Available online at www.sciencedirect.com
ScienceDirect
Int. J. Human-Computer Studies 69 (2011) 820–838
International Journal of Human-Computer Studies
www.elsevier.com/locate/ijhcs

An empirical investigation into the design of auditory cues to enhance computer program comprehension

Andreas Stefik[a,*], Christopher Hundhausen[b], Robert Patterson[c]

[a]Southern Illinois University Edwardsville, Department of Computer Science, USA
[b]Washington State University, Human-centered Environments for Learning and Programming (HELP) Lab, School of Electrical Engineering and Computer Science, USA
[c]US Air Force Research Laboratory, USA

Received 26 July 2010; received in revised form 18 April 2011; accepted 7 July 2011
Communicated by P. Mulholland
Available online 18 July 2011

Abstract

Decades of research have led to notable improvements in the representations used to aid human comprehension of computer programs. Much of this research has focused on visual representations, which leaves open the question of how best to design *auditory* representations of computer programs. While this question has particular relevance for visually impaired programmers, sighted programmers might also benefit from enhanced auditory representations of their programs. In order to investigate this question empirically, first, we introduce *artifact encoding*, a novel approach to rigorously measuring the comprehensibility of auditory representations of computer programs. Using this approach as a foundation, we present an experimental study that compared the comprehensibility of two alternative auditory program representations: one with *lexical scoping cues* that convey the nesting level of program statements, and another without such scoping cues. The results of our first experiment validate both artifact encoding and the scoping cues we used. To see whether auditory cues validated through our paradigm can aid program comprehension in a realistic task scenario, we experimentally compared programmers' ability to debug programs using three alternative environments: (1) an auditory execution environment with our empirically derived auditory cues; (2) an auditory execution environment with the current state-of-the-art auditory cues generated by a screen reader running on top of Microsoft Visual Studio; and (3) a visual version of the execution environment. The results of our second experiment showed that our comprehensible auditory cues are significantly better than the state-of-the-art, affording human performance approaching the effectiveness of visual representations within the statistical margin of error. This research contributes a novel methodology and foundational empirical data that can guide the design of effective auditory representations of computer programs.

Keywords: Auditory programming; Programming; Debugging; Program comprehension

1. Introduction

Computer programming is a notoriously difficult task. One challenging aspect is that of understanding the behavior of computer programs—an endeavor that becomes increasingly difficult as their size and complexity grow. How people comprehend computer programs has been widely studied (Mayrhauser and Vans, 1994), and modern integrated development environments (IDEs) have adopted many innovations to aid program comprehension. These include edit-time features such as enhanced typography (e.g., auto-indentation and syntax highlighting), as well as runtime features such as variable watch windows, the ability to execute a program both forwards and backwards (Snodgrass, 1984; LeDoux and Parker, 1985; Feldman and Brown, 1989; Pothier et al., 2007), and software visualizations of program code, data structures, and structure (see, e.g., Stasko et al., 1998).

Thus far, research into program comprehension aids has focused predominately on visual features and representations. Given that understanding a computer program requires a rich array of information, and given that humans possess a highly

*Corresponding author.
E-mail address: stefika@gmail.com (A. Stefik).

1071-5819/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ijhcs.2011.07.002

efficient visual processing system, this focus is altogether sensible. However, for human–computer interaction researchers interested in multimedia computing, this focus on purely visual representations leaves open at least two key research questions:

Research Question: *What influence do* auditory *representations of computer programs have on our ability to program, debug, or comprehend software?*

This first question leads to another:

Research Question: *How can we design effective auditory representations of computer programs?*

While answers to these questions are particularly relevant to the design of auditory programming environments for the blind and visually impaired, they can also guide the design of programming environments for the sighted, whose comprehension of computer programs might be enhanced by well-designed auditory representations (as in, e.g., Stefik and Gellenbeck, 2009).

In this article, we present a line of preliminary research that aims to address the above questions. One of the key problems in designing effective auditory representations of computer programs is that of measuring their comprehensibility. To address this problem, we present *artifact encoding*, a novel approach to rigorously measuring the comprehensibility of auditory program representations. Using artifact encoding as a basis, we developed auditory *lexical scoping* cues to aid human comprehension of the scope of program statements—one of the more difficult aspects of programs to comprehend aurally. In an experimental study, we found that such scoping cues significantly improved participants' ability to comprehend the behavior of source code. In order to explore whether auditory cues validated through our methodology might aid program comprehension in a more realistic scenario, we then experimentally compared human debugging performance using three alternative program execution environments:

- *experimental*: an audio-only environment with our empirically derived auditory cues;
- *state-of-the-art*: an audio-only environment with the state-of-the-art auditory cues generated by a screen reader running on top of Microsoft Visual Studio; and
- *control*: a visual version of the same environment with no auditory cues.

The results of our second experiment showed that the auditory cues in the *experimental* and *control* conditions promoted significantly faster debugging times than those in the *state-of-the-art* condition; however, there were no statistically significant differences between the *experimental* and *control* conditions. In other words, our empirically derived auditory cues promoted performance that was significantly better than the state-of-the-art, and that fell within the statistical margin of error of the performance promoted by a visual environment.

We make three key contributions to the literature on auditory program comprehension in this article. First, we have developed *artifact encoding*, a novel paradigm for analyzing the comprehensibility of auditory representations of computer programs. Second, we validate the effectiveness of both the paradigm and several auditory scoping cues in a formal empirical study. Third, we present evidence that the artifact encoding approach can be used to design auditory representations that are significantly better than the state-of-the-art, and that approach the effectiveness of visual representations within the statistical margin of error. Thus, our research provides a rigorous empirical foundation for future research into auditory representations of computer programs—research that could benefit non-sighted and sighted programmers alike.

The remainder of this article is organized as follows. After reviewing related work in Section 2, we present our *artifact encoding* approach in Section 3. Section 4 demonstrates the value of the artifact encoding methodology by using it as the basis for an experimental comparison of alternative auditory cues to promote comprehension of program scope. In Section 5, we present the Sonified Omniscient Debugger (SOD),[1] a custom program execution environment we developed in order to experimentally compare auditory cues in realistic comprehension tasks involving program execution. In Section 6, we use SOD in a follow-up experimental study to compare the effectiveness of two alternative sets of auditory cues, as well as the visual version of the environment, in promoting the comprehension of computer programs in a more realistic debugging task. Finally, in Section 7, we summarize our contributions and outline directions for future research.

2. Related work

This article presents a new approach to measuring the comprehensibility of auditory representations of computer programs, along with a pair of experimental studies that validate the methodology and provide empirical evidence that well-designed auditory cues can significantly improve the human performance. Below, we situate our work within the context of four lines of related work: (1) auditory display techniques, (2) novice programming environments, (3) empirical comparisons of alternative program representations, and (4) assistive auditory technologies for computer programming.

Auditory display techniques use audio to represent either data or information about data (Walker and Kramer, 2005). Gaver (1986) pioneering work on SonicFinder, a system that used audio to provide a richer and more informative desktop interface, introduced the idea of perceptual mappings between data and sound, and defined three different types: symbolic, metaphorical, and iconic. Auditory displays have been developed to assist the blind and visually impaired in

[1]Because they predominately use speech sounds, SOD (the Sonified Omniscient Debugger) and Sodbeans (the Sonified Omniscient Debugger in NetBeans) are technically auditory display environments, not sonification environments. The tools' misleading names are an artifact from their early days, when they generated musical sounds instead of speech.

a variety of tasks, including navigating web pages (Takagi et al., 2009; Rotard et al., 2005; Asakawa, 2005; Parente, 2004), reading tables (Abu Doush et al., 2009; Yesilada et al., 2004; Oogane and Asakawa, 1998), comprehending mathematics (Gellenbeck and Stefik, 2009; Bernareggi and Archambault, 2007; Karshmer et al., 2007, 2004; Soiffer, 2007; Stevens, 1996), browsing molecular structures (Brown et al., 2004), improving short term memory (Sánchez and Flores, 2004), and video games (Allman et al., 2009; Folmer et al., 2009; Yuan and Folmer, 2008).

A rich legacy of research into novice programming environments (see Kelleher and Pausch (2005) for an excellent overview) share our interest in easing the programming task and making computer programming easier. Within this large body of work, some of the themes that resonate with the work presented here include:

- studying the programming practices of novices as they use conventional programming languages (Bonar and Soloway, 1983; Soloway et al., 1983);
- studying the "natural" ways in which people program when they are not constrained by conventional programming languages, and building tools to accommodate those practices (Myers and Pane, 1996; McIver; Myers et al. (2004, 2008); Ko and Myers, 2009);
- using multimedia and graphics in order to engage learners and ease programming tasks (Gross and Powers, 2005; Pausch, 2008; Guzdial and Soloway, 2002; Mullins et al., 2009; Dougherty, 2007; Cooper et al., 2000; Stefik and Gellenbeck, 2009); and
- augmenting programming environments with visual representations of program code and its dynamic behavior, in order to enhance learners' understanding of their programs (Levy et al., 2003; Carlisle et al., 2005; Hendrix et al., 2004; Hundhausen et al., 2009).

A large body of empirical research has been concerned with comparing the relative effectiveness of alternative representations of computer programs. One significant line of work within this body has focused on comparisons between visual and textual representations (see Whitley (1997) for an overview). A key result that has emerged from this line of work is that the most effective representation is closely tied to the task to be performed with the representation (see e.g., Green et al. (1991)).

Another line of work within this body has focused on comparing the effectiveness of alternative typographical styles of source code (Baecker, 1988; Miara et al., 1983). This line of research has shown that typography does significantly impact program comprehension; Oman and Cook (1990) distill the empirical results of this research into a set of principles of good typographic style.

While computer programming (Green and Petre, 1996), debugging (Mayrhauser and Vans, 1997), and comprehension (Pennington, 1987a,b) tasks have been widely studied over the past three decades, relatively little research has considered the use of assistive auditory technologies for these tasks. Begel and Graham (2004) created Spoken Java, a tool designed to allow users to speak computer code instead of typing it. This work can be characterized as the reverse of the work presented here. Spoken Java provides means of inputting computer programs via speech, whereas our work provides means of outputting computer programs as, predominately, speech.

Boardman et al. (1995) created the language LISTEN to explore alternative mappings between source code and sound. In contrast to our work, LISTEN focused on facilitating code-to-audio mappings, not on the design of the auditory cues. Brown and Hershberger (1991) augmented algorithm animations in their Zeus system with "algorithm auralizations." Their musical auditory displays mapped higher-pitched tones to larger magnitude data in the algorithms being auralized. Brown and Hershberger claimed that their auditory displays assisted in the comprehension of the algorithms by communicating patterns, conveying additional information not depicted in the animations, and signaling exceptional conditions; however, they did not carry out any empirical evaluations to test these claims, nor did they attempt to measure comprehension.

Vickers and others (Vickers and Alty, 2002, 2005; Finlayson and Mellish, 2005; Berman and Gallagher, 2006) have promoted the use of musical cues as auditory representations. In Vickers case, he built a program auralization system called CAITLIN, which used musical auditory cues to represent the execution of computer programs written in Pascal. In experimental studies, Vickers failed to show that participants could find more bugs with musical cues, although he claimed to have found that the effectiveness of the musical cues increased with the cyclomatic complexity of the source code. Further, one could imagine using musical cues to supplement speech, for example, one might use continuous drones to convey lexical scoping depth. However, the latest literature has shown that musical auditory cues are, in fact, difficult to learn (Palladino and Walker, 2007), and as such, we explore here the use of speech-based cues as opposed to musical ones.

In a line of research perhaps most closely related to the work presented here, Francioni studied the use of sound to debug programs executing in parallel (Francioni et al., 1991a,b). Berman and Gallagher (2006) auralized program slices using musical structures, hypothesizing that could potentially help an individual to comprehend the context of a particular line of code (e.g., how does this line relate to others?). Finlayson et al.'s (2007) fisheye view similarly considers scope (or indentation) with an auditory environment, although their focus is on trying to determine what lines of code might be important, not on what auditory cues should be output to a user. Smith et al. (2004) developed a speech-based hierarchical structure analysis tool, JavaSpeak, specifically for non-sighted computer programmers. Others still have focused on outreach with the blind or visually impaired community (Bigham et al., 2008).

