LISTENING TO PROGRAM SLICES

Lewis I. Berman, Keith B. Gallagher

Department of Computer Science
Durham University
Durham DH1 3LE, UK
l.i.berman@durham.ac.uk
k.b.gallagher@durham.ac.uk

ABSTRACT

Comprehending a computer program can be a daunting task. There is much to understand, including the interaction among different portions of the code. Program slicing can help one to understand this interaction. Because present-day visual development environments tend to become cluttered, the authors have explored sonification of program slices in an attempt to determine if it is practical to offload some of the visual information. Three slice sonification techniques were developed, resulting in an understanding of how to sonify slices in a manner appropriate for the software developer undertaking program comprehension activities. The investigation has also produced a better understanding of sonification techniques that are musical yet non-melodic and non-harmonic. These techniques were demonstrated to a small set of developers, each reporting that the techniques are promising and useful.

1. INTRODUCTION

In program comprehension, much time and effort is spent reading code, often using an interactive development environment (IDE) such as Visual Studio [1] or Eclipse [2]. The IDE's visual field for this activity 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 have to 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 bookmarking in back-and-forth visual navigation.

The dependence among different lines or sections of code is useful knowledge in program comprehension. If, for example, a value 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. Developers can gain such an understanding by running the program in a debugger under different sets of constraints, which can be time consuming. Program slicing [3] is another tool that can aid such understanding. Program slices can be studied in a freely exploratory mode as opposed to being constrained by program flow in a debugger.

Accordingly, three techniques for hearing program slices were developed. These techniques were presented to a small group of developers, some with extensive musical training and some without. The developers, without exception, reported that these techniques would be useful in their own interactive program comprehension activities.

2. BACKGROUND

Program slices [3] are computed with respect to a selected *slicing criterion*, that is, a program point and its variable(s) of interest. A slice, more specifically a static, backward program slice, is the set of program statements that may contribute to the value of the variable(s) at the criterion. Both data and control dependences are captured. 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. Comprehensive surveys of program slicing can be found in [4], [5], and [6].

Figure 1 depicts two slices through the same method. Both slice criteria call out the same statement, but the criterion on the left is on two variables while that on the right is only only one of them. The right slice is a subset of the left slice.

Figure 1. *Two slices based on the same statement.*

Other comprehension techniques have been the benefactors of sonification. Sounds representing run-time events in algorithm animation have been successfully used to locate bugs [7]. The LISTEN tool [8] provides an environment to hear run time behavior. Vickers and Alty sonify nested program constructs, such as loops, at run time via musical sonification, using tonal, triadic note patterns [9]. Finlayson uses a similar technique to provide a static "audioview" of Java source code [10]. The present study's sonifications differ from those efforts in that its techniques are musical but non-harmonic and non-melodic, while their techniques are musical, triadic, and melodic.

Non-speech, interactive sonification is useful in gaining an overview of data in an exploratory mode, both for sighted and

non-sighted users, as in Kildal's sonification of data tables [11]. 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 pitch cluster with a distinct signature. As statements in an editor and objects within an object explorer have list-like characteristics, the sonification of these objects in the manner similar to that of Kildal would appear to be appropriate.

To gain an impression of impact at a program point, it is desirable to explore whether a separate statement, method, or object is in its slice, the amount of code in each object or method that's within 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. Distance from the slice point is measured by the number of edges in the dependence graph from the slice point to the contributing code. Fan-out means there are many contributing sections in different branches of the dependency graph. For example, many independent branches of code might call a common utility function such as a string formatter. Thus, a slice on a variable in the string formatter would fan out to its calling procedures. Homogeneity refers to the mixture of code in a method that's within the slice versus outside it. A highly homogeneous method is characterized by large chunks of code within the slice alternating with similar chunks of code outside the slice.

3. SONIFYING PROGRAM SLICES

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 slice versus non-slice lines of code within a method, (2) hearing an impression of one or more methods to determine how much code is within versus outside the slice, and (3) similarly hearing an impression of the amount of the object with respect to the slice. Technique 3 could be employed simultaneously, if desired, with techniques 1 or 2.

All three slice sonification techniques are intended to be used by a developer while examining source code using the IDE. An explorer showing the program's objects (or source files) and methods would be visible, as would an editor containing a particular method under examination. In the course of examining the code, the developer would first select variable(s) in a source statement as the slice criterion, then select or mouse over a second object, method, or source line to hear its relationship to the slice point. The developer could scan the items with the mouse quickly or slowly, resulting in a quick or slow succession of sonic events.

Slices are obtained at present using CodeSurfer [12], a standalone slicing tool, and the three types of sonifications are derived from each computed slice. The sound is realized using Csound [13], a sophisticated software sound generator and processor. 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 contrasting timbres and ranges while reducing the risk of interference by extra-musical tonal and melodic associations.

