Programming by Voice:

A Hands-Free Approach for Motorically Challenged Children

Author: *Amber (Krug) Wagner*
Advisor: *Dr. Jeff Gray*
Committee Member: *Dr. Eugene Syriani*
Committee Member: *Dr. Randy Smith*
Committee Member: *Dr. John Lusth*

Fall 2012

Department of Computer Science
College of Engineering
University of Alabama

Abstract

Computer Science (CS) educators frequently develop new methodologies, languages, and programming environments to teach novice programmers the fundamental concepts of CS. A recent trend has focused on new environments that reduce the initial challenges associated with the heavy syntax focus of traditional environments. There are numerous Initial Programming Environments (IPEs) available that have been created for student use that in some cases have fostered self-discovery and inquiry-based exploration. In this qualifier paper, three IPEs are surveyed: Scratch, Lego Mindstorms LabVIEW, and App Inventor. These three IPEs were selected due to the high number of students and teachers currently using them, as well as my personal experience with these environments through past teaching activities. The creative and graphical nature of these three IPEs have been shown to increase student and teacher interest in CS through adoption in classrooms worldwide at the K-12 levels, as well as interest in introductory university courses. An additional reason for selecting these environments for the focused study is that I have assisted in offering professional development workshops for teachers and student summer camps utilizing these environments.

Although the graphical nature of these IPEs can be helpful for learning concepts in CS, a small group of students is being left out from learning experiences and engagement in CS. Graphical environments often require the use of both a mouse and keyboard, which motorically challenged users often are unable to operate. Based on research performed and presented in this paper, a Vocal User Interface (VUI) is a viable solution that offers a "Programming by Voice" (PBV) capability (i.e., a capability to describe a program without the use of a keyboard or mouse). There are two primary disadvantages with VUIs as a technology to address the limitations of motorically challenged users: 1) vocal strain can emerge for technical solutions that require a deep amount of vocal interactions, and 2) the process of integrating voice controls into a legacy application (e.g., an IPE) is a very time consuming process. There are existing vocal tools (e.g., the generic Vocal Joystick) that could be integrated into various applications; however, due to the needs of IPEs and the duration of IPE usage, the Vocal Joystick is not feasible due to the potential vocal strain, which is why a more command-driven approach may offer benefits for the PBV concept. The command-driven approach leads to a time-consuming process in terms of adapting legacy applications, particularly, if multiple applications (like the three IPEs previously mentioned) require specialized VUIs. Each environment has its own layout and its own commands; therefore, each application requires a different VUI. In order to create a more generic solution, Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs) can be applied to create a semi-automated process allowing a level of abstraction that captures the specific needs of each IPE. From the specification of each IPE, a customized VUI can be generated that integrates with the legacy application in a non-invasive manner.

This paper presents background information on IPEs focusing on Scratch, LabVIEW, and App Inventor. Accessibility issues are discussed as well as previous work surrounding the concept of PBV. The possible application of MDE and DSL to support an integration solution is also introduced.

Table of Contents

Abstract 2

1. Introduction 4

2. Motivation for Improving Accessibility 5

3. Computer Science Education for First Learners 7

3.1 Learning Environments 7

3.2 Accessibility Issues 13

4. Previous Work in Voice-Driven Control of Programming Environments 14

4.1 Programming by Voice (PBV) 15

4.2 Myna 17

5. Summary of Future Research Plans 19

5.1 Myna 19

5.2 Overview of Possible Future Contributions from Extending Myna 20

6. Conclusion 22

Acknowledgements 23

References 23

1. Introduction

According to the US Census Bureau [39], there were approximately 54,030,469 children ages six to seventeen in the US in 2010. Among those, 3,212,225 (5.9%) participated in the College Board's Advanced Placement (AP) program. Furthermore, only 20,120 (0.037%) participated in AP Computer Science [9]. This low participation rate suggests that either students do not understand or are unaware of the opportunities in Computer Science, or schools are not offering Computer Science courses. Figure 1 references the trajectory of student participation in the AP Computer Science program over the past decade. A large increase in participation from 1991 to 2001 (146%) can be seen in the graph; however, the total number of students taking AP CS (A and AB) has declined since then. Because the numbers dropped primarily for the AB exam, the College Board discontinued the course in 2009, although the A exam is still offered.

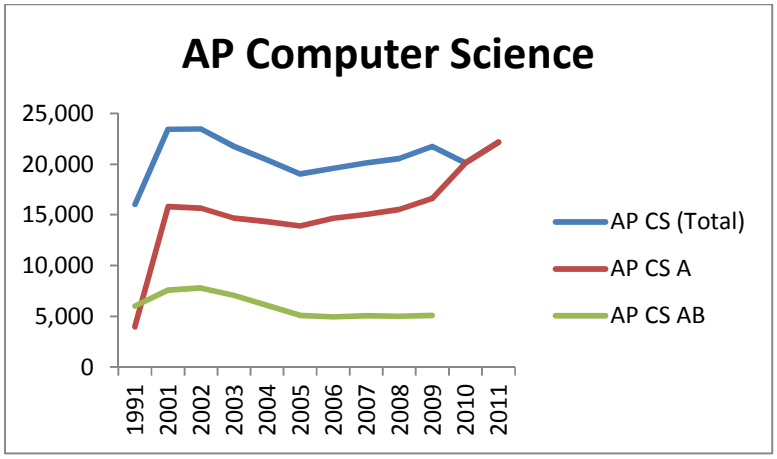

Figure 1. AP Computer Science participation, 1991-2011, provided by the College Board [9].

The Computer Science Teachers Association (CSTA) conducted two surveys: one in 2005 and an identical one in 2007 regarding High School Computer Science. In the analysis of the two surveys, Gal-Ezer and Stephenson [53] attribute the decrease in AP CS to three primary factors:

1. Rapidly changing technology;
2. Lack of student interest; and
3. Lack of staff support.

While teachers cannot control the speed at which technology changes or the amount of staff support available, teachers can provide a pedagogically sound yet interesting curriculum to motivate students rather than discourage them.

The current generation of students will not be impressed with the traditional "Hello World" program [21, 44] and pedagogical instruction techniques used with previous generations of students. The current generation of students has been playing video games with advanced graphics since they learned to walk, and the notion of internet accessibility is often viewed as a common utility. To increase the interest and attention of students, exciting new programs and technologies are being developed. Additionally, student engagement is often more successful when the context is driven by current topics that are of direct interest to students. For example, most teenagers are frequent users of smartphones, which

provides a unique context for engagement. The adoption rate of mobile computing among students age 12-17 in the US continues to grow and has been estimated at 23% for smartphone adoption (and 54% for general cell phone usage) [30]. Based on current US census results [40], this suggests that there are 4.8M middle and high school students with smartphones. Educators can take advantage of these devices as a springboard for motivating topics involving Computer Science [6, 9, 13, 25]. Mobile application development is not the only way to reach students, but based on the statistics, it is clearly an excellent starting point.

The realization that programming is a difficult topic for a first-time learner is not novel and has been a challenge for CS educators since the inception of our discipline. In fact, Kelleher and Pausch [21] reviewed 81 programming environments starting from 1963, in which the goal of each environment was to simplify the code-writing process for beginners. The improvements in programming environments over the past 40 years is very evident and likely have made a contribution toward how children are learning to program while concurrently raising their interest in CS. While this is positive, it is important to note that programming environments require the use of a keyboard and mouse. The dexterity required for these peripheral devices causes a substantial group of children to be left out of the CS education process. My long-term research focus is to investigate a new approach that makes an impact by offering motorically challenged children the same opportunities as an unimpaired child. The various topic areas and technologies that inform and drive this long-term vision are the focus of this qualifier paper.

This paper describes the context and motivation for this qualifier topic area in Section 2, followed by an explanation of initial CS learning environments and the accessibility issues associated with those environments in Section 3. Previous work is presented in Section 4, with a discussion including questions resulting from observations from the literature amid current needs for accessibility. The paper also provides a brief future research plan in Section 5, followed by concluding comments in Section 6.

2. Motivation for Improving Accessibility

Based on the statistics for AP CS participation presented in the previous section, methodologies and tools that introduce new students to CS and programming must be improved. Papert [56] suggests that programming languages should satisfy three conditions: low-floor (simple to start using), high-ceiling (ability to increase difficulty and complexity over time), and wide-walls (ability to create various types of projects). The current approach instructors are using to meet these suggested requirements is to utilize an environment with a graphical interface that eliminates the concern of syntax (e.g., Scratch [33], Alice [2], Lego Mindstorms LabVIEW [24], and App Inventor [3]) – these are often called Initial Programming Environments (IPEs) because their focus is on education and pedagogical concerns, rather than the traditional focus of a development environment for a programmer. Based on observations and empirical studies [55], the use of IPEs offers numerous benefits, such as: removal of the syntactic details of a language, the fostering of creativity thus increasing engagement and interest, and the ability to start problem solving on day one.