In general, the trend in the assistive technology of the programming literature is to focus either from a top-down perspective, analyzing auditory environments as a whole, or doing community outreach work with the blind and visually impaired community, without, necessarily, rigorous empirical data gathering. In contrast, our work includes a novel technique for bottom-up analysis (Stefik et al., 2007) (artifact encoding), top-down (Stefik and Gellenbeck, 2009) (analyze an environment as a whole), and outreach (e.g., we work with five schools for the blind across the U.S. and have two blind programmers on our active development team) components. In addition, in work that builds upon the auditory cues derived through the research presented here, we have built an auditory integrated development environment (IDE) tailored for the blind. This IDE—Sodbeans is integrated into Oracle's NetBeans (Stefik et al., 2009), is available through 〈http://www.sourceforge.net〉, and, while not discussed here, was deployed in January of 2011 to the Washington State School for the Blind as part of an 18-week programming course to blind students (Stefik et al., 2011).

3. Measuring comprehension with artifact encoding

Perhaps the most fundamental facet of an auditory tool's usability is whether the user understands the meaning of the audio. Poor comprehension of auditory cues can cripple usage of a real-world environment. Users can neither interact with nor modify components of a system that they do not understand. But how does one measure the comprehensibility of an auditory cue? In this section, we present a novel approach called *artifact encoding*.

Artifact encoding is an experimental paradigm for measuring human comprehension of computer programs. Artifact encoding uses free-form writing to try to estimate an individual's understanding of stimuli. While artifact encoding can be applied to a wide variety of contexts (e.g., visual stimuli), we focus our discussion here toward using it for auditory comprehension. The advantage to artifact encoding is that it generates a great deal of data and that it can be automatically analyzed by a computer. Previous approaches tend not to give enough information about a user's comprehension (e.g., multiple choice studies (Finlayson and Mellish, 2005)), or do not specifically study comprehension (e.g., usability studies (Ellis et al., 2007)). In the next subsection, we describe how artifact encoding works.

3.1. Creating artifact encoding studies

Artifact encoding studies have three primary phases: training, identification, and interpretation. When a study begins, participants are given information on what they will be doing in that study, including the type of sounds they will be hearing and how to write down their answers. Once participants are trained, they listen to a set of auditory cues—usually words in varying orders and contexts. Participants then use a shorthand notation to write down what they heard; they will use this shorthand to interpret the cues as a whole after listening. After the identification stage, participants are given several minutes to interpret the auditory cues they heard and make decisions regarding their meaning. Questions they might consider include: Does this auditory cue indicate a loop or a selection statement? Does this cue indicate a scoping relationship or the number of iterations in a loop?

After an artifact encoding study is complete, participants' answers are coded into strings, as summarized in Fig. 3. Because this coding process is subject to interpretation, multiple analysts must code a subset of the answers, and a reliability analysis must be performed in order to confirm the reliability of the coding system (see e.g., Dewey, 1983 and Hubert, 1977). As we will demonstrate, this process is very reliable. Once all strings are coded, they are sent to a computer program for automated analysis.

Consider, for example, Fig. 1, in which a user has indicated that she heard two `if` statements, one nested inside the first. The auditory cues the participant may have heard were, "if true, 1 nested if false." There are many choices for sounds indicating various program constructs (in pilot testing the artifact encoding paradigm and methodology, we tried many of them). The point is that participants hear some kind of auditory cue, write down what they think the cue meant, and then we document and analyze the similarities and differences. The participant answer in Fig. 1 would be coded as ITNIFU, where the I means `if`, the T and F mean true and false, and the characters N and U indicate nesting relationships. To be clear, the characters N and U are best interpreted in this context as the beginning and ending of lexical scoping blocks. For example, a loop with multiple iterations at runtime would have more Ns and Us than an `if` statement, as loops begin and end new lexically scoped blocks on each iteration. While this is true for our experiment in this paper, one can just as easily imagine creating a different artifact encoding scheme for static scoping relationships. This coding system expands on the one presented in Stefik et al. (2007).

Similarly, consider coding the loop in Fig. 2. In this case, the 3 inside the `while` indicates that the loop executed three times. The `if(T)` inside the loop means that this `if` statement was inside the loop and that it happened to be true on the first iteration, false on the second, and true on the third. Again, we code these participant answers into strings for computer analysis. In this case, the answer is correctly coded as LNITU LNIFU LNITU, where L means the beginning of the loop and U means the end of an iteration of a loop (its lexical scope). Note that the boolean is potentially meaningful for these statements as well, but that if it were coded directly, the boolean would be implicitly coded twice, once for the end of the loop and twice for recognizing the boolean itself. In other words, if

```
if(T) {
    if(F) {}
}
```

Fig. 1. An example of a conditional statement written by a participant.

```
while(3) {
    if(T, F, T) {}
}
```

Fig. 2. An example of a loop written by a participant. The comma after each T or F means a new iteration of the loop.

Table 1
An example comparison of a participant's answer (left) and the correct answer (right). The total comprehension score for this example is 4 out of 6, or 66.6%.

Human	Answer
I	I
T	T
	N
I	I
F	F
	U

the boolean is true, the loop continues, which is coded, and if it is false, there are no further entries for the loop.

Once strings are coded, they are graded by analyzing participant answer/correct answer pairs, like the one given in Table 1. In this case, the participant appeared to realize that two `if` statements executed (all of the examples here are of executing computer programs), but did not realize that the second `if` statement was nested inside the first. Blind programmers, we collaborate with in our outreach work who rely on auditory representations of programs, commonly complain about not knowing the current nesting level of a program statement. Thus, the issue of nesting is a prime candidate for further empirical investigation.

3.2. Analyzing artifact encoding studies

The first part of the grading process is to globally align the strings. Small strings can be matched by hand, but with longer examples this would be error-prone and tedious. Thus, we have developed a computer tool called the *Artifact Encoding Calculator* (AEC—pronounced "ack") to automate this process (see Fig. 4).

Using the Needleman–Wunsch amino acid sequence comparison algorithm (Needleman and Wunsch, 1970), we can obtain a *global alignment* between two strings. This algorithm is guaranteed to find the optimum match between the strings, using any chosen criteria for calculating what optimum means. There are, however, two minor differences in the typical implementation and our own.

The first difference is that, in the Needleman–Wunsch algorithm (Needleman and Wunsch, 1970), the language of acceptable characters in the algorithm is defined as the legal characters used for DNA, $\sum = \{A, G, C, T\}$, and the algorithm has a biological meaning. In our artifact encoding approach, however, we define $\sum = \{I, T, F, N, U, L, Z\}$. The only character here that has not been described is Z. This rarely used code indicates that the user made a mistake that could not otherwise be classified (see Stefik, 2008, for examples). This character difference is minor and has no bearing on the effectiveness of the algorithm.

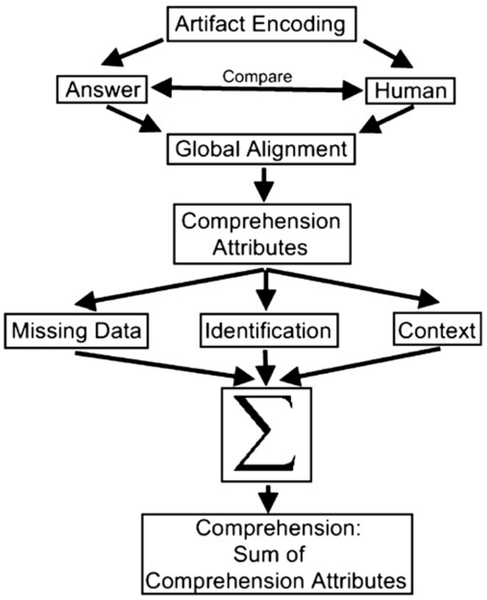
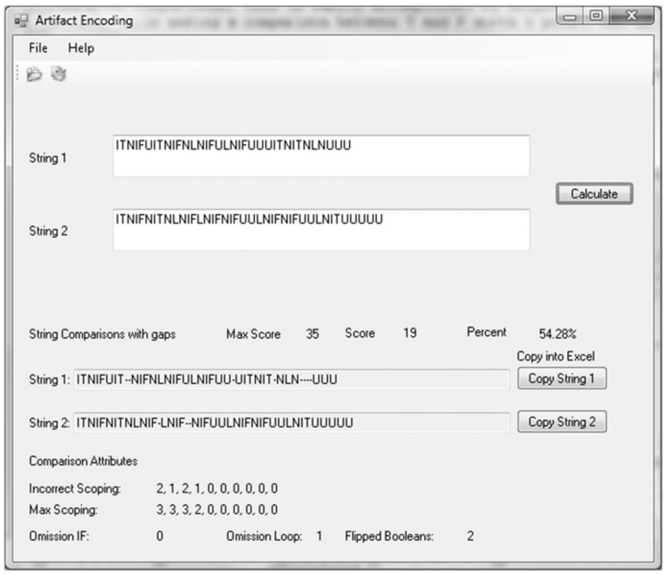


Fig. 3. A basic overview of Artifact Encoding, including the automatic grading process.



String 1 ITNIFUITNIFNLNIFULNIFUUUITNITNLNUUU

String 2 ITNIFNITNLNIFLNIFNIFUULNIFNIFUULNITUUUUU

String Comparisons with gaps Max Score 35 Score 19 Percent 54.28%

String 1: ITNIFUIT--NIFNLNIFULNIFUU-UITNIT-NLN----UUU

String 2: ITNIFNITNLNIF-LNIF--NIFUULNIFNIFUULNITUUUUU

Comparison Attributes

Incorrect Scoping: 2, 1, 2, 1, 0, 0, 0, 0, 0, 0

Max Scoping: 3, 3, 3, 2, 0, 0, 0, 0, 0, 0

Omission IF: 0 Omission Loop: 1 Flipped Booleans: 2

Fig. 4. A Screenshot of the artifact encoding automatic grading program, using a modification of the Needleman–Wunsch algorithm.

Second, in the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970), optimum is usually defined by assigning point values to character matches. For example, the character T, if matched to another T, might be worth 5 points, but N matching with N might be worth 3 points. In our case, all characters are given equal weight, with one exception. In `if` statements, we want to be able to detect whether participants determined the correct truth value for a statement, or alternatively, whether they "flipped" a boolean—meaning the participant said that the `if` statement was true or false, but the opposite was true.

To see why this might be a problem, consider globally aligning two strings ITNITU and ITIF. If all characters in the string are equally weighted, then it would be equally acceptable to match the string ITIF as either IT-IF- or IT-I-F, where dashes indicate a gap when compared to the string ITNITU. However, it would be desirable for the Needleman–Wunsch algorithm (Needleman and Wunsch, 1970) to automatically match up the string as IT-IF-, not IT-I-F, because then it can iterate through the string looking for flipped booleans. Fortunately, by assigning all like characters a weight of two points, T and F a weight of 1 point, and mismatches a weight of zero points, we can obtain the desired global alignment of our strings.

However, a global alignment is insufficient to inform us about the comprehension of auditory cues. For that, we need to post-process globally aligned strings. While many techniques are possible, we have defined five metrics of interest: (1) *Aggregate comprehension*, (2) *Incorrect scoping*, (3) *Missing* `if` *statements*, (4) *Missing loops*, and (5) *Flipped Booleans*.

The first metric is the *aggregate comprehension* score (see Table 1), a global measure of how well the participant correctly interpreted the auditory cues. To calculate the comprehension score, we sum the number of matched pairs in the two strings. For example, given the string ITNITU and ITIF, where the matched strings are ITNITU and IT-IF-, the comprehension score would be 4/6 or 66%. In Fig. 4, a much more complicated alignment and comprehension score is calculated, where the result is 54.28%.

The second metric is *incorrect scoping*. Consider the strings ITNITU and ITIT. To calculate this measure, we linearly post-process both strings in parallel. Whenever both strings are at an I or an L character, we check to see whether the scoping level of both constructs is equal, which we know by counting the N and U characters that are within the string.

