Increasing Fault Detection Effectiveness Using Layered Program Auralization

Andreas Stefik, Kelly Fitz, and Roger Alexander
School of Electrical Engineering and Computer Science
Washington State University
Pullman, Washington 99164
{*astefik,kfitz, rta*}*@eecs.wsu.edu*

Abstract

This paper presents a new approach to using music for debugging computer code, layered program auralization. We use layers of musical structure to represent the state and behavior of a computer program while it is running, taking advantage of metaphorical relationships between musical structure and programming constructs.

This paper makes three contributions to the debugging and program auralization literature. We use cadences, recognizable patterns of chords that usually dictate an ending, to metaphorically represent nesting depth and hierarchical control structures. Auralizations, in our system, use more than one perceptual mapping at a time. Further, we decompose our auralizations into units, which we call layers. This decomposition is similar to using classes in object oriented programming.

We see debugging as a time consuming, difficult, task, and present a solution where music is played to the programmer during program execution. Our goal is to increase debugging effectiveness and to improve a programmer's comprehension of the runtime behavior of computer programs.

Keywords: *Fault detection, music, sonified debugging, testing, layered program auralization.*

1 Introduction

In this paper, we propose the use of music during the debugging process to help find faults while computer programs are executing. Static analysis of a computer program allows the programmer to see information about what a program should do over time, but music allows the programmer to hear what a program is actually doing, without halting a program's execution. Thus, layered program auralization is a form of dynamic analysis.

Consider a graphics application running at 50 frames per second, and consider a programmer that knows a fault occurs at around the 200th frame. Using traditional tools, the programmer must either write print statements in the code to see the runtime behavior, or halt execution in a debugger. Layered program auralization offers an alternative, allowing the programmer to "listen" for where a problem occurs.

This paper makes three contributions to the debugging and program auralization literature. We use cadences, recognizable patterns of chords that usually represent an ending, to metaphorically represent nesting depth and hierarchical control structures. Users do not need to remember that a sound occurred to determine nesting depth, and instead can determine this information from the current musical context.

Second, we combine multiple perceptual mappings during auralization. Some program constructs, like an IF, can be mapped to metaphorically equivalent sounds, like cadences. Other characteristics, like an arbitrary string or number, have no known, general case, metaphorical mapping. To circumvent this problem, we have adopted an approach where metaphorical and nomic mappings are combined.

Lastly, we present a system of layers for program auralization. A layer is a small, encapsulated, section of the total sound produced, and we use it primarily in the way object oriented programming(OOP) developers use classes. Just like in OOP programming, classes can interact and communicate. Layers do the same, but after communication occurs, each layer's output is overlaid to create an auralization, or in other words, music. This final process of auralizing the code is analogous to using viewgraphs on an overhead projector. Each layer is like a viewgraph. When combined on an overhead, the viewgraphs overlap to present a final image. When layers are combined, they aurally represent a running computer program. Because layers encapsulate our total design, we discuss our first two contributions in the context of the third, for the remainder of this paper.

We are using this new technology to increase a programmers effectiveness while debugging computer code. To test

layered program auralization, an empirical study was recently conducted [11], with positive results. Further empirical work is ongoing.

2 Background

In program auralization, the state and behavior of computer programs are represented in sound. Program auralization has been used to assist programmers in debugging tasks [14], to help blind users navigate hierarchical data structures [9], and to provide a richer and more informative desktop operating system interface [6].

One important concept for program auralization was first defined by Gaver when working on the SonicFinder, a perceptual mapping [5]. Most pertinent for this paper are Gaver's categories of perceptual mappings, symbolic, metaphorical, and iconic [6] or nomic [5].

A symbolic mapping has meaning only by convention, and must be learned and remembered, like a langauge. A metaphorical mapping makes use of a non-literal relationship between an object and its representation. For example, the operation of copying a file from one part of a filesystem to another can be represented by the sound of water filling up a drinking glass. An iconic, or nomic, representation looks or sounds like the thing it represents. For example, a nomic mapping of deleting a file, by throwing it in a virtual recycle bin, could be represented by the sound of throwing physical objects into a real recycle bin [5].

