[By Paul Vickers and James L. Alty]

Siren Songs and Swan Songs

Debugging with Music

Program execution behavior can be mapped to a structured musical framework that helps locate and diagnose software errors.

In Greek mythology the sirens' beautiful songs lured sailors to their doom. Today, the CAITLIN musical program auralization system [12] renders the runtime behavior of Pascal programs into a structured musical framework allowing programmers to hear when their programs deviate from the expected path. Now, instead of the bugs luring programmers to their doom, with CAITLIN the bugs sing their swan songs to signal their own impending demise.

Our world is full of sound. Indeed, sound is so pervasive and integrated into our visual world that those of us fortunate to have both the visual and the auditory senses usually underestimate the importance of the auditory channel. It rates highly in our order of sensory priorities, probably because of its unique warning capability; auditory interrupts are difficult to ignore. Our brains process information heard very differently from information seen; moreover, audition has properties that complement vision. While vision provides excellent spatial perception, the auditory sense offers a number of temporal advantages. A single sound communicates many properties simultaneously, and humans can understand and separate out a series of simultaneous auditory sequences (music is an obvious example). Filmmakers use auditory sequences to carry the audience through major visual scene shifts.

Illustration by Jean-François Podevin

(a) Original Program

```
PROGRAM Demo;

VAR
   cntr1,
   cntr2          : Integer ;

BEGIN { Demo }
   cntr2 := 1 ;
   FOR cntr1 := 1 TO 7 DO
      WHILE cntr2 <= 4 DO
         BEGIN
         Writeln (cntr1, ':' cntr2) ;
         cntr2 := cntr2 + 1 ;
         END ;
END { Demo }.
```

(b) Auralized Program

```
PROGRAM Demo;

USES
   Miditoo1,
   Mididecs,
   Mmessage,
   M_voices,
   CGadgets,
   Crt ;

VAR
   cntr1,
   cntr2          : Integer ;
   pitch_1,
   pitch_2        : Integer ;
   pitch_cntr_1,
   pitch_cntr_2   : Integer ;
   effect_1,
   effect_2       : Integer ;
   flag_1,
   flag_2         : Boolean ;

BEGIN { Demo }
   IF ResetDSP = 0 THEN
      midiSendGSMessage (GSReset) ;
   SetOptions ;
   Delay (750) ;
   ProgramChange (channel_16, Organ_1) ;
   cntr2 := 1 ;
   SoundMetronome ;
   pitch_1 := caitlin_C + 24 ;
   pitch_cntr_1 := 0 ;
   FOR_signature (1, True) ;
   FOR cntr1 := 1 TO 7 DO
      BEGIN
      IF cntr1 = 7 THEN ;
         SoundNote (channel_10, sleigh_bell,
                    FOR_Data.velocity, 1) ;
      NoteOn (channel_12, pitch_1, FOR_Data.velocity) ;
      pitch_2 := caitlin_C + 24 ;
      WHILE_Signature (2, True) ;
      WHILE cntr2 <= 4 DO
         BEGIN
         WHILE_POI2 (2, True) ;
         Writeln (cntr1, ':' cntr2) ;
         SoundMetronome ;
         cntr2 := cntr2 + 1 ;
         SoundMetronome ;
         END ;
      WHILE_POI2 (2, False) ;
      WHILE_Signature (2, False) ;
      Wait (durationtable [FOR_Data.duration] DIV 2) ;
      NoteOff (channel_12, pitch_1, FOR_Data.velocity) ;
      Inc (pitch_1, GetInterval (FOR_Data.scale,
           pitch_cntr_1, True)) ;
      Inc (pitch_cntr_1) ;
      END ;
   Wait (durationtable [FOR_Data.duration]) ;
   ChangeVolume (FOR_Data.channel, FOR_Data.velocity) ;
   FOR_Signature_end (1, True) ;
END { Demo }.
```

Figure 1. Original source and auralized source: (a) list of a Pascal program and (b) the source as amended by CAITLIN. In normal operation, the user does not see the auralized source. In the same way intermediate object code is generated by compilers as input to the linker prior to generation of an executable image, the auralized code is an intermediary between CAITLIN and the language compiler.