If both I and L characters in the second string are at the same scoping level as the first, nothing happens, but if they are incorrect, we add one point to the number of incorrect scoping constructs at the appropriate scoping level. There is a problem, however. Consider the strings ITNITNITUU and ITITNITU. In this case, using the previous algorithm, we will count both the second and third `if` statements at the wrong scoping level, even though the participant must have known that the third statement was scoped inside the second, as shown in Fig. 5.

The type of problem, demonstrated by Fig. 5, is an example of a *cascading error*. Removing this type of error is important. If it is not removed, one mistake at the beginning of an answer will cascade through the rest of the answer. Fortunately, removing this type of error is simple. Suppose two characters are compared at a given scoping level and found to be incorrect. If this happens, the participant's answer is artificially set to the correct scoping level and processing is allowed to continue. The practical result of this is that participants are only docked points for every individual statement that is incorrect within the local scope, as opposed to being docked globally.

```
if ( T ) {                   if( T ) {}
   if ( T ) {         vs.    if( T ) {
      if ( T ) {}               if ( T ) {}
   }                         }
}
```

Fig. 5. On the left is the actual answer for a particular problem, whereas on the right is a participant's answer. If cascading errors were not taken into account, the participant would get two points docked for scoping errors, while with the cascading errors correction they would be docked one.

The third and fourth metrics are *Missing* `if` *statements* and *missing loops*. We count missing `if` statements and loops by counting the number of I or L characters in a participant string that are missing compared to the correct answer. The fifth and final metric is flipped booleans. A flipped boolean is counted whenever a T is matched with an F (or vice versa) in the globally aligned strings.

While we have applied the artifact encoding technique specifically to the use of auditory cues and basic programming, the paradigm itself is highly extensible. For example, adding extra characters to the list of possible codes, and defining what those codes would mean (e.g., code in the object-oriented or functional paradigms) would be quite trivial, and could then potentially represent a great deal more information than we represented here in our proof-of-concept studies. Similarly, there is no theoretical reason why the same techniques could not be applied to purely visual representations, through the use of flash cards or other media represented in sequence to humans. Effectively, artifact encoding is a technique for automatically measuring a human interpreted stimuli with a correct answer—the technique could be leveraged in a broad brush of potential experiments.

In sum, in the artifact encoding approach, the researcher runs an empirical study in which participants write down their interpretations of stimuli (e.g., auditory cues of computer programs). Participants' interpretations are then coded into strings. Using a global alignment algorithm, one compares those strings against the "correct strings" that encode the actual behavior of the computer code. Once strings are globally aligned, a series of comprehension metrics can be defined and automatically computed. In the studies presented in this article, we use this approach to test sighted users with no training in auditory code comprehension; however, the approach can be readily adapted for blind and visually impaired participants (e.g., Braille and/or screen readers for the non-sighted, large print for the visually impaired).

4. Experiment 1: auditory scoping cues

In order to validate the ability of artifact encoding to determine the relative effectiveness of auditory cues in a programming comprehension task, we ran an experiment with the following hypotheses: (H1) Auditory scoping cues significantly impact the ability of a participant to understand scoping relationships in a computer program, and (H2) Programming experience influences the comprehension of auditory cues. While there were any number of comprehension issues on which we might have focused in this study, we decided to focus on the issue of scoping because it is one of the most common issues identified by our blind and visually impaired programmer colleagues.

A scoping cue indicates relationships between the program constructs. There are many choices as to how one can represent scope in audio; for the executing block of code in Fig. 1, our cue is, "if true," followed by, "1 nested if false." The number 1 indicates the lexical scoping depth of the construct (e.g., it is inside the lexical block of another construct). One might immediately ask, "Why not indicate the end of the `if` statement with an auditory cue?" If the beginning and ending of a program construct are sonified without a scoping cue (a type of cue we call a *meta-auditory cue*), this would require the user to browse up and down a source file looking for the beginning and ending of a construct.

To test our hypotheses, we conducted a between-subjects experiment with two independent variables. Our first independent variable (scoping) had two levels: either with or without scoping auditory cues. The auditory cues heard in both conditions were identical except for the added scoping cue. Our second independent variable (experience) also had two levels. Specifically, some participants were enrolled in a programming course with no prerequisites, while others were enrolled in a course that required the successful completion of at least one previous programming course. The dependent variables used in this experiment were those metrics presented in the previous section. While our hypotheses were most concerned with the metric that indicates the number of scoping cues correctly placed at the appropriate scoping level, poor results for other comprehension metrics (e.g., flipped booleans, missing data) might also indicate that adding scoping cues has side effects.

4.1. Participants

Sixty-four participants were recruited from the computer science program at Central Washington University for this study. As an incentive, students were given approximately 3% extra credit toward their coursework. All participants could hear and see normally. None of the participants had previous experience using audio to comprehend computer software.

Out of the 64 participants in the study, 45 were male and 19 were female. Participants ranged in age from 18 to 48 years old, mean 22.8. Participants were self-reported having from 0 to 10 years of computer programming experience, mean 1.9 years, and were recruited from every class in the computer science department. Participants ranged from non-major freshman taking Visual Basic to seniors in the software engineering and advanced algorithms courses. Specifically, 39 of the students were in their first programming course, while 25 of them had already completed at least one course on programming. Table 2 shows the number of participants in each cell of the 2 $\times$ 2 design.

4.2. Materials and tasks

Participants completed a series of 10 experimental tasks, their responses to which were recorded in a provided booklet. Auditory cues were output from stereo speakers that were placed at the approximate center of the room. The volume of the speakers was loud enough for all participants to hear, as indicated by participants' responses to an exit questionnaire. The volume was kept constant between experimental conditions.

The stimuli used in this study were a stream of speech-based audio that related to an executing computer program. The cues conveyed the behavior of the executing computer program either with or without scoping indicators. The computer programs consisted largely of conditional statements and loops at various levels of scoping. Care was taken to make the programs semantically neutral, meaning that the participants could not determine the purpose of the program from the auditory cues.

The sounds for the audio were created using a custom computer tool we developed for this purpose, a pre-cursor to our Sodbeans environment called the Sonified Omniscient Debugger. Speech was read by Microsoft's text to speech engine, using the voice Microsoft Anna—English (United States). In order to facilitate understanding of the audio-based stimuli, we carefully tested the speed at which the running computer program was read, ultimately choosing a speed corresponding to two clicks below the label "Normal" in Microsoft's text-to-speech properties box.

Each experimental task consisted of computer code that was designed randomly, with some constructs being nested, some not, and loops executing any number of times. Each task was unique and therefore had unique auditory cues. Fig. 6 shows the code used to generate the auditory cues for task number six. Fig. 7 shows an example of the auditory cues from this experiment.

4.3. Procedure

This study was conducted in a quiet room over the course of two days. Each experiment required approximately an hour and a half to conduct, making it impractical to run all 64 participants through the experiment individually. Thus, participants were assigned to blocks of time. Each day there were four blocks. On the second day, the times for each experimental group were reversed.

The study was conducted double blind. We paid a proctor, who neither knew the hypothesis of the study nor was a computer scientist, to run all sessions with

Table 2
This table shows the number of participants in each cell of the 2 $\times$ 2 design.

Programming classes	Scoping cues	No. of scoping cues
0	19	20
1 or more	13	12

participants. This proctor was carefully trained on how to run the experimental sessions, was given a script for how to interact with participants, and had standard answers for many common questions students might have. In order to prevent any minor differences in reading the instructions during the training or experimental tasks, we paid a voice actor to record the sessions onto a compact disc. As such, except for scripted greetings by our proctor, all groups had a near identical experience.

In order to prevent cheating, we placed cardboard spacers between all students. Students of any height would have had to stand up and look over the cardboard spacers in order to cheat. The study proctor did not report any incidents of cheating during the course of the experiment.

```
int i = 0;
int d = 4;
if( d != d ) {}

while(i < d) {
   if(1 == i || 2 == i) {
      if(2 == i) {}
   }
   else {
      if(0 == i) {}
   }

   if( i == 0 ) {}
   i = i + 1;
}

if( d == d ) {}
```

Fig. 6. The code used by the Sonified Omniscient Debugger to generate the auditory cues for task 6.

```
int main() {
   int i = 0, k = 0;
   while(i < 2) {//execution begins here
      k++;
      if(i == 0) {
         k++;
      }
      else {
         k--;
      }
      i++;
   }
   return 0;
}
```

Fig. 7. An example from the training portion of this study. The caption in the study states: "The sounds in this example are the same as those before, but in this case, an IF statement is nested within a WHILE loop. This loop executes twice. During the first iteration of the loop the IF statement evaluates to TRUE and during the second iteration it evaluates to FALSE. The final sound produced by this execution sequence is LOOP ITERATION 1, 1 NESTED IF TRUE, LOOP ITERATION 2, 1 NESTED IF FALSE, END LOOP."

4.4. Results

Once the data collection was complete, participants' answers were coded into strings. To ensure that this encoding process was reliable, we conducted a Kappa analysis (see e.g., Dewey, 1983 and Hubert, 1977). Two researchers first trained on data not used in the study, and then independently coded approximately 20% of the actual data. A Kappa statistic of 0.904 (raw agreement 92.07%) was found, indicating that our coding system was highly reliable. A single researcher thus proceeded to code the remaining data.

After the data was coded, string representations of participants' answers were automatically graded by our computer tool (see Section 3). Univariate ANOVAs were used for the statistical analysis. For all measures, the interaction effect between scoping cues and experience level was non-significant, and as such, these results will not be considered further. Table 3 summarizes the results.

Fig. 8 presents the percentage of program constructs, loops or `if` statements, placed at the correct scoping level, either with or without scoping cues. Fig. 9 shows the same metric, but with experience level on the x-axis.

The mean percentage of program constructs placed at the correct scoping level for the group that received scoping cues ($M=79.8$, $SD=13.1$), was significantly higher than that of the no scoping cues group ($M=59.2$, $SD=10.1$), $F(1,60)=54.474, p<0.001$. The effect size was quite

Table 3
A summary table of the experimental results ($* = \alpha < 0.05$, $** = \alpha < 0.01$). The scoping and comprehension measures are percentage correct measures with a maximum value of 100%, whereas the missing if, loop, and flipped booleans measures are counts of the participants mistakes.

Measure	**Scoping cues**			**Experience**		
	p-val	*Yes*	*No*	*p-val*	*Yes*	*No*
Scoping	0.001**	$M=79.8$ $SD=13.1$	$M=59.2$ $SD=10.1$	0.841	$M=69.0$ $SD=14.35$	$M=70.3$ $SD=17.5$
Comprehension	0.051	$M=62.53$ $SD=25$	$M=52.26$ $SD=16.8$	0.038*	$M=64.69$ $SD=20.9$	$M=52.72$ $SD=22.0$
Missing if	0.451	$M=25.13$ $SD=29.9$	$M=18.97$ $SD=31.5$	0.008**	$M=9.60$ $SD=10.5$	$M=30.03$ $SD=36.31$
Missing loop	0.102	$M=8.81$ $SD=9.7$	$M=5.16$ $SD=8.35$	0.005**	$M=3.12$ $SD=3.29$	$M=9.46$ $SD=10.8$
Flipped boolean	0.144	$M=7.6$ $SD=8.1$	$M=5.4$ $SD=4.7$	0.750	$M=6.2$ $SD=8.3$	$M=6.7$ $SD=5.4$

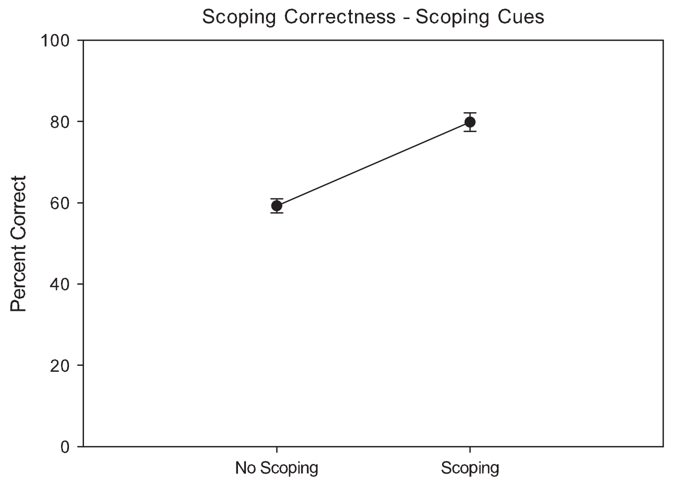


Fig. 8. This figure shows the percent of program constructs, loops or `if` statements, that were placed at the correct scoping level. The independent variable of scoping cues is on the *x*-axis, either without them (left), or with them (right).