Boardman created the language Listen, now reimplemented in Java and called JListen [2]. The original motivation for Listen was to create a tool for describing how computer source code can be auralized. This work included the creation of the LSL, Listen specification language, that allows auralization code to be put into other computer programs. This specification tells the Listen system how to interpret computer code it receives, and what to auralize in that code. So, in Listen, the focus was on inputting code and outputting auralized code, but not on the actual sounds themselves.

Paul Vickers uses musical structures to design his auralizations [13, 14]. With his program, CAITLIN, Vickers describes his method of program auralization for the Pascal programming language. Vickers ran empirical studies using his auralizations, in which he showed a positive correlation between use of the auralizations and debugging simple Pascal computer programs.

One interesting concept in auralization, to appear in Vickers, is the, so called, "Point of Interest," or (POI). Vickers describes this concept as as "a feature of a construct the details of which are of interest to the programmer at execution" [12]. In our design, a layer is a mapping over one or more related points of interest.

In Vickers' work, musical events correspond to the execution of program statements that are marked as points of interest. For example, a major chord could indicate a true in an IF statement, and a minor chord could indicate a false [14].

Bonar *et al.* conducted empirical studies of novice programmers in an attempt to understand why programming is difficult for novices [3]. Spohrer and Solloway discuss several of the most common, or high frequency, faults, although they call them bugs, created in novice programs [10]. Chmeil gave training to novice debuggers in an attempt to increase their debugging skills in a CS1 course [4]. Ahmadzadeh created a two phase debugging study, where student programs were first tested for common compilation errors, and second tested for logic errors [1]. Layered program auralization techniques literally create music mapped to an executing computer program, and as such, only runtime or logic errors are relevant to the current work.

3 Using Layers to Represent Computer Programs

In this section, we expand on the idea of a layer, and give an overview of how we partitioned our problem space, and our overall contributions, into these layers. Layers are similar to the object oriented idea of a class, where a layer encapsulates some idea or functionality. In our case, this functionality is an element of a perceptual mapping [5], although we, predominately, use metaphorical and nomic mappings [5].

Layers represent computer program behavior with music. We represent three different categories of program information [8]: control flow, state, and semantic data. Previous program auralizers, such as Vickers', represented only control flow information [14]. Our layered auralizer produces a musical score, which can be performed by a music rendering system, like a MIDI synthesizer.

Layers can generate music at various levels of specificity. Some layers generate information as specific as notes on the page, and others generate abstract information, like the musical key to perform in, or the volume. Some information, like volume, cannot be performed alone. This non-performable information constitutes metadata about the music being produced.

The auralization design in this paper consists of three primary layers. The root layer, referred to as the cadential layer, generates chord progressions over time as a form of musical metadata, and also generates specific notes for those chords. These chord progressions are communicated to lower level layers.

Lower level layers, the orchestration and lyrics layers, interpret metadata from the cadential layer and create mu-

