 corresponds to the boolean identity operation ==. Hence, this transformation function evaluates the expression $x == 0.00$ for each data value x.

A transformation function is used to generate the index of a sound control file to be executed. These are numbered, beginning at zero, in the order they appear in the widget control file, and are separated by a period (`.`). This sound control data is in the format described in §3.5.2.

Figure 3.15 outlines the syntax of widget control files. In summary, the name of the sonic widget class is followed by transformation function information (if necessary for configuration of the widget) and sound control data for each sound control file the widget manages. Transformation function information and sound control data must be in the format described in §3.5.3 and §3.5.2, respectively, and must be followed by a period.

3.6 User Interface

Configurable and variable arguments of sonic messages can be edited, through selection of valid arguments for the intended sound device, or selection of an appropriate transformation function, respectively. Sonic messages and the accompanying device default, device parameter, and message configuration files from the sound device server are enough to construct a graphical interface for this operation. A single class, `WclParams`, can generate an X resource file to configure a list of sonic messages (a sound control file), given the associated `SoundDevice` object.

For example, suppose the sonic message "Select Instrument" has a configurable parameter "Instrument". The device defaults table for the sound device indicates that "Instrument" is an enumerated type. The device parameters table has a list of all the valid instruments. The `WclParams` object writes WCL resource commands that use a scrolled list widget to present the available instruments to the user for selection. Another sonic message might need an integer parameter, "Volume," which the device parameters file indicates may range from 0 to 99. A slider with these bounds will be created to configure this parameter.

Since variable parameters are configured by selecting a transformation function, `WclParams` creates a push–button for each variable parameter that, when pressed, pops up a menu of available transformation functions. This menu, and the configuration panels for each transformation function, are specified by static X resource files that are also interpreted using WCL.

This functionality enables any application to provide a uniform, device–independent way to graphically configure sonic widgets. The configuration procedure for a sample sonic widget is illustrated in §3.7.2.

3.7 Application Programs

There are two demonstration application programs provided with Porsonify: `sdinterface`, which provides an interactive interface to a sound device server, and `porsonify`, which can create standalone sonifications. Because no sound device–specific knowledge is embedded in either of these applications, the applications can interact with newly written sound device servers without change.

3.7.1 Interactive Sound Device Interface: `sdinterface`

The `sdinterface` application connects to the host and port specified on the command line, and creates an X window panel of sliders and buttons that can be used to manipulate the device defaults, and a menu of sonic messages that can be sent to the sound device.

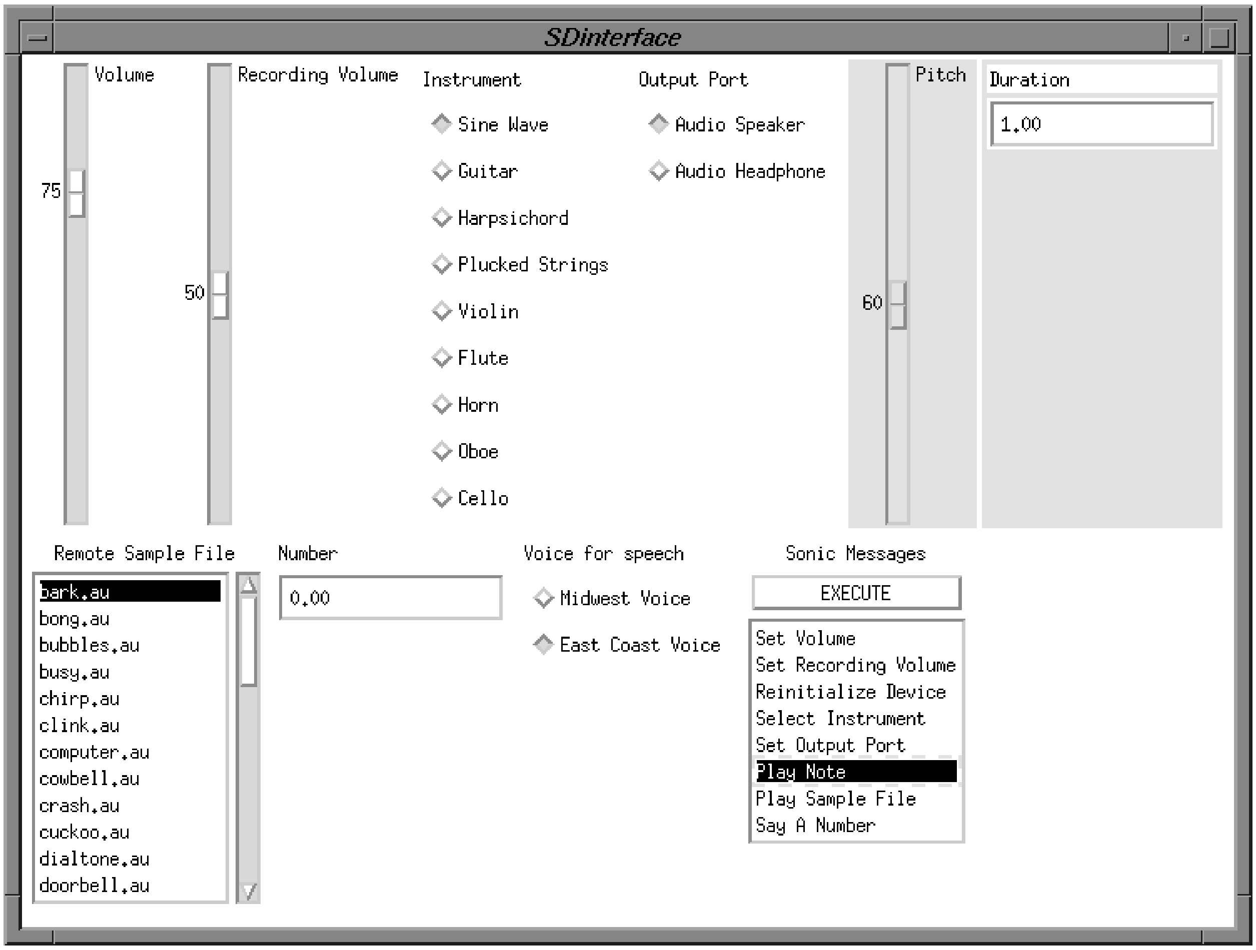

Figure 3.16: The **`sdinterface`** application, connected to SPARCstation audio device.

Figure 3.16 shows the default panel created by connecting to the port **`sparcaudio`** on the local workstation, with the command

```
sdinterface localhost sparcaudio
```

To send a command, one selects a sonic message from the **Sonic Messages** menu (in the bottom right hand corner), and clicks on the button directly above it, labeled **EXECUTE**. This action can also be accomplished by double–clicking on the menu selection. When a sonic message menu entry is selected, the arguments to that command are highlighted. Figure 3.17 describes the sonic messages supported by the Sun SPARCstation audio device.

In Figure 3.16, the **Play Note** sonic message has been selected, highlighting the arguments to that message: **Pitch** and **Duration**. Clicking on the button labelled **EXECUTE** will send the following sonic message to the Sun SPARCstation sound server:

Command	Description
Set Volume	Sets the play volume of the device.
Set Recording Volume	Sets the recording volume of the device. (Currently this is not too useful. Recording is not yet implemented.)
Reinitialize Device	Reinitialize the device defaults.
Select Instrument	Select an instrument to be used when playing notes.
Set Output Port	Direct output to the speaker or headphone.
Play Note	Play a note for a certain duration. The pitch number 60 corresponds to middle C on a piano keyboard, and each whole number is one key up or down.
Play Sample File	Play a sample file which has been installed on the system.
Say a Number	Say a number (this is done by concatenating sampled spoken numerals).

Figure 3.17: Sun SPARCstation audio device sonic messages.

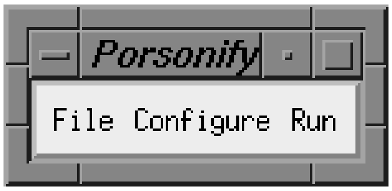

Figure 3.18: The `porsonify` application main menu.

```
"Play Note" 60 1
```

This causes the SPARC audio device to play a middle C (pitch number 60) for 1 second (the selected duration).

Graphically adjusting parameters has no effect until the appropriate message is sent to the sound device. For example, if you move the volume slider, this change will not take effect until you send the **Set Volume** command from the **Sonic Messages** menu.

3.7.2 Creating Sonifications: `porsonify`

The `porsonify` application creates a sonic widget; it accepts as an argument the name of a widget control file, allows one to select a data file to sonify, and permits configuration of the sonic messages that are sent to the sound device.

A static X resource file used with WCL defines the X interface to the `porsonify` application. It is simply a small menu bar, shown in Figure 3.18.