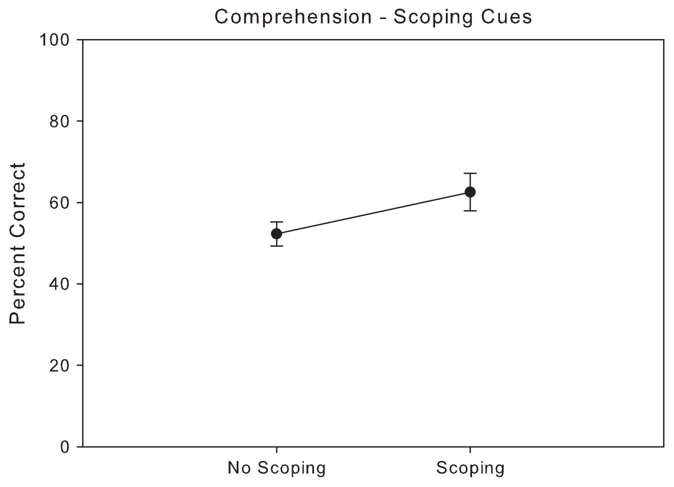


Fig. 10. This figure shows the percent correct for the aggregate artifact encoding score, called comprehension, on the *y*-axis. The independent variable of scoping cues is on the *x*-axis, either without them (left), or with them (right).

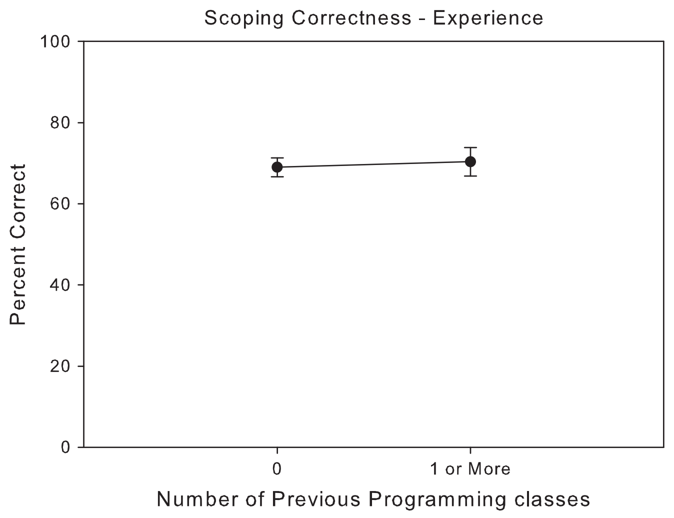


Fig. 9. This figure shows the percent of program constructs, loops or `if` statements, which were placed at the correct scoping level. The independent variable of experience is the *x*-axis, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

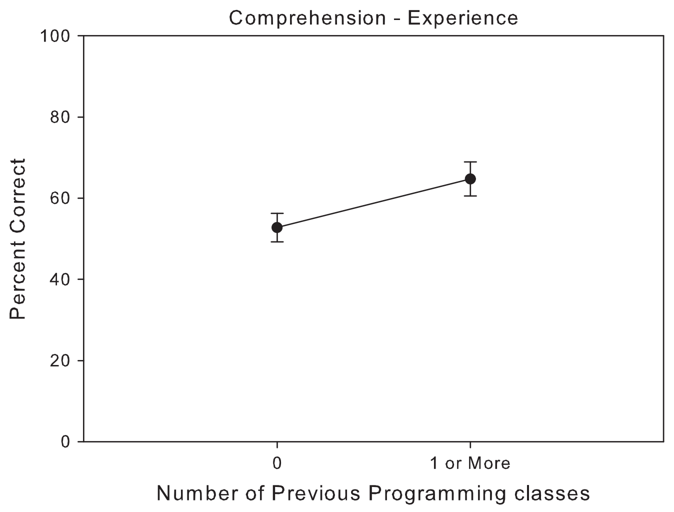


Fig. 11. This figure shows the percent correct for the aggregate artifact encoding score, comprehension, on the *y*-axis. The independent variable of experience is on the *x*-axis, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

large (partial eta-squared$=$0.476). However, the group that had no experience ($M=69.0$, $SD=14.35$) was not significantly different from the group with experience ($M=70.3$, $SD=17.5$), $F(1,60)=0.041$, $p=0.841$.

Fig. 10 shows the results of the total aggregate comprehension metric with scoping cues as the independent variable. Fig. 11 shows the same metric with experience on the *x*-axis. The difference between those who received scoping cues ($M=62.53$, $SD=25.8$) and those who did not ($M=52.26$, $SD=16.8$) approached significance, $F(1,60)=3.952$, $p=0.051$. The effect size of this value was modest (partial eta-squared$=$0.062). In contrast, the difference between those with experience ($M=64.69$, $SD=20.9$), and those without ($M=52.72$, $SD=22.0$) was significant $F(1,60)=4.52$, $p=0.038$. This effect was also modest in size (partial eta-squared$=$0.070). To give an idea of the size of the data set, the sum of the answers from all 10 tasks, if done perfectly, includes exactly 520 character codes. This means that after processing, to compute the percent correct for the scoping group, each individual would have had approximately 325.189 out of 520 correct answers, or 10,406 out of 16,640 overall. For the no scoping group, there was similarly 8696 of 16,640 potentially correct answers, giving a grand total for both sets summed of 19,102 of 33,280.

Two measures of missing data were extracted from the experiment (summed from all 10 tasks). Fig. 12 shows the effect the scoping cue variable had on missing loops and if statements. Those who received scoping cues ($M=8.81$, $SD=9.7$) had more raw missing loops than those who did not ($M=5.16$, $SD=8.35$); however, the difference was not significant, $F(1,60)=2.750$, $p=0.102$. Similarly, those who

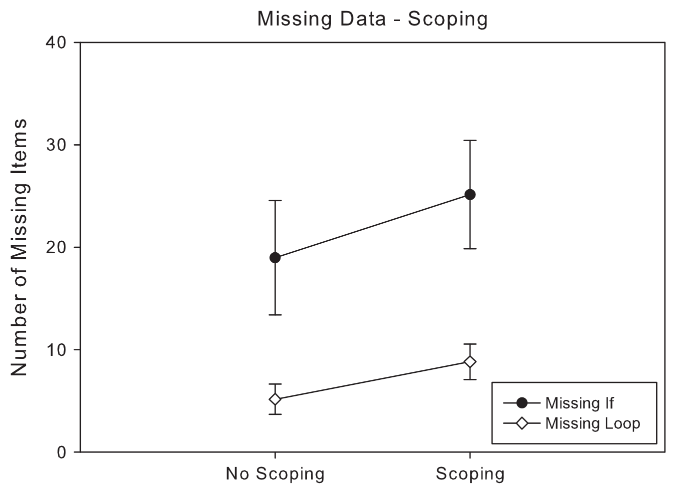


Fig. 12. This figure shows the number of missing loops and `if` statements, summed from all 10 tasks, on the *y*-axis. The independent variable of scoping cues is on the *x*-axis, either without them (left), or with them (right).

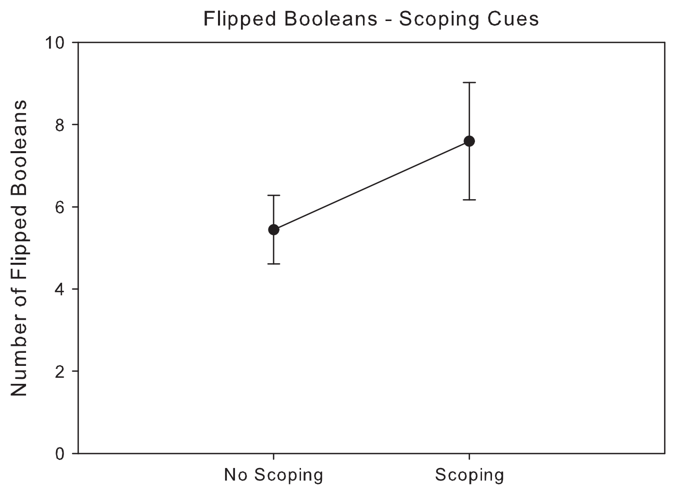


Fig. 14. This figure shows the number of flipped booleans in participants' answers. The independent variable of scoping cues is on the *x*-axis, either without them (left), or with them (right).

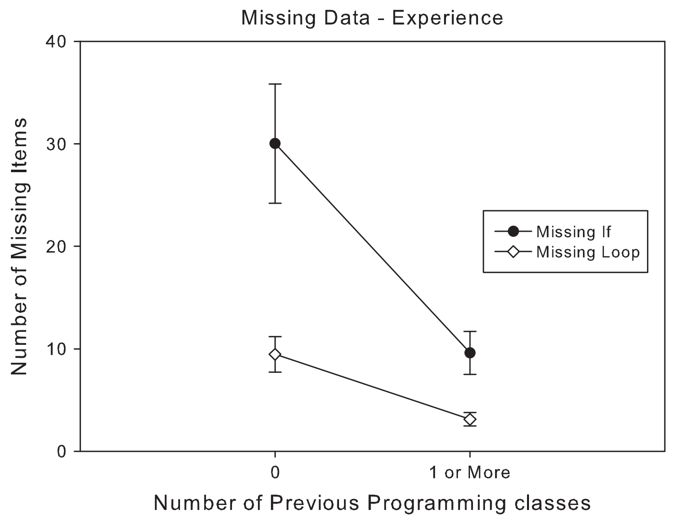


Fig. 13. This figure shows the number of missing loops and `if` statements, summed from all ten tasks, on the *y*-axis. The independent variable of experience is on the *x*-axis, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

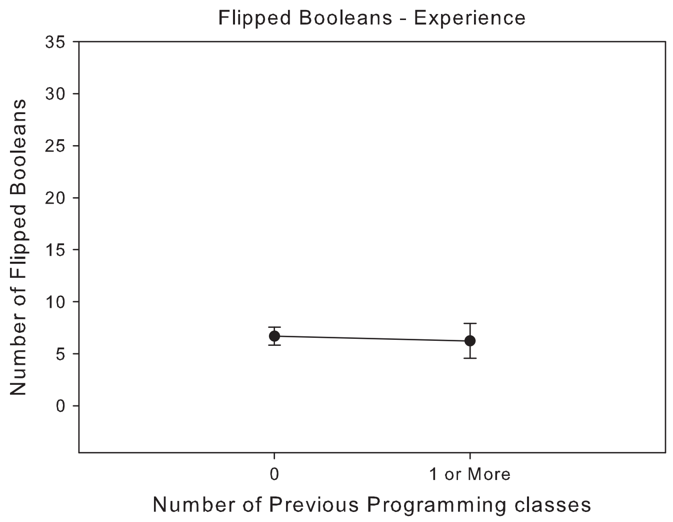


Fig. 15. This figure shows the number of flipped booleans in participants' answers. The independent variable of experience is on the *x*-axis, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

received scoping cues ($M=25.13$, $SD=29.9$) had more missing `if` statements than those who did not ($M=18.97$, $SD=31.5$). This result was also non-significant, $F(1,60)=0.574$, $p=0.451$.

Fig. 13 shows the effect with respect to the experience variable. Comparing those without experience ($M=9.46$, $SD=10.8$) to those with experience ($M=3.12$, $SD=3.29$), we found a significant difference between the two conditions with respect to the "missing loops" measure, $F(1,60)=8.521$, $p=0.005$. The same also held true for `if` statements, with the experienced group ($M=9.60$, $SD=10.5$) and the inexperienced group ($M=30.03$, $SD=36.31$) being significantly different $F(1,60)=7.486$, $p=0.008$. The effect size of both loops (partial eta-squared$=0.124$), and `if` statements (partial eta-squared$=0.111$) were moderate in size.