To address the concern over the challenges of learning the syntax of a programming language, many of the current IPEs have a form of “drag and drop” programming whereby a student connects program

constructs together in a manner that forces correct syntax (but may still produce logic errors). The mechanism for this capability is often through the usage of a mouse that requires a level of dexterity to connect the program statements together. However, because this new breed of IPEs requires motor functionality to operate a mouse and keyboard, those students who are motorically challenged are left out from these new learning experiences. The ACM code of ethics states, "[i]n a fair society, all individuals would have equal opportunity to participate in, or benefit from, the use of computer resources regardless of race, sex, religion, age, disability, national origin or other such similar factors" [50]. By not providing alternative means of access, users with disabilities are being denied learning opportunities that may allow them to explore career paths in computing. The driving motivation for the underlying theme of this qualifier paper is recognition that more children (or adults) should have the opportunity to learn about programming and CS using these new IPEs. Improving the diversity of a user base has several advantages, as noted by Kelleher and Pausch, who wrote that "[i]f the population of people creating software is more closely matched to the population using software, the software designed and released will probably better match [users'] needs" [21].

Many individuals suffer from a condition impairing motor functionality [29]. Of those with spinal cord injuries, 70% are unemployed [18]. If computer accessibility can be increased, new career opportunities may emerge that can help to reduce the unemployment rate among those with impaired motor function. Table 1 presents information collected from the National Institute of Neurological Disorders and Stroke [29], which lists four diagnoses impacting the motor function of children and/or young adults in the US. These individuals have the cognitive ability to understand the concepts of programming and CS, but they are unable to utilize the standard Windows Icon Mouse Pointer (WIMP) tools to interact with the computer.

Diagnosis	Number Impacted	Age of Onset
Spinal Cord Injury	250,000	56% occur between 16-30
Muscular Dystrophy	8,000(males 5-24 years of age)	400-600 males are born with MD each year
Multiple Sclerosis	250,000-350,000	20-40
Cerebral Palsy	800,000	10,000 babies born with CP each year

Table 1. List of a few major diagnoses impacting users' abilities, particularly children or young adults [7, 39, 40, 41].

A possible solution to widen the gap of accessibility to IPEs is to use a form of a voice-driven interface to assist as an input modality for those with motor impairments. The concept of "Programming By Voice" (PBV) is the long-term focus that drives the topic of this qualifier paper. Voice-driven applications, such as those presented in [5, 10, 11, 18, 34, 43], primarily cater to specific applications, like the Internet [14, 38]. These applications are discussed and compared in Section 4.

Scratch (discussed further in the next section) has a large number of users [33] and encourages a community of contributors where students can share and learn from each other. At the time of this writing, there are over 2.7M Scratch programs that have been shared among students and teachers. As an initial study to show the potential for extending an IPE to address accessibility needs, I have been leading the development of a prototype that explores the concept of PBV within the Scratch

environment [42]. This first effort is a fixed solution that is not extendable to other IPEs. It would be desirable to have a general approach that can assist in extending other IPEs with the concept of PBV. Kelleher and Pausch [21] described 81 programming environments that were developed over several decades. Although there are many environments that are no longer in use, CS educators and programming language researchers continue to develop better tools to teach programming. This qualifier paper is driven by the belief that the next step in IPE development will be to increase the usability [43] of all WIMP-based applications. However, incorporating voice-based control into an application is extremely time-consuming. The future research section of this qualifier (Section 5) outlines some ideas that point toward novel approaches to realize the goal of a generalized methodology for adding a VUI to legacy IPEs.

3. Computer Science Education for First Learners

Instructors are beginning to utilize "drag and drop" IPEs for novice programmers. These tools are plentiful in number, but I have chosen three for the purposes of this research – Scratch, LabVIEW, and App Inventor. These three environments and their common ancestry are described below along with the accessibility issues introduced by the environments.

3.1 Learning Environments

The College Board [9] indicates that high school CS enrollment is decreasing. The surveys conducted by the CSTA [53] highlight that student interest is an issue, which motivates the following question:

Question 1: How can we improve initial programming instruction?

Since 1963, researchers at universities have been developing initial programming environments to introduce programming concepts and problem solving with languages aimed at simplifying syntax [21]. Some environments (e.g., BASIC, Turing, and GNOME) were designed for college-level beginning programmers while others (e.g., Play, LogoBlocks, and Alice) were built for a younger target audience, such as elementary school children. The environments (e.g., Karel, GRAIL, and LegoSheets) reviewed by Kelleher and Pausch "tried to make programming accessible in three main ways namely, by simplifying the mechanics of programming, by providing support for learners and by providing students with motivation to learn to program" [21]. This subsection provides a brief history of the literature in this area of IPEs and compares and contrasts several popular existing environments.

The 81 programming environments surveyed by Kelleher and Pausch date back to 1963 when Dartmouth College created BASIC (Beginners All-Purpose Symbolic Instruction Code) to help teach "non-science students about computing through programming." BASIC was syntactically simpler than the more popular languages at the time (e.g., Fortran and Algol) [21]. The subtle simplification offered by BASIC was substantial in that it "sacrificed computer time for user time" [21]. Since the creation of BASIC, other universities have strived to design new languages with even simpler syntax such as GRAIL [49], created by Monash University in 1999. GRAIL was built on three principles: "maintain a consistent syntax; use terms that novice programmers are likely to be familiar with and avoid standard programming terms that have different meanings in English; and include only constructs that are fairly simple" [21].

GRAIL and BASIC are languages where textual programs are coded, as are many others created between 1963 and 1999 (e.g., JJ, Pascal, and Smalltalk). Language developers and those working in the initial days of CS Education, such as Seymour Papert [56] and Alan Kay [57], realized very early that children were suitable subjects for learning how to program. In 1967, MIT created Logo [21], which is a Lisp-based language that was made more appropriate for children by removing most of the need to understand syntax. While Logo was designed to "allow children to explore a wide variety of topics from mathematics and science to language and music," [21] it is most famous for the Logo turtle - a robotic turtle with a pen function that would illustrate the turtle's movements around the screen, given by the child's commands. Following from Logo's goal of being more approachable by students, other graphical-driven environments were investigated for novice programmers, such as TORTIS, which was developed in 1976 by MIT's Artificial Intelligence Lab. TORTIS, like many of its successors, was inspired by the Logo turtle in that children could programmatically control a robotic turtle to explore concepts of programming and even algorithm development. Many of the graphical languages were based on users creating rules, performing computations or I/O, or creating stories. With each new language development, more interesting and technical features became available, from simple environments such as Scratch [33], to more complicated environments such as Alice [2] by Carnegie Mellon University. Alice allows the creation of 3D interactive worlds simply by dragging and dropping program constructs into a syntax-directed editor, which removes the challenges of learning the details of syntax. Alice emerged as one of the most popular IPEs over the past decade among middle and high schools teachers, as well as many introductory courses within CS departments at the university level [54]. In addition to specific language environments, methodologies emerged that presented a common theme for exploration that are environment-independent, such as the Media Computation [17] approach that provides a context for understanding CS principles through manipulation of media files (e.g., changing properties of images or sound files). The following three learning environments are explained in more detail in the following subsections: Scratch [33], Lego Mindstorms LabVIEW [23], and Android App Inventor [3].

3.1.1 Scratch

MIT created Bongo in 1997, which allowed users to create their own video games and then share them with friends via the web [21]. MIT followed this same community-driven sharing model when creating Scratch, which was introduced in 2004 as an IPE that is simplified enough to teach to third graders, yet complex enough to teach to college freshmen or non-CS majors [58]. The University of California, Berkeley has adapted MIT's version of Scratch to a more advanced version called Build Your Own Blocks (BYOB) [6] (recently renamed SNAP), which Berkeley (and the University of Alabama) teaches to freshmen and non-CS majors. The creators of Scratch at MIT intended for Scratch to be a "networked, media-rich programming environment designed to enhance the development of technological fluency at after-school centers in economically-disadvantaged communities" [27]. The Scratch community has emerged into a social network for its 350,678 project creators who have shared 2,732,346 projects for all 1,187,229 registered members to view [33, 59]. Kelleher and Pausch [21] observed that social environments such as networked applications provide more motivation for students to learn how to program, and Scratch has provided further evidence to support this observation [58]. Scratch is available in 13 languages and is visited daily from 16,798 cities (see Figure 2) [33, 59]. As an IPE, Scratch is reaching hundreds of thousands of students and educators. The projects that are shared within the

community serve to inspire other members in numerous ways (e.g., students share their culture, political, and religious views, students collaborate to create scripts other students can utilize, and students learn how to critique one another). Scratch is more than just a programming environment; it is a socially-driven creative tool that children can use to express themselves and explore their own creativity.

Figure 2. The location and amount of Scratch users [33].

Scratch is heavily utilized throughout the world because of its ease of use and general availability (Scratch is freely available). Figure 3 is an image of the Scratch UI. All of the commands are "drag and drop" allowing the programmer to focus on logic and problem solving rather than syntax. Also, the programmer can develop his or her own images and import sounds, thereby fostering a creative environment that makes the underlying difficulty of the program transparent to the programmer.