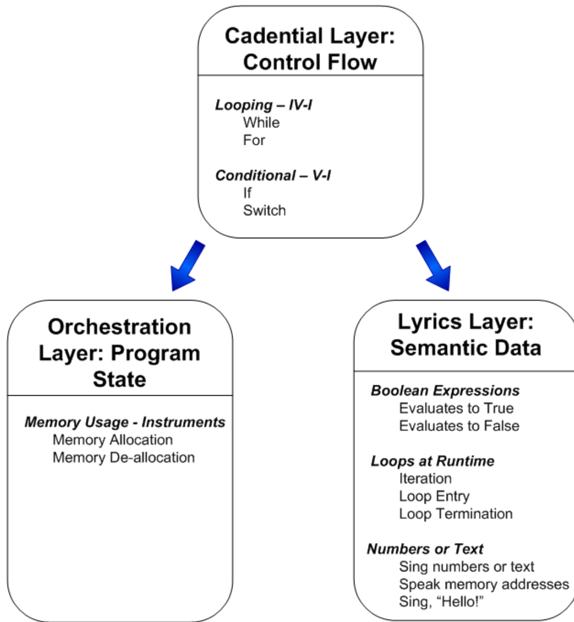

Figure 1. Overview of the auralization system

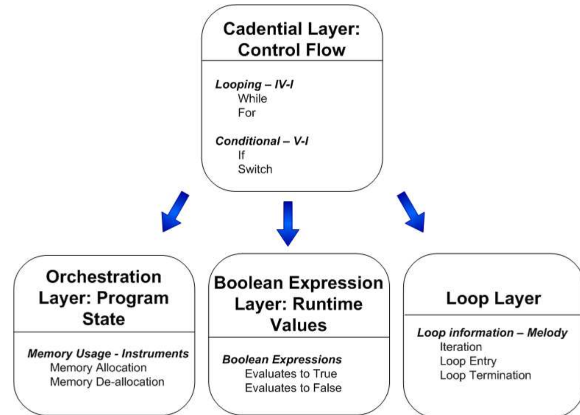

Figure 2. Overview with alternative layers

sic that can be performed from that data. The orchestration layer is based on a metaphorical mapping, and the lyrics layer is based on a nomic mapping. Figure 1 and Figure 2 give two alternative sets of layers. The second set of layers, in Figure 2, includes two layers not discussed in this paper, the boolean expression layer and the loop layer. These two layers consist of an alternative design that we tested in a recent empirical study on our auralization system. See Stefik [11] for more information on the empirical study or these alternative layers.

For a programmer, auralizations such as these may be useful, as it may inform the debugger what element of a running program is currently executing, like a branch of an IF statement. Note that the cadential layer is a metaphorical mapping. Other layers are not limited to any particular perceptual mapping [5].

To auralize C++ code, we designed a system to represent various program constructs, like loops and conditional statements, semantic data, like numbers or text, and dynamic memory allocation and deallocation. Thus, we are, whenever possible, mapping program constructs to metaphorically similar musical structures. We use these metaphorical mappings where possible, but not all behavior or data has an obvious metaphorical mapping. When this occurs, we create a new layer that uses a nomic mapping.

4 Auralization Examples

This section discusses the specifics of several layers in our layered program auralization design. For the examples in each subsection, we give the computer code, written in C++, and show the exact mapping from that code into music. Mappings with the same input always produces the same, in this case musical, output.

4.1 Cadential Layer - Control Flow

A *cadence* is a chord progression used to mark the end of a musical phrase. The *cadence* is a centuries-old technique. We used them in our system, largely, because they are aurally recognizable. For example, an "amen," technically called plagal, cadence, is often used at the end of a hymn.

Cadences give a strong "pull" to the end of a phrase or piece of music. So, in our work, we might play the introduction to a two chord cadence at the beginning of an IF statement, in a running computer program, and play the end of the same cadence at the end of the IF statement.

One goal in our design for this system is for a computer programmer to know where in the control flow a program is currently executing, without using a debugger or halting the program. This is especially useful for applications where halting the program is either inconvenient or difficult. Examples include graphics applications that make calculations on each frame, sound synthesis applications with thousands of calculations a second, or programs, in general, that run in an infinite loop.

Cadential patterns were chosen to represent control flow because of their likeness to computer code structures, and their recognizability as part of a musical structure.

