Listening to Program Slices

Lewis I. Berman
University of Durham
Durham, DH1 3LE, UK
l.i.berman@dur.ac.uk

Keith B. Gallagher
University of Durham
Durham, DH1 3LE, UK
k.b.gallagher@dur.ac.uk

Abstract

Comprehending a computer program can be a daunting task. There is much to understand, including the interaction among different parts of the code. Program slicing is a vehicle that can help one understand this interaction. Because present-day visual development environments tend to become cluttered, and because a program slice may involve the interaction among a number of sections of code, the authors have explored the sonification of program slices. This investigation has produced an understanding of how to sonify slices in a manner appropriate for the software developer as program comprehender. Secondarily, the investigation has produced a better understanding of musical sonification techniques that are non-melodic and non-harmonic. This exploration is a proof-of-concept and pilot for further research.

Background

The dependence among different lines or sections of code is useful knowledge in program comprehension. If a critical variable is found to be incorrect at the execution of a given line of code, the maintainer would like to know which other code impacts that line.

Programmers can gain this understanding by running the program in a debugger under different sets of constraints, which can be time consuming. Program slicing is another tool that can aid such understanding.

Slices are computed with respect to a selected slice point, that is, a program point and variable(s) of interest. A static backward program slice is the subset of computation in the program upon which the selection is dependent [Weiser 1984].The slice at one program point may vary radically from the slice at a nearby program point. A given execution of the program up to the slice point is equivalent to one possible path through the slice.

To gain an impression of impact to a program point, it is desirable to explore whether a separate object or method is in a slice, the amount of code in each object or method that's in the slice, how far removed the statement, object, or method is from the slice point, fan-out from the slice point, and the homogeneity of slice versus non-slice code. At a detailed level, it is desirable to know exactly which variables and local statements impact those at the slice point.

A high percentage of modern programmers use integrated development environments (IDE's) such as Visual Studio and Eclipse. Typically, these environments are visually cluttered, containing numerous overlapping and paneled windows and controls. A notable disadvantage of the contemporary IDE is the necessity to locate and load multiple windows in order to understand relationships among different sections of code. Unless the display area of each window containing program code is severely restricted, the

interface becomes modal as the developer is forced to switch between views. Whilst the visual capacity of such IDE's is stressed, their auditory capacity is exploited only trivially if at all. The number of information channels available to the developer can be increased through use of sonification.

Sonification techniques have been applied to problems in software comprehension. Sounds representing run-time events in algorithm animation have been successfully used to find bugs [Baecker, 1997]. The LISTEN tool [Boardman 2001] provided an environment to hear run time behavior. Vickers and Alty sonified nested program constructs, such as loops, at run time via musical sonification, using tonal, triadic note patterns [Vickers 2003]. Finlayson has used a similar technique to provide a static "audioview" of Java source code [Finlayson, 2005].

It has been shown that non-speech, interactive sonification is useful in gaining an overview of data in an exploration mode, both for sighted and non-sighted users. Recently, Kildal sonified data tables [Kildal 2005]. Each table cell's data maps to a pitch realized in a fixed, piano-like timbre. A row or column can be played quickly enough that it sounds like a distinct pitch cluster.

Sonifying Program Slices

Program slices have been visualized [Gallagher VIA], but they can also be sonified. In performing program comprehension, much time and effort is spent textually reading code. The IDE's visual field for this activity primarily contains a tree-structured explorer

providing navigation to objects and methods, along with an active window showing the code for the selected object or method. To understand the impact of one object or method to another, one would typically navigate between them. This paradigm promotes linear detail at the expense of overview and exploration. The ability to hear characteristics of selected or hovered-over objects and methods provides an added data dimension without cluttering the screen, reducing effective area for existing display, or requiring modal changes and bookmarking in back-and-forth visual navigation.

Three program sonification techniques were developed on a stand-alone basis, with the intent of integrating them into an IDE. Each technique sonifies a slice at a different observational level: (1) hearing actual slice versus non-slice lines of code within a method, (2) hearing a quick impression of a method with respect to the slice, and (3) hearing an impression of the amount of the object with respect to the slice. #3 could be heard simultaneously, if desired, with #1 or #2.

All three slice sonification techniques are intended to be employed by a developer while examining source code using the IDE. An explorer showing the program's objects (or source files) and methods is visible, as is an editor containing a particular method under examination. In the course of examining the code, the developer would select variable(s) in a source statement as the slice point, then select or mouse over a second object, method, or source line to hear its relationship to the slice point. In a step toward this goa, slices are obtained at present using the CodeSurfer standalone slicing tool [Grammatech], and the three types of sonifications are derived from each slice so obtained. The sound is

realized using Csound, a sophisticated software sound generator and processor [Csound]. A consistent sound universe is employed: the timbral space consists of actual and synthesized plucked instruments, and the tonal space consists of consecutive diatonic or chromatic pitches. The mappings are neither triadic nor musical phrases; instead, they are simply mappings to pitches in defined ranges, allowing the listener's focus to be directed to timbre and range whilst reducing the risk of interference by extra-musical tonal and melodic associations.