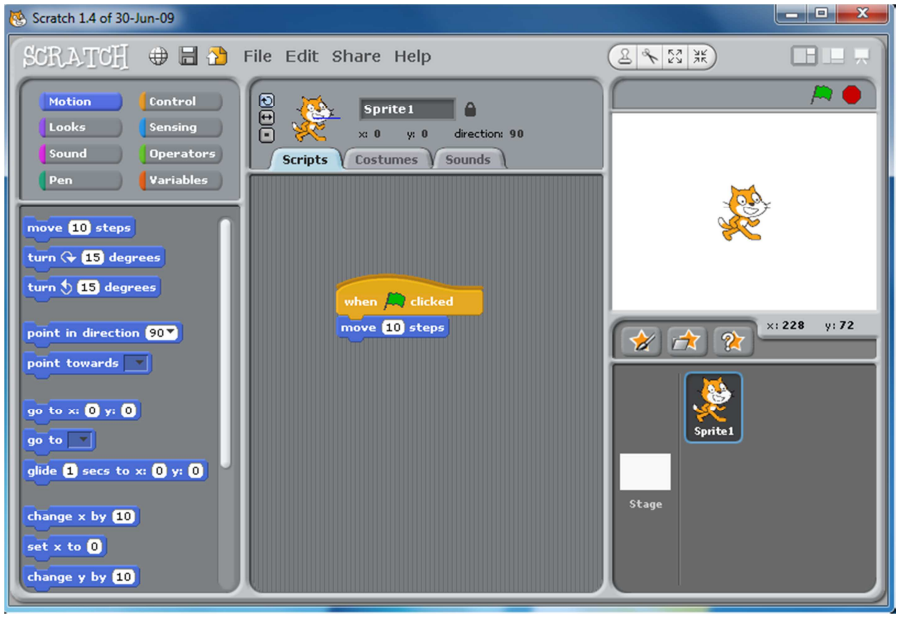

Figure 3. MIT's Scratch interface [33].

3.1.2 Lego Mindstorms Software

The University of Colorado created LegoSheets in 1995 to work with the MIT Programmable Brick ("intelligent, built-in computer-controlled LEGO brick that lets a MINDSTORMS robot come alive and perform different operations" [60]). LegoSheets was primarily graphical and allowed users to progress slowly from controlling basic motor functionality of a robot to adding conditional statements in a program. In 1996, the MIT Media Lab developed LogoBlocks, which was "a graphical programming language designed for the Programmable Brick" [21]. With LogoBlocks, users could "drag and drop" code blocks to manipulate the brick and also allowed the user to learn about the creation and use of parameterized procedures to support generic reusable functionality.

The successor of the programmable brick is the commercially available Lego Mindstorms LabVIEW environment developed by the MIT Media Lab in 1998. Since then, the programming environment has improved along with the hardware (e.g., the initial RCX Mindstorms robot used a very fragile infrared process for downloading programs, but the newer NXT platform supports Bluetooth and USB connections). The latest Mindstorms hardware is the NXT 2.0. A special programming template has been developed for National Instruments' LabVIEW [51], which provides a graphical programming environment that is used by many K-12 schools. The software comes with 46 tutorials to assist educators and/or students in learning how to program and command the NXT robot. The tutorials demonstrate how to build the robot (this is helpful if specific sensors are needed) and how to build the blocks for the code [24]. Lego has created a social media forum for users called NXTLOG 2.0 where users can post their projects and share ideas [23], similar to Scratch. Figure 4 is a screenshot of the Mindstorms LabVIEW interface [24], which is similar to MIT's original design. There is a block palette on the left, the program editor in the center, and the tutorials are on the right.

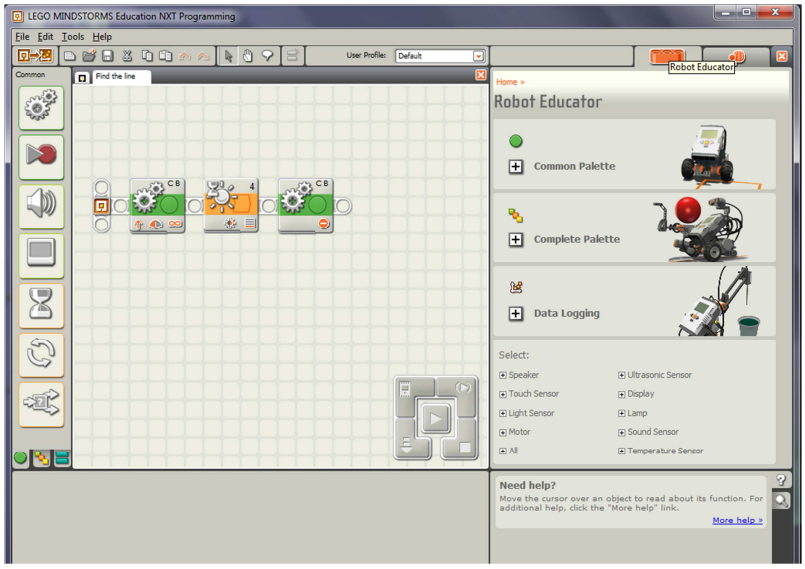

Figure 4. Lego Mindstorms' LabVIEW interface [24].

3.1.3 Android App Inventor

Google's Android App Inventor [3, 45], now maintained by MIT's Center for Mobile Learning, enables educators to take advantage of the interest that high school students have in smartphones. App

Inventor is a visual programming language that allows users to write apps using a block-oriented "drag and drop" interface to create an app's user interface and to specify its behavior and functionality (the latter is based off of Scratch's visual block structure). An emulator is available so that apps can be executed on a local desktop. App Inventor also integrates with Android smartphones and tablets, which enables the user-made applications to be tested on a physical device. Similar to Scratch, app developers can package their apps for a phone and then share that app with friends on a community-based gallery; thus, creating a networked social environment and motivating the students further [21].

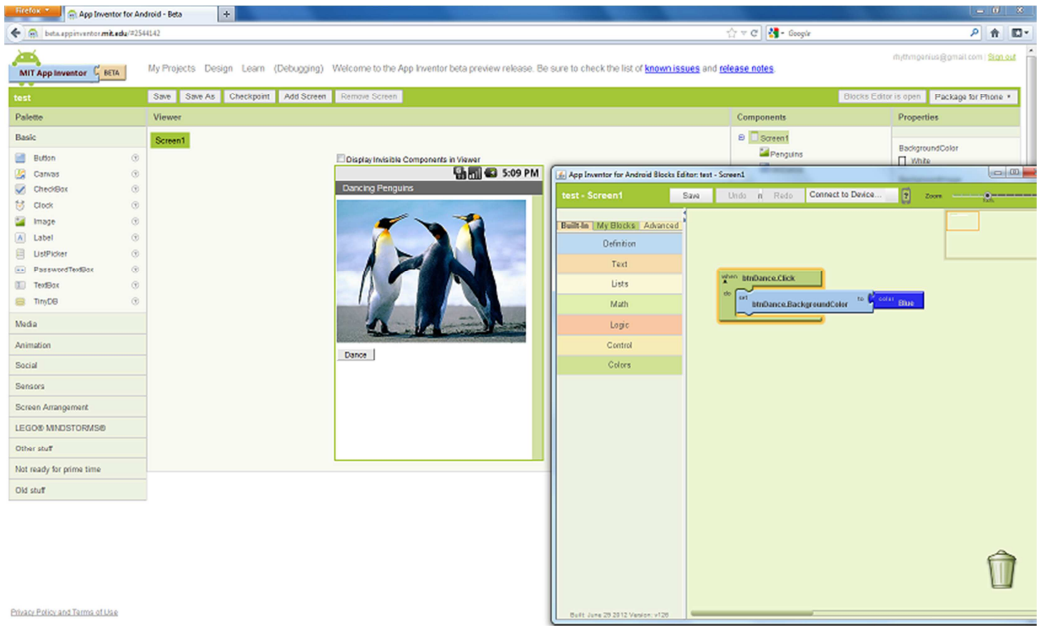

Figure 5. Google's App Inventor interface (now maintained by MIT) [3].

The popularity of mobile devices has inspired much interest as a context for teaching CS principles [26]. In fact, a new learning model has emerged, LOCAL (Location and Context Aware Learning) [4], which combines mobile devices and wireless networks to create a new learning context. David Wolber integrated App Inventor into a general education course over three semesters at the University of San Francisco (USF) and found that "teaching App Inventor has been [his] most satisfying teaching experience in seventeen years" [44]. Before teaching App Inventor, he used Lego Robots [23] and Media Computation [17], similar to the approach followed at the University of Alabama for summer camps. Using App Inventor at USF taught students not only how to program, but also offered a context for the potential impact that apps can have on the community [44]. Although the students attending the general education course at USF are not CS majors, the students successfully learned how to solve problems, and more importantly, the students felt successful at the conclusion of the course. Because students were able to start building applications immediately with App Inventor, they were motivated to learn "how to solve hard logic problems and specify interactive behavior with a static (though visible) language" [44]. As described in [13], Fenwick and Kurtz from Appalachian State University and Hollingsworth from Elon University taught their senior classes using the Android SDK [12] and App Inventor. The Elon University course started as a lecture-based course with hands-on activities followed by a project-based approach. Appalachian State University's course was project-based from the start.