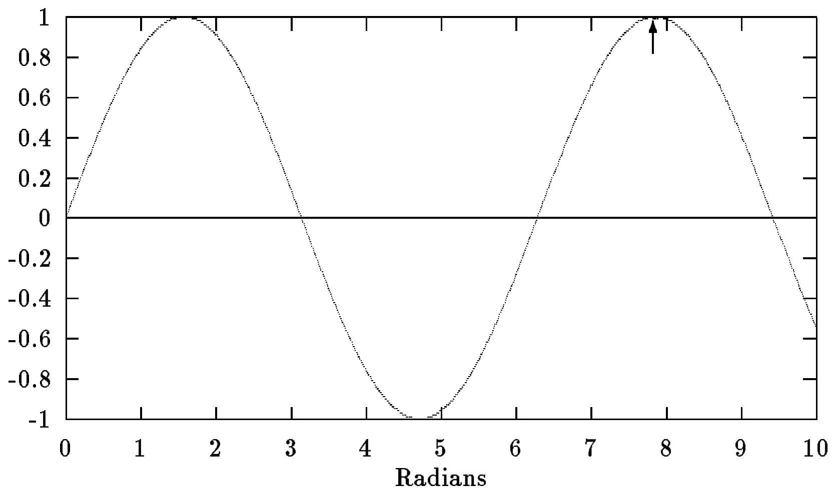

Figure 3.19: A sine wave with errors.

The rest of this section describes the use of `porsonify` to create and configure a demonstration sonification from the widget control file shown in Figure 3.14. The communication between clients and servers incurred by this sonification is depicted in Figure 2.3.

3.7.2.1 Selecting a Data File

Because the `porsonify` application maps a sequence of data values to sound, a data file must be specified. This can be accomplished by selecting **New data file** from the **File** menu of the `porsonify` main menu. As an example, we have selected a data file that contains samples of a "defective" sine curve ($y = sin(x)$) with x ranging from 0 to 10 radians. Figure 3.19 shows a plot of this curve, with the errors marked by an arrow. In the second peak, two data values are slightly smaller than they should be. The irregularity in this curve is difficult to detect visually, but it is easy to hear. When each data value is mapped to pitch, instead of a smooth rise and fall over the peak, one hears a "bumpy" pattern of high and low notes that is distinct from the pattern of notes heard in the first peak.

3.7.2.2 Configuring the Sonification

In this example, all the information needed to sonify each data value is contained in the widget control file `Demo.wcf`, shown in Figure 3.14. It is possible to interactively change this informa-

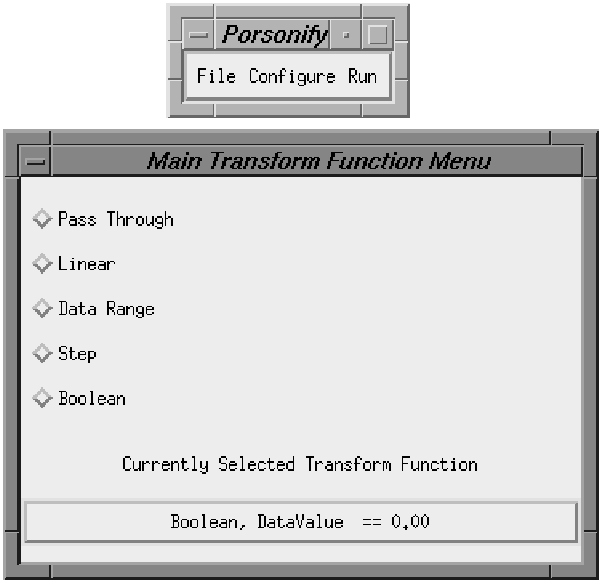

Figure 3.20: Configuration of the main transform function for `Demo.wcf`

tion to customize the sonification. Furthermore, some valid widget control files do not specify transformation functions for variable arguments of sonic messages; these may be selected at configuration. `Demo.wcf` describes a `MultiSonicWidget`; first we configure the transformation function that controls which sound control file will be executed, and then the sonic messages that are sent.

Selecting **Configure** from the `porsonify` main menu produces the display shown in Figure 3.20. As Figure 3.14 shows, `Demo.wcf` contains two sound control files. The first sound control file uses a MIDI audio device on the local host to play a note for each data value. The pitch of the note is controlled by the data value. The second sound control file uses the Sun SPARCstation audio device to play a file of audio samples. The Boolean transformation function in Figure 3.14 will perform some operation on the input data to produce a number which corresponds to the index of the sound control file which should be executed.[2] Then that sound control file is executed with the *original* input data value. Any variable parameters in the sound control file are instantiated using the unmodified input.

In this example, all values of $sin(x)$ are mapped to pitches in the range 30 to 90, except for the value 0, which causes a sampled file to be played on the remote SPARC audio device. This

[2] Recall that constituent sound control files are numbered beginning at 0 in the order they appear in a widget control file.

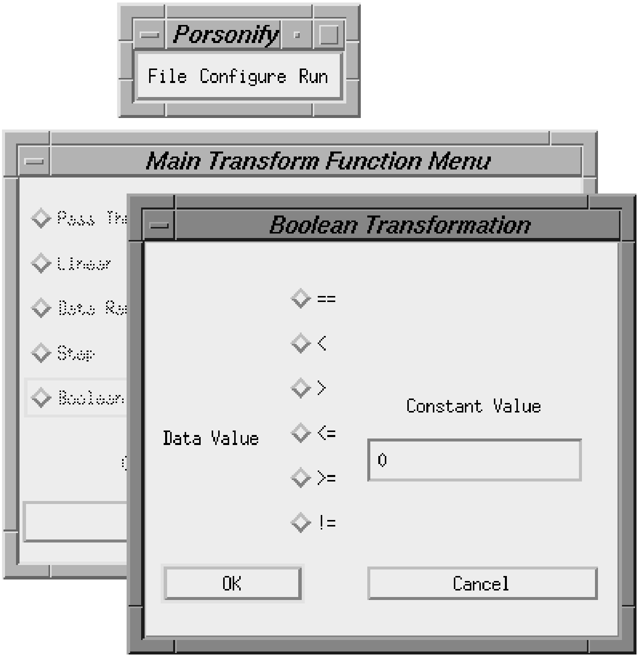

Figure 3.21: Configuration of the boolean transform function for `Demo.wcf`

alarm provides a reference point that can be used to determine when the x–axis is crossed. A boolean transformation function, shown in the label on the button at the bottom of Figure 3.20, has been selected. Each data value x is tested in the expression $x == 0$. If the test returns true, the transformation generates a 1, indicating that the second sound control file in `Demo.wcf` should be executed. If the test fails, a 0 is returned, and the first sound control file is executed.

Pressing the button at the bottom of the screen will continue configuration of the sonification without changing this transformation function. To change the currently selected transform function, one would select one of the options by clicking the mouse on the radio button to the left of the desired data transform. These transformation functions are the same as described in §3.5.3. Each transformation function has its own associated menu to prompt for required information, described by a WCL file. For example, clicking on the radio button beside **Boolean** produces the panel shown in Figure 3.21. A different constant value or a new boolean operation may be selected. Pressing the button marked **Cancel** would cancel any changes. Clicking on the **OK** button accepts any changes, dismisses all windows related to transformation function configuration, and continues with sonification configuration.

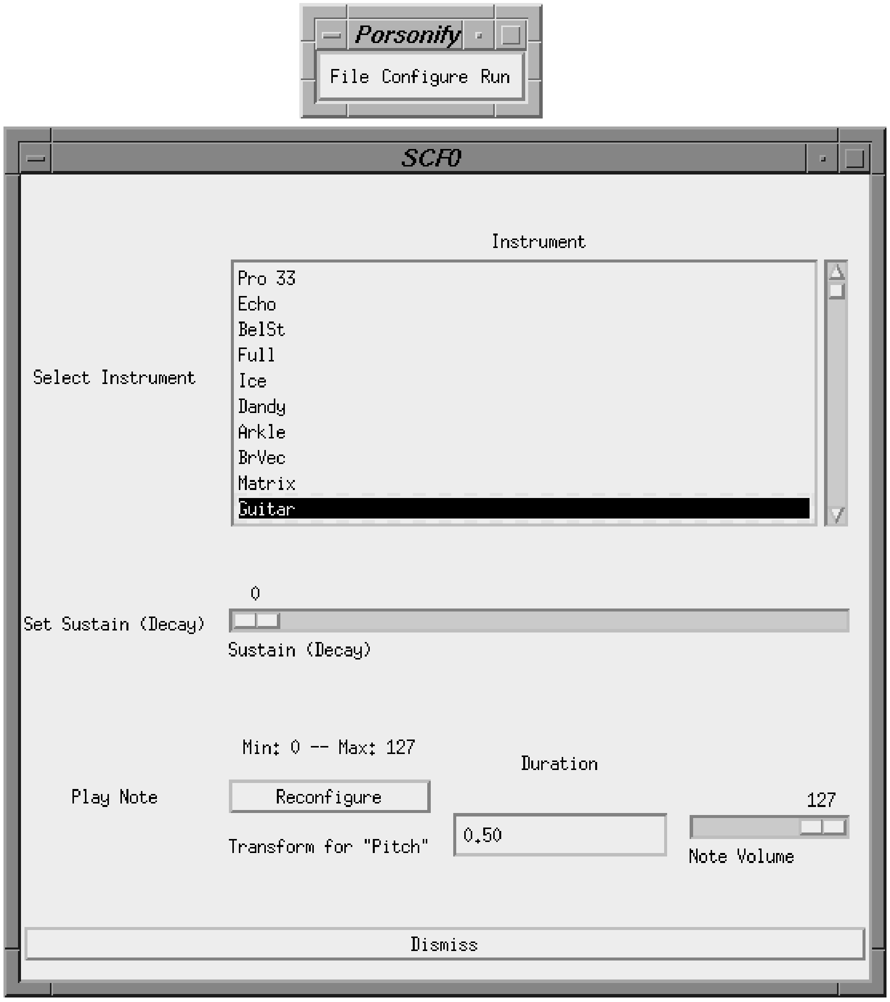

Figure 3.22: Configuration of the first sound control file in `Demo.wcf`.