Fig. 14 shows the flipped boolean measure, with the scoping cues independent variable on the *x*-axis. Fig. 15 shows the same metric with the experience independent variable on the *x*-axis. The group that received scoping cues ($M=7.6$, $SD=8.1$), had more flipped booleans than the group that did not receive scoping cues ($M=5.4$, $SD=4.7$), although this result was not significant $F(1,60)=2.193$, $p=0.144$. Those participants with experience ($M=6.2$, $SD=8.3$) had fewer flipped booleans than those who did not ($M=6.7$, $SD=5.4$), but this effect was also non-significant, $F(1,60)=0.103$, $p=0.750$.

Some readers of complex statistical analysis prefer a Bon-Ferroni correction be applied when multiple tests are used, while others feel that such a correction is too conservative and provides little practical insight into the data. While we are neutral on the issue, when conducting only a small to moderate number of tests, note that most of our primary results are very strong and hold regardless of whether this correction is applied. Since this correction is a simple division, the exact details are left as an exercise to the reader, both for this study and the next.

4.5. Discussion

Our results show that scoping cues that state the nesting level explicitly (e.g., 1 nested if true) increase a participant's ability to understand the scope of a program construct. This effect was shown to be invariant of experience level.

Moreover, no negative side effects could be detected by adding scoping cues. While the group that received scoping cues did have higher rates of missing data (especially loops) and a higher incidence of flipped booleans, these differences were non-significant, and appeared to be a result of random error. However, while those who received scoping cues made fewer scoping errors, even with these cues, they were able to obtain only about 80% accuracy. This suggests that other auditory cues may be more effective.

Interestingly, scoping cues did not universally offer better aggregate comprehension, as seen in the list of tasks in Fig. 16. Notice that in some cases, the two groups were approximately equal, and in the case of tasks 1 and 8, the no scoping cues group actually outperformed the scoping cues group, although not significantly so. Similarly, Fig. 17 shows the effect of scoping cues on each task.

While participants made fewer scoping errors for all tasks in this experiment, qualitative inspection of participants' data suggests reasons for task differences. For example, it is easy to determine the scoping relationship between any individual `if` statement if it is the first `if` statement inside a loop. The auditory cue for a loop, even if it only executes one time, was, "Loop iteration *x*." The cue "End loop" indicates when the conditional inside the loop has evaluated to false. This implies that for certain

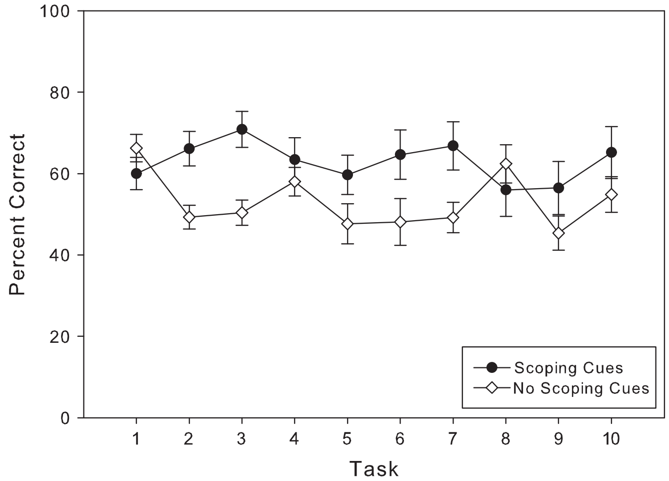


Fig. 16. This figure shows the percent correct for the comprehension metric for every task.

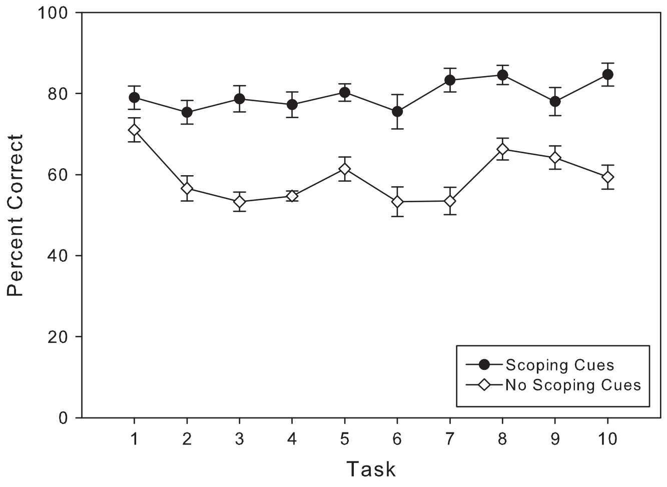


Fig. 17. This figure shows the percent correct for scoping in each task.

loops, with certain structures, one can determine the scope of the construct without needing a scoping cue.

This works for loops because, in a debugger, the loop eventually has to evaluate to false, unless it is an infinite loop. But `if` statements do not repeat, making the end of the construct difficult to determine unless the closing brace of a statement is sonified. However, data from the experiment suggests that participants were able to determine the scoping level, even without the closing brace sonified, approximately 80% of the time. This means that the current scoping cues effectively allow the user to identify scoping information without having to browse up and down the source code in search of a closing brace.

Finally, we note that the participants in this study completed what were essentially note taking tasks—something that programmers would not typically do. In order to explore whether these results might hold up in more realistic tasks, we first had to create an interactive program execution environment capable of outputting our auditory cues. In the next section, we present that environment, which was used as a basis for performing the follow-up experimental study presented in Section 6.

5. The sonified omniscient debugger

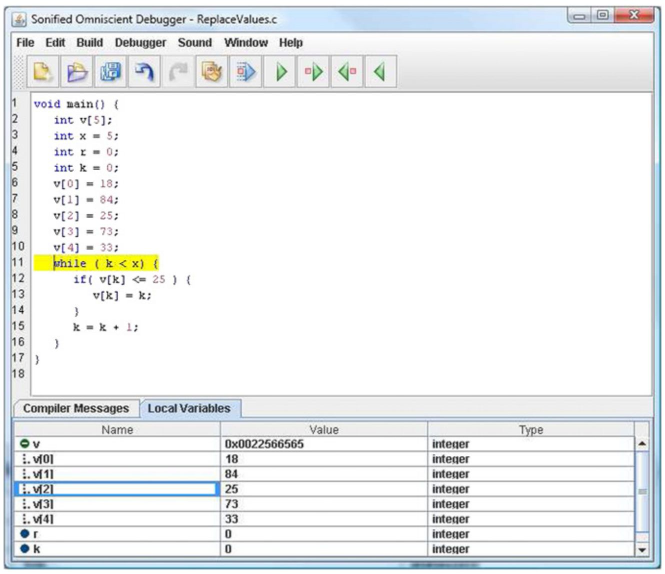
In order to test the effectiveness of auditory cues derived through artifact encoding studies like the one presented in the previous section in realistic program comprehension tasks, we developed a custom program execution environment called the Sonified Omnsicient Debugger (SOD). In addition to the scoping cues considered in the previous section, we conducted informal pilot studies on an extensive list of auditory cues (e.g., musical cues, different word choices, prosodic elements) related to several different program constructs (e.g., loops, `if` statements, variable assignment, expressions, scoping). While we were ultimately unable to test every auditory cue that went into SOD, we incorporated all of the cues that we were able to test empirically, and made reasonable choices for the rest.

For a thorough taxonomy of the auditory cues used in SOD, and significantly more details on their design, see Stefik (2008).

As the name implies, SOD is a sonified (i.e., narrated) and omniscient program execution environment. SOD allows for any possible runtime or static attribute of a computer program to be presented to the user via auditory cues. SOD is, if the reader will excuse the standard term in the literature, omniscient (Ducassé, 1999; Lewis and Ducassé, 2003; Pothier and Tanter, 2009) because, as SOD executes a program, it stores a record of computation and all changes to the state of the program. As a result, SOD is able to execute programs in reverse, and can provide users with information on program execution history.

In order to provide a feel for how the SOD environment works, we now walk through a sample session in which we use the SOD environment to complete a portion of one of the tasks performed by participants in our second experimental study (see Section 6). The task is to answer a comprehension question about the "Replace Values" algorithm, which replaces values in an array that are less than 25 with zero. For comparison purposes, we describe both SOD's auditory cues, and the state-of-the-art auditory cues that are generated by the JAWS 9 screen reader (plus scripts) coupled with Microsoft Visual Studio 2005, which was one of the experimental conditions in the study presented in the next section. We remind the reader that newer versions of Visual Studio (2010) and JAWS (11) are largely unchanged in the auditory cues they present.

Fig. 18 presents a visual snapshot of the SOD environment. The C source code for the same buggy "Replace Values" algorithm used in our second experiment is loaded into the environment. In this example, we consider how a user might answer the comprehension question, "How many items have been replaced after the third iteration of


Sonified Omniscient Debugger - ReplaceValues.c
File Edit Build Debugger Sound Window Help
Compiler Messages
Local Variables

```
1  void main() {
2     int v[5];
3     int x = 5;
4     int r = 0;
5     int k = 0;
6     v[0] = 18;
7     v[1] = 84;
8     v[2] = 25;
9     v[3] = 73;
10    v[4] = 33;
11    while ( k < x) {
12       if( v[k] <= 25 ) {
13          v[k] = k;
14       }
15       k = k + 1;
16    }
17 }
18
```

Name	Value	Type
v	0x0022566565	integer
v[0]	18	integer
v[1]	84	integer
v[2]	25	integer
v[3]	73	integer
v[4]	33	integer
r	0	integer
k	0	integer

Fig. 18. Snapshot of SOD interface while performing "replace values" comprehension and debugging tasks.

the loop?" The user could answer this question using three different methods. Illustrating these three methods will reveal SOD's repertoire of auditory cues.

In the first method, which is the most difficult, the user would first place the source code into working memory, and then mentally simulate the algorithm to arrive at an answer. To do this, the user would listen to the source code by pressing the up and down arrow keys, which cause the environment to play cues corresponding to static information on the line being visited. Whereas pressing the arrow keys produces semantic cues in SOD, it produces syntactic cues in Visual Studio+JAWS. For example, when visiting the statement `m[3]=14;` a SOD user would hear "m sub 3 equals 14." In contrast, a Visual Studio+JAWS user would hear "m left bracket three right bracket equals fourteen semicolon."

The second method the user could employ would be (a) to execute the loop three times by repeatedly pressing the F9 key to forward execute each line of code, and then (b) to count the number of times the assignment statement that replaces array values (line 13 in Fig. 18) is executed. To illustrate how this would be done, let us assume that the user is positioned at the `while` statement highlighted in Fig. 18. From there, the user would need to press F9 four times in order to navigate through one complete loop iteration.

On the first F9 press, the user would hear "loop iteration one" (line 11). On the second F9 press, the user would hear "one nested if true" (line 12). This cue conveys two pieces of information: (a) that the `if` statement is nested inside the `while` loop, and (b) that the `if` statement evaluated to true, since the value at `v[0]` is, in fact, less than or equal to 25 (as can be seen in the Locals Window). By pressing the F9 key a third time, the user would hear the cue "set array v sub 0 to 0" (line 13), indicating that an array element had been assigned a value. An alert user would also recognize this statement as a bug, since the value of `k` will not always be 0, as required. The cues we have used in other studies are slightly different (Stefik and Gellenbeck, 2009).

Hitting F9 a fourth time, the user would hear the loop increment statement on line 15: "set variable k to one." Finally, on the fifth press of F9, the user would begin the second iteration of the loop ("loop iteration two"), at which point the entire process would start over.

Contrast the above set of auditory cues to those generated by Visual Studio+JAWS. Each time F9 is pressed, the Visual Studio+JAWS user would hear "F9." In order to hear information regarding the line that was navigated to, the user would have to press the up arrow key followed by the down arrow key. For example, if the user pressed F9 to execute the `if` statement on line 12, and then hit the up and down arrow keys, the user would hear the cue "graphic 53 if left paren v left bracket k right bracket less than equals 25 right paren left brace."