Both universities found that student engagement was high and students once again exhibited an "entrepreneurial and independent spirit" [13].

To assist in teaching the importance of human-computer interaction, Loveland used App Inventor as a platform to motivate students in a new course [8]. At the conclusion of the semester, one student commented on his appreciation for using new technologies in the course. This is why many educators (e.g., as evidenced by positive adoption stories detailed in [4, 13, 15, 25, 26, 44]) are taking advantage of mobile computing as a teaching context. Students are able to learn about programming using tools that are very personal and relevant to their daily lives. Another successful educational use of AI is the creation of data collection apps for the betterment of a community [1]; particularly, Hal Abelson mentions a medical app, which collects data in Nicaragua and is used as a mobile laboratory [1]. In our own state of Alabama, Gina McCarley from Lawrence County attended one of our summer teacher workshops and then encouraged her students to design an app for tracking wild hogs, which was the national winner of Samsung's "Solver for Tomorrow" technology contest worth $125,000 [52].

3.1.4 Observations from Common and Distinguishing Characteristics among Modern IPEs

Scratch, Lego Mindstorms LabVIEW, and App Inventor have a common ancestor: LogoBlocks. LabVIEW is a direct descendent of LogoBlocks [21], Lego blocks were the inspiration for Scratch [58], and App Inventor followed the trend set by LogoBlocks in 1996. Scratch was intended for eight to 16 year old students [58]; however, Malan and Leitner [55] found success using it in introductory CS courses at the college-level. Similarly, Lego Mindstorms LabVIEW was designed for K-12 classrooms, yet Ranganathan, Schultz, and Mardani [60] found success using it in a freshman Electrical Engineering course. App Inventor was built for a broader audience and has had success in college classrooms [61, 62] and is slowly being integrated into high school classrooms [62, 63]. Unlike Scratch and LabVIEW, there is no evidence that App Inventor is appropriate for students younger than high school; however, it is still appropriate for this research as it provides a higher ceiling [56] for the older students.

While Scratch, LabVIEW, and App Inventor have many commonalities (e.g., similar age range, graphical blocks that snap together, a command palette, and a program editor), there are differences among them. Both LabVIEW and App Inventor have physical components that allow students to observe their programs run on a device other than the computer. Because of the associated physical devices, App Inventor and LabVIEW have a duality in that the user must design a Graphical User Interface (GUI) for the screen of the phone in addition to programming the behavior of the phone, and the user must ensure the robot is built appropriately for the LabVIEW program. While Scratch is intended to run on the computer only and is not associated with a physical device, it does have a similar duality as users must design the stage (or background) of their world prior to building the program.

An obvious commonality among all of these graphical programming environments is the dependence on the WIMP metaphor, which "provides ease of use but assumes dexterity of human hands to use a mouse and keyboard" [42]. Clearly, this dependence results in a failure to achieve universal usability [43] and "[address] the needs of all users" [35]. Based on the ACM code of ethics [50], it is important that these accessibility issues be addressed.

3.2 Accessibility Issues

Despite the growing popularity of IPEs as a context for introducing CS to young students, there are several challenges that remain related to extending the impact of such environments to a broader audience. This issue is motivated by the following question:

Question 2: The programming environments described in Section 3.1 depend on the WIMP metaphor; can all children participate?

Although Table 1 identifies four primary diagnoses affecting motor function, particularly in children, it is not all-inclusive. It also does not describe the severity of each diagnosis. Understanding how users with motor impairments currently use a computer and what issues they have with a standard computer is imperative to creating a better solution to "fit the abilities" [43] of these users. Harada et al. [18] performed a study to evaluate the effectiveness of a voice-controlled cursor, in which five of the participants suffered from various degrees of motor impairment. Participant number one in this study was 52 years old and was afflicted with multiple sclerosis for seven years. Although he could type, he used Dragon Naturally Speaking instead because he easily tires from the arthritic pain he feels in his hands. Participant number three (age 20) has suffered from muscular dystrophy since birth. Muscular dystrophy causes "progressive weakness and degeneration of the skeletal muscles that control movement" [29]; thus, this participant has a small range of mobility and must be in a wheelchair. Because her mobility is limited to the point that she cannot turn her hands so that her palms are face down, she has taught herself how to type with the backs of her hands. She tried using voice recognition in the past, but she was displeased with the performance. Participant number four (age 30) has had cerebral palsy since birth. Due to her condition, she has violent spasms frequently and weakened muscles. In addition to cerebral palsy, she suffers from fibromyalgia, a "chronic condition" [18] causing the nerves to misfire constantly. Her input device of choice is a touchpad on her laptop as a keyboard and traditional mouse are tiring for her. She is also on a ventilator, which concerned the researchers since the study was regarding voice-control (although the ventilator makes some noise during operation, it did not impact the application).

Changing input devices from a traditional device to a new modality like voice is a very difficult process; both the user and the application must learn or train. The user needs to learn the grammar, and the application needs to learn the user's accent/articulation [5, 10, 11, 18, 34, 43]. Because the traditional input devices for a computer cause extreme fatigue for individuals suffering from these conditions and they truly are not designed with their specific needs in mind, these individuals are willing to devote the extra time it takes to learn and train the tools. Harada et al. [18] point out the positive attitude these users have towards taking the time to learn new tools, "a number of people with disabilities…are willing to invest more time in learning a system, especially if there are no other alternatives due to situational, monetary, or availability constraints." Data from experiments performed by Dai et al. [10] confirms Harada et al.'s [18] claim. Dai et al. [10] presented a voice-controlled navigation system using grids, and the participants in the experiment found the grid-based solution to be fast, accurate, and comfortable. Conversely, Wobbrock et al. [43] argue that the "burden of change" should fall on the system rather than the user, and designers should focus on ability-based design resulting in adaptive systems (discussed further in Section 4). Similar to Wobbrock et al., Gibson [14] complains that "assistive

technology is not always able to interpret the user interaction model" implying that the system is not adapting to the user appropriately. Until individuals with impairments become the developers of the tools, "[a]ccommodating diverse human perceptual, cognitive, and motor abilities [will be] a challenge to every designer" [35].

Many legacy applications (e.g., Scratch, LabVIEW, and App Inventor) were not built with a Vocal User Interface (VUI) in mind; therefore, adding a VUI is complicated. Despite the need for "out of the box" solutions to lower barriers for those with impairments [43], an "out of the box" voice recognition system (e.g., Dragon Naturally Speaking) will not integrate with these legacy applications without a bridge to connect the voice recognition system to the legacy application. This bridge needs to connect vocal commands to those the user sees on the screen. Moreover, the voice recognition system must be able to accommodate the behavior of the application, which is a far more difficult process. There is a significant difference in adding a "click" or "drag and drop" action than an action to "drag and drop" one block within another.

4. Previous Work in Voice-Driven Control of Programming Environments

The key to creating a successful, universally usable tool is to apply an ability-based design, in which developers strive to take advantage of what abilities the user possesses and to make the system adapt to the user, rather than make the user adapt to the system [43]. This should be a general design strategy for systems/tools/applications for all users, not just those with impairments. To ensure a design is ability-based rather than disability-based, Wobbrock et al. [43] developed seven principles: Ability, Accountability, Adaptation, Transparency, Performance, Context, and Commodity (See Table 2 for more detailed information on these seven principles).

STANCE	**1. Ability.**	Designers will focus on ability not *dis*-ability, striving to leverage all that users *can* do.	*Required*
	2. Accountability.	Designers will respond to poor performance by changing systems, not users, leaving users as they are.	*Required*
INTERFACE	**3. Adaptation.**	Interfaces may be self-adaptive or user-adaptable to provide the best possible match to users' abilities.	*Recommended*
	4. Transparency.	Interfaces may give users awareness of adaptations and the means to inspect, override, discard, revert, store, retrieve, preview, and test those adaptations.	*Recommended*
SYSTEM	**5. Performance.**	Systems may regard users' performance, and may monitor, measure, model, or predict that performance.	*Recommended*
	6. Context.	Systems may proactively sense context and anticipate its effects on users' abilities.	*Recommended*
	7. Commodity.	Systems may comprise low-cost, inexpensive, readily available commodity hardware and software.	*Encouraged*

Table 2. Seven principles for ability-based design [43].

Wobbrock et al. [43] presented a strong case for utilizing low-cost technology, "cost, complexity, configuration, and maintenance of specialized hardware and software are perpetual barriers" for technology use. Additionally, they observed that less than 60% of users needing "access technologies" actually use them. Some of these devices can cost thousands of dollars, and it may be unreasonable for the users to be required to spend that much money over and above any medical costs they have. Voice-driven tools offer a modality for input that is often cost effective and addresses the needs of a broad range of users who have motor challenges, but are still cognitively capable and have an ability to speak (please note: This specific qualifier does not consider in scope those issues faced by individuals with other impairments, such as speech or vision impairments).