Layers take advantage of a variety of musical metaphors. Each program construct is put in context of a cadence.

```
if ( node->getKey() == key )
{
   return node;
}
```

Figure 3. An IF statement

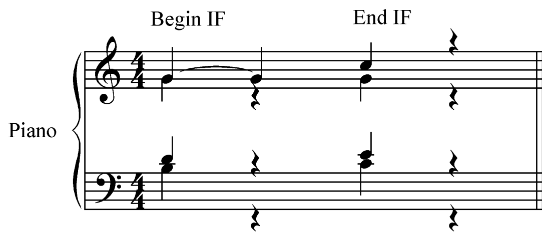

Figure 4. An auralization for an IF statement

Computer programs have structure, and cadences have musical structure in a similar way. Layered program auralization uses musical key, cadences, melody, and other tools, to give a stronger metaphorical mapping from computer programs into music.

Another advantageous element of cadences is that they can be extended by inserting chords in between the typical cadential pattern. This helps distinguish which IF-ELSE block, in a series of IF-ELSE blocks, is executing. In music theory, roman numerals are used to indicate chord progressions, and these notations were useful in our technical musical design. Roman numeral analysis of music is a simple way to show chord progressions in a key neutral way. While the specifics of the music theory are outside the scope of this paper, examples of the results of our cadence based approach are given.

Figure 3 shows code, and Figure 4 an auralization, for an IF statement in C++. Later, we will build upon this and other examples, and show how adding program state or semantic information does not alter this design. Encapsulation between layers makes adding new information similar to adding a viewgraph on an overhead projector. The total picture from the projector is altered, but the viewgraphs beneath the addition are unchanged.

If multiple IF statements occur one after the other, in a typical IF-ELSE construct, such as the one shown in Figure 5, the listener should be able to determine which block of the code is executing. Figure 6 gives a second example with several IF-ELSE combinations. This second auralization combines several IF-ELSE blocks into a short set of ca-

```
if (d < 10)
{ //1
   //statements to be executed
   //when d is less than 10
}
else if (d > 10 && d <= 30)
{ //2
   //when d is greater than 10
   //and less than or equal to 30
}
else if (d > 30 && d < =50)
{ //3
    //when d is greater than 30
    //and less than or equal to 50
}
else
{ //4
   //when d is none of the above
} //5
```

Figure 5. IF-ELSE statements

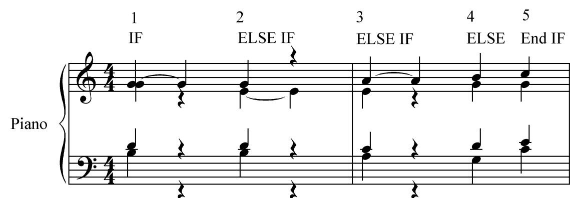

Figure 6. An auralization for IF-ELSE statements

dential patterns. The music represented in Figure 6 is what would play if, in the running computer program, the value of the variable *d* were, for example, 59.

Looping constructs, WHILE and FOR, are represented with a different set of cadential patterns. Each iteration of the loop, at runtime, changes the chords. Figure 7 shows an auralization for a FOR loop with ten iterations. The first chord is the beginning of the auralization. Each successive chord indicates another iteration of the loop. The last two chords indicate the end of the FOR loop.

Nesting of control structures, as shown in Figure 8, is common. We use key changes to indicate a change in nesting depth. As control flow leaves the nested region, the key changes back to the original, eventually leading to cadences in the original key. Figure 8 gives an example of a nested IF statement and Figure 9 gives its corresponding auralization. In addition to a cognitive benefit from a metaphorical mapping using cadences, they also sound musical!

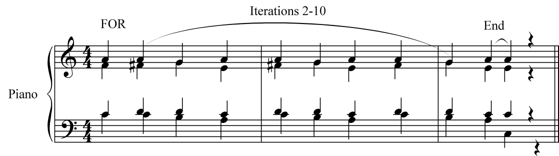

Figure 7. An auralization for a FOR loop

```
if (d < 10)
{ //1
   //executes if d is less than 10
   if (q < 15)
   { //2
      //executes if d is less than 10
      //and q is less than 15
   }
   else
   { //3
       //executes if d is less than 10
       //and q is greater than 15
   }//4
}//5
```

Figure 8. A nested IF statement

4.2 Orchestration Layer - Program State

Orchestration, at least for our purposes here, includes adding to, and removing from, the instruments in a musical score. Thus, in the cadential layer, we saw that only a piano was used, but when the orchestration layer is included, new instruments, like a saxophone or guitar, may be added, depending on the current program state.

