INTERNATIONAL JOURNAL OF SPEECH TECHNOLOGY 4, 103–116, 2001

VoiceGrip: A Tool for Programming-by-Voice*

A. DESILETS
National Research Council of Canada, Bldg M-5, Montreal Road, Ottawa, ONT K1A 0R6
alain.desilets@nrc.ca

Received August 11, 2000; Revised February 13, 2001

Abstract. In recent years, there has been an increase in the number of computer programmers suffering from Repetitive Strain Injury (RSI)—an umbrella term covering a series of musculoskeletal disorders caused by repetitive motion of the hands and arms. For those individuals, or any programmer with a handicap that precludes keyboard and/or mouse input, Speech Recognition (SR) is an attractive alternative because it could allow them to do their work without using such devices. Unfortunately, programming-by-voice with current SR systems is awkward because programming languages are not meant to be spoken. In this paper we describe various usability problems with programming-by-voice and show that none of the existing programming-by-voice tools address all of those barriers. We then present VoiceGrip, a programming-by-voice tool that adresses the widest range of programming-by-voice problems to date. VoiceGrip uses a unique approach where programmers first dictate code using an easy to utter pseudo-syntax, and then translate that automatically to native code in the appropriate programming language. The system has been downloaded by 343 individuals, and postings on a neutral programming-by-voice mailing list indicate that it is being used by at least some of them. We also present an experiment evaluating the performance of the system's symbol translation algorithm. In this experiment, the system exhibited low error rates in the range of 2.7% when confusion between homophonic symbols (i.e. symbols that have the same spoken pseudo code form) was ignored and 6.6% when confusion between homophonic symbols was taken into account. Finally, even though VoiceGrip is the tool that currently addresses the widest range of programming-by-voice problems, we conclude that a better tool can be developed by combining features of VoiceGrip with features of other existing programming-by-voice tools.

Keywords: programming-by-voice, speech recognition, HCI, programming environments, assistive technologies

1. Introduction

In recent years, there has been an increase in the number of computer users suffering from Repetitive Strain Injury (RSI)—an umbrella term covering a series of musculoskeletal disorders caused by repetitive motion of the hands and arms. Many of the individuals affected cannot use a mouse or keyboard without experiencing severe pain (Pascarelli and Quilter, 1994). The US Bureau of Labor Statistics reported that 64% of worker compensation claims in 1994 were attributable to RSI—a 10% increase from the same figure in 1993 (US Bureau of Labor Statistics, 1994). This study did not specify the number of RSI cases that were related to computer use, however an Australian study found that 10% to 50% of women doing keyboard work suffer from RSI at some degree (Kiesler and Finholt, 1994).

Although incidence of RSI in computer programmers has not been specifically investigated, anecdotal evidence indicates that they are also widely affected. For programmers with RSI, Speech Recognition (SR) is an attractive alternative to mouse and keyboard, because it could allow them to do their work without exerting their injured limbs. Unfortunately, programming

*This work was supported by the National Research Council of Canada.

using current SR technology is at best cumbersome, because programming languages are not meant to be spoken. Pervasive punctuation and abbreviations make dictation of programming code very difficult.

For example, suppose you want to dictate a simple C statement such as:

```
if (currRecNum < maxOffSet)
{
```

Using a typical continuous SR system, you might have to say:

> *"if open-paren charlie uniform romeo romeo cap romeo echo charlie cap november uniform mike less-than max begin-capitalize begin-no-space off set end-capitalize end-no-space close-paren new-line open-brace"*

Programming-by-voice in this way is not practical because it involves large vocal and cognitive load. By *vocal load*, we mean the stress caused to the voice apparatus by uttering the code. By high *cognitive load*, we mean that the user has to think too hard about how to say things, which distracts her from the programming task (an activity which is cognitively demanding in its own right). The issue of vocal load is important because it can lead to voice strain, an injury similar to RSI that affects the vocal tract muscles and can cause total and long-term voice loss. This is particularly dangerous for programmers suffering from RSI, since a recent study (Kambeyanda et al., 1997) showed that people with a history of RSI are particularly susceptible to SR induced voice strain. High vocal load can occur if the programmer is required to say too many utterances per line of code. It can also occur if the programmer is required to pause frequently between utterances, because abrupt stops in the speech flow are detrimental to the vocal chords.

In this paper we present VoiceGrip, a programming tool designed specifically for voice input. The system allows programmers to dictate code using a pseudo code syntax that is easier to utter than the native code itself. Later, these pseudo code statements can be automatically translated in real time to native code in the appropriate programming language. For example, to type the above C statement, you can simply say:

> *"if current record number less than max off set then"*

pausing wherever it feels natural to do so. At first, the statement would get typed literally in pseudo code, but you could later utter a single voice command to automatically translate it to the appropriate native C code. This approach to code dictation imposes a lower vocal load on the user because it is much shorter to utter. It also imposes a lower cognitive load because the user does not have to struggle to dictate the exotic punctuation and symbols which are present in the native code.

Note that while this approach is aimed at solving problems encountered in programming-by-voice, it is directly applicable to any editing-by-voice task where the spoken form of the text differs significantly from its written form. Examples of such tasks outside of the programming realm are voice-editing of HTML pages and scientific formulas. Also, while the VoiceGrip system was originally designed to help programmers with RSI, it can be useful for any programmer with a handicap that precludes use of the keyboard and/or mouse (but not use of the voice). Finally, while the approach was originally developed to facilitate voice input of programming code, it could be modified to support reading of the code to visually impaired programmers. In this case, the opposite translation would be carried out, that is the native code would be translated to a more ear-friendly pseudo code form before being read by a Text To Speech (TTS) system. This is in fact the approach taken by Raman et. al with their ASTER system (Raman et al., 1995).

The paper is organized as follows. Section 2 presents problems with programming-by-voice using current SR technology. We focus particularly on code editing problems and argue that they are central to the problems encountered in most other programming-by-voice activities. Section 3 reviews various programming-by-voice tools proposed to date and situates them in terms of what specific code editing problems they address. Section 4 presents the VoiceGrip system. We describe the system's overall approach and discuss how it addresses most of the code editing problems described in Section 2. In addition, Section 4 contains details of the algorithm for translating from pseudo code to native code. Section 5 discusses anecdotal user feedback on the tool. It also presents results of an experiment evaluating the performance of the system's symbol translator. Finally, Section 6 presents conclusions and directions for future research.

2. Programming-by-Voice Problems

All tools that have been developed to date for programming-by-voice focus on code editing. There is a good reason for that. Although programming-by-voice involves more than just code editing, the usability problems encountered in editing code by voice also happen to be the central usability problems when performing other programming activities by voice. In this section, we describe what those code editing problems are and then explain why they are central to other programing-by-voice activities.