A slice of an example program, the open-source ACCT, has been sonified and placed online [14]. The online files are also included as an attachment. ACCT has been chosen because it has a low number of slices that are equivalent to one another [15]. A slice through the program was obtained. The slice extends through multiple files; the file *ac.c* as displayed by CodeSurfer is attached [IMAGE ac_main.pdf]. The slice point (on page 6 of 11), underlined, is the following line in the method *update_system_time*:

```
Struct login_data *l = hashtab get value (he);
```

The statement was chosen to provide a variable that is referenced previously in the same method and multiple times elsewhere in the program.

3.1. The First Technique

The first technique is intended to allow source statements within a method to be heard as the developer passes the mouse over them, possibly quite quickly in succession, as in the Kildal table sonification. Each statement is heard as a single note produced by a plucked instrument. The lexical order of the statements maps to the succession of notes. Figure 2, on the following page, depicts the first few notes of the slice obtained using the slicing criterion described above, in the method *main* as listed in [IMAGE ac_main.pdf]. (A statement within the slice is one containing code printed in red. The *static struct* is considered to be one statement.) Statements within the slice are notes heard within a bounded pitch range, and statements outside the slice are heard in a lower range, as shown in Figure 2a. The pitch ranges are merely bounds; relative pitch within each range has no meaning except to differentiate consecutive statements via the pitches rising and falling within the range. Higher-range pitches, those in the slice, are sustained to leave the aural impression, when scanning statements quickly, that there were indeed statements within 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 2b. The first three notes represent the first three statements in the method, all within the slice. The fourth note, middle C, represents the fourth statement in the method, which is outside the slice. Similarly, the following sequences of six, one, one, three, and two notes represent six notes within the slice, one note outside the slice, one within, three within, and two outside. We hear that the beginning of this method displays low homogeneity with respect to this particular slice, and we hear that a majority of that code is within the slice. The full realization of ACCT's main method is captured in the audio file [SOUND sonif1.wav].

The listener can gain an impression of different dimensions of the slice. The number of statements in the method maps to the number of notes and therefore the excerpt length. Percentage of statements within the slice maps to the ratio of higher-pitched notes to the total number of notes. Homogeneity is heard through the clustering of high or low-pitched sounds.

Figure 2. *The first seventeen statements of ACCT's main method.*

The listener can gain an impression of different dimensions of the slice. The number of statements in the method maps to the number of notes and therefore the excerpt length. Percentage of statements within the slice maps to the ratio of higher-pitched notes to the total number of notes. Homogeneity is heard through the clustering of high or low-pitched sounds.

3.2. The Second Technique

The second technique depicts methods, rather than individual statements, as sonic events. Its objective is to leave an impression of each method's size and how much of it is in the slice. The intent is to hear an event as one passes the pointer over each method in the explorer. Again, the slice may be heard quickly or slowly, controlled by the listener's mouse rate over the methods. Each event consists of zero or more higher-pitched notes representing code within the slice, followed by zero or more lower-pitched notes representing code outside the slice, all in very rapid succession to retain the impression of a single event. Each note in the event 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.

A method having no statements in the slice and one to ten statements outside the slice is represented by the single sound heard in [SOUND sounf2_1out.wav]. A method having one to ten statements within the slice and one to ten statements outside the slice is represented by the high sound followed by the low sound in [SOUND sonif2_1in_1out.wav]. If more than ten statements are within or outside the slice, more notes are heard. For each method, an impression of the amount of the code within and outside the slice, respectively, is heard through the perceived density of the higher and lower parts of each event.

Table 1 shows some methods in ACCT's file *ac.c*. Figure 3 shows the realization. Each measure represents a method.

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.

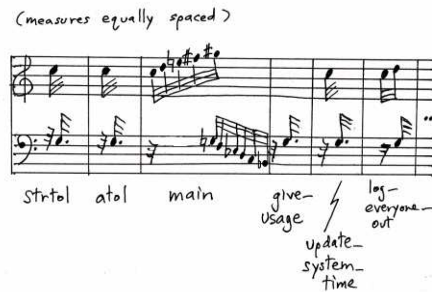

Figure 3. *Realization of methods in the program ac.c.*

The entire program ac.c is realized as [SOUND sonif2.wav], in the order listed in [IMAGE ac_code_data.xls]. Each of the first two events represents a method having one statement within the slice and one statement outside the slice. The main method of *ac.c*, having 51 statements within the slice and 47 statements outside the slice, is heard third as an event of greater density. All thirteen of the methods in *ac.c*, starting with those shown in Table 1, are realized, followed after a pause by each of the remaining methods in the other four major source files.

3.3. The Third Technique

The third technique operates at the highest level, allowing the listener to compare 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 [16]. Each sound cloud, corresponding to one object (i.e., one file in this case), consists of numerous sound grains randomly distributed according to parameters that are fixed over the cloud's duration. 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 listener hears a distinct cloud continuously during a visit to its corresponding object in the explorer, which could be lengthy.

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 object boundaries 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.

3.4. Evaluation

As a preliminary, informal evaluation, a group of four sighted, software-literate listeners were able to "hear" and describe the characteristics of a slice. Two listeners were highly trained musically and two were not. One of the musically trained listeners was a programmer; the other wasn't. The less highly-trained listeners were both programmers.

