ADSL: An Auditory Domain Specification Language for Program Auralization

Dale S. Bock
Department of Electrical and Computer Engineering
Syracuse University
121 Link Hall
Syracuse, NY 13244
Email: dsbock@mailbox.syr.edu

Abstract

ADSL (Auditory Domain Specification Language) is a program auralization specification language which encourages users to actively diagnose software bugs as opposed to just monitoring predetermined known program events. Instead of focusing on individual program lines, users specify general domains of program information, called tracks, by using a customizable specification language. Based on the set of component tracks chosen, different sound domains are heard during program execution. The customized tracks can be refined or enlarged, forming an abstract continuum of auditory information. A preprocessor parses a user's source code, compares it with a specification file, and places the necessary audio routines. Tracks can be modulated by variables or other tracks along various sound dimensions, allowing for a significant increase in auditory information.

1 Introduction

This chapter describes a methodology that relies on sound to aid programmers in understanding how a program works, and more importantly, to determine if it is running correctly. The approach taken is very different from other program auralization aids [1, 2, 3, 4, 5] that require users to map audio cues to specific program lines. With this system there are no required explicit mappings between program constructs and their associated auditory signals. Instead, users define auditory cue sound domains and associate them with entire software functions. Program constructs and data, along with a high-level component specification file, determine how the sound mappings are assigned and modulated. Because the sound domains are defined beforehand, a user knows approximately what to expect and can diagnose potential software bugs by listening for unusual deviations in a sound dimension, or for the lack or inclusion of certain auditory cues during program execution. This approach encourages users to actively *diagnose* software and discover unknown errors, instead of simply monitoring known events.

The sound domain paradigm lets one use audio in a very natural way. Our dynamic sense of hearing appears to be well suited to detecting when something does not "sound right." The ear is capable of detecting when a single note in an entire symphony is played incorrectly. In contrast, our visual senses easily ignore misspelled, duplicate, or missing words in a document. Although most people are unaware of the exact sound of a properly tuned engine, they can often tell when there *may* be a problem that requires closer *visual* inspection. Sounds occurring outside of one's *expectations* for an engine (e.g., a rubbing sound) can lead to diagnostic insights. However, these

```
Track_Name=Loop
{
1 Track=Status('for'):Snd("for_sound");
2 Track=Status('while'):Snd("while_sound");
}
```

Figure 1: Example of a program auralization track used to monitor software loops.

same sounds may be considered within the "normal" range for a different domain context, such as a car's braking system.

2 Creating Sound Domains

Because software supplies no auditory diagnostic clues when it executes, it was necessary to develop an approach that allowed users to create customized sound domains for their software components. A specification language called ADSL (**A**uditory **D**omain **S**pecification **L**anguage) was developed for this purpose.

ADSL is a user-defined language. Its most basic elements are referred to as tracks. A track contains a list of program constructs and associated audio cues. Conditional run-time constraints, such as the value of a program variable (e.g., $i > 10$) can also be included in the track definition. By analogy, the different sounds produced from an automobile engine or braking system, can be viewed as types of tracks. We define a program construct as any line or fragment of program source code. A "Loop" track might consist of **for** and **while** statements. This is illustrated in Figure 1. Other tracks could monitor such things as function calls, input/output, graphical operations, arithmetic expressions, resource allocation, logical operations, and so forth. Higher level tracks may be defined to analyze sorting algorithms, object-oriented class invocations, searching procedures, program typo errors not caught by the compiler (e.g., i instead of $\&i$ in C), etc.

The sounds used consist of digitized recordings, synthesized speech, and MIDI (Musical Instrument Digital Interface) output to a synthesizer. Different tracks will tend to suggest the most appropriate sounds. Gaver's auditory icons [6] can be used to form an analogy between program events and everyday sounds. A "cranking" sound might be used to signal a loop. "Stamping" sounds could be associated with program assignment statements. Synthesized tones, such as Blattner's earcons [7], may be more appropriate in cases where a good sound analogy cannot be found or the high repetition of the event requires fast recognition (earcons are short, rhythmic sequences of pitches that can be combined to express complex auditory messages).

The sounds within a track domain may be grouped together to form a multilayered information structure. With this type of grouping, track events form a family of related occurrences, allowing a user to make comparisons among its members. Once a basic sound is established for a track, it can be varied across different audio dimensions to supply event data. For example, a "Loop" track may be denoted by a certain percussive instrument. The rhythmic pattern of this percussive sound could signify which family event occurred (on each encounter a **for** loop might produce two sounds and a **while** loop just one).