2.1. Code Editing Problems

Below is a list of problems with editing code using current SR technology.

- code dictation
  - punctuation
  - symbols
- code navigation
  - global navigation
  - local navigation
- error correction
- mouse-free operation

This list was generated through ad-hoc exchanges with experienced programmers-by-voice, and by observing what problems these programmers-by-voice have tried to solve with their home grown SR and editor macros for code editing.

The first problem, *code dictation*, has to do with dictation of new code, or modifications of existing code. This problem is illustrated by the sample utterance given in Section 1. The usability problems for *code dictation* are similar to those which have already been recognized for Text To Speech (TTS) rendering of technical material such as scientific formulas (Raman et al., 1995). Like programming code, this kind of material is not meant to be spoken and is therefore hard to understand when read exactly as written. Therefore, specialized text readers such as ASTER and EmacsSpeak (Raman et al., 1995) have been developed to translate the native material to a more ear-friendly form. Note that in the context of programming-by-voice, the user needs the opposite transformation, that is she needs to get from an easy to utter form to a hard to utter native form.

Theoretically, code dictation seems natural and easy to perform. Indeed, Damper and Leedham (1997) studied the suitability of SR for the various types of input tasks described in (Foley et al., 1984) and argued that SR is best suited for *string* input tasks (of which code dictation is a particular example). However, this analysis does not hold in the case of code dictation, because of *punctuation* and *symbols*. Most programming languages rely heavily on *punctuation* to allow unambiguous parsing by computers. While a punctuation mark can be typed easily with a single keystroke (either simple or composite), dictating it requires the programmer to say one or more words (e.g. "*open-paren*", "*close-paren*", "*open-brace*"). Given the high frequency of punctuation in programming code, this causes a significant vocal load. Also, it increases the user's cognitive load since people are not used to uttering punctuation while speaking. Note that some languages (e.g. Cobol, Ada and Pascal) rely on English keywords rather than punctuation for their syntax and are consequently easier to dictate. However, the languages most commonly used today (e.g. C, C++, Java, Perl) tend to have heavily punctuated syntax and therefore programming-by-voice tools must cater to that situation.

Another problem with code dictation is caused by programmer-defined *symbols* like variable and function names. These symbols often contain abbreviations that must be spelled (e.g. *currRecNum*). But spelling with most current SR systems is not an easy task. Letters are very hard to recognize because they are so short in duration. Consequently, most SR systems require the user to spell using a special "*alpha-bravo*" alphabet. For most people, this involves a significant cognitive load.

Even symbols that do not contain abbreviations can be hard to dictate if they mix words with different capitalization (e.g. *maxOffSet*). In those cases, the user must invoke commands to carry out the proper capitalization (e.g. "*begin-capitalize*", "*end-capitalize*"), which again is lengthy and increases the vocal load. Also, uttering the right capitalization command at the right moment requires concentration and increases the user's cognitive load.

It could be argued that the availability of SR eliminates the need for cumbersome abbreviated symbols because programmers can now afford to enter verbose unabbreviated symbols. This argument is based on the assumption that programmers have historically adopted abbreviated symbols only because they were faster to type. While this is partly true, there are a number of reasons why programming-by-voice must still support the dictation of abbreviated symbols. Firstly, abbreviated symbols take up less screen real estate, and will probably continue to be used for that reason.

Secondly, we cannot expect all programmers to immediately adopt SR, and in fact it may be that speech will never completely replace the keyboard. So unless programmers-by-voice isolate themselves from the rest of the programmer community, they will need to work on code written with mouse and keyboard. Finally, even if speech became the most common input modality for programming, it would still have to support maintenance of legacy code that contains hard to dictate symbols.

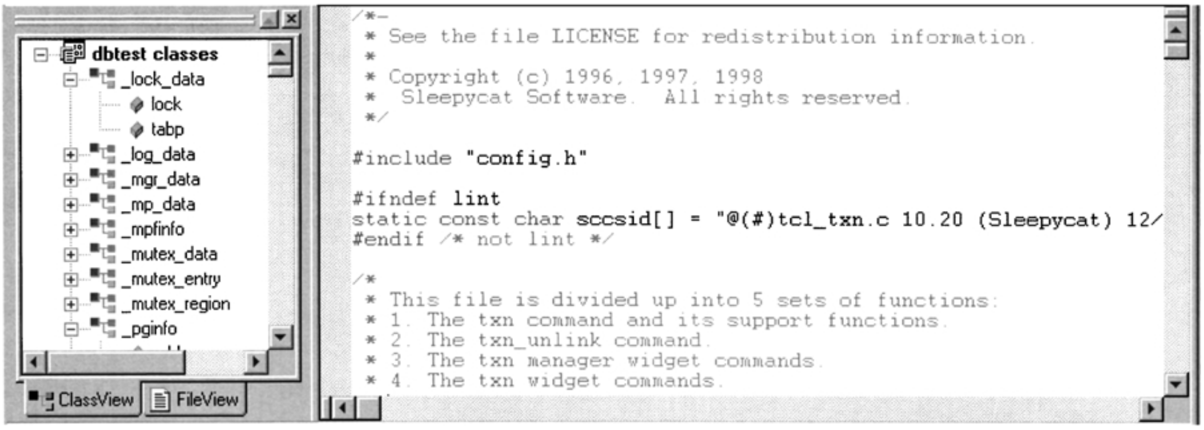
The second problem, *Code navigation*, has to do with moving the cursor and/or scrolling window to specific parts of the code. When modifying existing code, navigation is used to identify code that needs modification and move the cursor there so that changes can be dictated. We distinguish between two types of navigation. In *local navigation*, users move from one part of a file to a neighboring part of the same file. For example, scrolling an editor down three pages in a source file is considered local navigation. In *global navigation*, users jump from one part of a file to a non-neighboring part of a possibly different file. For example, moving from a line in a source file where a particular variable is used to the line in an other source file where this variable is defined, is considered global navigation.