There are disadvantages, too. By their nature, auditory interruptions are intrusive, and precise quantitative information is difficult to represent sonically. However, the capacity of our auditory memory is great and can be very precise. After only a few hearings, an experienced listener can memorize a whole symphony (typically containing more than 20,000 notes), recognize subtle differences between performances, and certainly tell if a performer plays a wrong note. Since a compact disc stores up to 650MB, this is quite a feat. Moreover, there is no visual equivalent; we do not remember pictures in so much detail.

Surprisingly, audio has received little attention in human-computer interaction in recent decades, with applications restricted mostly to limited speech or trivial sounds. This was certainly not the case in the early days of computing. Many researchers and engineers in the 1950s and 1960s cite examples of auditory output. An oft-repeated anecdote is that of the early programmers who tuned AM radios to pick up the interference produced by their computers. Listening to the patterns of sounds, they learned to monitor CPU activity and even to identify aberrant program behavior. Why were these lessons forgotten? While graphical capability has been available for years, inexpensive audio facilities are more recent. By the time affordable audio was available to the average computer user the study of the visual medium was well advanced despite audio storage requirements being much less demanding than video storage requirements.

Visual interfaces do offer significant advantages. First, they can be browsed, whereas auditory interfaces are heard through a narrow continuously moving aperture. Second, they can be quantitative, whereas audio is mainly qualitative. Above all, visual interfaces are not intrusive.

However, reliance on visually oriented interfaces produces its own stresses. Screens become cluttered with multiple windows, and information in one window is routinely hidden by subsequent windows. We cannot attend to the entire screen, especially in the case of multi-window graphical user interfaces (GUIs). Good design approaches ameliorate some of these problems, but video alone still creates problems for the visually impaired. These users, reliant on screen readers (and in some cases Braille) cannot use graphical programming and debugging tools. Use of screen readers is made more difficult by the multi-window implementations of modern program development environments.

The past decade or so has been characterized by renewed interest in auditory interfaces through the efforts of researchers worldwide. One important motivation is the provision of interfaces for visually impaired computer users for whom development of

GUIs is problematical; examples include the Sonic Finder [5] and the Soundtrack system [4]. Another motivation is the increasingly crowded nature of visual interfaces attempting to exploit the multimedia capabilities of sound and vision. In 1995 Alty [1] suggested that interface designers move beyond simple sounds and speech to exploit the rich potential offered by music, illustrating this idea with a graphical interface for blind users using only music.

The Debugging Problem

Observing that program debugging is often a "struggle against complexity," Lieberman [8] advocated using computers to help programmers visualize program behavior and relate the static information contained in the source code to the dynamic runtime activity.

The richness of musical representation offers fairly precise bug location possibilities (whether in isolation or in conjunction with visual representations). One possible representation is to map the tracing of the execution path through different modules to different instruments. This is not handled well by visual media that cause frequent screen shifts. Using timbre (the quality given to a sound by its overtones), allows us to easily follow switches between modules, letting our eyes concentrate on program detail. Indeed, in very large programs, visual debugging methods are not only tedious, they can be misleading in pointing to the real problem source; giving sound to a running program gives a broader picture of what is happening [6].

Program events take place in the time domain, whereas visual mappings are mostly spatial. Graphics provide good descriptions of spatial relations and structural details (as Fourier analysis does for sound waves) but do not naturally represent temporal details. Using sound presents us with a complementary modality that increases the number and type of diagnostic tools available to us by providing a temporal view of software (as the wave-form plot does for a sound wave).

Program auralization—the mapping of program data to sound—has attracted research interest in recent years. Using sound to aid the visualization of software was suggested in [3, 9]. The idea was further explored through a visual programming language to add audio capabilities to a debugger [7] and the Auditory Domain Specification Language [2]. Although these systems demonstrated technical feasibility, little formal evaluation of program auralization was ultimately done.