The configuration menu for the first sound control file will now appear, as shown in Figure 3.22. This sound control file, intended for the Yamaha TG33 Tone Generator sound device, executes four commands for each data value it presents. It selects an instrument, sets the stereo balance, sets the sustain, and plays a note.[3] However, only three commands have configurable or variable parameters (see Figure 3.14); the "Set Balance" command has a fixed parameter so it cannot be changed interactively. The default values for the selected instrument, sustain, duration, and note volume correspond to those in Figure 3.14.

[3] Setting the instrument and balance for every data value is usually unnecessary; this is shown for illustration.

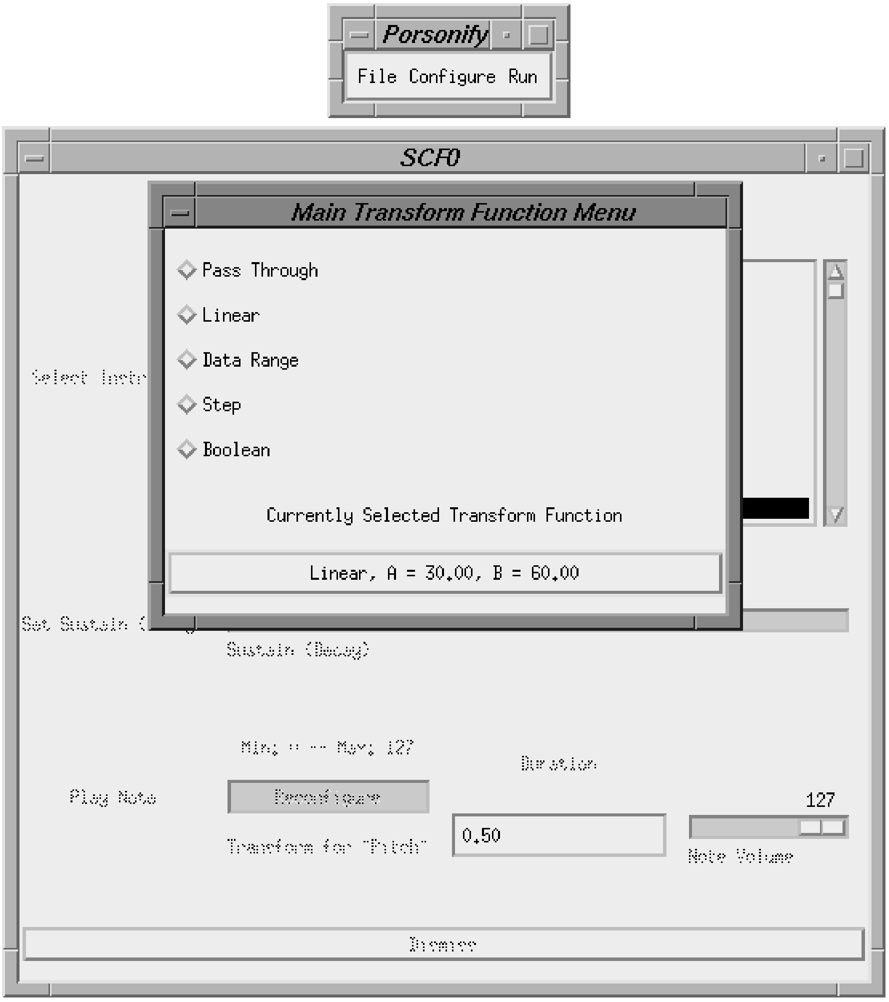

Figure 3.23: Configuration of **Pitch** in `Demo.wcf`.

Pitch is a variable argument, and has an associated transformation function instead of a default value. The button marked **Reconfigure** next to the **Play Note** label is used to reconfigure this transformation. Pressing this button creates another transformation function menu, as shown in Figure 3.23. Notice that the label for the currently selected transformation function is **Linear, $\mathbf{A = 30}$, $\mathbf{B = 60}$**. This means that the pitches of the notes that are played for data values is determined by the formula $30 * x + 60$. Since the values of $sin(x)$ in our data file fall within the range -1 to 1, this expression maps them into the range 30 to 90, a reasonable range of pitches. This transform function can be changed just like the boolean transformation function earlier. For example, it is possible to play the same note (e.g., pitch 60) for each data value by selecting the linear transformation function $0 * x + 60$. To do this, one would click

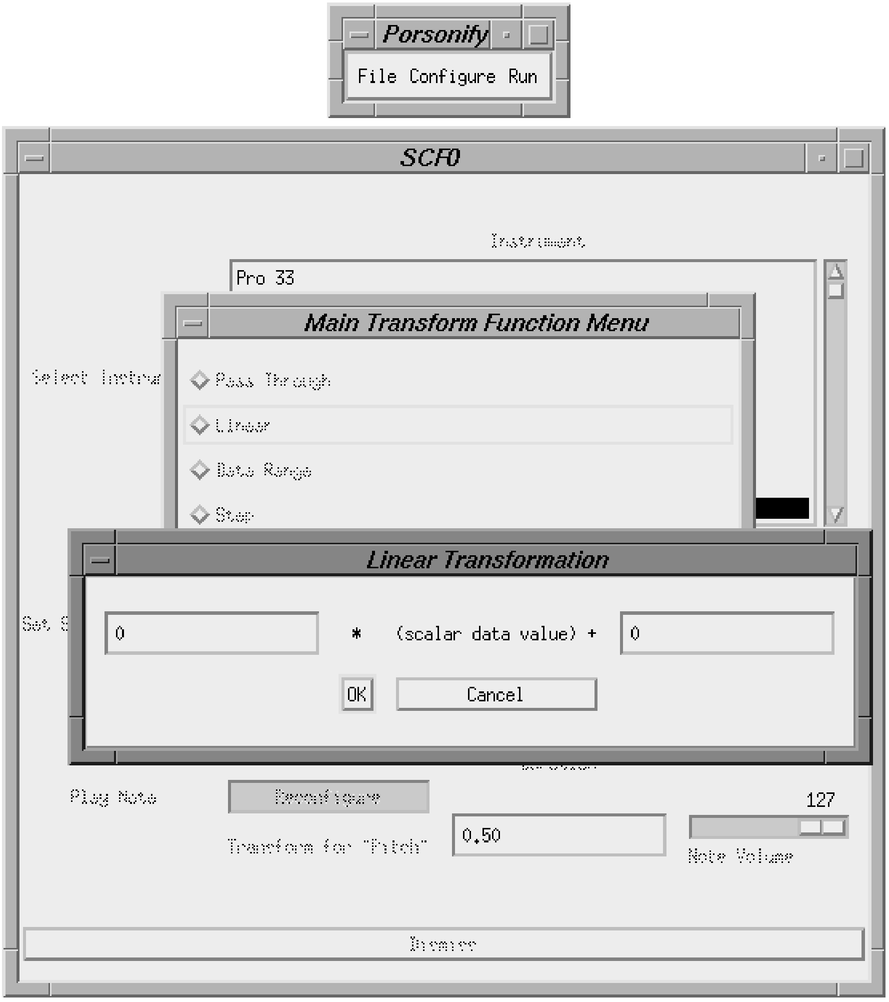

Figure 3.24: Changing the linear transformation function for **Pitch** in `Demo.wcf`.

on the button marked **Linear**. The linear transformation configuration panel will pop up, as shown in Figure 3.24. The new values for the constants can be typed into the text boxes on the right and left. As before, pressing **OK** will dismiss all transformation configuration panels and continue with sonification configuration.