Most local navigation operations fall in the category of what Foley et al. (1984) call *position* input tasks (identifying a point in a two-dimensional space). As argued by Damper and Leedham (1997), this type of task is hard to do by voice because it calls for an essentially analogue, continuous device, whereas SR produces a discrete output. Global navigation may typically involve three of the Foley et al. types of input task: *string*, *position* and *select* (picking an item from a list). For example, to move the cursor to the definition of a particular class, the programmer could:

- dictate the name of the class in a *class name* text field of a *find class* dialogue (string)
- move the cursor over an instance of the class and invoke a command to move to its definition (position)
- select the class name from an object browser such as the one in Fig. 1 (select)

As we pointed out earlier, position input tasks are difficult to perform by voice and so are string input tasks when it comes to symbol dictation. As for select input tasks, Damper and Leedham (1997) argue that they are easy to do by voice as long as the list of alternatives is not too long. In situations where this list is long, the user must scroll through it by voice, which presents problems similar to position input tasks. In the context of globally navigating source code, the list of alternatives consists of all possible places in the source code that the user might want to jump to. This is typically very large and therefore not suited for selection by voice.

The third problem, *error correction*, is an issue for all voice input tasks. However, it is particularly problematic in a programming context. When using a SR system, it is important that the user corrects every recognition error otherwise the accuracy of her voice model will degrade gradually over time. In a normal dictation context, when the user corrects a misrecognition, the SR system can automatically untype the misrecognized words and type the correct ones instead. But in a programming context, this is often not possible because programmers-by-voice frequently use editor



```
/*-
 * See the file LICENSE for redistribution information.
 *
 * Copyright (c) 1996, 1997, 1998
 *  Sleepycat Software.  All rights reserved.
 */

#include "config.h"

#ifndef lint
static const char sccsid[] = "@(#)tcl_txn.c 10.20 (Sleepycat) 12/
#endif /* not lint */

/*
 * This file is divided up into 5 sets of functions:
 * 1. The txn command and its support functions.
 * 2. The txn_unlink command.
 * 3. The txn manager widget commands.
 * 4. The txn widget commands.
```

Figure 1. An object browser window (left side).

macros that cannot be undone by the SR system. For example, suppose the programmer has defined a voice command "*for loop*" to invoke an editor macro that: (i) types some template code, (ii) indents it automatically at the proper level and (iii) moves the cursor to a particular position inside the template. If at a particular point the SR erroneously recognizes "*for loop*", it has no way of undoing the actions fired by that voice command because it doesn't know what the editor macro did (in fact, the SR system may not even be aware that the sequence of events it sent to the editor ended up invoking an editor macro). Therefore, most SR systems do not attempt to automatically undo the result of a misrecognized utterance if control characters and/or non-keyboard events were sent as a result of that utterance. For programmers-by-voice, this means they have to manually undo all the commands up to and including the misrecognized one, and then reutter that command and all the ones that followed. Given the error rate of current continuous desktop SR systems (in the order of 5%) this situation arises relatively frequently and can become time consuming.

Finally, the last problem is the requirement for *mouse-free operation*. It could be argued that some of the code editing problems (e.g. local navigation) could be solved by requiring the programmer to use the mouse for at least some operations. Although this is possible for programmers with mild cases of RSI, it is simply not an option for severely afflicted individuals or for individuals with handicaps that completely preclude the use of the mouse (such as paraplegia). Therefore, there is a requirement for programming-by-voice tools to be completely operatable without a mouse.

2.2. *Impact of Code Editing Problems on other Programming Activities*

As we pointed out before, there is more to programing-by-voice than mere code editing. However, in this section we argue that the code editing problems described in Section 2.1 (punctuation, symbols, local navigation, global navigation, error correction, mouse-free operation) are central to most other programming-by-voice activities.

In their empirical study of software engineer work habits, Singer et al. (1997) observed that only 2.9% of activities done by software engineers were related to code editing. This figure doesn't give an accurate picture of the importance of code editing problems in programming-by-voice for three reasons. Firstly, the software engineers observed by Singer et al. were involved only in software maintenance. Although this has never been verified empirically, one would expect code editing to be more frequent for programmers involved in development projects. Secondly, the Singer figures only measure the frequency of activities and say nothing about the length of time spent on each. In a programming-by-voice context, code editing with current SR technology is so lengthy that it could end up taking a significant portion of a programmer's time even if it was a relatively infrequent activity. Finally and most importantly, the code editing problems described in Section 2.1 creep up in other programming activities as we will now demonstrate.

The study by Singer et al. identified the following 14 types of activities as being typical of a software engineer's work. Since the present paper is concerned with programming only, we will distinguish between programming and non-programing activities. The 14 types of activities are:

Programming activities

i. Searching source
ii. Viewing source
iii. Following call traces
iv. Using a debugger
v. Issuing shell commands
vi. Configuration management
vii. Using in-house tools
viii. Editing source
ix. Compiling sources

Non-programming activities

x. Manipulating hardware
xi. Consulting documentation
xii. Consulting other software engineers
xiii. Taking/consulting notes
xiv. Management activities

In the Singer study, the 9 programming activities accounted for 74.8% of activities carried out by software engineers. Figure 2 shows their relative frequencies.

We now argue that from a programming-by-voice point of view, all but one of these 9 programming activities (*following call traces*) present problems similar to those described previously for code editing. One reason for this is that programming operations often require arguments which are portions of code or programming symbols. Another common type of argument is the name of a file or directory (e.g. *myFile.prj*),