Techniques 1 and 3 were found to be intuitive by all listeners after a few sentences of introductory explanation. Technique 2

required greater explanation. The one-to-ten mapping of source statements to discrete notes was somewhat troublesome to one of those with less musical background, requiring some training. The programmer with high musical training felt that Technique 1 could communicate more information through use of variable pitch ranges. That listener and the musically-trained non-programmer had initial expectations that relative pitch within the range be meaningful. All, including the non-programmer, were able to describe characteristics of the slice from its musical realization, especially using Technique 1: how much of the method was in the slice versus its complement, the homogeneity of the slice, and the mapping to individual lines of code. Several of the listeners pointed out the advantage of Technique 3 as a continuously-heard sound signature that could be analyzed over any necessary time span, as opposed to the one-time nature of the events of the other two techniques. All participants reported that the techniques appear promising.

4. DISCUSSION AND FUTURE DIRECTION

The preliminary evaluation suggests that slice sonification merits further investigation. The chosen techniques appear to be effective, as they were understood by each of the interviewed developers. Different preferences were voiced regarding how much information should be represented using each technique. The amount may have a correspondence to the listener's level of musical sophistication.

Questions of interest for further exploration concern utilization of slice sonification in the IDE, wider use of sonification in program comprehension, and the 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 usage scenarios. Ability to hear and rapidly compare multiple slices is of particular interest. Differences in slice profiles of several programs should be detectable by the user. A typical calculator program, for instance, has a low number of large, equivalent slices, differing from ACCT, which has a large number of non-equivalent slices. This difference should be readily hearable. One dimension that can be added to the existing sonification is the distance within the slice, i.e. number of edges of the graph, to each object or method. An audio impression of physical distance is a possible mapping.

The experiment with slice sonification points to larger questions: which information can be offloaded from the visual IDE to the audio realm? How much of that information can be perceived at one time? How effective is the sound domain compared with the visual domain, especially given its temporal nature? Finally, does the sonification of particular data affect the succession of tasks in the program comprehension process?

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 amount of simultaneous audio information 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.

5. CONCLUSION

The authors have explored the sonification of program slices, with encouraging early results. Three promising techniques were developed using musical but non-thematic, non-triadic realizations, the third involving granular synthesis. Further research will refine the techniques, addressing each technique's communication capacity and its ability to provide a relatively quick overview. Future steps include refinement of the sonifications, integration with an IDE, and empirical study of both their interactivity and sonic effectiveness.

6. REFERENCES

[1] Microsoft Corp., http://msdn.microsoft.com/vstudio/

[2] Eclipse Foundation, http://www.eclipse.org/

[3] M. Weiser, "Program slicing," in *IEEE Transactions on Software Engineering*, Vol. 10 No. 4, 1984, pp.332-357.

[4] Delucia, A. "Program Slicing: Methods and Applications," *First Int'l Workshop on Source Code and Manipulation*, 2001, pp. 142-149.

[5] D. Binkley and K. Gallagher, "A Survey of Program Slicing," in *Advances in Computers*, M. Zelkowitz, ed., Academic Press, 1996.

[6] F. Tip, "A Survey of Programming Slicing Techniques," in *Journal of Programming Languages*, vol. 13, no. 3, 1995, pp. 121-189.

[7] R. Baecker, C. DiGiano, and A. Marcus. *Software vi for Debugging*. Communications of the ACM, Vol. 40 No. 4, April, 1997, pp. 44-54.

[8] D. Boardman, V. Khandelwal, G. Greene, and A. Mathur, "LISTEN: a Tool to Investigate the Use of Sound for the Analysis of Program Behavior," in Proceedings of the 19th Annual International Computer Software and Applications Conference (COMPSAC '95), Dallas, TX, August 1995, IEEE Press, pp. 184-193.

[9] P. Vickers and J. Alty, "Siren Songs and Swan Songs: Debugging with Music," in *Communiations of the ACM*, Vol. 46, No. 7, July 2003, pp. 87-92.

[10] J. Finlayson and C. Mellish, "The 'Audioview' – Providing a Glance at Java Source Code," in *International Conference on Auditory Display (ICAD)*, Limerick, Ireland, July 2005, pp. 127-133.

[11] J. Kildal and S. Brewster. "Explore the Matrix: Browsing Numerical Data Tables using Sound," in *International Conference on Auditory Display (ICAD)*, Limerick, Ireland, July 2005.

[12] Grammatech, Inc., http://www.grammatech.com/.

[13] http://www.csounds.com/.

[14] http://mysite.verizon.net/vze8b1hf/.

[15] D. Binkley and M. Harman. "Locating Dependence Clusters and Dependence Pollution," Int'l Conference on Software Maintenance (ICSM), Budapest, Sept. 2005.

[16] C. Dodge and T. Jerse, *Computer Music, Second Edition*, Schirmer, 1997, pp. 262-271.