Program state is "the connections between execution of an action and the state of all aspects of the program that are necessarily true at that point in time" [7]. Examples of program state include the number of nodes in a linked list or data stored in one particular node.

In our auralizations, we use orchestration to represent dynamic memory allocation in the implementation of a linked list data structure. As the computer program adds elements to the linked list, instruments are added to the auralization. The newly added instruments use metadata from the cadential layer to determine what melody to play and what musical key to play in. Similarly to addition, when nodes are removed from the linked list, instruments are removed from the auralization.

Figure 10 shows the program code for a method that removes all elements from the list, and releases the associated memory. Figure 11 shows the auralization of the *clear*

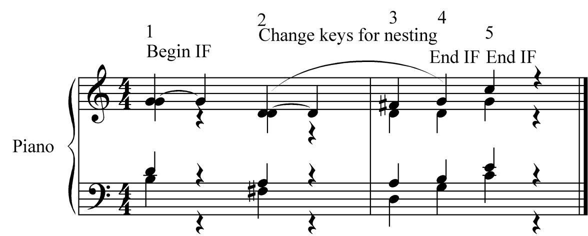

Figure 9. An auralization for a nested IF statement

```
void LinkedList::clear( void )
{
   Iterator it(this);
   while( it.hasNext() )
   {      //Cadential layer

      ListNode* node = it.next();
      Object* ob = node->getObject();

     delete node;
     delete ob;
            //Orchestration layer

  }
   head = 0;
   numNodes = 0;
}
```

Figure 10. The clear method in a linked list

method as it deletes four nodes. In this figure, the piano part is created by the cadential layer. The last five parts are created by the orchestration layer.

The calls to *it.next()* and *node→getObject()* are not auralized. The *delete* operation is also not auralized directly, but, instead, the orchestration changes from this operation occurring. Note that the orchestration of the passage "thins out" as the memory is deleted. This operation works well when you do not need to know the exact number of objects removed. The thicker the orchestration, the more memory is consumed.

4.3 Lyrics Layer - Semantic Data

Previous work in program auralization included musical information with non-speech based audio only [12]. This musical information is typically auralized using either a symbolic or metaphorical mapping. However, using this approach, it is difficult to represent numbers and text. Further,

Figure 11. An auralization for the clear method

Figure 12. An auralization for a nested IF with lyrics

Figure 13. An auralization for adding a node to a linked list

while musical elements may be created with good reason, or a strong sense of logic, they must be memorized to be understood by the listener. Our approach includes speech based audio, specifically singing, with lyrics based on the computer code. Using speech based audio, in conjunction with our existing cadential structure, combines a metaphorical mapping with a nomic one.

In the previous musical examples, we see several difficulties in determining the location of a running program in code from cadential sounds alone. The largest difficulty is making the cadential structures sufficiently different, so that an untrained listener can tell them apart. To solve this problem, words can be used to help the listener understand the current location in an executing program. Figure 12 shows a nested IF statement auralized with the cadential layer, the piano, and the lyrics layer, the tenor.

It is impractical to map numeric data into pitch numbers in the general case, and an obvious intuitive mapping from music into text does not exist. As an example, consider trying to represent the text “Hello, how are you Sally?” To assign auralizations to this text, we could define melodies for each character, melodies for each word, cadential patterns, or other techniques, but as the number of text strings the user tries to remember grows in size, it becomes increasingly difficult to determine an appropriate set of sounds. Since a metaphorical mapping does not exist for this type of information, a different solution was necessary.