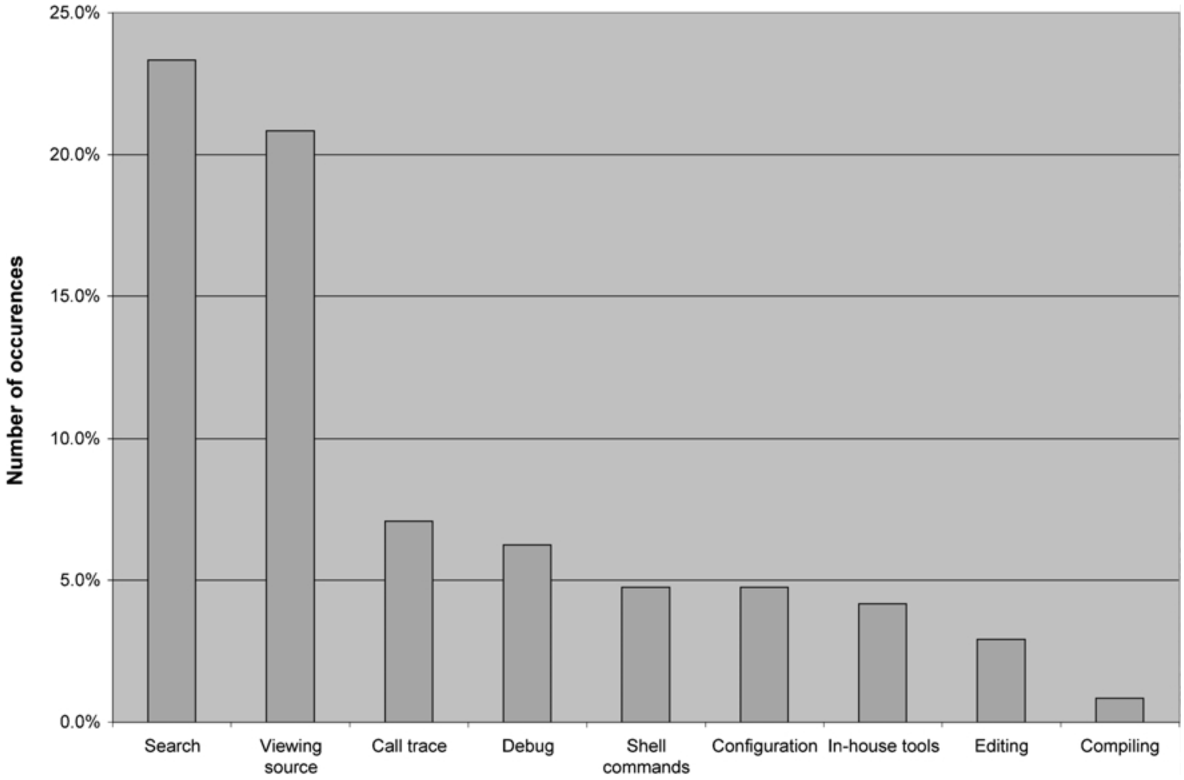


Figure 2. Relative frequencies of programming activities (Singer et al., 1997).

which present the same problems for dictation as programming symbols. A second reason why code editing problems occur in other programming-by-voice activities is that local and global navigation are often required by those other activities as well.

Note that in most modern programming environments, users do not necessarily have to type the name of files, directories and symbols on which they want to operate. Most of those environments allow the programmer to pick them from a list. For example, the object browser window depicted in Fig. 1 allows the programmer to select a class/attribute/method by selecting it from a hierarchical list. However as we pointed out earlier, in the context of programming the list of alternatives is typically too large to make selection by voice practical.

We now look in turn at each programing activity and describe how they are affected by code editing problems. In *Searching source* for example, the programmer needs to enter the name of files to search as well as a search pattern. According to a survey (Sim et al., 1998), in 89% of the cases this search pattern is a programming symbol or a portion of code. In *Using a debugger*, the programmer frequently needs to enter the name of a variable to inspect or a function to break at. Also, the programmer may need to do *local navigation* to move the cursor to a line where he wants to insert a break point. *Shell commands* usually use a syntax similar to programming languages, and dictation presents the same problems as *code dictation*. Also, in the context of programming, *Shell commands* are typically used for navigating through directories and manipulating files (e.g. deleting, copying, renaming, compiling, grepping), which means the names of directories and files often need to be dictated as command arguments. When *Viewing code* in an editor, programmers do local and global navigation as they selectively browse through portions of the source. *Version management* and *Compilation* both require the user to specify files on which to carry out those operations. The particulars of *In-house tools* will vary from organization to organization but in all likelihood, if they are designed to operate on source code they will involve manipulation (and therefore specification) of files and pieces of source code. The only programming activity that does not present problems similar to those of code editing is *Following call traces*. While this activity does involve local and global navigation, the problems there are typically different from navigation of source code because the call trace files do not consist of programming language statements.

3. Review of Existing Tools for Programming-by-Voice

Having identified the code editing problems as being central to programming-by-voice in general, we now review existing programming-by-voice tools and discuss how they addresses those problems. This discussion is summarized in Table 1 which shows the coverage of code-editing problems for each of the programming-by-voice tool.

Columns in the table correspond to programming-by-voice tools. The last column corresponds to version 3 of VoiceGrip which will be described later in this paper. The rows of the table correspond to specific code editing problems. An **X** in the table indicates that a particular tool addresses a particular problem.

In addition to the usability problems listed in Section 2.1 (punctuation, symbols, local navigation, global navigation, error correction, mouse-free operation), we discuss two portability issues: *cross SR* and *cross editor*. A tool is *cross SR* if it can be ported to different SR systems (e.g. Dragon NaturallySpeaking, IBM ViaVoice, L&H VoiceXpress) with minor customization. Similarly, a tool is *cross editor* if it can be ported to different programming editors (e.g. Emacs, MS Developer Studio), with minor customization. These issues are important because most programmers have invested much time in customizing a particular SR and editor for their personal needs and are understandably reluctant to switch to new environments. Note also that we subdivided the *symbols* problem into 2 rows. The *existing symbols* row refers to the ability of dictating symbols which are already defined in the source code while the *new symbols* row refers to the ability of dictating new symbols which do not yet appear in the source code.

Research on programming-by-voice started in the mid 80's. Leggett and Williams (1984) conducted a first experiment to compare the speed of code editing tasks for subjects using a discrete SR versus subjects using keyboard and mouse. Subjects doing the tasks by voice used a language-directed editor which addressed the punctuation issue by automatically inserting punctuation marks without requiring the subjects to utter them explicitly. The authors found that voice was less efficient than mouse and keyboard but close enough to be competitive. Impact of the language-directed editor on the performance of the subjects doing the tasks by voice was not evaluated. Note that one must interpret these results carefully because of the artificial nature of the task which subjects were asked to perform. In one of the tasks, subjects were asked to exactly transcribe a predefined program, while in the other task they were asked to exactly reproduce predefined changes to a program. Because subjects did not have to think about what code to enter and/or modify, they would not have had to do any global navigation and their cognitive load would have been significantly lower than in a real programming situation. Also, the predefined programs and changes only contained symbols which were

Table 1. Coverage of code editing problems for existing tools.

	Legtt and Willaims	Petry and Pierce	Bettini and Chin	DEMACS	ELSE	CachePad	EmacsListen	VoiceGrip 2	VoiceGrip 3
Usability									
Punctuation	X			X	X				X
Existing symbols				X		X	X	X	X
New symbols									X
Local navigation			X		X	X			X
Global navigation			X						
Error correction							X		
Mouse-free operation	X	X	X	X	X	X		X	X
Portability									
Cross SR				X	X	X	X		X
Cross editor								X	X

in-vocabulary words. This means that the subjects would not have encountered the *symbols* problem.

Petry and Pierce (1984) present a BASIC programming-by-voice environment. This work focused on the SR aspect of the system and did not discuss any of the usability issues of programming-by-voice. Bettini and Chin (1990) present a program debugging environment where users issue spoken natural language commands to a source level debugger. The natural language commands allowed the user to navigate locally and globally by specifying line numbers or subroutines to break at (although the later involved spelling the name of the subroutine).