Auralization for Debugging

We built the CAITLIN system to allow the systematic study of musical program auralizations in debugging tasks. Our starting point was the fact that music perception is primarily temporal, that is, people more readily perceive those features in music that are in temporal relationships. When we listen to music we perceive them not as an arbitrary sequence of note durations but as a temporal structure in which notes are grouped. In particular, two features of temporal musical structure—succession and overlap—have analogs in the program domain (sequence and construct nesting). Also, as the structure of music (like that of programs) has multiple levels and given that the events of an executing program occur in a time-ordered framework, it seems worthwhile to attempt to map program events to musical events.

The CAITLIN preprocessor auralizes programs at the construct level (iterations and selections). That is, a `WHILE` loop is auralized in one way, and `REPEAT`, `FOR`, `CASE`, and `IF` constructs in other ways. CAITLIN recognizes the constructs in a source program and creates subroutine calls to stored musical sequences that are subsequently sent by the program to a MIDI (musical instrument digital interface) synthesizer (see Figure 1). The user can change various aspects of each construct's auralization, including instrument, note length, MIDI channel, and volume. The auralized program executes in the normal way (though it has to be slowed down for comprehension); and the musical sequences are heard (via the synthesizer) as the constructs are executed. The system works for Pascal, though its principles can be applied to other languages, including C and Java.

> NOW that we have shown that musical auralizations can be used by programmers, the next step is to see how auralizations compare with visualizations.

The CAITLIN auralization approach is based on the notion of a point-of-interest (POI), or "a feature of a construct, the details of which are of interest to the programmer during execution" [11]. Pascal's `IF`, `FOR`, `CASE`, `WHILE`, and `REPEAT` constructs each contain a number of POIs (such as entry to the loop

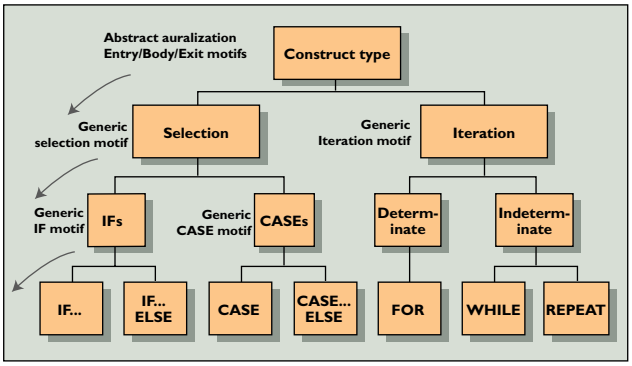


Figure 2. Program constructs in Pascal and their relationships with musical motifs.

and evaluation of a Boolean expression). Each construct is represented by a musical motif, or a short recurring theme associated with a particular thought or character; well-known examples are the themes given to the principal characters in Sergei Prokofiev's *Peter and the Wolf* (1936).

Motif Design

In the first prototype, the motifs were arbitrary in their design, the only consideration being to make each one distinct from the others to avoid ambiguity. In the second version, we tried to relate the motifs musically, hoping to aid memorization and recognition. Thus the two basic types of construct—selection and iteration—are based on different motifs; lower-level ones are related to the ones higher in the tree (see Figure 2). Each motif was played on a different timbre (musical instrument).

Common to all the constructs (with the exception of the `FOR` loop) is the idea of success or failure (more correctly, the evaluation of Boolean expressions that yield values of true or false). An `IF` statement either succeeds (and executes its statement block) or fails (and executes the next program statement); in the `IF...ELSE`, failure results in the `ELSE` part being executed. `WHILE` iterates until failure; `REPEAT` iterates until success. We mapped this with the metaphor of a major chord for success (true) and a minor chord for failure (false). Although most users would not be able to define these chords, they could still easily recognize them.

We also decided that a construct should have an opening sound and a mirroring closing sound (usually an inversion of the opening). This is analogous to HTML tags and the `IF..FI`, `DO..OD` pairings often used in pseudocode. We used it because listeners need to identify when constructs terminate, possibly at a time much later than the onset.