When configuration of the first sound control file is completed, its configuration panel can be dismissed by pressing the **Dismiss** button at the bottom. The configuration panel for the second sound control file, which plays a sample file on the Sun SPARCstation audio device, appears as shown in Figure 3.25. This sound control file contains commands to set the output port, volume, and play a sample file for each data value it presents. The output port, volume,

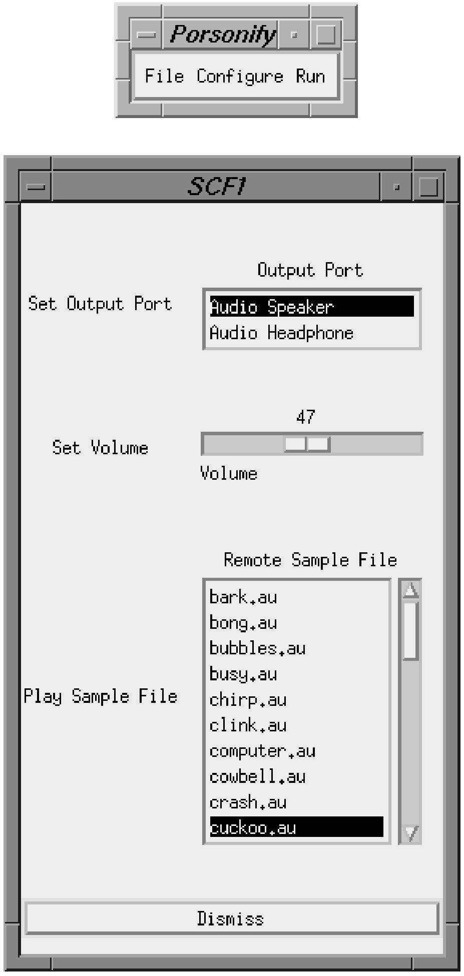

Figure 3.25: Configuration of the second sound control file in `Demo.wcf`.

and sample file are configurable parameters and may be edited. Clicking on **Dismiss** will close this panel.

At this point, the main transformation function and all sound control files managed by `Demo.wcf` have been configured and the sonification can be run.

Changes to the sonification configuration can be saved by writing a new widget control file. This file can be used with `porsonify` to recreate the sonification with the new characteristics. To do this, one would select **Write WCF File** from the **File** menu.

3.7.2.3 Running the Sonification

Selecting **Start** from the **Run** menu begins the sonification. The sonification can be stopped by choosing the **Stop** menu selection from the **Run** menu. If you choose to start it again, it will start from where it was last stopped. When all the data values have been read from the data file, the sonification stops. Either another data file can be sonified, or the current file can be "rewound" and played again.

Chapter 4

Sound and Visualization

The `porsonify` application is instrumental in the creation of sonifications, however, it provides no *graphical* output of the data. Ideally, we would like to accompany sonifications with visual displays. More generally, integration of sound into arbitrary applications should be easy to encourage experimentation with sonification.

This chapter describes the integration of Porsonify into the Pablo performance analysis environment [18]. The device–independent design of Porsonify permits Pablo to utilize any available sound hardware without any device–specific code. Information about available sound devices and how to construct sonifications for them is kept separately in a configurable directory. Porsonify can be integrated into any application using a similar approach.

4.1 The Pablo Visualization Environment

The Porsonify software coupled with the Pablo performance analysis environment is an excellent arena for experimentation with coordinated sonification and visualization of data.

Pablo is an extensible, portable performance analysis environment capable of supporting the analysis of data drawn from both shared and distributed memory parallel systems. It consists of a set of data transformation and visualization modules which can be interactively interconnected in a graphical workspace to form a directed, acyclic data analysis graph. These graphs can be configured, saved and loaded. At the highest analysis level, performance trace events are converted to a self–describing data format [1] that includes internal definitions of data types, sizes, and names. A trace file in this format consists of a set of record definitions

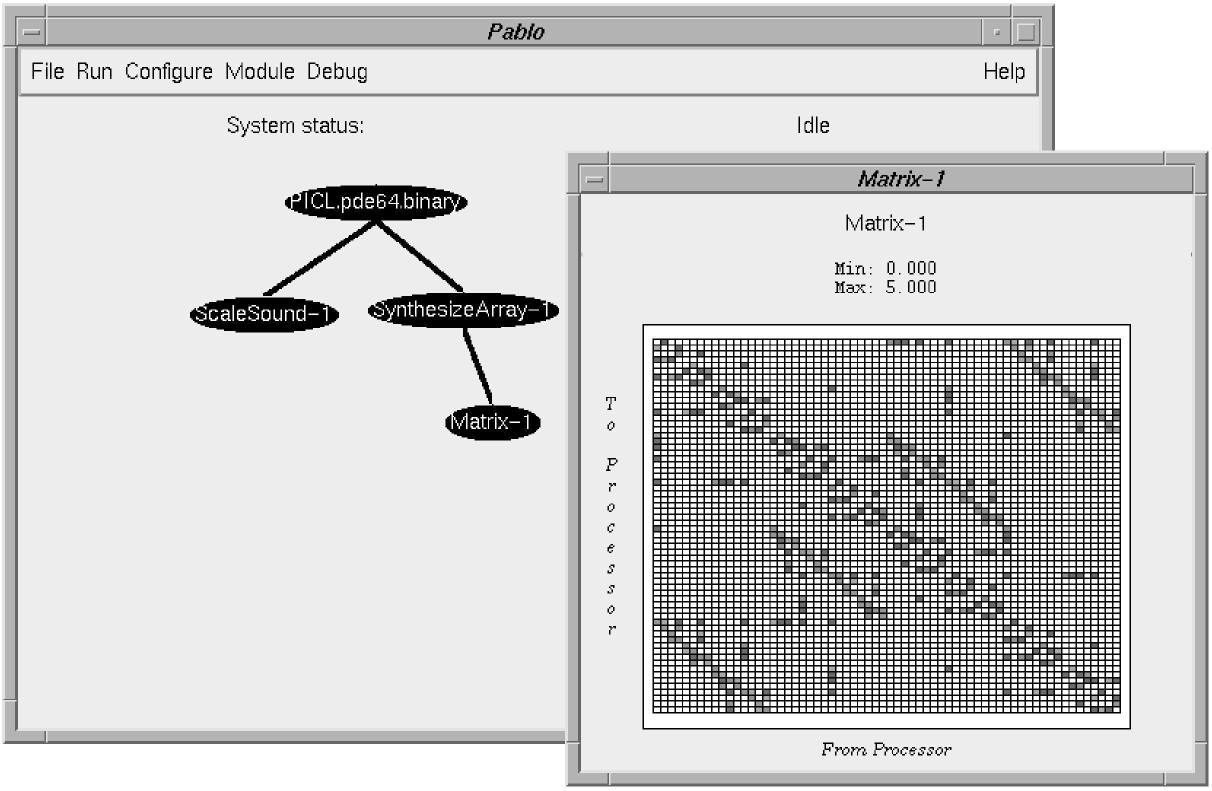

Figure 4.1: A very simple Pablo configuration.

(much like C structures) and a stream of record tag/data pairs. Each tag identifies the type of the record to follow.

Using the tags and record descriptions from the input trace data file, one can select record fields to be passed into the input ports of each module, and define new record types to be output to descendent modules. Although intended for analysis of performance data, the flexibility of the input data format and analysis graph configuration enable Pablo to accommodate any kind of data. Figure 4.1 is a simple example of how this environment works.

This graph creates a matrix of counts of messages sent between processors, and maps the number of each sending processor to a note in a scale. At the root of the tree is the node labeled `PICL.pde64.binary`. This node represents the input trace file; it was generated by a partial differential equation solver running on a 64 node Intel hypercube. It contains records representing events such as sends and receives between processors and communication statistics. In the left subtree, `ScaleSound-1` has been configured to map the processor identifier field from send records to notes in a chromatic scale. In the right subtree, `SynthesizeArray` uses the sending and receiving processor identifiers as indices to construct an array of message counts, which is displayed on the `NumberMessagesSent` matrix.

Integration of the Porsonify sound toolkit into Pablo involved creation of a set of sonic displays, selectable as leaves of the data analysis graph like graphical displays (e.g. Chart, Matrix).

4.2 Implementation of Sonification Modules

Within the Pablo infrastructure, data transformation and display modules are called *functional units*. A set of very simple sound functional units has been added to Pablo, but using or incorporating arbitrarily complex sonifications is trivial. The collection of local or remote sound devices that support each sonic mapping, and even the execution of the sound, is completely configurable by each user. For example, the Sun SPARCstation is monaural; it does not support mapping of data to stereo balance. A creative Pablo user can use all the SPARCstations in an office as audio output devices to implement a simple stereo mapping simply by writing the widget control file to do this and placing it in a specified directory. This change would not have to be built into the system, nor would it affect anyone else.

Each sound functional unit is associated with one sonic widget, which defines the sonification. Certain functional units are named for the widget control files used to to create the core sonic widgets, and others permit the user to select a widget control file by name. The sound functional unit is just a medium to invoke a sonic widget.

Functional units in Pablo are polymorphic, performing operations on data of different types. All sound functional units accept scalars and vectors of integers and single and double precision floating–point numbers, which are used as parameters to execute a widget control file. The functional unit itself does not check whether the size of the vector corresponds to the number of variables in the widget control file; this way, the widget control file can be edited, changing the sonification, after the Pablo data analysis graph has been constructed.