Adding a lyrics layer simplifies representing strings and numbers by making their audio representation nomic [6]. Figure 13 and 14 show an example of program constructs causing the auralizer, using the lyrics layer, to sing a memory address. With music alone, representing the address is difficult, but using the lyrics layer, the representation becomes non-ambiguous.

5 Conclusion

We have discussed a new system of auralizing program constructs for use in debugging computer programs. We

```
void LinkedList::add( ListNode * node ) //1
{
   if( NULL == head ) //2
   { //Cadential layer
      head = node; //3
            //Lyrics layer

      // ERROR here:
      // should increment numNodes
   }
   else
   {  //Cadential layer

      node->setNext(head);

      ++numNodes;
      head = node;
            //Lyrics layer
   }
} //4
```

Figure 14. Adding a node to a linked list

used this system of auralization to write music that potentially helps programmers debug computer programs more effectively. Recently, we conducted an empirical study to determine the effectiveness of the techniques for debugging computer code [11]. In this study, 29 sophomore level programmers, broken into three experimental groups, used the auralizations presented in this paper. All three groups were given debugging tasks in C++. One group was given no auralizations, another was given musical auralizations, and the third was given musical auralizations with the addition of lyrics. Using standard ANOVA statistical tests, we found, in one trial, that the music only group was more effective at finding faults than the control group, with a p-value of 0.003. On the other hand, we found that, given no training in the use of auralizations, that the lyrics group was less effective than the control group, with a p-value of 0.022. Other trials did not lead to statistically significant results.

In this study, we measured the number of faults novice programmers found while debugging C++ computer programs with and without auralizations. The results of this empirical study indicated that, in one case, an example where programmers are working predominately with control flow, and without lyrics, the programmers found significantly more faults than those without auralizations. With positive results in hand from our empirical study, we are currently designing and implementing a sonified debugger into Microsoft Visual Studio.NET and preparing for further empirical work.

References

[1] M. Ahmadzadeh, D. Elliman, and C. Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM Press.

[2] D. B. Boardman, G. Greene, V. Khandelwal, and A. P. Mathur. Listen: A tool to investigate the use of sound for the analysis of program behavior. In *Computer Software and Applications Conference, 1995. COMPSAC 95. Proceedings., Nineteenth Annual International*, pages 184–189, 1995.

[3] J. Bonar and E. Soloway. Uncovering principles of novice programming. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 10–13, New York, NY, USA, 1983. ACM Press.

[4] R. Chmiel and M. C. Loui. Debugging: from novice to expert. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 17–21, New York, NY, USA, 2004. ACM Press.

[5] W. W. Gaver. Auditory icons: Using sound in computer interfaces. *Human-Computer Interaction*, 2:167–177, 1986.

[6] W. W. Gaver. The sonic finder: An interface that uses auditory icons. *Human-Computer Interaction*, 4:67–94, 1989.

[7] N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, E. Soloway, and B. Shneiderman, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113, Westport, CT, USA, 1987. Greenwood Publishing Group Inc.

[8] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[9] A. C. Smith, J. S. Cook, J. M. Francioni, A. Hossain, M. Anwar, and M. F. Rahman. Nonvisual tool for navigating hierarchical structures. In *ASSETS '04: Proceedings of the ACM SIGACCESS conference on Computers and accessibility*, pages 133–139, New York, NY, USA, 2004. ACM Press.

[10] J. G. Spohrer and E. Soloway. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 230–251, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[11] A. Stefik. An empirical comparison of program auralization techniques. Master's thesis, Washington State University, Pullman, WA, Dec 2005.

[12] P. Vickers. *CAITLIN: Implementation of a Musical Program Auralisation System to Study the Effects on Debugging Tasks as Performed by Novice Pascal Programmers*. PhD thesis, Loughborough University, 1999.

[13] P. Vickers and J. L. Alty. When bugs sing. *Interacting with Computers*, 14(6):793–819, 2002.

[14] P. Vickers and J. L. Alty. Siren songs and swan songs debugging with music. *Communications of the ACM*, 46(7):86–93, jul 2003.