In the early 90's, availability of the first desktop large vocabulary discrete SR systems coincided with the first wave of computer induced RSI cases. This prompted a growing community of injured programmers to adopt SR and develop tools and techniques to facilitate programming-by-voice. Although their work has not been published in academic forums, most of it can be found on the World Wide Web.

For example, *DEMACS* (Nielsen, 1996) defines a series of voice and editor macros that address the *punctuation* problem by typing pre-punctuated templates for common code constructs in various languages. For example, a voice macro "*for loop*" can be used to enter the following C code:

```
for ( ; ; )
{
}
```

and position the cursor before the first semicolon. *DEMACS* also supports dictation of *new symbols* through the concept of a cache pad which we will discuss in more detail later. Note that although template commands alleviate the *punctuation* problem, they require the user to do a lot of *local navigation* during code dictation. This is because once the template code has been entered, users need to navigate to its various slots in order to fill them. This requires the user to pause before and after each navigation command because current SR systems do not allow continuous utterance of multiple commands (although they support continuous dictation of words interspersed with formatting commands like "*cap-that*"). Such pauses make dictation lengthy and increase the cognitive load because they interrupt the user's natural speech flow. Note also that code templates have other advantages besides addressing the *punctuation* problem. For example, they allow the programmer to type more code with fewer actions, and they help the programmer remember the syntax of common programming language constructs. However these advantages are not specific to programming-by-voice and are therefore not taken into account in the present review.

ELSE (Emacs Language Sensitive Editor) developed by Peter Milliken and customized for programming-by-voice by Hans Van Dam (Milliken and Van Dam, 2000) also deals with *punctuation* through a series of voice commands and voice friendly menus for entering code template. In addition, it addresses *local navigation* with voice commands that automatically move the cursor from one slot of a template to the next.

CachePad, originally proposed in *DEMACS* and later perfected by Epstein (Epstein, 1998), deals with the *existing symbols* problem by allowing the programmer to select them from a short list of recently used symbols. For example, uttering a voice command "*symbol 3*" inserts the CachePad's third symbol at the current cursor location. The system also implements commands allowing the user to add symbols to the CachePad list without having to point them with the mouse. For example, a command "*cache at second bravo on 47*" can be used to add the symbol starting at the second occurrence of the character *b* on line 47. Similar commands allow the user to do *local navigation* by specifying a line number and a character or punctuation mark.

EmacsListen (Klarlund, 2000) is a programming-by-voice environment that takes a string of continuously dictated text and interprets it as a string of code editing commands. Because interpretation of the string of commands is implemented at the editor level, the system can deal with *error correction* by automatically undoing the actions of misrecognized utterances and redoing the actions of the correct ones that followed it. *EmacsListen* also implements interaction modes that combine voice commands with cursor and mouse position to easily copy and paste portions of code. This can be used to type *existing symbols* by copying an existing occurrence. For example, the user might click the cursor where he wants to insert an existing symbol, then move the mouse pointer over an instance of that symbol (without clicking there) and say "*copy word*".

Version 2 of *VoiceGrip* (Désilets, 1997) allows programmers to automatically create voice commands for entering symbols defined in a series of source files. The system uses typographical conventions and a user provided dictionary of abbreviations to associate a phrase of in-vocabulary words to every symbol. For example, if appropriate expansions have been defined for abbreviations *curr*, *rec* and *num* in the abbreviations dictionary, then an existing symbol *currRecNum* would be added to the SR vocabulary as a new word with spoken form "*current record number*" and written form *currRecNum.*

As can be seen from Table 1, none of the existing programming-by-voice tools adress all the code editing problems in Section 2.1, although they collectively cover them all. This means that a satisfactory solution to the code editing problems—and therefore to programming-by-voice in general, could be achieved by combining features from different tools. The table also indicates that there is a greater range of existing solutions for some code editing problems (punctuation, existing symbols, local navigation, mouse-free operation) than for others (new symbols, global navigation, error correction). In particular, the only tool that supports *global navigation* (Bettini and Chin, 1990) was a research prototype that never became available as a product. As for the portability issues, most tools are *cross SR* but not *cross editor*. Note however that while most tools have been implemented in an editor-specific fashion, the techniques they use are not themselves editor-specific and could easily be replicated in other editors.

4. The VoiceGrip System

We now describe version 3 of VoiceGrip (not to be confused with version 2 which was briefly described earlier in this paper), a programming-by-voice tool that implements a relatively complete set of solutions to the code editing problems described in Section 2.1.

4.1. System Overview

The general VoiceGrip approach is to allow programmers to dictate code continuously using a pseudo code syntax that is easier to utter than the code itself. These pseudo code utterances are recognized as ordinary text and initially get typed literally into the editor. The programmer can subsequently utter a VoiceGrip command to automatically translate those pseudo code statements to native code in the appropriate language. For example, to dictate the sample code of Section 1, the programmer could say the single utterance:

> *"if current record number less than max off set then"*

into a code editor, then utter a VoiceGrip voice command to automatically translate it in real time to the desired C code. This is much shorter and easier to say than the "raw" utterance shown in Section 1. This code translation approach is unique to VoiceGrip and alleviates most of the code editing problems discussed in Section 2.1. It addresses the *punctuation* problem by allowing programmers to omit certain punctuation marks (e.g. parenthesis after the reserved word "*if*") and use easy to utter aliases for common sequences of punctuation marks (e.g. "*then*" for *close-paren* followed by *newline* and *open-brace*). It also alleviates the *existing symbols* and *new symbols* problems by allowing the programmer to dictate symbols by saying what they mean (e.g. "*current record number*") instead of struggling to dictate them exactly as they are written.

The VoiceGrip code translation approach also enables an effective form of local navigation. To navigate to a particular portion of code, the user simply invokes a "*find code*" command and dictates the code she wants to jump to. This code could be a symbol, or any part of a programing statement. For example, if the user wants to move the cursor to the next occurrence of a code snippet: *currRecNum = 0*, she can simply say:

> *"search code forward [pause] current record number equals zero enter"*

VoiceGrip supports *mouse-free operation* of all its functionality. Finally, it is designed to be *cross editor* and *cross SR*, as we shall explain later.

The architecture of the system is depicted in Fig. 3. The user utters *Pseudo code* and *VoiceGrip commands* which are recognized by the *SR system. Pseudo code* is recognized as ordinary text by the *SR system's* dictation grammar and is entered literally into the *Programming Editor. VoiceGrip commands* on the other hand, are recognized as voice macros (*SR Macros*) implemented specifically for VoiceGrip at the level of the *SR system's* command grammar. These macros invoke *Programming Editor* level macros (*Editor Macros*) also implemented specifically for VoiceGrip. These *Editor Macros* in turn invoke modules of the VoiceGrip system proper.