Since each sound functional unit is essentially just a wrapper around a widget control file used to create a sonic widget, sound functional units are very similar. The most general units, `SynchSoundFU` and `AsynchSoundFU`, prompt for the name of a widget control file and use it to create a sonification that is either synchronized with visual displays, or not, respectively. The other sound functional units have names that correspond to the name of a widget control file. This convention is intended to make configuration of commonly used sonifications easier.

A new sonification can be integrated into Pablo by writing a widget control file to describe it and selecting that widget control file to configure either `SynchSoundFU` or `AsynchSoundFU`. The operation of saving a Pablo configuration graph causes the sonic widgets in the sound functional units to save their configurations. When the Pablo graph is reloaded, the saved widget control file will be used to reconfigure the sound functional unit. If the sonification is heavily used, a sound functional unit which is named for that particular widget control file can be added to the Pablo palette of sound functional units.

An environment variable, `WCF_DIR`, is set to the name of a machine–specific directory where the widget control files are kept, in subdirectories corresponding to each sound device accessible from the workstation running Pablo. Each subdirectory contains its own set of widget control files for that sound device, since these files contain sound device specific commands. There is a file called `.description` in `WCF_DIR` which has a description of the sound device associated with each subdirectory.

Accessible sound devices are any devices on remote or local machines that could potentially be used for sound display. For example, suppose there are two Sun SPARCstations in an office, named `sparc1` and `sparc2`, and that `sparc1` has a MIDI synthesizer connected to it. The `WCF_DIR` on the host `sparc1` might have three subdirectories: `SPARC-local`, `MIDI-local`, and `SPARC2`. The `.description` file might look like this:

```
SPARC-local    SPARC audio chip on the local host (sparc1).
MIDI-local     MIDI synthesizer on the local host (sparc1).
SPARC2         SPARC audio chip on host sparc2.
```

When a sound functional unit that corresponds to the name of a widget control file (e.g., `ScaleSound`), is configured, each subdirectory of `WCF_DIR` is searched for a widget control file of the name `ScaleSound.wcf`. There are three possibilities. If no widget control file by that name exists in any of the subdirectories, the functionality represented by that sonification is not supported by any accessible sound devices. Therefore, the sound functional unit cannot be configured and will never be run. If there is only one widget control file of the appropriate name, that widget control file is used, regardless of what device it is written for. If there is more than one widget control file, a menu of the descriptions of each device (from the `.description` file) is offered to the user. The selected device determines which widget control file will be used.

In this way, new sound devices can be integrated seamlessly into Pablo by creating a subdirectory in `WCF_DIR` for the new device, adding its description to the `.description` file, and writing widget control files for each supported sonification.

The current set of sound functional units with associated sonic widgets is listed in Figure 4.2. Most of these play a note for each data value, varying one sound characteristic according to the data. Although very simple, these kinds of displays can be useful for augmenting standard graphical displays. When many different events are represented in several different visualizations, they can provide aural feedback to indicate where to look. For example, changing pitch might be associated with one type of data record and changing volume with another. **SayNumber** is similar, although it "displays" a data value by "speaking" it. **NoteAlarm** and **SampleAlarm** are simple audio cues. They play a note or a sampled sound, respectively, when the input data satisfies some condition. These are useful in the Pablo environment for calling attention to certain interesting events; one does not need to constantly monitor visualizations. **NoteOn** and **NoteOff** turn a note on and off, respectively. These may be used in conjunction with **VaryPitch**, which changes the pitch of the sustained note, or by themselves. These are useful for playing overlapped notes to provide a sense of magnitude or to indicate that many events are happening concurrently.

This set is provided as a starting point for sonification design. Once the value of varying one sound characteristic at a time is determined, the widget control files used to create these sonifications can be extended to allow multiple characteristics to be varied with more input data parameters.

Functional Unit Name	Description
BalanceSound	A note is played for each scalar input data value. The stereo balance of each note is determined by the input data.
DurationSound	A note is played for each scalar input data value. The duration of the note is determined by a configurable transformation of the data value.
NoteAlarm	A note is played if a configurable boolean condition is satisfied for each input data value.
NoteOn	A note whose pitch corresponds to the scalar data input, or the first element of an input vector, is begun. It continues playing until a corresponding **NoteOff** functional unit is executed.
NoteOff	Turns off a note started by previous execution of a **NoteOn** functional unit, whose pitch corresponds to the scalar data input.
Player	Accepts a character string describing a sound command and a timestamp, pauses for the delta between the previous and current timestamp, and executes the sound command.
SampleAlarm	A file of sampled audio data is played for each data value that satisfies a configurable boolean condition.
SayNumber	The input data value is "spoken".
ScaleSound	A note is played for each scalar input data value. The pitch of the note is determined by a configurable transformation of the data value.
TimbreSound	A note is played for each scalar input data value. The timbre, or voice, of the note is controlled by a configurable transformation of the data value.
VaryPitch	Controls the pitch of a note that has been turned on by previous execution of a **NoteOn** functional unit, according to the input scalar data value.
VolumeSound	A note is played for each scalar input data value. The volume of each note is controlled by a configurable transformation of the data value.

Figure 4.2: Sonifications in Pablo.

Chapter 5

Sonification Experiences

Once Porsonify was integrated with the Pablo performance analysis environment, it was easy to construct a variety of sonifications of different kinds of data. This section describes some of the sound mappings which seemed to be most useful, and general observations and conclusions about various styles of aural "displays." No formal experiments were conducted to determine the effectiveness of any of these methods of presentation. They illustrate the flexibility of the software and are suggestions to inspire further work in this area.

5.1 Multivariate Data

Multivariate data with more than three dimensions is challenging to visualize, because more creative means than plotting must be used to represent the higher dimensions. Chernoff's faces [5], for example, is a technique to aid data clustering by mapping variables of a data sample to facial characteristics, e.g., length of nose, size of eyes. Each face represents an individual sample.

Chernoff faces are analogous to Bly's approach to sonifying multivariate data, where the four variables of each data sample were mapped to pitch, volume, duration, and fundamental waveshape of a note [3]. The resulting notes enabled listeners to classify each sample into one of three related sets. Sound mappings of this type share many of the same problems as Chernoff faces. Face parameters are not completely independent, and extreme values of some face parameters cause others to lose their effect [7]. In the same way, sound characteristics are interrelated, and it can be hard to distinguish the timbre of a very short or high–pitched note, or the pitch of a very short note, etc. Some assignments to face parameters are better than others for illustrating set groupings; the same is true of sound. Not only do some sound parameters carry more weight than others (for example, pitch and rhythm characterize a melody that can be carried by any number of instruments, thus can be considered more significant than timbre), but the range of each sonic parameter differs widely across sound devices and among other parameters on the same device. Certain mappings could imply loss of critical information.

5.1.1 Direct Mapping

This technique of direct mapping is easy to replicate in Porsonify (although the Sun SPARC-station and MIDI hardware platforms support variation of sound characteristics different than those used in Bly's experiments). The data used to experiment with multivariate data sonification was taken from the Places Rated Almanac [4]. In this data set, each of three hundred and twenty–nine cities in the United States has a rating for climate and terrain, housing, health care and environment, crime, transportation, education, the arts, recreation, and economics, in addition to a latitude, longitude, and population. These twelve parameters provided ample opportunity to experiment with different mappings without attempting to solve any particular analysis problem.

For example, in one sonification, the recreation index for each city was mapped to pitch (one of 36 over four octaves in a C major scale). Population was scaled to generate a duration for each note up to five seconds. The climate index determined the timbre; three instruments were chosen to represent the range of weather. Bad weather was a raspy digital sound, average was a flute, and above average weather was a pleasant mixture of bells and strings. The volume was controlled by the housing cost index. Longitude was mapped to stereo balance, since it is inherently locational and most people can envision maps of the U.S. where higher longitudes are to the right. In this way, each note represented four variables of each sample.

5.1.2 Sonic Scatter Plots

It seems that this sort of presentation overlooks potential interpretations of time. Like a drawing without perspective, it does not attempt to create the illusion of higher dimension. Specifically, in the above example, both the absolute and relative positions of notes in the sonification are quite arbitrary. Additional information could be provided simply by sorting the notes by a fifth, "silent" variable, or by one of the four variables controlling the sonification. Furthermore, the duration of the entire sonification has no meaning, except as the sum of the variable mapped to duration of notes. This can be useful if comparing two data sets where this statistic is valuable. Otherwise, it could be used to mean something more significant.

Notes can be "plotted" in time, creating a multi–dimensional sonic scatter plot. One variable, (e.g., population) is selected to be the time axis. The entire data set is sorted by that key. Next, the duration of the plot is chosen, and the key variable is scaled to that range of seconds. For example, population in this data set ranges from 62820 to 8274961; if the user wishes the sonification to last for one minute, the population will be scaled to the range 0 to 60. These scaled values represent logical timestamps for notes; a note whose pitch corresponds to some other variable (e.g., the arts) in the current sample is sustained until the timestamp of the next sample.