A third important feature we wanted to capture was that a construct persists over time. Subordinate statement blocks (which could themselves contain constructs) are often carried out within the body of a construct. Therefore, we added a drone (a continuously held note) between the opening and closing audio signatures to remind listeners that the construct is still active and that what is being executed is happening within a construct block. Nested constructs have differently pitched drones, so the multiple nesting can still be heard (as a chord); Figure 3 includes an example of two constructs.

Our hypothesis was that the musical program auralizations generated by CAITLIN could assist novice programmers in locating bugs that manifest themselves either directly or indirectly in terms of program flow.

A concern in this work is whether musical ability or experience is needed to use such a system. We are encouraged by other work [11] showing that while musical ability certainly aids recognition, the differences between musicians and non-musicians are not as great as one might expect, particularly for simple melodies. Most people are good at recognizing and memorizing music; the international success of the popular music industry supports this view. Cultural differences are another important issue, but again the evidence is that these differences become important only in complex musical structures; for example, most of the world's music systems are based on the same chromatic, or 12-tone, scale (or a subset of it). We have found that subjects' differing musical experience does not affect their ability to use auralizations.

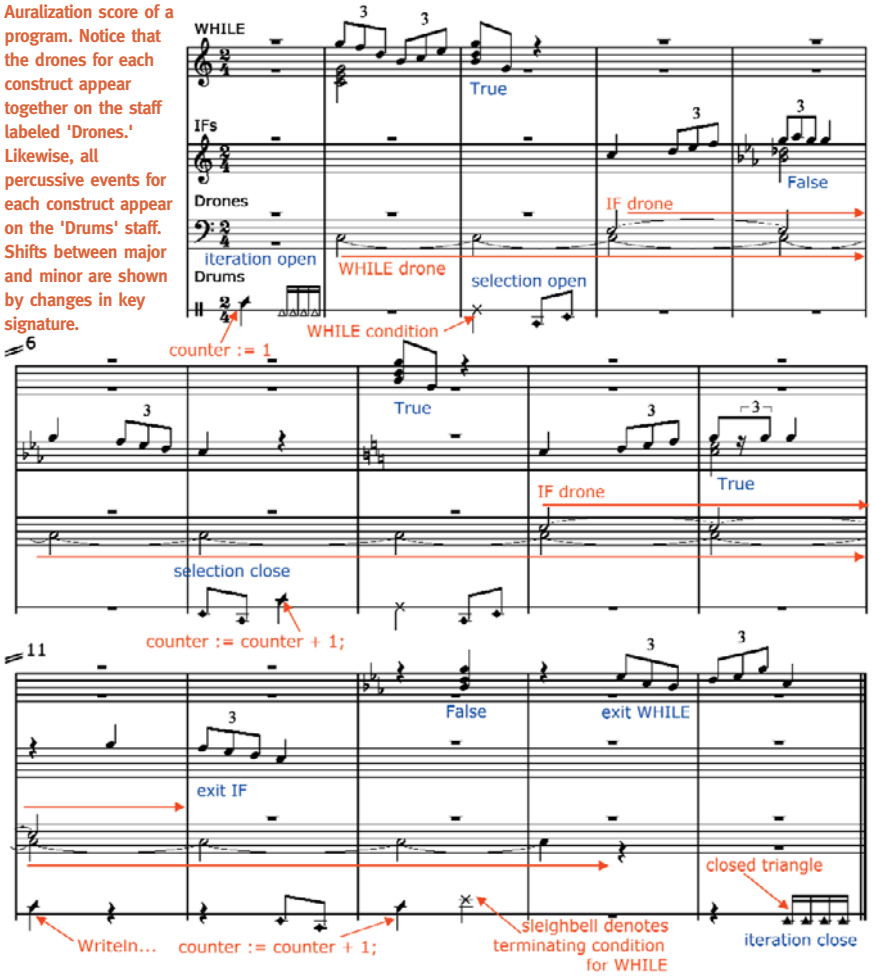
Evaluation

We conducted two studies to explore the usefulness of musical auralizations. In the first [10], 22 second-year computer science undergraduates at Loughborough University in the U.K. were asked to identify the Pascal constructs represented by 60 auralizations. We presented 40 single-construct auralizations followed by 10 construct-pair auralizations. Of the paired auralizations, 50% were of sequential constructs (such as a `WHILE` followed by an `IF`) and the other 50% of nested constructs (such as an `IF` within a `WHILE`). Because the motif design is hierarchical (see Figure 2) each construct could be identified at any of three levels: class (iteration or selection), subclass (such as `IF`s and `CASE`s), and specific identity (such as `IF`, `CASE...ELSE`).