What does "graphic 53" mean in the preceding auditory cue? It turns out that when Visual Studio+JAWS reads the line that is about to be executed, it literally translates the execution arrow as "graphic 53." In contrast, when a user navigates to the line that is about to be executed in SOD, the user hears "cursor at the execution line" followed by a semantic depiction of the line itself (e.g., "if true").

A third method for answering the question "How many items have been replaced after the third iteration of the loop" is identical to the second method just described, except that the user would use the Locals Window (bottom of Fig. 18) to inspect the contents of the array after completing three loop iterations, rather than keeping track of the number of times that the set statement on line 13 executes.

In both the SOD and Visual Studio+JAWS interfaces, the user can navigate to the Locals Window by pressing ALT+F3, at which point both SOD and Visual Studio+JAWS say "locals." Once in the Locals Window, users of both interfaces can navigate from variable to variable using the up and down arrows. When a variable is visited, the variable name, value, and type are read in sequence. For example, suppose the user hits the down arrow to visit variable r (second from the bottom in Locals Window of Fig. 18). In both interfaces, the user would hear "r zero integer."

The auditory cues for array variables differ slightly between SOD and Visual Studio+JAWS. To illustrate this, suppose that the user navigates to the array variable v (top-most variable in Locals Window of Fig. 18). In both interfaces, the user would hear the following: "v zero x zero zero one zero zero zero nine five nine zero integer" (the long sequence in the middle is the memory address of the array). In order to see/hear the individual array values of v, the user would need to open the array by hitting the right arrow key. When the user does this in the SOD interface, the user hears "opening array v". In contrast, hitting the right arrow key to open an array in Visual Studio+JAWS does not generate an auditory cue. Once the array is opened, the user can navigate to individual array values with the up and down arrows. The auditory cues produced for array values are identical to those produced for regular variables, with one notable exception: SOD says an array variable using "sub" language (e.g., "v sub zero"), whereas Visual Studio+JAWS says an array variable by speaking it literally (e.g., "v left bracket zero right bracket").

While the experiments presented in this work were conducted on the auditory environment SOD, our current implementation efforts are focused on Sodbeans 2.0, an open-source version of SOD that was integrated into the NetBeans Platform. To contrast, SOD included an auditory omniscient debugger, a talking text editor, and could connect to text-to-speech through Microsoft SAPI (a speech library). Sodbeans, however, can connect through SAPI or multiple screen readers, is cross-platform, and has additional auditory features (e.g., auditory brace matching, auditory runtime exceptions, auditory code completion, auditory editor annotations).

6. Experiment 2: experimental evaluation of SOD

In order to evaluate the effectiveness of our auditory cues (including the scoping cues tested in the study of Section 4) embodied in the SOD environment, we conducted a second experimental study with the following hypothesis: (H3) In program comprehension and debugging tasks, SOD's auditory cues will promote significantly faster and more accurate performance than the auditory cues generated by Visual Studio+JAWS+JAWS scripts; however, a visual version of the environment will promote significantly more efficient performance than both SOD and Visual Studio+JAWS. Obviously, such a comparison would be impossible to conduct with blind individuals, as they would be unable to use a visual environment. As such, the goal of this study was to measure how effectively sighted individuals with no experience could use the two auditory environments, and to compare their performance with the auditory environments against sighted performance.

To test this hypothesis, we conducted a within-subjects experiment with three conditions defined by task environment:

1. *SOD*: An audio-only version of the environment presented in the previous section embedded with the SOD auditory cues;
2. *JAWS*: An audio-only version of the environment presented in the previous section embedded with the auditory cues produced by JAWS 9 in Visual Studio 2005 (effectively the same as the state-of-the-art); and
3. *Visual*: The visual environment presented in the previous section with no auditory cues. This can be considered the "gold standard" that establishes an upper bound on human performance.

Comprehension and debugging outcomes were assessed according to three dependent measures: (a) accuracy on six comprehension questions; (b) ability to locate, describe, and explain how to fix two bugs; and (c) time to complete the comprehension questions and debugging tasks.

6.1. Participants

We recruited 19 students (13 male, 6 female; mean age 22.9) out of the Spring, 2008 offering of CptS 443/580, the undergraduate/graduate human computer interaction course at Washington State University. Participants were either juniors, seniors, or graduate students, and reported a mean of 5.47 years of prior programming experience. One participant, despite being told otherwise, thought aloud, which is known to be a confound for experiments involving audio interfaces (Tsujimura and Yamada, 2007). Another participant failed to complete the tasks, and was a significant outlier. These two participants were removed from the statistical analysis.

6.2. Materials and tasks

All participants worked on a computer running the Windows XP operating system. While the machine was equipped with a mouse and keyboard, participants were allowed to use only the keyboard. In the Visual condition, a 19″ LCD color display was set to a resolution of 1024 × 768. In the audio conditions, the computer display was turned off, and the Microsoft Mary—English (United States) voice was used in the text-to-speech speech engine. No user could use the mouse during the experiment.

Prior to working on the tasks in each condition, participants completed an informationally equivalent training task that introduced them to the environment version they would be using by having them navigate the code structure, execute the code in the debugger, and use the local variable window.

In each condition, participants worked with one of three algorithms of approximately equivalent complexity: Find Max (locates the largest element in an array), Replace (replaces values smaller than 25 with 0), and Count (sums the number of array values larger than 50). The task in each condition was two-fold: (a) answer six comprehension questions related to the runtime behavior of the algorithm (e.g., "How many times does the loop execute?); and (b) locate, describe, and explain how to correct two bugs seeded in the algorithm. Participants were provided with a work booklet containing instructions for all tasks, and spaces to enter their answers to comprehension and debugging questions.

We used Morae ® Recorder to make lossless recordings of participants' computer screens (which the participants themselves could not see in the SOD and JAWS conditions). An overhead camera, focused on participants' work booklets, captured their work in a smaller inset image. These recordings allowed us to accurately gauge participants' time on task.

6.3. Procedure

In order to guard against task order effects and any possible asymmetries between the tasks, we counterbalanced the task and treatment orders using a standard Latin-square design. This gave nine possible orderings: three different task orderings crossed with three different treatment orderings. Thus, roughly two study participants performed each of the nine possible task-treatment orderings.

We ran participants through the experiment individually over the course of a 10 day period. In each study session, which lasted roughly 90 min, participants began by filling out an informed consent form. Next, they were read a general description of what they would be doing in the study. Before performing tasks within a given treatment, participants completed a 10-min training task for that treatment. Participants were then instructed to complete the tasks in a given treatment as quickly as possible, without sacrificing accuracy, with the stipulation that all of the tasks in a given treatment had to be completed within 20 min. After 20 min, or whenever they finished, participants moved on to the tasks in the next treatment. After finishing the tasks in all three treatments, participants were given 10 min to fill out an exit questionnaire.

6.4. Results

Before analyzing our data, we first verified that there were no order or task effects. An analysis of variance (ANOVA) found no significant differences in (a) time for the total order, $F(7,43) = 0.927$, $p = 0.495$; (b) performance in the three tasks (Find Max, Count, Replace) $F(2,48) = 0.345$, $p = 0.710$; and performance with respect to task sequence, $F(2,48) = 0.026$, $p = 0.974$. We conducted the same analysis on the comprehension accuracy and debugging accuracy metrics, but similarly found non-significant order and task effects. We thus concluded that our tasks were reasonably isomorphic and that our Latin square successfully removed order effects from our design. In addition, using normal probability plots, we confirmed that our accuracy and time-on-task data were normally distributed.

Figs. 19 and 20 plot time on task, comprehension accuracy, and debugging accuracy by condition. These figures indicate that participants in the Visual condition promoted better performance than the other two conditions with respect to all three dependent measures.

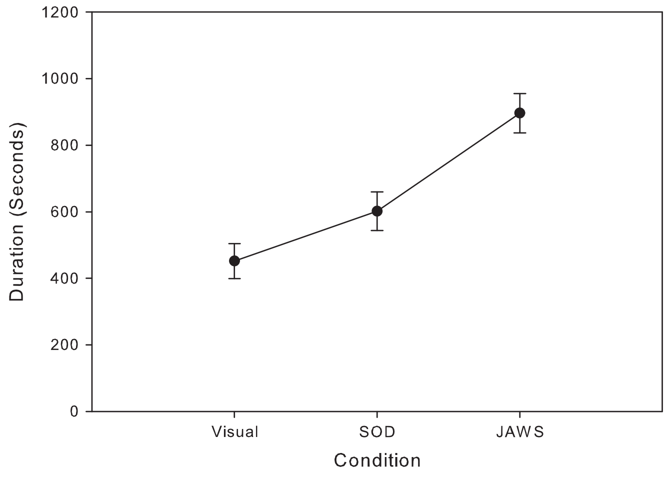


Fig. 19. Total time on task by condition.

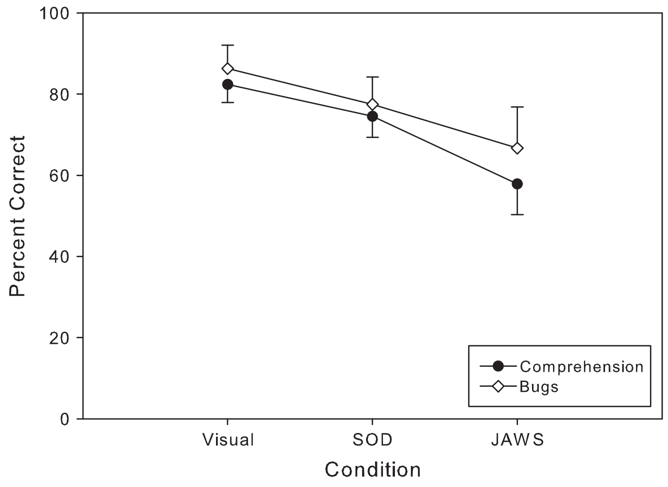


Fig. 20. Comprehension and debugging performance by condition.

Moreover, the SOD condition promoted better performance than the JAWS condition in all three measures.

To determine if any of these differences was statistically significant, we employed univariate ANOVAs. With respect to time on task, we found that the overall model was significant, $F(2,48)=15.925, p<0.001$ (partial eta-squared $=0.399$). Post hoc Tukey tests revealed that the difference between the Visual group ($M=452.1$ s, $SD=216.6$ s) and the JAWS group ($M=896.2$ s, $SD=244.3$ s) was significant, $p<0.001$, and that the difference between the JAWS group and the SOD group ($M=601.4$ s, $SD=238.8$ s) was significant, $p=0.002$. However, the difference between the Visual group and the SOD group was not significant, $p=0.160$.

With respect to comprehension accuracy, the ANOVA model was also significant, $F(2,48)=4.538, p=0.016$ (partial eta-squared $=0.159$). Post hoc Tukey tests revealed that the difference between the Visual group ($M=4.94$ out of 6, $SD=1.09$) and the JAWS group ($M=3.47$, $SD=1.87$) was significant, $p=0.013$. However, the difference between JAWS and SOD ($M=4.47$, $SD=1.28$) was not significant, $p=0.122$, nor was the difference between the Visual and SOD groups, $p=0.615$.

With respect to debugging accuracy, the overall ANOVA model was non-significant, $F(2,48)=1.606, p=0.211$ (partial eta-squared $=0.063$). Finally, we wanted to see if participants' subjective opinions about each environment accorded with their objective performances with each environment. Fig. 21 plots participants' responses to five exit questionnaire questions designed to elicit participants' subjective opinions on each environment. All responses were on a Likert scale ranging from 1 (strongly disagree with statement) to 7 (strongly agree with statement).