This kind of sonification is especially useful for detecting the correlation between different parameters. For example, the arts index is highly correlated with the population; the pitches played are generally increasing. It is also easy to get a qualitative feel for the distribution of the key variable. In the sonic scatter plot of population and the arts, it is distinctly audible that although the population range is large, most cities have relatively small populations; many notes are played within a very short time at the beginning. It is also clear that the average arts index is low as well; most of the pitches heard are low, except for the ones corresponding to very highly populated cites (e.g., New York), at the very end of the sonification.

These characteristics can best be appreciated in contrast to the same sonification of two different variables (e.g., climate and recreation indices) which have little correlation. If the climate index is used as the time axis, it is obvious that most cities have average climates. Clustering of many low pitches, indicating low recreation ratings, occurs in the middle of the sonification.

This sonic scatter plot can be increased to three or more dimensions in several ways. Perhaps most direct approach is to use other variables to control the location, timbre, etc. of each note. As in the first sonification, stereo balance can be mapped to longitude, weather to timbre, and the recreation index to pitch for a total of four dimensions. One concern is that as the number of variables encoded in each note increases, the tempo of the sonification should decrease, in order to allow time to assimilate the extra information.

Another approach to increasing the number of data dimensions presented is to play two sonic scatter plots simultaneously, one in the left speaker and one in the right. Obviously, balance cannot be used in this case to convey a separate variable. This stereo scatter plot approach seemed to work well at the same brisk tempo with two variables encoded in pitch, plotted against a third.

5.2 Performance Data

The method of using each note to represent a data point can work with other kinds of data sets. This technique was used to sonify an execution trace from PDE code (a partial differential equations solver) running on an Intel iPSC/860 hypercube of 64 nodes, a message–based parallel system. This trace file was captured using the PICL subroutine library [11], and converted into SDDF format to be used with Pablo. The trace file consists of records of different types. Each record includes a timestamp, and other information depending on the record type. Records document events such as hardware sends and receives between processors, message sizes, processor idle time, and other such statistics.

This application is communication bound; processors are idle roughly 70 percent of the time. To convey an impression of the heavy communication patterns, a note is played for each send and receive. The pitch of the note corresponds to the sending processor identifier (processors 0–63 were mapped into pitches 30–93). The location (left or right) in the stereo balance field indicates whether the note represents a send or a receive, respectively. The order of the notes corresponds to the ordering of sends and receives in the trace file, but there is no indication of how much trace time passed between events.

In the sonification of the PDE code, notes are scattered throughout a wide pitch range, and sends and receives are relatively balanced; although in the beginning of the application there are bursts of sends (long sequences of notes in the left of the stereo balance field) and corresponding bursts of receives, communication reaches a stable state where a few sends are followed by a few receives. The pitches of these notes are scattered over the entire range; it is immediately evident that no single cluster of processors is responsible for a large portion of the communication.

This sonification can best be appreciated in contrast to the same sonification of a trace file with a strikingly different communication pattern. For example, in a trace of a recursive matrix transpose, also executed on a 64 node hypercube, all processors first send and then receive. Therefore, all 64 pitches are heard in the left, and then all 64 are heard in the right.

Instead of playing a discrete note for each send and receive, it is possible to start a note at the send and resolve it at the corresponding receive. This approach seemed impractical for trace files from 64 processors; too many notes were being sustained to detect any patterns. It provided interesting insight, however, into a trace of simplex code running on an 8 node Intel iPSC/2 hypercube. The simplex method is an algorithm used to solve linear optimization problems. This trace file was generated using the Crystal trace capture library [19]. As above, the pitch of a note corresponded to the processor identifier, however, nodes 0–7 were mapped to pitches in a C major scale to create more musically recognizable patterns. A note was started for each send and finished at the corresponding receive. All notes originated from the same location, and a synthesized instrument with a long, natural sustain (Bells and Strings) was selected to carry the notes without sounding strained.

In each iteration of the simplex code, a global minimum communication process is used several times to identify and distribute the global minimum. In this process, first all local minima are passed from the upper half of the hypercube to the lower half. Then the lower half passes the newly computed minima from its upper half to its lower half. The process repeats until one node (processor 0) has the global minimum, and it broadcasts that value to every other node. This computation creates a very distinct pattern of notes that is repeated frequently. Broadcasts from processor zero can easily be detected as the same pitch repeated seven times in succession. The effect of a broadcast is that the pitch corresponding to that processor sounds extremely powerful until the broadcast message is received, and it starts dying down as the individual notes are resolved.

One can quickly detect from this sonification message transmissions without corresponding receipt; they are audible as trailing sustained notes. In fact, this sonification revealed an error which had not been detected in multiple years of visualization; it was erroneously recorded that a processor had sent a message to itself. A variation on the sonification helped to isolate the error, which was difficult to identify by pitch alone. A different voice was assigned to each processor. Among eight processors, it was possible to immediately detect which processor had an unmatched send by the timbre of the sustained note.

It might seem counterintuitive, given the nature of the trace data, that neither the duration of notes nor their starting times correspond directly to any actual timing information in the trace file. For example, in the matrix transpose trace, although the relative order of the sends among the processors can be heard, the amount of time between each event could vary radically. As described below, time can be used to great advantage in conveying additional information. However, there are several practical reasons why we might not always want to do this.

A natural correlation between some characteristic of the data and time in the sonification might not always exist. Even if a logical mapping is obvious, such as using timestamps in a trace file to control when notes are played, it may not be feasible to do this. Events might occur in very short, sparse bursts over a long period of time, so that it would be impossible to hear what is happening during busy periods, and tedious to listen to the entire sonification. Scaling the time (i.e., letting each second of trace time be ten seconds of sonification time) alleviates the problem only if one can selectively listen to bursts of activity; otherwise it can quickly become impractical to listen to the entire trace. Furthermore, if the sonification is to be played with a corresponding visualization, the time to update graphical displays and process records that do not contribute to a sonification can easily exceed the data time. In such cases, a graphical data–time clock could be used in conjunction with the sonification to illustrate time distortion. However, when no reasonable correlation between data and time exists, or when it is difficult to successfully incorporate time into the sonification, it is best to default to a simple mapping.

5.2.1 Aural Timelines

Rhythm seems to be a natural mechanism for conveying a qualitative impression of the relative duration and timing of events, represented by the individual notes.

We obtained an execution trace of code from the Stanford Parallel Applications for Shared Memory (SPLASH) benchmark suite [21] running under the *Choices* operating system on the Encore Multimax. The benchmark run is a many–body molecular dynamics simulation (WATER). *Choices* is an object–oriented operating system designed to support experimentation with resource management policies on shared and distributed memory parallel systems. To support this experimentation, Choices provides software tools to collect performance data from the operating system. The *Choices* context switching code was instrumented to record the transitions among *Choices* system processes and the WATER application processes. Each of the context switch records contains a timestamp, processor identifier, the duration for which the previously running process executed, and the identifiers of the previously executing process and the process that is about to gain control of the processor.

These kind of timestamped events lend themselves nicely to construction of aural timeline sonifications. Every note that is represents an event that is occurring for the entire duration of the note. Other qualities of the note (e.g., timbre, stereo location, volume, pitch) may be used to represent additional or even redundant information about the event, as simple mappings do. The time notes are played in the sonification corresponds to the time they actually occur.

If pitch variation is used, this kind of sonification can draw upon people's natural ability to recognize melodies. It also uses the fact that sound is temporal to great advantage. At any time one hears only the events that are actually occurring. When a "normal" aural pattern has been established, abnormalities are immediately evident as breaks in the normal rhythm.

Several sonifications of the *Choices* context switch data on a dual–processor Encore Multimax were performed using aural timelines, to characterize the behavior of processes under the *Choices* operating system. The purpose of these sonifications was to gain a qualitative sense of the length of time spent in each application process. A note with a pitch assigned to each process started when the process began to execute, and stopped when that process was preempted by another.

First this sonification was heard for each of the two processors individually. Notes generated this way do not overlap; however, certain patterns were immediately evident. For example, the `IdleProcess`, an operating system process which runs when nothing else is ready to run on the CPU, was assigned a low pitch. This could frequently be heard as part of staccato cycles between other network daemons with higher pitches.

To listen to both processors together, stereo balance is used to distinguish events. All processes executing on the first processor are heard in the left of the stereo balance field, and all processes on the second are heard in the right. This sonification reveals an even more interesting pattern. When a process is preempted on one processor, it is usually quickly picked up by the other, effectively bouncing back and forth across the stereo field amidst the low tone of the `IdleProcess`.

As soon as the application process, marked by a comparatively high pitch, began to execute, aural patterns changed dramatically. Typically the application process would execute for substantially longer periods of time when nothing else was available to run; this was distinctly audible as sustained high and low notes, representing the application process and the `IdleProcess`, respectively. Furthermore, this implementation of the WATER code forked and executed in parallel on both processors. During this period a high pitch could be heard in the left and the right.