3 Application

Once auditory information tracks have been created, they are used to auralize software in terms of component sound domains. The sound domains are created by what are called *track sets*. A track

```
Fun1
{
1 Track=ID:Snd("Fun1");
2 Track=Math;
3 Track=Loop;
}
```

Figure 2: A component auralization specification. Math and Loop are predefined auralization track sound domains.

set consists of a group of user-defined tracks assigned to a software component. The particular combination of tracks in a track set determines the allowable range of sound cues generated when the auralized component is executing. In this context, we define a program auralization to be a mapping between software components and track sets.

Component track sets are defined in a specification file. The format is similar to the way program functions are declared. Instead of writing software functions, users define how the component is *permitted* to auditorially behave (the actual sounds produced depend on the particular track event activated). This is illustrated in Figure 2. Here, component **Fun1** is being analyzed in terms of a sound domain consisting of "Math" AND "Loop" tracks. The ID track (line 1) identifies the beginning of a component with an audio cue. It lets a user know which component they are hearing when a component auralization becomes active.

A preprocessor realizes a specification file by scanning over a copy of a user's source code. It looks for component events that are within the scope of a component track set's specified audio domain, and inserts the associated sound mappings when a match is found. The process is repeated for each auralized component and the modified code is then compiled for execution.

3.1 Refinement

Depending on how tracks are defined, the domain of a track may be too detailed for a given application, resulting in an overload of auditory information that can make fault isolation difficult. For instance, when using a "Loop" track (refer to Figure 1), one might want to auralize a specific **for** loop, or only those involving variable i. We introduce the concept of track refinement to deal with this issue. A refinement changes the domain of a track by imposing additional syntactic and run-time constraints. By analogy, an automobile's brake sounds can be considered to be a refinement of the broader sound domain encompassing the entire vehicle.

Figure 3 depicts several types of track refinement. In line 1, "Loop" track events are limited to **for** loops using variable i as an index. Any combination of Boolean operators (AND/OR/NOT) can be applied. A Limit (Lim) constraint is also supplied. In this case, only the first iteration of the loop will be auralized. Line 2 contains a conditional refinement. Before the "Graphics" track audio is heard, variable x must also be greater than five, OR variable y below two. Finally, line 3 demonstrates how a domain can be generalized. The inline sound file "math_sound," replaces all other "Math" track event audio. This will notify users when a "math" event has occurred, without specifying its type (we are raising the abstraction level of information).

3.2 Modulation

Suppose one hears a car horn while driving at night. That single horn sound can supply us with many types of information. At the most basic level we are simply aware of the event (attention getting). Using our terminology, this can be viewed as a type of track event. However, more

Acknowledgments

I would like to thank Gary Craig, Amrit Goel, Can Isik, and Dan Pease for their help and support. I am also indebted to my father, Gary Bock, and sister, Nancy Bock, for their editorial assistance. Many thanks to Gregory Kramer for giving me a forum and the opportunity to express my ideas.

References

[1] Brown, M. H. and J. Hershberger. "Color and Sound in Algorithm Animation." *IEEE Computer* **25(12)** (1992): 52–63.

[2] DiGiano, C., R. Baecker, and R. Owen. "LogoMedia: A Sound-Enhanced Programming Environment for Monitoring Program Behavior." In *Proceedings of Human Comp. Interaction—INTERCHI '93*, edited by S. Ashlund, 301–302. Held in Amsterdam, the Netherlands, April 24–29, 1993. Reading, MA: Addison-Wesley, 1993.

[3] Francioni, J. M., J. A. Jackson, and L. Albright. "The Sounds of Parallel Programs." In *Proceedings of the Sixth Distributed Memory Computing Conference*, edited by Q. Stout and M. Wolfe, 570–577. Portland, OR: IEEE, 1991.

[4] Jameson, D. H. "Sonnet: Audio-Enhanced Monitoring and Debugging." In *Auditory Display*, edited by G. Kramer, 253–265. Reading, MA: Addison-Wesley, 1994.

[5] Sonnenwald, D. H., B. Gopinath, G. O. Haberman, W. M. Keese III, and J. S. Myers. "InfoSound: An Audio Aid to Program Comprehension." In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, Vol. II, 541–546. Held in Honolulu, Hawaii, 1990. Washington, DC: IEEE 1990.

[6] Gaver, W. W. "Auditory Icons: Using Sound in Computer Interfaces." *Human Comp. Inter.* **2** (1986): 167–177.

[7] Blattner, M. M., D. A. Sumikawa, and R. M. Greenberg. "Earcons and Icons: Their Structure and Common Design Principles." *Human Comp. Inter.* **4(1)** (1989): 11–44.