4.1 Programming by Voice (PBV)

Programming by voice is not a radical concept. Prior efforts, exemplified by [5, 10, 11, 18, 34], present some form of a voice-driven application. Begel [5] and Désilets et al. [11] discussed voice-controlled applications for textual programming, whereas Dai et al. [10] and Harada et al. [18] focused on voice-driven cursor control techniques. Dai et al. [10] evaluated a voice-driven math program for the visually-impaired, but experienced the same issues as Begel [5] and Désilets et al. [11]. The goal for Begel [5] was a design based on "natural verbalization." The problem that both Begel [5] and Désilets et al. [11] experienced is that code and natural language are far different, and algorithms had to be created to provide the required flexibility a user would need. Begel [5] provided an illustrative example that highlights this challenge within the context of verbalizing a common loop structure:

```
for (int i = 0; i < 10; i++) {
}
```

If a programmer were to speak this statement aloud, it might be spoken as:

"for int i equals zero i less than ten i plus plus"

As a first effort, this appears to be a reasonable way to state the loop structure, and it is quite probable that every programmer says this in his/her head while typing a "for" loop. Unfortunately, this statement has two problems: 1. the brackets and punctuation are not dictated, and 2. some of these terms are not usual for a typical natural language processor. Instead of the above verbalization, Begel [5] explains that the voice recognition engine may understand the words spoken as:

"**4** int **eye** equals **0 aye** less **then** ten i plus plus"

Although Begel solves some of the issues, particularly the punctuation, there is still some responsibility placed on the programmer to learn the structure and terminology. Désilets et al. [11] discovered the same issue and solved it in a similar manner. There is a proper grammar the user must learn; however, this same grammar translates to any programming language. Désilets et al. [11] also interjected code navigation and error correction abilities. In the SpeechClipse tool, Shaik et al. [34] utilized a rule-based grammar in coordination with Java's Robot class to translate the commands recognized by the speech recognition engine into activities performed by the keyboard. In their approach, as the user speaks, the mouse or keyboard performs actions based on the commands (i.e., the mouse and keyboard are programmatically controlled by the Java Robot class, which is driven by parsed words spoken by the user).

Rather than understand code, Dai et al. [10] and Harada et al. [18] focused on cursor movement. Dai et al. [10] used a grid-based solution to allow the user to identify where on the screen he/she wishes to click. The grid begins as the entire screen is divided into nine sections. The user selects in which of those nine segments he/she wishes to click by verbalizing the number. That portion of the screen is then partitioned into nine sections, and again the user verbalizes the section in which he/she wishes to click. This process repeats three times until the specific item of interest is identified. Dai et al. [10] conducted

a small study that compared whether a grid-based system works best with nine cursors or with one cursor; the difference being that with nine cursors, the user makes three verbal clicks, and with the one cursor solution, the user makes four verbal clicks. The nine cursor solution was determined the fastest method.

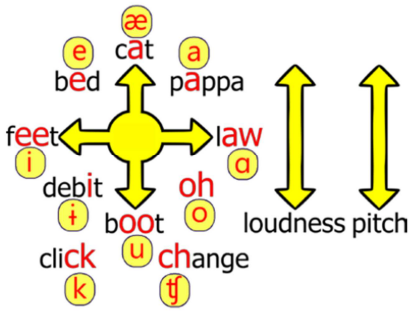

Figure 6. Vocal Joystick vowel mapping [18].

The specific approach explored by Harada et al. [18] used a very different method for cursor control. Rather than use words such as "move right, move right, stop," they identified ten syllable sounds that are mapped to direction and speed (Figure 6). The syllables are selected based on the International Phonetic Alphabet (IPA). Of the ten syllable sounds, eight are vowel sounds, and they "were chosen because they represent the eight most distinct sounds in the vowel map periphery that are present in as many of the world's major languages as possible" [18]. A disadvantage of using the syllable mapping is that it is not instinctive and some syllables are difficult to pronounce at first; therefore, it requires some training. The actual tool, called the Vocal Joystick, was developed with the goal of user-friendliness. It provides multiple methods of feedback for the user, which is particularly helpful in the training sessions. If a certain phonetic sound is difficult for a user to pronounce, the system can be modified for that user. Additionally, it is built in such a way that applications can be developed around it such as VoiceDraw, which is supported by Wobbrock et al. [43] as meeting three of their seven guidelines: adaptation, performance, and commodity. The most important aspect of the Vocal Joystick, and thus VoiceDraw, is the ability to adapt to the user rather than the user having to adapt to the system.

Several papers in the area of PBV [5, 10, 11, 18, 34] demonstrate that voice can be a reasonable alternative to the mouse and keyboard. However, an undesirable consequence is the potential for vocal strain. Haxer et al. [47] reported a case study of a doctoral candidate/lecturer who began experiencing pain due to tendonitis and decided to utilize a voice recognition program instead of her keyboard. She used voice recognition approximately four to six hours daily. Due to the heavy usage, she began to experience vocal strain and vocal fatigue. She then went to the Multidisciplinary Voice Clinic of the University of Michigan Vocal Health Center for evaluation. After performing several tests, it was determined that her vocal quality and sound were below average. She then started speech therapy where she was taught proper vocal techniques such as warming-up her vocal chords before speaking for long periods of time and hydrating periodically during vocal use. After learning how to properly exercise her vocal chords, she began using voice recognition again for four to six hours daily, and she no longer experienced vocal strain or fatigue. Moreover, she was re-evaluated, and her vocal quality and sound had improved to normal levels. De Korte et al. [48] performed a study to determine if voice recognition

in the workplace could improve posture and productivity and how user friendly this technology was perceived among participants in the study. While voice recognition did improve posture, most participants found they were more productive, and most participants found the voice recognition to be user friendly; five of the 15 participants complained about sore throats as a result of the frequent speaking.

4.2 Myna

The PBV research above indicates that voice is a viable methodology for providing accessibility for users with motor impairments. This leads to the following:

Question 3: How do we implement PBV to allow children to take advantage of creative, "drag and drop", programming environments?

Student colleagues from UAB previously created a tool affectionately named Myna, which is a Hindi term for a species of birds that are well-known for their imitative skills. Myna is a Java program that runs parallel to Scratch. When Myna is started and Scratch is opened, the user can begin creating a program within Scratch solely through voice. The grammar for the user is relatively simple as the verbal commands match the commands on the screen with the exception of select action commands (e.g., "drop after"). A prototype was created and populated with a small vocabulary to perform basic functionality. The grammar was then extended to include all commands native to the program. Myna development continues at UA with deep extensions to the grammar and capabilities related to navigation. Participants from United Cerebral Palsy (UCP) of Birmingham will soon be working with us to evaluate the application and how feasible it is for someone with Cerebral Palsy to work with a voice-driven IPE. Stevens and Edwards [37] present an approach for designing a solid evaluation for assistive technology, which will be consulted during our evaluation design process. The future interactions with UCP require a fully functional application rather than a prototype.

There are three types of navigation in Myna [42]:

1. *Drag and Drop Navigation*: This mimics the idea of clicking on an object, dragging it to another location, and dropping it. The user will say, "drag and drop" followed by the command block they wish to add to the program, and the block will be placed after the last block in the program.
2. *Continuous Navigation*: The user drives the cursor by stating, "move right" and "keep moving," which contradicts the research in [18, 43]; therefore, this might be an area for improvement.
3. *Transparent Frame Navigation*: The transparent frames allow small numbers to be placed next to commands within the program (see Figure 7). The user will state "drag" followed by the command on the desired block, and upon determining where the user wishes to place the block, he/she will state one of three macro commands ("drop before," "drop in," or "drop after") and the number from the label.

The primary design goal for Myna was to avoid invasive modification of the Scratch source code (e.g., the actual source code of Scratch itself). Myna is a separate entity that runs parallel to Scratch, as a type of monitor that sits on top of Scratch to process the voice-driven commands. This allows

Myna to be flexible in that it can be applied to other IPEs such as LabVIEW or App Inventor. This generalized capability will be discussed further in Section 5.