Chapter 6

Conclusions

6.1 Future Enhancements

The sonic widget set provided by Porsonify is just a start at providing a framework for data–driven sonification. These sonic widgets do not offer control structures; they simply perform actions based on very simple transformations on data. A general purpose data analysis program (such as Pablo) reduces the data to the point where these sonic widgets can be applied. Instead, one might envision tightening the connection between data analysis and display, so that algorithms used to create sound can dynamically provide feedback to data analysis modules, changing the sonification as appropriate. A single control mechanism should exist for creating data–driven soundtracks and animations. For example, one could imagine weaving a path through a three–dimensional scatter plot, "brushing" against points that resound with different timbres and pitches, revealing additional variables in each sample.

The sample sonifications described in Chapter 5 are simply suggestions to inspire more experimentation. There are many possible sonification techniques besides simple mappings. For example, a data sample need not correspond to a single note or chord; it is possible to use data to change parameters of an entire musical phrase. The tempo or key of a simple, repeated motif might change with the data, perhaps while another sonification is being played in the foreground.

The effectiveness of different kinds of sonifications must be carefully researched before we can establish a set of standard aural "displays." Once guidelines are created, much work could be done to create some high–level algorithms for reading in a data set, characterizing it, and given some rules for a particular sound device, generating appropriate data to sound mappings. This is a non–trivial endeavor; not only would one have to know the commands the device supports, but things like the ranges of different instruments, what kinds of voices sound good together, etc. Clearly, much work remains to be done before sonifications are as recognized, accepted, and understood as their visual counterparts.

6.2 The Porsonify Design

In practice, Porsonify has satisfied our goals for a sonification system. It provides a uniform network interface to sound devices through table–driven servers without hiding unique hardware functions. The device–dependent aspects of a sonification can be encapsulated in a single file. A user–interface to configure sound devices and sonifications can be generated independently of the underlying hardware. Porsonify was easily integrated into the Pablo performance analysis environment, enabling experimentation with synchronized sonification and visualization. This kind of portable experimental environment is critical to further work in sonification.

Appendix A

Recording

This appendix describes the recording that accompanies this thesis. The recording contains samples of the sonifications described in Chapter 5. All of the sonifications were generated by a MIDI–based Yamaha TG33 Tone Generator.

A.1 Multivariate Data

The following set of sonifications present the data from the Places Rated Almanac [4]. A note is played for each data sample (each city), where pitch corresponds to the recreation index, duration is controlled by population, longitude determines the stereo balance, and the climate index is used to select the timbre. More specifically, the recreation index is scaled from the range 300–4800 to a value in the range 0–35 and mapped to a note in a C major scale. Population is scaled from 62820–8274961 to a duration between 0.01 and 5 seconds. Longitude is scaled from -127.202– -68.772 to a stereo balance between 0 (far left) and 4 (far right). The climate index is scaled from 105–910 to a value between 0 and 2. This scaled value, in turn, was used to select a synthesized instrument. A zero corresponds to the "rude" instrument, a raspy digital sound, 1 corresponds to a flute, and 2 corresponds to the "bells and strings" instrument. Finally, the housing cost index is scaled from 5159–23640 to a volume setting between 50 and 99.[1]

The data samples are grouped into states, and samples from Illinois, California, Florida, New York, New Jersey, and Alabama are played in sets, in that order. It is immediately clear

[1] The synthesizer supports volume settings from 0–99, but notes too soft are difficult to hear.

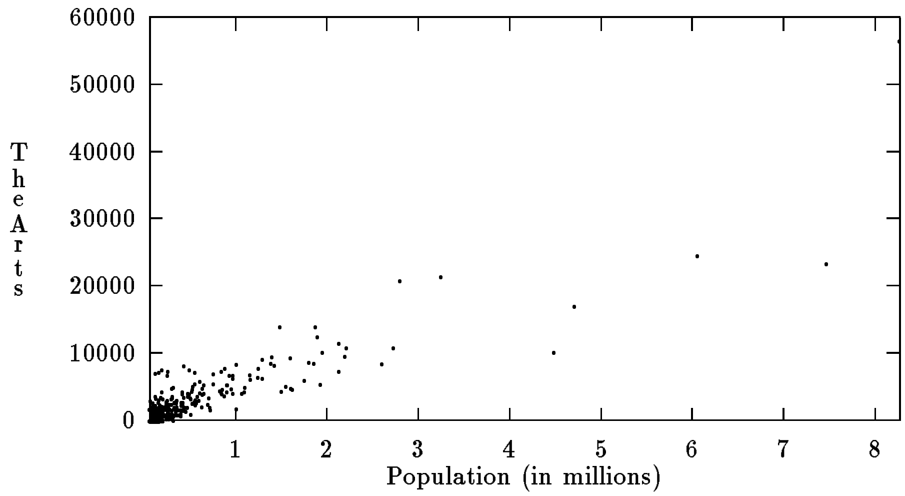

Figure A.1: Population and the Arts.

that the stereo balance indicates the location of the state. New York and New Jersey are, overall, "louder" than, for example, Alabama, indicating a higher average housing cost. New York City is the most highly populated city in the United States, represented by the last note in the New York set. Listen for the long, loud, high pitch; it is expensive to live there but it has a high recreation rating. California clearly has the best weather of these sample states, judging by the timbre of the notes. Compare it with weather in Florida.

A.1.1 Sonic Scatterplots

The next two sonifications are examples of a two–dimensional, sonic scatter plot; see Figure A.1 for the graphical scatterplot corresponding to this sonification. In the first, population is the time axis, and the samples for all cities are sorted by that variable. Each population is then scaled to a timestamp in the overall duration of the sonification, selected arbitrarily to be 1.5 minutes. A pitch is played at each timestamp to correspond to the arts index for that city. As in the previous sonifications, the pitch is mapped to a note in a five octave C major scale.

The arts index is highly correlated with population. Listen for the generally increasing pitches. Also notice that most of the populations are clustered at the beginning.

In the second sonification, the climate index is used as the time axis, and pitch is used to represent the recreation index. Although obvious, clustering is not as severe, so the duration

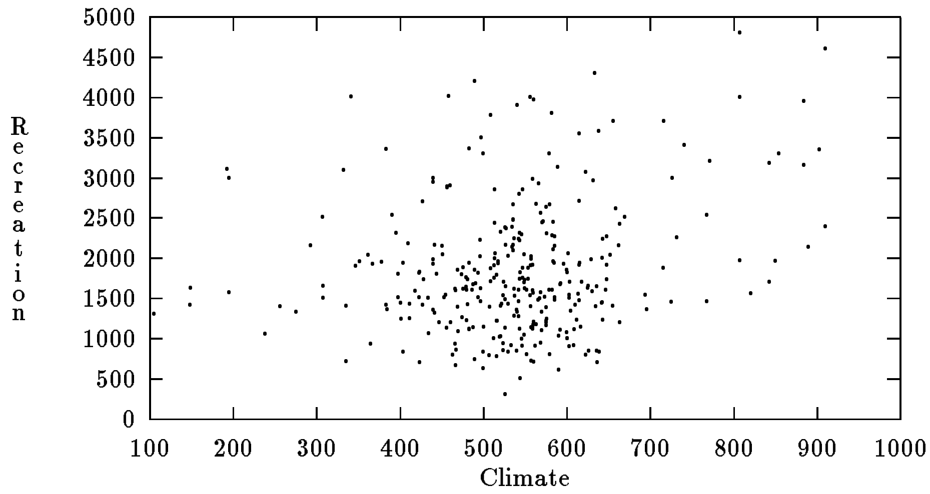

Figure A.2: Climate and Recreation.

of this sonification is only one minute. There is little correlation between these variables, in contrast to the previous sonification.

Figure A.2 shows the scatter plot corresponding to this sonification.

A.1.2 Stereo Sonic Scatter plots

The next sonification is the result of "overlapping" two sonic scatter plots. Pitches in the left speaker correspond to one variable and those in the right correspond to another. Both sequences of notes are played synchronously along the same time axis, generated as above by scaling a third variable.

In this sonification, the health care score is mapped to a pitch in the right, the education rating is mapped to a pitch in the left, and the arts index is the time axis. The sonification runs for one minute. There is a strong linear correlation between scores for the arts and health care, and between education and health care. Figure A.3 shows the corresponding graph.

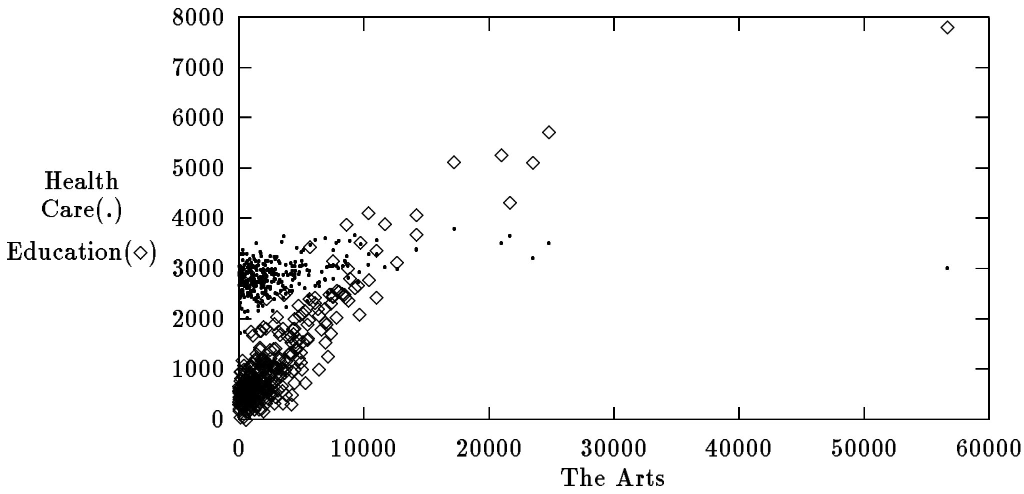

Figure A.3: The Arts, Health Care, and Education.