There are two types of VoiceGrip commands:

- *Compilation commands*
- *Translation commands*

The user issues *Compilation commands* when he wants VoiceGrip to scan a series of source files for

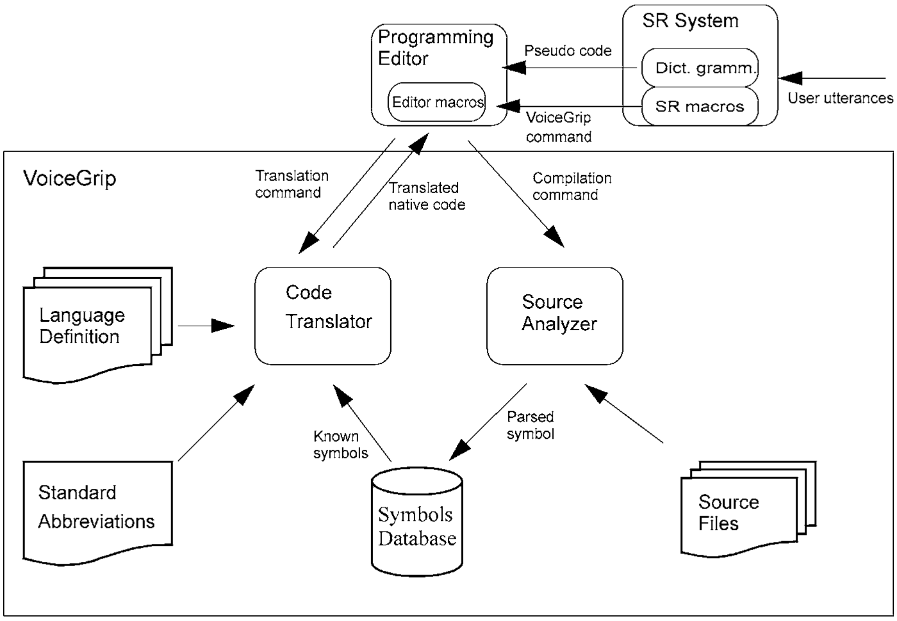


Figure 3. VoiceGrip architecture.

programming symbols. *Compilation commands* are forwarded to the *Source Analyzer* which scans the source files, identifies all symbols they contain and stores them in a *Symbols Database*. The user issues *Translation Command* when he wants VoiceGrip to translate a line of pseudo code to native code. *Translation Commands* are forwarded to the *Code Translator* module which takes the line of pseudo code and outputs its translation in the native syntax of the appropriate programming language. During translation the *Code Translator* uses *Language Definition Files* containing information about the pseudo code and native syntax of various programming languages. It also uses information about known symbols stored in the *Symbols Database*, as well as a *Standard Abbreviations Dictionary* which defines expansions for commonly used abbreviations (e.g. *pointer* translates to *ptr*). The details of the code translation process are described in Section 4.2.

The system was designed to be as cross editor and SR as possible. Supporting VoiceGrip from a new *SR system* simply involves implementing *SR Macros* (8 in total) to recognize VoiceGrip commands and invoke the appropriate *Editor Macros*. Similarly, supporting VoiceGrip from a new *Programming Editor* involves writing *Editor Macros* (also 8 in total) that use some form of interprocess communication to pass *VoiceGrip commands* to VoiceGrip and receive its response. At the moment of writing, VoiceGrip could be used with three commercial continuous SR systems (Dragon NaturallySpeaking, IBM ViaVoice and L&H VoiceX-press), and supported two programming editors (Emacs and MS Developer Studio).

4.2. *Code Translation Algorithm*

When translating pseudo code to native code in a particular language, VoiceGrip uses a simple deterministic parsing algorithm. At each iteration, it translates a substring starting at the beginning of the remaining pseudo code. It then removes that substring from the pseudo code and proceeds with the next iteration. The process continues until there is no more pseudo code to translate. The utterance is translated in a single pass with no backtracking. At each iteration the following translations are attempted:

- translation to a programming language construct
- translation to a known native symbol
- translation to a new native symbol

The first translation to succeed is applied and the algorithm proceeds with the rest of the pseudo code. The order of those 3 steps goes from the most constrained (*programming language construct*) to the

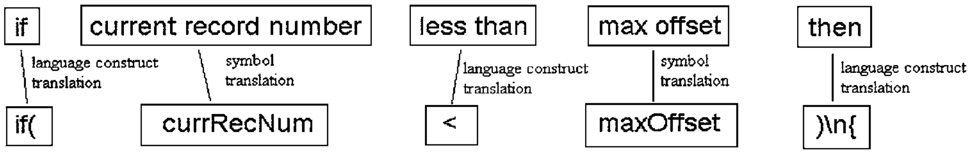


Figure 4. Example of pseudo code to native code translation.

least constrained (*new native symbol*) type of translation. This will be explained in more detail later. Figure 4 illustrates an example of translation.

Note that the translation from pseudo code to native syntax is completely handled at the level of the *Programming Editor*. The *SR System* plays no role other than to recognize the pseudo code using its dictation grammar and entering it "as is" into the *Programming Editor* so that the later can subsequently translate it.

We now describe the various translation steps in more detail.

4.2.1. Translation to a Programming Language Construct. The first thing the *Code Translator* attempts is to translate a portion of pseudo code to a programming language construct (e.g. the beginning of an *if* or *for* statement). Rules for such translations are specified by a series of Perl regular expressions—called *SUBSTITUTE patterns*, which are defined in the *Language Definition File*. For example, the *SUBSTITUTE* pattern defined by statement:

```
SUBSTITUTE('(\s|^)then(\s|$)', ')\n{\n');
```

means that the word *then* delimited by blanks (or by the start/end of the pseudo code) should be substituted by a closing parenthesis followed by an opening brace on a new line. In the example of Fig. 4, this rule was used to translate "*then*" at the end of the pseudo code.

VoiceGrip tries each of the *SUBSTITUTE* patterns in the *Language Definition File* until it finds one that matches. If no matching patterns are found, it proceeds with the next type of translation: *translation to a known native symbol*.