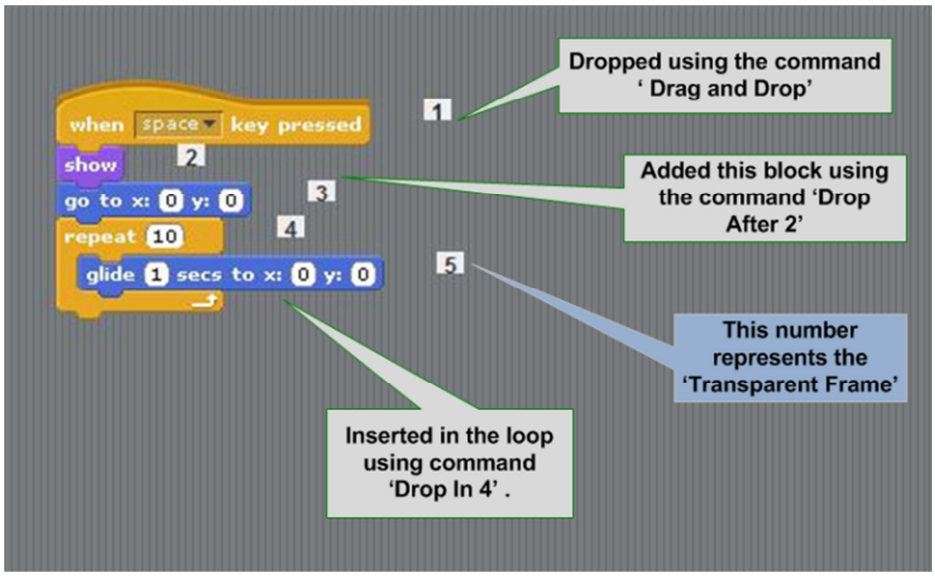

Figure 7. Image of prototype and use of transparent frames.

As the user creates a program in Scratch using Myna, the mouse moves without the need for any physical activity from the user. This is due to the Java Robot class executing the mouse and keyboard events translated by the speech recognition engine. SpeechClipse [34] was also developed using this approach. Myna allows the user to have full control over the interface with just his/her voice. The following is an example of how Myna maps the voice commands to actions (refer to Figure 8) [42]:

1. *User gives a voice command.*
2. *The input command is identified by the speech recognizer and checked against the grammar file.*
3. *If the command is present in the grammar file, an appropriate action is invoked in the Command Executor.*
4. *The Command Executor obtains the current mappings of the component.*
5. *If necessary, the Command Executor requests the Model to change its current state.*
6. *The Command Executor calls into the Java Robot [class] to perform the corresponding mouse/keyboard action.*

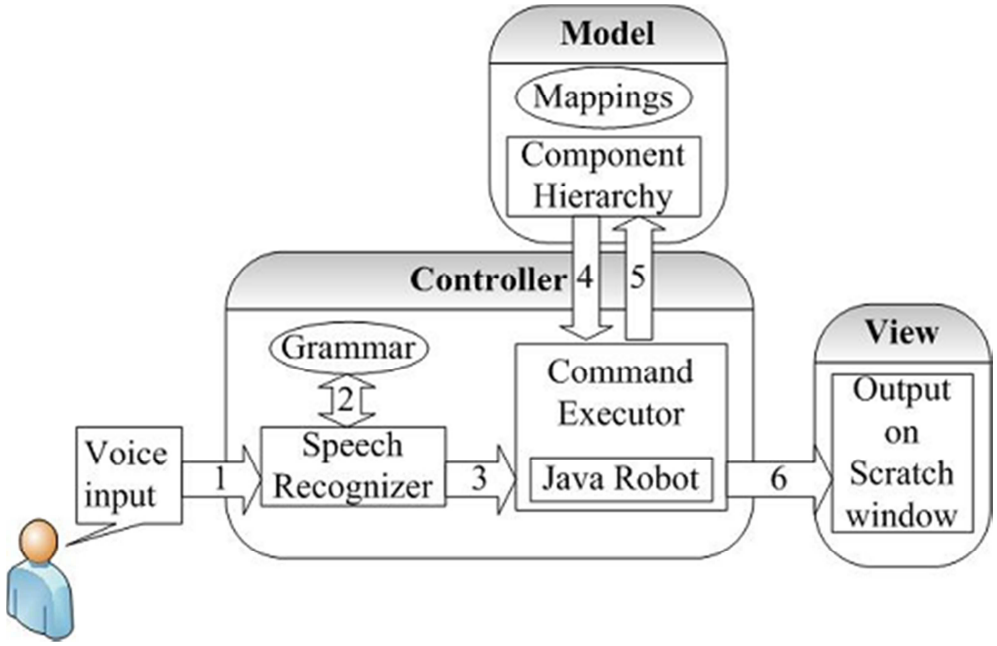

Figure 8. Myna Workflow – mapping of voice commands to actions in Myna.

5. Summary of Future Research Plans

Although there are potential advantages for the Programming By Voice idea to help children learn more about Computer Science, it is costly to customize each separate environment individually. The effort to provide PBV for Scratch is likely not possible to be reused for a similar purpose on some other environment, such as App Inventor or Mindstorms LabVIEW. This leads to the final question considered in this qualifier paper, which points toward future work in this area:

Question 4: What are some potential ideas that can assist in generalizing the process of voice enabling initial programming environments?

It is not realistic to create a new environment for every IPE that exists, or will emerge in the future. Furthermore, every time the arrangement of GUI elements changes in a new version of the IPE, the hardcoded dependencies often require much effort to re-tool the voice commands. Section 5.1 presents research on two concepts that could work together to semi-automate the voice integration process and potentially reduce the cost of creating voice-controlled solutions for existing IPEs. Then, in Section 5.2, a list of the possible contributions of this future work is enumerated.

5.1 Myna

The first step in voice-enabling Scratch through Myna was to reverse-engineer the application to determine the necessary commands. There have been several approaches proposed to reverse-engineer GUIs, based on techniques such as static analysis of source code [36], dynamic execution of the application [22, 28], and reverse engineering of system resource files [31]. Chang et al. [8] and Yeh et al. [46] present a different methodology. Their solution enables users to select regions of the screen that are mapped to scripting behavior. In relation to the desire to bring PBV to many new environments, tool developers could move through the static screens and modes of an IPE and select the GUI components

that are of interest in the IPE. The properties of these components could be stored, such as the physical coordinates of the component, the user-defined name of the component, and the type of the component. For example, the "repeat" block in Scratch could be selected from the list of controls and mapped to an internal representation.

The second idea is to utilize Model-Driven Engineering (MDE) [16, 32] to further automate the process. Aside from the static properties of the IPE screens, the dynamic behavior must also be captured. This represents the valid execution states of the interaction among various screens or modes of an application. For example, the user of a speech-enabled application should only be allowed to vocalize commands that make sense in the current context. The tool developer must specify all of the interactions among the execution paths of the IPE. A future research task associated with the PBV idea will be explored to design a domain-specific modeling language to capture such interactions. Domain-specific software environments [16] are emerging as a powerful tool for rapid development of software from higher-level visual models. Metamodeling tools have been shown to maximize reuse of domain knowledge by capturing an appropriate level of task abstraction that can be easily and efficiently used to synthesize new applications [16, 32]. The level of abstraction coupled with capabilities to generate code from higher level models seems to be a viable technique for considering the variability among different IPEs and their specific needs for those who are motorically challenged. The envisioned plan may be able to automate many of the dynamic and static configuration tasks associated with mapping a command to a speech recognition engine. A contribution could be a new process that generates the code needed to customize the grammars and Java code needed to programmatically control the mouse and keyboard with the Java Robot class. Figure 9 provides an outline of the envisioned framework to support creation of speech-enabled control of legacy applications that are highly dependent on GUIs and mouse/keyboard interfaces. This figure illustrates the scope of the future work that will be developed for a possible PhD proposal.

5.2 Overview of Possible Future Contributions from Extending Myna

Myna will be the model from which future IPEs will be integrated with voice controls. The following are the required tasks that need to be completed in order to have a fully functional PBV framework that can be instantiated for specific IPEs.

Enhancement of the Myna/Scratch User Interface: The following represent very basic engineering tasks that are necessary to support interface enhancements needed to make Myna suitable for general use:

- Currently, the Myna prototype from the initial effort does not support navigation of the horizontal and vertical scroll bars, which is needed in many cases when the program becomes large (i.e., when the Scratch scripts extend past the real estate on a single screen).
- Enabling Myna to work in all screen resolutions and in all size and shapes of the Scratch window (i.e., dynamically adjusting the properties of components, such as location and size as the user modifies the main Scratch window size).
- Including the Remove command and subsequent support for undo and redo of Scratch actions using voice commands (as implemented by the Command pattern).

Voice Recognition Flexibility: The initial Myna prototype used Sun's Java Speech API (JSAPI) [19] as the speech recognition interface. The prototype was designed such that the current speech recognition API

can be replaced easily with a new implementation to support other platforms and recognition engines. The Java Speech Grammar Format (JSGF) [20], a platform-independent and vendor-independent textual representation of grammars, was used as a standard to specify the grammar for Myna. This grammar establishes the communication path for translating commands into events.

Grammar Customization: Users may utilize a specific function more frequently and desire to change the command for that function. To achieve this functionality, a wizard will be created to allow users to customize the grammar. In addition to creating shortcuts, this will allow users to change words that they find challenging to articulate, thus, improving the adaptability of Myna [43].