A.2 Parallel Program Behavior

The following sonifications use sound to illustrate aspects of the behavior of various parallel programs. These were all generated in concert with standard visualizations in the Pablo performance analysis environment [18].

A.2.1 Sixty–Four Processors

The next two sonifications are designed to illustrate communication patterns in parallel applications. The data comes from a 64 node Intel iPSC/860 hypercube, and was generated with the Portable Instrumented Communication Library (PICL) [11]. In each sonification, a note is played for each message send and receive. The sending processor identifier (0–63) is mapped to a pitch from (30–93). Notes corresponding to sends are played in the left of the stereo balance field, and receives are played in the right. The duration of all notes is 0.05 seconds, a duration intentionally chosen to be small so as not to "slow down" the corresponding visualization. Pauses between notes are caused by the time required to process instrumentation trace records that are not sonified and to update other visual displays. Although it is possible to use the duration of each note to convey more information about the messages sent and received (e.g., the message size), this could be confusing when combined with uneven pauses between notes due to other computation and displays.

The first of the PICL sonifications is of part of an execution trace of a partial differential equations solver. This code is characterized by heavy communication. It should be evident that the communication pattern stabilizes to a state where a few sends are followed by a few receives. The pitches are scattered over a wide range, indicating that no single group of processors is responsible for sending all the messages.

The second sonification is of an execution trace of a matrix transpose application. It should be audible that a different kind of communication pattern is used here; first all processors send, then receive. With the exception of a few outliers, all 64 pitches are first heard in the left, then in the right as the messages are received.

A.2.2 Eight Processors

This sonification of part of the execution of simplex linear optimization code running on an 8 node Intel iPSC/2 hypercube [23] demonstrates another way of revealing communication patterns. As in the previous sonifications, the identifier of the sending processor is mapped to pitch. However, the smaller number of processors permits notes started at each send to be sustained until their corresponding receives. This approach would not work with 64 processors due to synthesizer hardware limitations (the Yamaha TG33 Tone Generator can play at most 32 notes at a time) and the fact that 64 notes sustained at once would be overwhelming. The smaller number of processors also permits mapping each identifier to a pitch in a major scale beginning at middle C, producing more musically recognizable patterns. In the previous sonifications, to avoid using extremely high or low pitches which could be hard to distinguish, a chromatic scale was used. The "Bells and Strings" instrument is used to play the notes due to its long, natural sustain.

In this sonification, a sequence of seven repeating notes indicates a message broadcast. These repeated notes always have the same pitch (C4), which is the lowest note; it corresponds to processor zero. It should be clear from listening for a while that there is a distinct, predictable pattern to the message sends.

A.3 Operating System Behavior

This sonification represents the processes running on the *Choices* operating system, on a two–processor Encore Multimax. There are thirteen processes; of these, ten are actually operating system daemons and services, one is the `IdleProcess`, which runs when nothing else is ready to run, and two are the WATER benchmark (from the SPLASH benchmark suite) and its child. Far left and far right in the stereo field are selected to distinguish activity on the two processors. A pitch is associated with each process; a note is played when a process begins to execute and stops when it is preempted by another. The timestamps from the execution trace are used to create this sonification in simulated real–time.

The `IdleProcess` and the WATER code are of greatest interest to us. To enable them to be easily identified, the other processes were mapped to notes in a chromatic scale beginning at middle C. The `IdleProcess` was assigned a pitch an octave lower than middle C, and the WATER application and its child process were assigned pitches an octave higher than the last operating system process.

When the application starts running, it dramatically affects the context switch patterns. The *Choices* scheduling algorithm permits a process to run as long as nothing else is waiting to execute. Long, sustained, high notes indicate that the WATER application code is executing, typically while the `IdleProcess` is on the other processor. After the application spawns a child, the two high–pitched notes representing these two processes can be heard together in the left and the right, as they execute in parallel.

Bibliography

[1] Aydt, R. A. SDDF: The Pablo Self-Describing Data Format. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Mar. 1992.

[2] Blattner, M. M., Sumikawa, D. A., and Greenberg, R. M. Earcons and icons: Their structure and common design principles. *Human–Computer Interaction 4* (1989), 11 – 44.

[3] Bly, S. *Sound and Computer Information Presentation.* PhD thesis, University of California, Davis, 1982.

[4] Boyer, R., and Savageau, D. *Places Rated Almanac.* Rand McNally, New York, 1985.

[5] Chernoff, H. The use of faces to represent points in k-dimensional space graphically. *Journal of the Ameerican Statistical Association 68* (1973), 361 – 368.

[6] Dodge, C., and Jerse, T. A. *Computer Music: Synthesis, Composition, and Performance.* Schirmer Books, New York, NY, 1985.

[7] Flury, B., and Riedwyl, H. Graphical representation of multivariate data by means of asymmetrical faces. *Journal of the Ameerican Statistical Association 76* (1981), 757 – 765.

[8] Francioni, J. M., Albright, L., and Jackson, J. A. The Sounds of Parallel Programs. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press.

[9] Frysinger, S. P. Applied research in auditory data representation. In *Proceedings of the SPIE, Conference 1259, Extracting Meaning from Complex Data: Processing, Display, Interaction* (1990), pp. 130 – 139.

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

[11] Geist, G. A. A Portable Instrumented Communication Library, C Reference Manual. Tech. Rep. ORNL/TM-11130, Oak Ridge National Laboratory, 1990.

[12] Jackson, J. A., and Francioni, J. M. Aural signatures of parallel programs. In *Proceedings of the 25th Hawaii International Conference on System Sciences* (1992).

[13] Keefe, R. Composing by musical analog: A look at planetary orbits. *Computer 24* (1991), 72 – 75.

[14] Lunney, D., and Morrison, R. C. Auditory presentation of experimental data. In *Proceedings of the SPIE, Conference 1259, Extracting Meaning from Complex Data: Processing, Display, Interaction* (1990), pp. 140 – 146.

[15] MIDI Manufacturers Association. *MIDI – Musical Instrument Digital Interface, Specification 1.0.* International MIDI Association, Los Angeles, CA, 1988.

[16] Moore, F. R. *Elements of Computer Music.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[17] Rabenhorst, D. A., Farrell, E. J., Jameson, D. H., Linton, T. D., and Mandelman, J. A. Complementary visualization and sonification of multi–dimensional data. In *Proceedings of the SPIE, Conference 1259, Extracting Meaning from Complex Data: Processing, Display, Interaction* (1990), pp. 147 – 153.

[18] Reed, D. A., Olson, R. D., Aydt, R. A., Madhyastha, T. M., Birkett, T., Jensen, D. W., Nazief, B. A. A., and Totty, B. K. Scalable Performance Environments for Parallel Systems. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press, pp. 562 – 569.

[19] Rudolph, D. C., and Reed, D. A. CRYSTAL: Operating System Instrumentation for the Intel iPSC/2. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989).

[20] Scaletti, C., and Craig, A. B. Using Sound to Extract Meaning from Complex Data. In *Proceedings of the SPIE, Conference 1459, Extracting Meaning from Complex Data: Processing, Display, Interaction II* (San Jose, CA, 1991).

[21] Singh, J. P., Weber, W. D., and Gupta, A. Splash: Stanford parallel applications for shared-memory. Tech. Rep. CSL-TR-91-469, Computer Systems Laboratory, DEECS, Stanford University, Apr. 1991.

[22] Smith, S., Bergeron, R. D., and Grinstein, G. G. Stereophonic and surface sound generation for exploratory data analysis. In *Proceedings of the CHI'90 Conference on Human Factors in Computer Systems* (Reading, MA, 1990), pp. 125 – 132.

[23] Stunkel, C. B., and Reed, D. A. Hypercube Implementation of the Simplex Algorithm. In *Proceedings of the Third Conference on Hypercube Computers and Concurrent Applications* (Pasadena, CA, Jan. 1988), Association for Computing Machinery, pp. 1473–1482.

[24] Sun Microsystems. *Sun System User's Guide.* Mountain View, CA, 1991.

[25] Wenzel, E. M., Wightman, F. L., and Foster, S. H. A virtual display system for conveying three–dimensional acoustic information. In *Proceedings of the Human Factors Society* (1988), pp. 86 – 90.