An example program, the open-source ACCT, has been sonified and placed online [online demo]. ACCT has been chosen because of its low number of slices that are equivalent to one another. [Binkley ??].

The first technique is intended to allow source statements to be heard as the developer passes the mouse over them, possibly quite quickly in succession, as in the table sonification referenced above. Each statement is heard as a single note produced by a plucked instrument. Statements within the slice are heard within a bounded pitch range, and statements outside the slice are heard in a lower range, as shown in Figure 1a. To help differentiate consecutive statements, the sequence of pitches rises and falls within the range. Higher range pitches, those in the slice, are sustained to leave the aural impression, when scanning statements quickly, that statements were indeed in the slice. Stereo separation helps differentiate pitches in each range. The number of consecutive pitches in the same range, along with the succession of segments within each range, indicates homogeneity. The beginning of ACCT's main method maps as shown in Figure

1b. There are three statements in the slice, followed by one out of the slice, followed similarly by 6, 1, 1, 3, and 2.

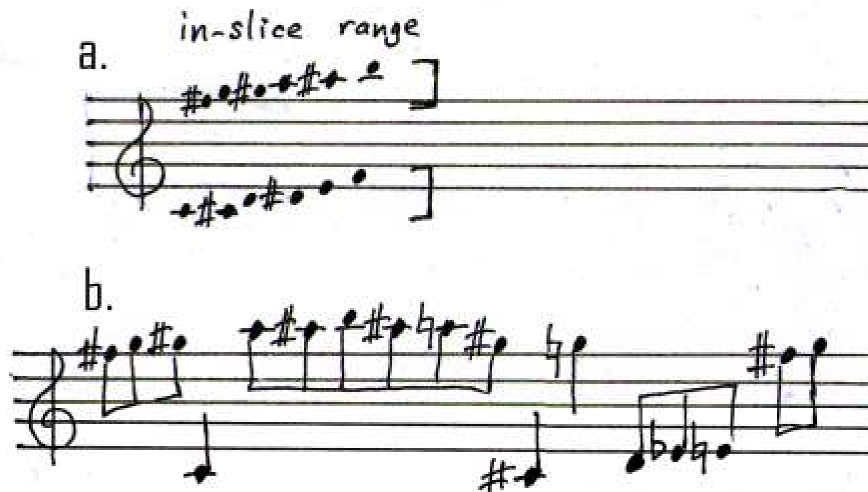

Figure 1. Beginning of ACCT's main method.

The second technique depicts methods rather than individual statements. Its objective is to leave an impression of each method's size and how much of it is in the slice. One hears an event as one mouses over each method in the explorer. Again, this may be done quickly or slowly. Each event consists of zero or more higher-pitched notes followed by zero or more lower-pitched notes, all in very rapid succession. Each note represents up to ten source statements. A method that has five statements within the slice and twenty-five statements outside the slice will result in one higher-pitched pluck followed by a cluster of three lower-pitched plucks. Table 1 shows some methods in AC's file ac.c. Figure 2 shows the corresponding realization in sound.

Method	Statements in slice	Statements not in slice
strtol	1	1
atoll	1	1
main	47	51
give_usage	0	1
update_system_time	7	1
log_everyone_out	13	11

Table 1. Some methods in the file ac.c.

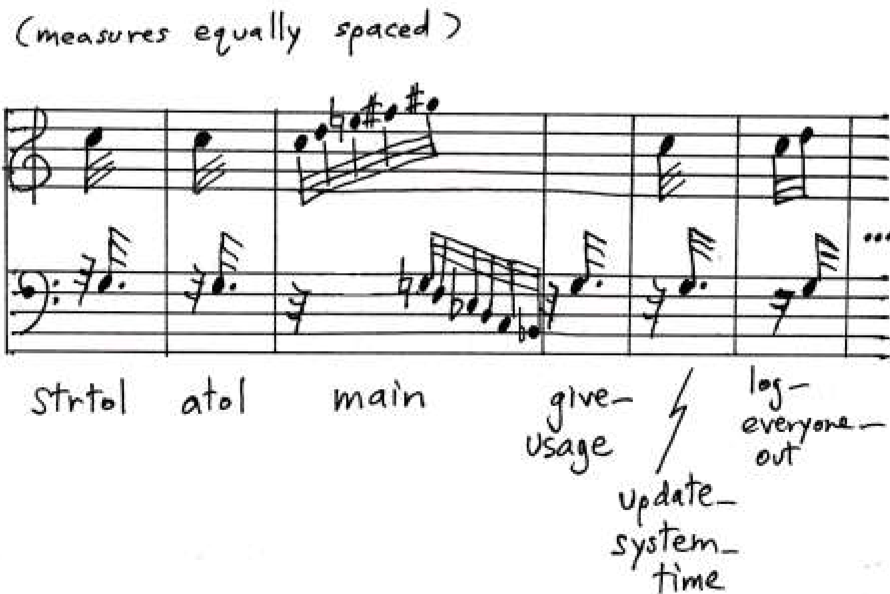

Figure 2. Realization of methods in ac.c.