Multi-lingual Support for Code Dictation: Almost all computer programming languages have reserved words that are derived from the English vocabulary (e.g., "if" "for" "begin"). Correspondingly, current PBV tools assume that the user will be speaking English to drive the recognition engine. Yet, many users (especially young children, who are the target group of this research) may feel more comfortable speaking in their native tongue such that the development environment translates their native expression into the corresponding code. Under the proposed future work, wizards for adapting the grammar to match a specific spoken language will be provided. This is not proposed as a major contribution, but such adaptability can make the resulting tools more amenable to those with disabilities who are not native English speakers. United Cerebral Palsy will help to identify a participant who is a non-native speaker (e.g., a child who is fluent in Spanish) in order to evaluate the effectiveness of multi-lingual support in IPEs.

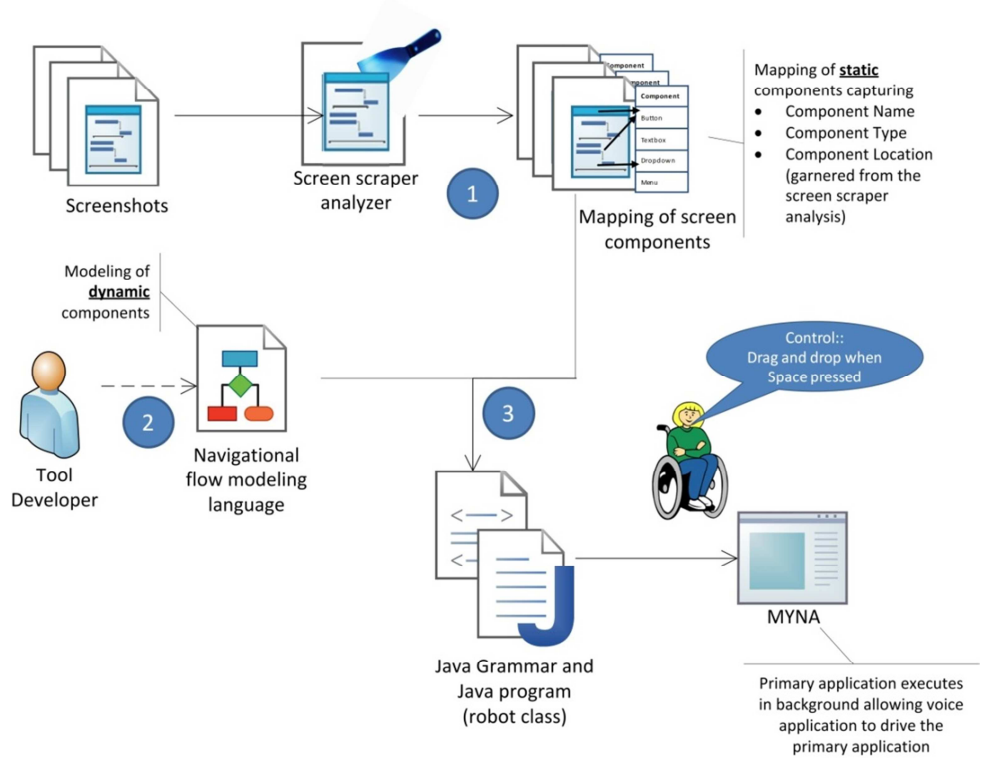

Figure 9. The Generalization of Myna for Customized Applications.

After Myna becomes a more mature application (such as meeting the guidelines set by [43]), the next phase will be to create a semi-automatic means by which voice can be integrated into other applications, such as LabVIEW and App Inventor:

Step 1 – *Importing the Static Structure of the GUI:* A first step in speech-enabling a GUI is to understand all of the widgets that are of interest and must be clickable from voice commands. Using a similar approach to [8, 46], screen shots of the tool can be collected, and the menu commands recorded and then inferred from the screenshots.

Step 2 – *Specifying the Dynamic Execution Behavior of the IPE:* A metamodel [16, 32] will be created for a language that can capture the commonalities of IPEs. This language will be informed from the initial hardcoded effort of Myna.

Step 3 – *Code Generation of Grammars and Programmatic Control Code:* After the static structure and dynamic behavior of the IPE have been obtained, it may be possible to generate the grammar needed by the speech recognition system, as well as the programmatic control of the mouse and keyboard through the Java Robot Class.

6. Conclusion

Students of all ages need to be made aware of the opportunities available in Computer Science. The best approaches that have been proposed thus far are based on programming environments designed specifically for initial learners that are inquiry-based, motivate the learning experience from a specific context, and assist in reducing the burden of learning the details of concrete syntax [4, 13, 15, 17, 21, 26, 44, 45, 54, 55, 56, 58, 60, 61, 62, 63]. While there are numerous methodologies and tools to choose from, this qualifier paper focused on Scratch, LabVIEW, and App Inventor as example IPEs. Scratch is useful for students of all ages, but has found a special niche for upper elementary to middle school students, while Berkeley's BYOB extension of Scratch is more appropriate for high school and college students due to its support for parameterized abstraction. LabVIEW is also very appropriate for upper elementary to middle school students. App Inventor can be more complicated and is recommended for high school and college students. These tools, when presented at the appropriate age, can spark a student's interest in Computer Science [44, 54]. The question that has emerged from this qualifier paper is whether ALL students of diverse physical capabilities can participate using these learning environments?

Motorically challenged students cannot use these applications due to the dependence most IPEs have on the WIMP metaphor. Wobbrock et al. [43] encourage ability-based design and support voice-driven applications if they are adaptive, perform well, and are cost effective. The research plans for this project include an adaptability feature (i.e., the grammar customization wizard) and are cost effective through adoption of open source tools or tools that are standard on many platforms.

The concept of a voice-driven IPE is an excellent tool, but can suffer from a static implementation strategy that is focused on hardcoded adaptation for each IPE that is considered. It is time consuming to fully integrate voice into each new IPE. The use of MDE combined with screen scraping techniques may offer an aid in creating a semi-automated process to generate the needed commands and the required grammar. We believe that the realization of the vision illustrated in Figure 7 would provide a capability

to create voice-controlled alternatives for GUI applications in a way that reduces or eliminates the amount of hardcoded dependencies (i.e., the intelligence of the code generator can use the configuration information of the static and dynamic information to produce what was previously implemented as a fixed-point solution). These ideas will drive the focus of the next phase of this doctoral research, leading to a proposed topic in this area.

Acknowledgements

I would like to sincerely thank my committee members: Dr. Jeff Gray, Dr. Eugene Syriani, Dr. Randy Smith, and Dr. John Lusth for their guidance through this process.

This project is supported by NSF grant IIS-1117940, with previous support from a Google Research Award.

References

[1] Abelson, H. Mobile Ramblings. *EDUCAUSE Quarterly,* vol. 34, no. 1. 2011.

[2] Alice. Carnegie Mellon University. http://www.alice.org.

[3] Android App Inventor from Google Labs (now maintained by MIT). http://www.appinventor.org.

[4] Barbosa, J., Hahn, R., Rabello, S., and Barbosa, D. LOCAL: A model geared towards ubiquitous learning. In *Proceedings of the 39th ACM SIGCSE Technical Symposium on Computer Science Education*. Portland, OR. March 2008, pgs. 432-436.

[5] Begel, A. Programming by voice: A domain-specific application of speech recognition. In *AVIOS Speech Technology Symposium – SpeechTek West*, February 2005.

[6] BYOB 3.1. University of California, Berkeley. http://byob.berkeley.edu.

[7] Center for Disease Control and Prevention. Muscular Dystrophy. Retrieved July 9, 2012. http://www.cdc.gov/ncbddd/musculardystrophy/data.html.

[8] Chang, T., Yeh, T., and Miller, R. GUI testing using computer vision. . In *Proceedings of the 28th ACM SIGCHI International Conference on Human Factors in Computing Systems*. 2010, pgs. 1535-1544.

[9] College Board. Retrieved August 10, 2012 from http://professionals.collegeboard.com/profdownload/AP-Exam-Volume-Change-2011.pdf.

[10] Dai, L., Goldman, R., Sears, A., and Lozier, J. Speech-based cursor control: A study of grid-based solutions. In *Proceedings of the 6th International ACM SIGACCESS conference on Computers and Accessibility.* Atlanta, GA. October 2004, pgs. 94-101.

[11] Désilets, A., Fox, D., and Norton, S. VoiceCode: An innovative speech interface for programming-by-voice. In *Proceedings of the 24th International Conference on Human Factors in Computing Systems*. Montréal, Québec, Canada. April 2006, pgs. 239-242.

[12] Eclipse. ADT plugin for Eclipse. Retrieved August 20, 2012 from http://developer.android.com/tools/sdk/eclipse-adt.html.

[13] Fenwick Jr., J., Kurtz, B., and Hollingsworth, J. Teaching mobile computing and developing software to support Computer Science education. In *Proceedings of the 42nd ACM SIGCSE Technical Symposium on Computer Science Education*. Dallas, TX. March 2011, pgs. 589-594.

[14] Gibson, B. Enabling an accessible web 2.0. In *Proceedings of the 2007 International Cross-Disciplinary conference on Web accessibility (W4A)*, Banff, Canada. August 2007, pgs. 1-6.