The table here cites the average scores for the two sets of auralizations. If a construct is recognized correctly, then a Specific Identity score is given. If a mistake is made at this level (such as confusing an `IF` with an `IF...ELSE`) or a subject simply described the construct as one of the `IF`s, then a subclass score

(a) Source Program

```
PROGRAM EXemplar ;
VAR
    counter : Integer ;
BEGIN
counter := 1 ;
WHILE counter <= 2 DO
  BEGIN
▫    IF counter MOD 2 = 0 THEN
▫        BEGIN
▫        Writeln ('Counter is even') ;
▫        END ;
   counter := counter + 1 ;
   END ;
END.
```



Auralization score of a program. Notice that the drones for each construct appear together on the staff labeled 'Drones.' Likewise, all percussive events for each construct appear on the 'Drums' staff. Shifts between major and minor are shown by changes in key signature.

Figure 3. A program and its corresponding auralization: (a) IF statement within a WHILE statement and (b) auralization of the program as annotated musical notation revealing points of interest, success/failure signals, and drones.

is given. A class score meant a construct was identified correctly as either a selection or an iteration. While construct-specific recognition was not high (46% and 49%), subjects were good at recognizing construct subclasses (76% and 84%, respectively). We were pleased to find almost no confusion between nested and sequential constructs (recognition rate 97.5%).

The information presented to these subjects was without context; that is, they had no domain information to help them understand the problem. In a real debugging situation subjects would be listening to auralizations at the same time they would be looking at the program source (even blind programmers would have some textual/verbal representation of the code). The extra context provided by the presence of other constructs and understanding the aims and structure of the program would aid the recognition process. The programmer already knows from the source code which constructs have been used and whether or not the selections have ELSE branches. Therefore, the low specific identity scores do not worry us, as we would expect the added context of the program source to fill in the gaps. Given that the subjects had only half an hour to become familiar with the system prior to the study, more exposure might be expected to make the motifs more recognizable.

One feature of the hierarchic design is that a programmer should be able to listen to an auralization at varying levels of abstraction. To understand program flow it might be enough simply to hear that there is some form of selection here and some form of unbounded loop there; exact details can be gleaned from the listing. With training and continued use, programmers might become adept at distinguishing between all the construct auralizations even without contextual clues.

Therefore, to explore the role of the auralizations in bug location tasks we conducted a debugging test comprising eight different debugging exercises, numbered A1 to A8 [12]. For each, another 22 subjects were given a program specification, a pseudocode program design, sample input data, the expected output data, and the actual output data. Each program was syntactically correct and contained a single logical error (bug). Subjects were allowed as much time as they wanted to read the documentation. When ready to proceed they were given the program source code and had 10 minutes in which to locate

the bug. The eight tasks were performed in order, beginning with A1. Half the subjects had auralizations for tasks A1, A3, A5, and A7; the other half had the auralizations for tasks A2, A4, A6, and A8. Thus, each program was tested in both the auralized and the nonauralized states. As the 22 subjects were asked to locate one bug in each of the eight programs (four in the auralized state, four in the normal state) there were a possible 88 auralized bugs and 88 normal bugs. The subjects found a total of 60 auralized bugs, compared with 46 in the nonauralized state.

Set	Specific Identity	Subclass	Class	Total	Nested/ Sequential
Set 1	46%	30%	21%	97%	NA
Set 2	49%	35%	14%	98%	97.5%

Auralization recognition scores.