We used a MANOVA to test for significant differences in participants' questionnaire responses. Using Wilks' Lambda, we found that the entire model was significant,

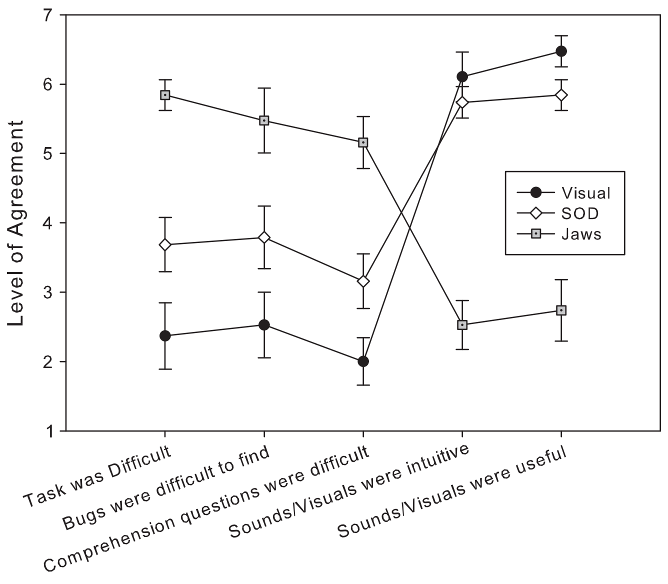


Fig. 21. Exit questionnaire results by condition.

$F(10,88)=12.801, p<0.001$ (partial eta-squared $=0.593$). According to post hoc Tukey tests, participants found that the SOD environment's auditory cues, as compared to the JAWS environment's auditory cues, (a) made the task less difficult, $p<0.001$, (b) made it easier to find bugs, $p<0.001$, (c) made it easier to answer comprehension questions, $p<0.001$, (d) were more intuitive, $p<0.001$, and (e) gave more useful information, $p<0.001$. While the same differences were found between the Visual and JAWS environments, differences between the Visual and SOD environments were less pronounced. In particular, while participants found that tasks with the Visual environment were easier than with SOD, $p=0.038$, the Visual and SOD environments did not differ significantly with respect to any of the other questions.

6.5. Discussion

Our results provide strong empirical support for our hypothesis that the SOD auditory cues promote significantly faster performance than the JAWS auditory cues for sighted users with no experience in an auditory programming environment. Indeed, the JAWS group took 4.9 min longer to complete the comprehension and debugging tasks than the SOD group—a difference that is both statistically and practically significant.

We also predicted that the Visual environment would be a "gold standard," promoting significantly faster task performance than either of the two audio environments. We found a significant difference between the Visual and JAWS environments, with the JAWS group taking 98.3% longer to complete tasks; however, the 33% difference in task speed between the Visual and SOD environments was not found to be statistically reliable. We found this result surprising: even though participants had never before used an auditory program execution environment, their performance with SOD approached their performance with the visual environment within the statistical margin of error. This suggests that well-designed auditory cues can indeed make a large difference for sighted individuals who are using an auditory environment for the first time.

With respect to comprehension and debugging task performance, we found only one statistically significant difference between the Visual and JAWS groups. Especially in the debugging task, we suspect that the lack of differences had to do with a ceiling effect: There were only two bugs to find within computer programs totaling fewer than 20 lines each, and most participants succeeded in finding the bugs.

Why was it the case that our main hypothesis, that the SOD auditory cues hold a significant advantage over the JAWS auditory cues, was substantiated? It seems plausible that *design of the auditory cues*, *semantic priming* and *temporal masking* may have played at least some role:

Perhaps the most plausible explanation of the differences we have documented is in the design of the auditory cues themselves. For example, the JAWS cues tend to focus information at the syntactic level—indeed, they give the listener a literal representation of the source, but little context. On the other hand, the SOD cues only present the information most critical to an understanding of the source, but leave out extraneous detail while supplementing with higher level context information (e.g., scoping cues). This makes the cues shorter without sacrificing the information required to complete pertinent tasks. After reviewing our recordings of the experimental sessions, we find this explanation to be especially plausible. In the JAWS sessions, participants would often have to listen to the auditory cue for each line multiple times. In contrast, in sessions with the SOD environment, which removed these extra characters and made them available only by pressing the left and right arrows, we noticed that participants rarely had to listen to the same line multiple times.

Semantic priming (Whitney, 1998; McNamara, 2005) is a psychological theory that suggests that a participant's response to one stimulus (e.g., an auditory cue) can alter the speed at which the participant can mentally retrieve, through a process called spreading activation, another stimulus. Why would semantic priming cause the SOD auditory cues to have an advantage? Consider the auditory cues generated when the participants execute a line of code (by pressing the F9 key). In the JAWS environment, the auditory cue is "F9," which has no meaning in the context of the executing program. In contrast, in the SOD environment, the auditory cue provides information relevant to context, such as "if true." This information may prime the participant's mental model of both the executing program's behavior, as well as the participant's temporal location within that program.

Finally, temporal masking (Zhang and Formby, 2007) is the theory that a sound stimulus can have an effect on the audibility of another stimulus either before or after the original. This phenomenon *may* be related to the auditory cues generated when the user is navigating the code with the arrow keys. When auralizing a line of code, for example, the JAWS environment reads every character literally, including brackets, braces, semicolons, and parentheses. It is possible that this large influx of superfluous details masks the information most relevant to the user, such as whether the line has a loop or conditional in it. This explanation may be even more plausible with expert level screen reader users that are listening at high speed, although we did not test this hypothesis here. In short, human auditory processing is a highly complex system and we do not pretend to know the ratio by which each of our explanations mattered; we offer them largely as a hunch. If anything, the work presented here is just a beginning toward an explanation of which auditory cues can or should be used in an auditory system.

6.6. General discussion

The auditory cues presented by a state-of-the-art screen reader coupled with a state-of-the-art IDE provided little guidance to our sighted participants. In our own artifact encoding tests, we decided not to test the comprehension of the auditory cues provided by Visual Studio, largely because it is relatively easy to tell, a priori, that a cue such as "F9" or "graphic 53" would obtain a comprehension score at or near zero. As such, we spent our time analyzing cues we thought had potential merit, and chose to present an example of one experimental study of cues that are deceptively difficult to design: scoping cues.

We also underscore that the programs participants worked with in our second study were small—less than 20 lines of code. Thus, the results we obtained in that study may not generalize to larger programming, debugging, or comprehension tasks. It could be the case that our study failed to capture the real benefits of an auditory program execution environment like SOD within larger, real-world tasks. Alternatively, it could be the case that, while SOD is helpful for understanding the runtime behavior of programs, it has other problems that prevent users from using it in larger, real-world, tasks. Whatever the case, further empirical studies involving larger programs and more complex tasks are clearly needed in order to draw more definitive conclusions.

And finally, we want to be clear that our testing in this work was conducted with sighted individuals who could not see the screen. Are such individuals an accurate representation of *all* blind individuals? Over the past five years, we have worked with a considerable number of blind individuals, including: (1) professional programmers who set their screen readers to very high speed and are very familiar with tools like JAWS (amongst other screen reading technologies), (2) a retired programmer using Mac OS X Voice Over at medium to slow speed, (3) children with no programming experience and no JAWS experience (e.g., Bigham et al. (2008) also worked with children in this category), and (4) a number of legally blind individuals that use screen magnification without a screen reader, including one member of our development team.

In other words, we are hard pressed to argue that our results here do, or do not, apply to the blind in some broad, population-level, sense. Indeed, if we have learned anything from our outreach work, it is that it is difficult to paint the community with a broad brush. We imagine that highly experienced blind programmers, with significant expertise in JAWS, would perform better than those in our samples did. It is unclear if differences for a specialized group like this would have benefited from our more comprehensible auditory cues. As for the benefits to blind children learning programming and a screen reader for the first time, there is no way to be sure from our work here, but we suspect that creating better and more easily understood auditory representations is likely a positive step forward and we hope that our empirical data and paradigm (artifact encoding) provide a *foundation* for studying the issue in more detail.

7. Summary and future work

This research makes three key contributions to the literature on auditory programming. First, it furnishes a rigorous empirical methodology for measuring the comprehension of computer programs, which we happened to apply to auditory processing—a methodology that can form a foundation for future research in the field. Second, it provides empirical evidence validating that the artifact encoding approach can detect differences in the human comprehension of auditory cues. And third, it presents empirical evidence that we can create auditory representations that are significantly better than the state-of-the-art (an IDE coupled with a screen reader), and that approach the effectiveness of visual representations within the statistical margin of error.

While the preliminary studies presented here focused on the performance of sighted individuals, an obvious next step for future research is to see if the results apply to non-sighted individuals, who have a greater stake in auditory representations of computer programs than sighted individuals. To that end, we recently ran a summer camp in which we performed a field test of the successor to SOD (Sodbeans) with 12 blind and visually impaired high school students. Sobeans includes the latest version of our narrated debugger, an in-place screen reader replacement (Sappy), and a custom computer programming language (Hop) designed specifically for this population through philosophically similar empirical studies. During this camp, we gathered significant data on how students interacted with Sodbeans in real programming tasks and we tested general usability issues (e.g., opening files using audio, creating projects, navigating windows). We are also designing a custom, 18-week, introductory programming curriculum, rooted in Sodbeans, that will be offered at several schools for the blind throughout the U.S. We look forward, in future publications, to reporting more detailed results regarding the effectiveness of tools designed using the methodology presented here in blind and visually impaired populations. Our hope is that these tools will ultimately empower the blind and visually impaired to write their own programming tools—tools that are *for* the blind and *by* the blind.

Acknowledgments

We would like to thank the contributors to the Sodbeans project for their help with creating our technology, especially Richard Most and Sina Bahram. We would also like to thank the Washington State School for the Blind for their help in brainstorming ideas on how to improve our tools and we appreciate the extraordinary technical expertise of Tim Boudreau (Previously with Oracle) and Tom Wheeler (A NetBeans Platform expert), who have helped make the NetBeans platform integration possible. And we extend our thanks to Central Washington University, especially Ed Gellenbeck, who helped us to prepare for our artifact encoding study. Finally, we wish to express gratitude to the National Science Foundation under award (CNS-0940521) and the school of Electrical Engineering and Computer Science at Washington State University for their generous support of this work.

References

Abu Doush, I., Pontelli, E., Simon, D., Son, T.C., Ma, O., 2009. Making Microsoft Excel: multimodal presentation of charts. In: Assets '09: Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 147–154.

Allman, T., Dhillon, R.K., Landau, M.A., Kurniawan, S.H., 2009. Rock vibe: Rock band ® computer games for people with no or limited vision. In: Assets '09: Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 51–58.

Asakawa, C., 2005. What's the web like if you can't see it?. In: W4A '05: Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A), ACM, New York, NY, USA, pp. 1–8.

Baecker, R., 1988. Enhancing program readability and comprehensibility with tools for program visualization. In: Proceedings of the 10th International Conference on Software Engineering. IEEE Computer Society Press, Singapore, pp. 356–366.

Begel, A., Graham, S., 2004. Spoken language support for software development. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE Computer Society Press, Los Alamitos, CA, pp. 271–272.

Berman, L.I., Gallagher, K.B., 2006. Listening to program slices. In: Proceedings of the 12th International Conference on Auditory Display, London, UK.

Bernareggi, C., Archambault, D., 2007. Mathematics on the web: emerging opportunities for visually impaired people. In: W4A '07: Proceedings of the 2007 International Cross-disciplinary Conference on Web Accessibility (W4A). ACM, New York, NY, USA, pp. 108–111.

Bigham, J.P., Aller, M.B., Brudvik, J.T., Leung, J.O., Yazzolino, L.A., Ladner, R.E., 2008. Inspiring blind high school students to pursue computer science with instant messaging chatbots. In: SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education. ACM, New York, NY, USA, pp. 449–453.

Boardman, D.B., Greene, G., Khandelwal, V., Mathur, A.P., 1995. Listen: a tool to investigate the use of sound for the analysis of program behavior. in: Computer Software and Applications Conference, COMPSAC 95 Proceedings. Nineteenth Annual International, Dallas, TX, pp. 184–189.

Bonar, J., Soloway, E., 1983. Uncovering principles of novice programming. In: POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM Press, New York, NY, USA, pp. 10–13.

Brown, M.H., Hershberger, J., 1991. Colour and sound in algorithm animation. In: Proceedings of the IEEE Workshop on Visual Languages. IEEE Computer Society Press, Los Alamitos, CA, pp. 10–17.