4.2.2. Translation to a Known Native Symbol. If translation to a standard programming language construct fails, the *Code Translator* assumes that the current portion of pseudo code starts with a pseudo symbol. It then finds the next occurrence of a *SUBSTITUTE* pattern and considers that to mark the end of the pseudo symbol. It then tries to translate that pseudo symbol to a known native symbol. In the example of Fig. 4, the portion of pseudo code between "*if*" and "*less-than*" (i.e. "*current record number*") did not match any of the *SUBSTITUTE* patterns and was therefore translated to a known native symbol ("*currRecNum*").

When translating a pseudo symbol, to a known native symbol, the *Code Translator* loops through symbols in the *Symbols Database* and looks for a likely match. To determine whether a known native symbol N is a likely match for a pseudo symbol P, we do as follows.

Let:

- $N_i, i = 1, \ldots, n$ be the ith alphanumeric character in N, shifted down to lowercase
- $P_j, j = 1, \ldots, m$ be the jth alphanumeric character in P, shifted down to lowercase

we consider that N matches P if there is a mapping of characters:

$$f\text{: } \{1, \ldots, n\} \text{ -> } \{1, \ldots, m\}$$

such that:

- $N_{f(i)} = P_{f(j)}, i = 1, \ldots, n$
- $f(i) < f(k)$ for all $i < k$
- if P_j corresponds to the first character of a word in P, then there is an i such that $f(i) = j$

In other words, mapped characters must be equivalent (in a case insensitive sense), the mapping must preserve character ordering and the first character of each word in P must be mapped. In addition, we require the native symbol N to have at least 3 alphanumeric characters. The rationale for this is that short native symbols can potentially match too many pseudo symbols, which results in too many false matches. Note that this restriction means that short symbols cannot be dictated as pseudo symbols and must be spelled explicitly. But in the case of short symbols, spelling is usually as quick and almost as natural as dictation as pseudo symbols.

If more than one known symbols match P, the *Code Translator* chooses the one that has been dictated most recently. The rationale for this is that recently dictated symbols are probably relevant to the current programming task and are therefore more likely to be dictated in the near future than arbitrary symbols. If neither of the matching known symbols have been dictated recently, then the *Code Translator* chooses the one with highest frequency in the source files that have been compiled by the *Source Analyzer*. The rationale for this is that symbols with high frequency have been dictated (or typed) many times in the past and are therefore more likely to be dictated in the future than symbols with low frequency. If two or more of the matching known symbols have the same frequency, the *Code Translator* then chooses the longest one. Again, the rationale for this is that short native symbol are more likely to falsely match a given pseudo symbol. If there are no matches at all, the *Code Translator* proceeds with the next type of translation: *translation to a new native symbol*.

Note that even for a relatively long native symbol S, there can be a huge number of falsely matching pseudo symbols. This means there is a good chance that S could falsely match pseudo code for a programming language construct defined by one of the *SUBSTITUTE* patterns. This is why we attempt symbol translation only after translation to a programming language construct has failed. For example, suppose the user says "*for loop*" to enter the beginning of a for loop. If we were to do symbol translation first, this pseudo code could falsely match existing native symbols such as: *flop* ("*floating point operation*"), *fr_LP* ("*frequent LP*"), *f o_lp* ("*file output line pointer*")

4.2.3. *Translation to a New Native Symbol.* If translation of a pseudo symbol P to a known native symbol fails, VoiceGrip assumes that P represents a new symbol. It then abbreviates each word in P and joins them to form a new native symbol that gets added to the *Symbols Database*. In the example of Fig. 4, pseudo code "*max record*" did not match a *SUBSTITUTE* pattern nor a known native symbol, and was therefore abbreviated as new symbol *maxRec*. Note that the pseudo code form of new symbols is completely unconstrained, that is, any sequence of words could be construed as the pseudo code form of a new native symbol. This is why translation to a new symbol is attempted only after the more constrained types of translation (*programming language construct and existing native symbol*) have failed.

Abbreviation of a word $W = (w_1, \ldots, w_n)$ in P is done using heuristics that approximate the way in which programmers do it. These heuristics are controlled by preset thresholds l_{min} and l_{max} denoting the minimum and maximum length of an abbreviation (currently set to 3 and 5 respectively).

First we check if $i \leq l_{max}$ or if W is made up of consonants only. In such cases, we consider the word to be already abbreviated and leave it alone.

If not, we try to abbreviate W. We first look in the *Standard Abbreviations Dictionary*. This is a file which specifies the expansion for common abbreviations used by programmers (e.g. *pointer* translates to *ptr*). The user can also add his own idiosyncratic abbreviations. If W appears in the *Standard Abbreviations Dictionary*, we replace it by its standard abbreviation. If W does not appear in the *Standard Abbreviations Dictionary*, we try to truncate it at a position $l_{min} \leq i \leq l_{max}$ such that w_i marks the end of a sequence of consecutive consonants in W (e.g. *symbol* translates to *symb*).

If truncation fails, we remove all vowels from W except for consecutive leading vowels. Truncation is then attempted again as described above (e.g. *transport* translates to *trnsp*).

If the second truncation also fails, the whole word W is returned as the abbreviation.

5. System Evaluation

The VoiceGrip system has been available for free download at *http://ii2.ai.iit.nrc.ca/VoiceCode/aboutVG.html* since November 1998. To date, 343 users have downloaded it. While a formal usability study has not been done on the system, messages posted by users on a neutral programming-by-voice mailing list (VoiceCoder: *http://www.egroups.com/community/VoiceCoder/*) indicate that the system is actually being used by a number of those who downloaded it. One user has commented that VoiceGrip is the first tool to make code editing practical. However, programing-by-voice, even with VoiceGrip, is still not generally perceived to be as productive as programming with mouse and keyboard.

We have also done an experiment to evaluate the precision of the algorithm for translating pseudo symbols to known native symbols. We collected a total of 16734 native symbols from the following sources: (i) the C code for the Emacs editor, (ii) the header files for the standard C libraries and (iii) the standard Perl libraries. We then manually translated each of those native symbols to a likely pseudo symbol, and asked

Table 2. Error rate of symbol translation algorithm.

Number of symbols	16734
Errors	
WITH homophone confusion	1111 (6.6%)
WITHOUT homophone confusion	446 (2.7%)

the *Code Translator* to match it to one of the native symbols. The results of this experiment are summarized in Table 2. We found that only 1111 of the 16734 pseudo symbol were not matched to the correct native symbols (6.6%). In 665 of those cases we found that the native symbol matched by the algorithm was *homophonic* to the correct one, meaning that they had the same pseudo code pronunciation. Some of the homophones differed only in non-alphanumeric characters or case of characters (e.g. *x_rectangle* vs *XRectangle*) while others corresponded to different ways of abbreviating the same English words (e.g. *ypos* vs *y_position*). If we ignore errors that are due to homophonic symbols, there remains only 446 errors (2.7%).