Conclusion

This research indicates that music can communicate information about program flow and assist with bug identification and eradication. The results highlight two areas where music seems particularly useful: where the program's output contains no clues as to the bug's location and where programs contain complex Boolean expressions. Otherwise, when the output gives clues (when, say, a loop displays only three lines of output instead of an expected 10), then a bug's location is relatively easy to deduce. However, when the program gives no such clue, it is more difficult to hypothesize about the bug's location. The auralization quickly shows it is the loop that is at fault. In the case of multiple complex Boolean expressions, the auralization made it easy to hear which ones were at fault; without auralization, subjects had to evaluate the expressions by hand.

Now that we have shown that musical auralizations can be used by programmers, the next step is to see how auralizations compare with visualizations. We have looked at program flow, but what about data flow and data structure? How might sound assist our understanding of these features?

Rather than replacing visual displays (though it would be useful for the visually impaired) we anticipate that combination audio/visual displays will be the most effective; for example, an animation of a data structure could be monitored while listening to an execution trace. Meanwhile, the message passing between objects in different threads of a Java program could be heard while inspecting the values of variables on the screen. We are conducting further research to explore these issues.

Whether program auralization really is the swan song of debugging problems remains to be seen. But we have strong hopes that combining auditory and visual external representations of programs will lead to new and improved ways of understanding and manipulating code. Ultimately, we hope the sound and light displays of multimodal programming systems will be standard items in the programmer's toolbox. More information and audio examples are at www.paulvickers.com/auralization. C

References

1. Alty, J. Can we use music in computer-human communication? In *Proceedings of HCI'95, People and Computers X,* M. Kirby, A. Dix, and J. Finlay, Eds. Cambridge University Press, Cambridge, U.K., 1995, 409–423.
2. Bock, D. ADSL: An auditory domain specification language for program auralization. In *Proceedings of the 2nd International Conference on Auditory Display (ICAD'94)* (Santa Fe, NM, Nov. 7–9). Santa Fe Institute, Santa Fe, NM, 1994, 251–256.
3. DiGiano, C. and Baecker, R. Program auralization: Sound enhancements to the programming environment. In *Proceedings of Graphics Interface'92* (Vancouver, BC, May 11–15, 1992), 44–52.
4. Edwards, A. Soundtrack: An auditory interface for blind users. *Hum. Comput. Interact. 4,* 1 (1989), 45–66.
5. Gaver, W. The SonicFinder: An interface that uses auditory icons. *Hum. Comput. Interact. 4,* 1 (1989), 67–94.
6. Hotchkiss, R. and Wampler, C. The auditorialization of scientific information. In *Proceedings of Supercomputing'91* (Albuquerque, NM, Nov. 18–22). ACM Press, New York, 453–461.
7. Jameson, D. Sonnet: Audio-enhanced monitoring and debugging. In *Auditory Display,* G. Kramer, Ed. Addison-Wesley, Reading, MA, 1994, 253–265.
8. Lieberman, H. The debugging scandal and what to do about it (special section). *Commun. ACM 40,* 4 (Apr. 1997), 27–29.
9. Stasko, J., Domingue, J., Brown, M., and Price, B., Eds. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA, 1998.
10. Vickers, P. and Alty, J. Musical program auralisation: A structured approach to motif design. *Interact. Comput. 14,* 5 (Oct. 2002), 457–485.
11. Vickers, P. and Alty, J. Using music to communicate computing information. *Interact. Comput. 14,* 5 (Oct. 2002), 435–456.
12. Vickers, P. and Alty, J. When bugs sing. *Interact. Comput. 14,* 6 (Dec. 2002), 793–819.

Paul Vickers (paul.vickers@northumbria.ac.uk) is a principal lecturer and Director of Research (Computing) in the School of Informatics at Northumbria University, Newcastle Upon Tyne, U.K. **James L. Alty** (j.l.alty@lboro.ac.uk) is a professor of computer science in the Department of Computer Science and Dean of the Faculty of Science at Loughborough University, Loughborough, U.K.

We would like to thank Loughborough University for providing the facilities and support to conduct this research. Paul Vickers would also like to thank former colleagues in the School of Computing & Mathematical Sciences at Liverpool John Moores University for participating in the pilot studies and the university itself for its financial assistance during his Ph.D. research.