The third technique operates at the highest level, comparing different objects. Its goals are to differentiate the objects, leave an impression of the size of the object, and leave an impression of the percent of the object's code within the slice. This technique makes use of sound clouds generated through granular synthesis [Dodge]. The object's size corresponds to the overall pitch range of the cloud, and the percentage of the object's code within the slice corresponds to the cloud's density.

The third technique differs radically from the others so that it can be heard as background along with the others. It is intended to change as one progresses between objects while hovering over the methods in an explorer. Thus, the objects are differentiated in time through unique signatures even if the listener is not actively listening to the clouds.

As a preliminary, informal evaluation, a small group of sighted, software-aware listeners were able to "hear" and describe the characteristics of a slice. Most of the listeners were not highly trained musically. Techniques #1 and #3 were found to be intuitive after a few sentences of explanation. Technique #2 required greater explanation. The one-to-ten mapping of source statements to discrete notes was somewhat troublesome, requiring some training.

Discussion and Future Direction

The preliminary evaluation suggests that slice sonification merits further investigation. The chosen techniques appear to be effective. Questions abound concerning utilization of slice sonification in the IDE, wider use of sonification in program comprehension, and the sonification techniques themselves.

The interactive nature of slice sonification in the IDE has yet to be explored. The next step is to integrate the CSound sonification mechanism with Eclipse, along with an embedded slicing component, and evaluate actual usage scenarios. The ability to hear and rapidly compare multiple slices is of particular interest. Differences in slice profiles of several programs should be detectable to the user. A typical calculator program, for instance, has a low number of large, equivalent slices, differing from the ACCT program mentioned above. This should be readily hearable. One dimension that can be added to the existing sonification is the distance within the slice, i.e. number of nodes of the graph, to each object or method. Audio distance is a possible mapping.

More generally, slice sonification is seen as part of an effort to offload information from the IDE to the audio realm and evaluate its effectiveness. The information that can be heard during active debugging is one candidate for exploration.

The chosen timbral and non-harmonic techniques differ from previous software sonification techniques exploiting harmony, melody, and to a lesser extent, rhythm. The question of which techniques are more appropriate for which comprehension tasks therefore arises. A combination may be appropriate, raising the question of the number of simultaneous audio that the user can process, especially given foreground versus background events. Conversely, it may be possible to maximize information flow via the chosen combination of techniques. Another area of evaluation is clarity. For example,

changing the attack times between near-simultaneous notes may help the user to discriminate them.

Summary

The authors have explored the sonification of program slices, with encouraging early results. Three techniques were developed, two involving musical but non-thematic, non-triadic realizations, and the third involving granular synthesis. Future steps include refinement of the sonifications, integration with an IDE, and empirical study of both the interactive and sonic natures of the realizations.

References

Weiser 1984	Weiser, M. Program slicing. IEEE Transactions on Software Engineering, 10(4):332-357, 1984.
Gallagher ?	??
Vickers 2003	Vickers, P. and Alty, J. Siren songs and swan songs: debugging with music. Communications of the ACM, 46(7):87-92, July, 2003.
Csound	http://www.csounds.com
Online demo	TBD
Binkley ??	Binkley, D. and Harman, M. Locating dependence clusters and dependence pollution. TBD, TBD.
Dodge	Dodge, C. and Jerse, T. Computer Music: Synthesis, Composition, and Performance, second edition. Schirmer, 1997, pp. 262-271.
Boardman 2001	Boardman, D., Khandelwal, V., Greene, G., and Mathur, A. LISTEN: a tool to investigate the use of sound for the analysis of program behavior.
Kildal 2005	Kildal, J. and Brewster, S. Explore the matrix: browsing numerical data tables using sound. Proceedings of ICAD, July, 2005.
Gallagher VIA	Gallagher, K. Visual impact analysis. ?? ??

Baecker 1997	Baecker, R., DiGiano, C., and Marcus, A. Software vi for debugging. Communications of the ACM, 40(4):44-54, April, 1997.
Finlayson 2005	Finlayson, J. and Mellish, C. The ‘audioview’ – providing a glance at java source code.
Grammatech	Grammatech, Inc., http://www.grammatech.com