[15] Goadrich, M. and Rogers, M. Smart smartphone development: iOS versus Android. In *Proceedings of the 42nd ACM SIGCSE Technical Symposium on Computer Science Education*. Dallas, TX. March 2011, pgs. 607-612.

[16] Gray, J., Tolvanen, J.P., Kelly, S., Gokhale, A., Neema, S., and Sprinkle, J. "Domain-Specific Modeling," *Handbook on Dynamic System Modeling*, (Paul Fishwick, ed.), CRC Press, ISBN: 1584885653, 2007.

[17] Guzdial, M. and Ericson, B. *Introduction to Computing and Programming in Python, A Multimedia Approach.* Pearson Education, 2009.

[18] Harada, S., Wobbrock, J., Malkin, J., Bilmes, J., and Landay, J. Longitudinal study of people learning to use continuous voice-based cursor control. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*. Boston, MA. April 2009, pgs. 347-356.

[19] JSAPI, "The Java Speech Application Programming Interface." http://java.sun.com/products/javamedia/speech. January 2004.

[20] JSGF, "The Java Speech Grammar Format." http://java.sun.com/products/javamedia/speech/forDevelopers/JSGF. January 2004.

[21] Kelleher, C. and Pausch, R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. In *ACM Computing Surveys (CSUR)*, vol. 37, no. 2. June 2005, pgs. 83-137.

[22] Kumar, N. and Sasikumar, M. Automatic generation of speech interface for GUI tools/applications using accessibility framework. In *Techshare India*, 2008 – Breaking the Barriers Conference, February 2008.

[23] Lego Mindstorms. http://mindstorms.lego.com.

[24] Lego Mindstorms Software. http://www.legoeducation.us/eng/product/lego_mindstorms_education_nxt_software_2_1/2240.

[25] Loveland, S. Human computer interaction that reaches beyond desktop applications. In *Proceedings of the 42nd ACM SIGCSE Technical Symposium on Computer Science Education*. Dallas, TX. March 2011, pgs. 595-600.

[26] Mahmoud, Q. Best practices in teaching mobile application development. In *Proceedings of the 16th ITiCSE Joint Conference on Innovation and technology in Computer Science education*. Darmstadt, Germany. June 2011, pg. 333.

[27] Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., and Resnick, M. Scratch: A sneak preview. In *The 2nd International Conference on Creating, Connecting, and Collaborating through Computing*. Kyoto, Japan. January 2004, pgs. 104-109.

[28] Memon, A., Banerjee, I., and Nagarajan, A. GUI ripper: Reverse engineering of graphical user interfaces for testing. In *International Journal of Man-Machine Studies*, vol. 21, Issue 6, December 1984, pgs. 493-520.

[29] National Institute of Neurological Disorders and Stroke. National Institutes of Health. http://www.ninds.nih.gov/disorders/disorder_index.htm.

[30] Lenhart, A.*Teens, Smartphones & Texting*. Retrieved August 21, 2012 from http://pewinternet.org/Reports/2012/Teens-and-smartphones/Cell-phone-ownership/Smartphones.aspx. ResHacker, 2010. http://download.cnet.com/Resource-Hacker/3000-2352_4-10178588.html.

[31] ResHacker, 2012. http://download.cnet.com/Resource-Hacker/3000-2352_4-10178588.html.

[32] Rosado da Cruz, A. M. and Pascoal Faria, J. A metamodel-based approach for automatic user interface generation. In *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Oslo, Norway. October 2010, pgs. 256-270.

[33] Scratch. Massachusetts Institute of Technology. http://scratch.mit.edu.

[34] Shaik, S., Corvin, R., Sudarsan, R., Javed, F., Ijaz, Q., Roychoudhury, S., Gray, J., and Bryant, B. SpeechClipse – An Eclipse speech plug-in. In *Eclipse Technology eXchange Workshop (OOPSLA)*, Anaheim, CA. October 2003.

[35] Shneiderman, B. and Plaisant, C. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, 5th Edition*. Addison-Wesley, Boston, MA, 2010.

[36] Staiger, S. Reverse engineering of graphical user interfaces using static analyses. In *Working Conference on Reverse Engineering,* Vancouver, British Columbia, Canada, October 2007, pgs. 189-198.

[37] Stevens, R. D. and Edwards, A. D. N. An approach to the evaluation of assistive technology. In *Proceedings of the 2nd ACM Conference on Assistive Technologies*. Vancouver, British Columbia, Canada. April 1996, pgs. 64-71.

[38] Trewin, S. Physical usability and the mobile web. In *Proceedings of the 2006 International Cross-Disciplinary conference on Web accessibility (W4A)*, Edinburgh, UK. May 2006, pgs 109-112.

[39] U.S. Census Bureau. Retrieved August 20, 2012. http://quickfacts.census.gov/qfd/states/00000.html.

[40] U.S. Census Bureau. Age and Sex Composition: 2010. 2010 Census Briefs, May 2011. Retrieved July 9, 2012. http://www.census.gov/prod/cen2010/briefs/c2010br-03.pdf.

[41] US Census Bureau. 2009. CPS October 2009 – Detailed Tables. Retrieved August 20, 2012 from http://www.census.gov/hhes/school/data/cps/2009/tab01-01.xls.

[42] Wagner, A., Rudraraju, R., Datla, S., Banerjee, A., Sudame, M., and Gray, J. Programming by voice: A hands-free approach for motorically challenged children. In *Proceedings of the 30th ACM SIGCHI International Conference on Human Factors in Computing Systems*. Austin, TX. May 2012, pgs. 2087-2092.

[43] Wobbrock, J., Kane, S., Gajos, K., Harada, S., and Froelich, J. Ability-based design: Concept, principles, and examples. In *ACM Transactions on Accessible Computing*, vol. 3, no. 3, article 9. April 2011.

[44] Wolber, D. App Inventor and real-world motivation. In *Proceedings of the 42nd ACM SIGCSE Technical Symposium on Computer Science Education*, Dallas, TX, March 2011, pgs. 601-606.

[45] Wolber, D., Abelson, H., Spertus, E., and Looney, L. *App Inventor: Create Your Own Android Apps*. O'Reilly, 2011.

[46] Yeh, T., Chang, T., and Miller, R. Sikuli: Using GUI screenshots for search and automation. In *ACM Symposium on User Interface Software and Technology,* October 2009, pgs. 183-192.

[47] Haxer, M., Guinn, L., and Hogikyan, N. Use of speech recognition software: A vocal endurance test for the new millennium? In *Journal of Voice*, Vol. 15, No. 2, 2001, pgs. 231-236.

[48] De Korte, E. and Van Lingen, P. The effect of speech recognition on working postures, productivity, and the perception of user friendliness. In *Applied Ergonomics*, Vol. 37, 2006, pgs. 341-347.

[49] McIver, L. Grail: A zeroth programming language. In *Conference in Computers in Education*. 1999.

[50] Association for Computing Machinery (1992). *ACM code of ethics and professional conduct.* Retrieved August 12, 2012 from http://www.acm.org/constitution/code.html.

[51] National Instruments, LabVIEW. http://www.ni.com/LabVIEW/.

[52] Samsung Solve for Tomorrow Contest, http://explore.appinventor.mit.edu/stories/award-winning-wild-hog-tracker.

[53] Gal-Ezer, J. and Stephenson, C. The current state of computer science in U.S. high schools: A report from two national surveys. In *Journal for Computing Teachers*, Spring 2009.

[54] Rodger, S. H., Hayes, J., Lezin, G., Qin, H., Nelson, D., Tucker, R., Lopez, M., Cooper, S., Dann, W., and Slater, D. Engaging middle school teachers and students with alice in a diverse set of subjects. In *Proceedings of the 40th ACM technical symposium on Computer science education SIGCSE 2009,* March 2009, pgs. 271-275.

[55] Malan, D. and Leitner, H. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education SIGCSE 2007*, March 2007, pgs. 223-227.

[56] Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[57] Kay, A. The early history of smalltalk. In *ACM SIGPLAN Notices.*1993, Vol. 28, No. 3, pgs. 69-96.

[58] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. Scratch: Programming for all. In *Communications of the ACM*. November 2009, Vol. 52, No. 11, pgs. 60-67.

[59] Scratch Statistics. Retrieved August 22, 2012 from http://stats.scratch.mit.edu/community/.

[60] Ranganathan, P., Schultz, R., and Mardani, M. Use of Lego NXT Mindstorms brick in engineering education. In *Proceedings of the 2008 ASEE North Midwest Sectional Conference*. October 2008.

[61] Uludag, S., Karakus, M., and Turner, S. W. Implementing IT0/CS0 with Scratch, App Inventor for Android, and Lego Mindstorms. In *SIGITE* October 2011, pgs. 183-189.

[62] Gray, J., Abelson, H., Wolber, D., and Friend, M. Teaching CS principles with App Inventor. In ACMSE March 2012.

[63] Roy, K. App Inventor for Android: Report from a summer camp. In SIGCSE February 2012, pgs. 283-288.