Note that the error rate of the algorithm in an actual programming situation should be somewhere between 2.7% and 6.6%. This is because in our experiment, homophonic symbols were essentially indistinguishable. In fact, one cannot say that a native symbol is a better match than one of its homophones, without referring to an actual programming context (e.g. symbol *A* is better than homophone *B* when entered at line *X* of file *Y* in program *Z*). In a real programming situation, the translation algorithm would be able to use context information such as recent dictation history and frequency of the symbol in the current program, to better disambiguate between homophones. In any case, even the error rate with homophone confusion is relatively small and feedback from users indicate that it is within a tolerable range.

6. Conclusions and Future Research

In this paper, we have discussed various problems with editing code by voice using current Speech Recognition systems. We also demonstrated that these code editing problems were central to programming-by-voice in general. We reviewed tools which have been developed so far for programming-by-voice and showed that although none of them adress all code editing problems, they collectively cover all of them. We have also described VoiceGrip, a tool that addresses the widest range of code editing problems to date. Messages posted by individuals on a neutral programming-by-voice mailing list indicate that the system is being put to actual use by at least some of the 343 people who downloaded it.

Some VoiceGrip users have commented that they would prefer to dictate pseudo code and see it being typed directly as native code into the editor (currently, pseudo code first gets typed literally and must be translated later by invoking the *Code Translator*). Also, because pseudo code can be ambiguous (for example with respect to homophonic symbols), the *Code Translator* is not always 100% accurate. In cases where it mistranslates portions of pseudo code, users have commented that they would like to have a correction mechanism similar to the one used in SR systems for recognition errors. For example, the user might select the mistranslated portion and ask VoiceGrip to display a selection of other plausible translations. We plan to add those two improvements in the near future.

While VoiceGrip has been useful in practice for programmers-by-voice, it is still not generally perceived to be as productive as mouse and keyboard. Also, while VoiceGrip is the tool that currently covers the widest range of code editing problems, its solution to each individual problem may not always be optimal when compared with other tools. For example, *CachePad* may be a better alternative than VoiceGrip for entering symbols whose pseudo code form is very long. For example, if you plan to enter symbol *PollConnThreadNonBlock*, many times in the next five minutes, you may be better off adding it to the *CachePad* and dictate it as "*symbol N*" (where *N* is the current position of the symbol in the CachePad) than say "*poll connection thread non blocking*" every time.

We believe that it is possible to build a programming-by-voice environment which would be completely competitive with mouse and keyboard, by integrating together features from the various programming-by-voice tools described in Section 3. Consequently, the National Research Council of Canada has started an Open Source initiative to bring together programming by voice best practices into a coherent cross editor toolbox. This initiative—called VoiceCode, currently involves 15 developers from 5 countries (Canada, US, UK, Australia, Netherlands) and involves the authors of 4 of the tools described in this paper (VoiceGrip, EmacsListen, CachePad and ELSE). The first version of this toolbox is expected to be available in early 2001.

Acknowledgments

I wish to thank the following programmers-by-voice for their constant and stimulating input throughout the development of VoiceGrip: Jonathan Epstein, Rick Gehrs, Eric Johansson, Nils Klarlund, Peter Milliken, Hans Van Dam, Barry Jaspan and David Jeschke. I would also like to thank Janice Singer and Normand Vinson for their excellent comments on this paper.

References

Bettini C. and Chin, S. (1990). Towards a speech oriented programming environment. *IEEE Region 10 Conference on Computer and Communication Systems*. Hong-Kong: IEEE, pp. 592–595.

Damper, B. and Leedham, G. (1997). Human factors. In *Speech Processing*. McGraw Hill, NRC, Ottawa, pp. 360–393.

Désilets, A. (1997). VoiceGrip 2.0: A utilities package for programming by voice (Report NRC 40220). *Ottawa, Ontario, Canada: National Research Council of Canada*. Retrieved July 18th, 2000 from the World Wide Web: ftp://ai.iit.nrc.ca/pub/iit-papers/NRC-40220.pdf.

Epstein, J. (1998). The CachePad Emacs extension. Retrieved July 18th, 2000 from the World Wide Web: http://voicerecognition.org/developers/jepstein/.

Foley, J.D., Wallace, V.L., and Chan, P. (1984). The human factors of graphic interaction techniques. *IEEE Computer Graphics and Applications*, *4*:13–48.

Kambeyanda D., Singer L., and Cronk, S. (1997). Potential problems associated with use of speech recognition products. *Asst Technol*, *9*:95–101.

Kiesler, S. and Finholt, T. (1994). The mystery of RSI. In C. Huff and T. Finholt (Eds.), *Social Issues in Computing: Putting Computing in Its Place*. McGraw-Hill, NY, pp. 94–118.

Klarlund, N. (2000). EmacsListen Emacs extension. Retrieved July 18th, 2000 from the World Wide Web:http://www.research.att.com/projects/EmacsListen/.

Leggett, J. and Williams, G. (1984). An empirical investigation of voice as an input modality for computer programming. *Int. J. Man-Machine Studies*, *21*:493–520.

Milliken, P. and Van Dam, H. (2000). Emacs language sensitive editor (ELSE). Retrieved July 18th, 2000 from the World Wide Web:http://members.xoom.com/pmilliken/.

Nielsen, T.R (1996). DEMACS Emacs extension. Retrieved July 18th, 2000 from the World Wide Web: ftp:/ftp.cl.cam.ac.uk/a2x-voice/demacs.tar.

Pascarelli, E. and Quilter, D. (1994). *Repetitive Strain Injury*. NY: John Wiley and Sons.

Petry, F.E. and Pierce, W. (1984). Input of tiny BASIC programs by voice. In Proc. *IEEE Southeastcon '84*, Louisville, Kentucky, USA: IEEE, pp. 245–247.

Raman, T.V. and Gries, D. (1995). Audio-formatting- making spoken text and math comprehensible. *Int. J. of Speech Technology*, *1*:21–31.

Sim, S.E., Clarke, C.L.A., and Holt, R.C. (1998). Archetypal source code searches: A survey of software developers and maintainers. *Proceedings of IWPC'98*. Los Alamitos, CA: IEEE Comput. Soc., pp. 180–187.

Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. (1997). An examination of software engineering work practices. In *Proceedings of CASCON 97*, Toronto, ON, NRC, Ottawa, pp. 209–223.

US Bureau of Labor Statistics (1995). Workplace injuries and illnesses in 1994. (Report USDL-95-508). US Washington, DC: Bureau of Labor Statistics.