Brown, A., Pettifer, S., Stevens, R., 2004. Evaluation of a non-visual molecule browser. In: Assets '04: Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility. ACM Press, New York, NY, pp. 40–47.

Carlisle, M., Wilson, T., Humphries, J., Hadfield, S., 2005. RAPTOR: a visual programming environment for teaching algorithmic problem solving. In: Proceedings of the ACM SIGCSE 2005 Symposium. ACM Press, New York, pp. 176–180.

Cooper, S., Dann, W., Pausch, R., 2000. Alice: a 3-d tool for introductory programming concepts. In: CCSC '00: Proceedings of the Fifth Annual CCSC Northeastern Conference on the Journal of Computing in Small Colleges. Consortium for Computing Sciences in Colleges, USA, pp. 107–116.

Dewey, M.E., 1983. Coefficients of agreement. Br. J. Psychiatry 143, 487–489.

Dougherty, J.P., 2007. Concept visualization in cs0 using alice. J. Comput. Small Coll. 22, 145–152.

Ducassé, M., 1999. Coca: an automated debugger for C. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 504–513.

Ellis, B., Stylos, J., Myers, B., 2007. The factory pattern in api design: a usability evaluation. ICSE 2007. Twenty-ninth International Conference on Software Engineering, pp. 302–312.

Feldman, S.I., Brown, C.B., 1989. IGOR: a system for program debugging via reversible execution. SIGPLAN Not. 24, 112–123.

Finlayson, J.L., Mellish, C., 2005. The audioview – providing a glance at java source code. In: Proceedings ICAD'05, London, UK.

Finlayson, J., Mellish, C., Masthoff, J., 2007. Fisheye views of java source code: an updated LOD algorithm. In: Stephanidis, C. (Ed.), Universal Access in Human–Computer Interaction. Applications and Services, Lecture Notes in Computer Science, vol. 4556. Springer, Berlin/Heidelberg, pp. 289–298.

Folmer, E., Yuan, B., Carr, D., Sapre, M., 2009. Textsl: a command-based virtual world interface for the visually impaired. In: Assets '09: Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 59–66.

Francioni, J.M., Jackson, J.A., Albright, L., 1991a. The sounds of parallel programs. In: The Sixth Distributed Memory Computing Conference, pp. 570–577.

Francioni, J.M., Albright, L., Jackson, J.A., 1991b. Debugging parallel programs using sound. In: PADD '91: Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging. ACM Press, New York, NY, pp. 68–75.

Gaver, W.W., 1986. Auditory icons: using sound in computer interfaces. Human–Computer Interact. 2, 167–177.

Gellenbeck, E., Stefik, A., 1998. Evaluating prosodic cues as a means to disambiguate algebraic expressions: an empirical study. In: Assets '09: Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 139–146.

Green, T.R.G., Petre, M., 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. Visual Lang. Comput. 7, 131–174.

Green, T., Petre, M., Bellamy, R., 1991. Comprehensibility of visual and textual programs: a test of superlativism against the 'Match-Mismatch' conjecture. In: Empirical Studies of Programmers Fourth Workshop, Papers, pp. 121–146.

Gross, P., Powers, K., 2005. Evaluating assessments of novice programming environments. In: ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research. ACM Press, New York, NY, USA, pp. 99–110.

Guzdial, M., Soloway, E., 2002. Teaching the Nintendo generation to program. Commun. ACM 45, 17–21.

Hendrix, T., Cross, J., Barowski, L., 2004. An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. In: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education. ACM Press, New York, pp. 387–391.

Hubert, L., 1977. Kappa revisited. Psychol. Bull. 84, 289–297.

Hundhausen, C.D., Farley, S.F., Brown, J.L., 2009. Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study. ACM Trans. Computer–Human Interact. 16, 1–40.

Karshmer, A.I., Gupta, G., Pontelli, E., Miesenberger, K., Ammalai, N., Gopal, D., Batusic, M., Stöger, B., Palmer, B., Guo, H.-F., 2004. Uma: a system for universal mathematics accessibility. In: Assets '04: Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 55–62.

Karshmer, A., Gupta, G., Pontelli, E., 2007. Mathematics and accessibility: a survey. Technical Report, University of Texas at Dallas.

Kelleher, C., Pausch, R., 2005. Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. ACM Comput. Surv. 37, 83–137.

Ko, A.J., Myers, B.A., 2009. Finding causes of program output with the java whyline. In: CHI '09: Proceedings of the 27th international conference on Human factors in computing systems. ACM, New York, NY, USA, pp. 1569–1578.

LeDoux, C.H., Parker Jr., D.S., 1985. Saving traces for ada debugging. In: SIGAda '85: Proceedings of the 1985 Annual ACM SIGAda International Conference on Ada. Cambridge University Press, New York, NY, USA, pp. 97–108.

Levy, R.B., Ben-Ari, M., Uronen, P., 2003. The jeliot 2000 program animation system. Comput. Educ. 40, 1–15.

Lewis, B., Ducassé, M., 2003. Using events to debug java programs backwards in time. In: OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. ACM Press, New York, NY, pp. 96–97.

Mayrhauser, A.v., Vans, A.M., 1994. Comprehension processes during large scale maintenance. In: ICSE '94: Proceedings of the 16th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp. 39–48.

Mayrhauser, A.V., Vans, A.M., 1997. Program understanding behavior during debugging of large scale software. In: ESP '97: Papers Presented at the Seventh Workshop on Empirical Studies of Programmers. ACM Press, New York NY, pp. 157–179.

McIver, L., The effect of programming language on error rates of novice programmers, The 12th Annual Workshop of Psychology of Programmers Interest Group (PPIG), Cosenza, Italy.

McNamara, T.P., 2005. Semantic Priming: Perspectives from Memory and Word Recognition. Psychology Press, New York, Hove.

Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B., 1983. Program indentation and comprehensibility. Commun. ACM 26, 861–867.

Mullins, P., Whitfield, D., Conlon, M., 2009. Using alice 2.0 as a first language. J. Comput. Small Coll. 24, 136–143.

Myers, B., Pane, J., 1996. Usability issues in the design of novice programming systems. School of Computer Science Technical Report CMU-CS-96-132, Carnegie Mellon University.

Myers, B.A., Pane, J.F., Ko, A., 2004. Natural programming languages and environments. Commun. ACM 47, 47–52.

Myers, B.A., Ko, A.J., Park, S.Y., Stylos, J., LaToza, T.D., Beaton, J., 2008. More natural end-user software engineering. In: WEUSE '08: Proceedings of the 4th International Workshop on End-user Software Engineering. ACM, New York, NY, USA, pp. 30–34.

Needleman, S.B., Wunsch, C.D., 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. 48, 443–453.

Oman, P.W., Cook, C.R., 1990. Typographic style is more than cosmetic. Commun. ACM 33, 506–520.

Oogane, T., Asakawa, C., 1998. An interactive method for accessing tables in html. In: Assets '98: Proceedings of the Third International ACM Conference on Assistive Technologies. ACM, New York, NY, USA, pp. 126–128.

Palladino, D.K., Walker, B.N., 2007. Learning rates for auditory menus enhanced with spearcons versus earcons. In: Proceedings of the 13th International Conference on Auditory Display, Montréal, Canada, pp. 274–279.

Parente, P., 2004. Audio enriched links: web page previews for blind users. SIGACCESS Accessibility and Computing 77–78, 2–8.

Pausch, R., 2008. Alice: a dying man's passion. In: SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education. ACM, New York, NY, USA, pp. 1.

Pennington, N., 1987a. Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychol. 19, 295–341.

Pennington, N., 1987b. Comprehension strategies in programming. In: Olson, G.M., Sheppard, S., Soloway, E., Shneiderman, B. (Eds.), Empirical Studies of Programmers: Second Workshop. Greenwood Publishing Group Inc., Westport, CT, pp. 100–113.

Pothier, G., Tanter, E., 2009. Back to the future: Omniscient debugging. IEEE Softw. 26, 78–85.

Pothier, G., Tanter, E., Piquer, J., 2007. Scalable omniscient debugging. In: OOPSLA'07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. ACM, New York, NY, USA, pp. 535–552.

Rotard, M., Knödler, S., Ertl, T., 2005. A tactile web browser for the visually disabled. In: HYPERTEXT '05: Proceedings of the Sixteenth

ACM Conference on Hypertext and Hypermedia. ACM, New York, NY, USA, pp. 15–22.

Sánchez, J., Flores, H., 2004. Memory enhancement through audio. SIGACCESS Accessibility and Computing 77–78, 24–31.

Smith, A.C., Cook, J.S., Francioni, J.M., Hossain, A., Anwar, M., Rahman, M.F., 2004. Nonvisual tool for navigating hierarchical structures. In: The Tenth International ACM SIGACCESS Conference on Computers and Accessibility, 77–78. ACM Press, New York, NY, pp. 133–139.

Snodgrass, R., 1984. Monitoring in a software development environment: a relational approach. SIGPLAN Not. 19, 124–131.

Soiffer, N., 2007. Mathplayer v2.1: Web-based math accessibility. in: Assets '07: Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, pp. 257–258.

Soloway, E., Bonar, J., Ehrlich, K., 1983. Cognitive strategies and looping constructs: an empirical study. Commun. ACM 26, 853–860.

Stasko, J., Domingue, J., Brown, M., Price, B., Domingue, J., Brown, M., Price, B., 1998. Software visualization. MIT Press, Cambridge, MA.

Stefik, A., 2008. On the design of program execution environments for non-sighted computer programmers. Ph.D. Thesis, Washington State University.

Stefik, A., Gellenbeck, E., 2009. Using spoken text to aid debugging: an empirical study. In: ICPC '09: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09). IEEE Computer Society Press, Los Alamitos, CA, pp. 110–119.

Stefik, A., Alexander, R., Patterson, R., Brown, J., 2007. WAD: a feasibility study using the wicked audio debugger. In: ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE Computer Society Press, Los Alamitos, CA.

Stefik, A., Haywood, A., Mansoor, S., Dunda, B., Garcia, D., 2009. Sodbeans. In: ICPC '09: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09). IEEE Computer Society Press, Los Alamitos, CA.

Stefik, Andreas M., Hundhausen, Christopher, Smith, Derrick, 2011. On the design of an educational infrastructure for the blind and visually impaired in computer science. In: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, Dallas, TX, USA, ACM, New York, NY, USA, pp. 571–576, ISBN 978-1-4503-0500-6.

Stevens, R., 1996. Principles for the design of auditory interfaces to present complex information to blind people. Ph.D. Thesis, The University of York.

Takagi, H., Kawanaka, S., Kobayashi, M., Sato, D., Asakawa, D., 2009. Collaborative web accessibility improvement: challenges and possibilities. In: Assets '09: Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 195–202.

Tsujimura, S., Yamada, Y., 2007. A study on the degree of disturbance by meaningful and meaningless noise under the brain task. In: Nineteenth International Congress on Acoustics. Madrid, Spain.

Vickers, P., Alty, J.L., 2002. When bugs sing. Interacting Comput. 14, 793–819.

Vickers, P., Alty, J.L., 2005. Musical program auralization: empirical studies. ACM Trans. Appl. Percept. 2, 477–489.

Walker, B.N., Kramer, G., 2005. Mappings and metaphors in auditory displays: an experimental assessment. ACM Trans. Appl. Percept. 2, 407–412.

Whitley, K., 1997. Visual programming languages and the empirical evidence for and against. J. Visual Lang. Comput. 8, 109–142.

Whitney, P., 1998. The Psychology of Language. Houghton Mifflin Company, Boston, MA.

Yesilada, Y., Stevens, R., Goble, C., Hussein, S., 2004. Rendering tables in audio: the interaction of structure and reading styles. SIGACCESS Accessibility and Computing 77–78, 16–23.

Yuan, B., Folmer, E., 2008. Blind hero: enabling guitar hero for the visually impaired. In: Assets '08: Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, New York, NY, USA, pp. 169–176.

Zhang, T., Formby, C., 2007. Effects of cueing in auditory temporal masking. J. Speech Lang. Hear. Res. 50, 564–575.