UNIVERSITY OF CALIFORNIA
Santa Barbara

Enabling Wide-Scale Computer Science Education through Improved Automated Assessment Tools

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Bryce A. Boe

Committee in Charge:

Dr. Diana Franklin, LSOE, Chair

Professor Timothy Sherwood

Professor Danielle Harlow

September 2014

The Dissertation of
Bryce A. Boe is approved:

Professor Timothy Sherwood

Professor Danielle Harlow

Dr. Diana Franklin, LSOE, Committee Chairperson

September 2014

Enabling Wide-Scale Computer Science Education through Improved Automated Assessment Tools

by

Bryce A. Boe

To my brother, Daniel Jacob Boe. You will forever be missed.

Acknowledgements

The path to my Ph.D. was a long and tiresome one. I would never have reached this point in my life without the love, influence, and support from other people.

First and foremost, thank you Diana Franklin for taking me under your wing these last few years. I could not have performed the research and completed this dissertation without your patience and guidance.

The loss of my brother nearly resulted in me abandoning the completion of my Ph.D. I would have done so without the generosity of Klaus Schauser who mentored me and provided me funding through Appfolio, Inc. during the final stage of my Ph.D. Thank you Klaus, and thank you Appfolio, Inc.

Mom, thank you for providing me with the freedom to learn. Dad, thank you for showing me how the world works. Collectively, you influenced my development of both an inquisitive nature and a desire to learn. Furthermore, thank you for tolerating me breaking the computer on a plethora of occasions.

Matt, Daniel (1988–2014), and Connor, thank you for being the best brothers anyone could ask for. Buz, thank you for introducing me to HTML in 6^{th} grade, and being a great friend through all these years. Adam and Scott, thank you for the camaraderie, support, and encouragement during our computer science education. To all the gentlemen of the GALAGA house, both past and present, thank you for all the great events and outings. I am eternally grateful to have met such amazing friends.

Tim, thank you for organizing the 2004 programming battle and subsequently mentoring me throughout my college education. Andy, thank you for the encouragement and all the challenging tasks you entrusted to me at WorldViz, LLC. Cha and Bob, thank you for convincing me to apply to the Ph.D. program by attempting to discourage me from doing so. Janet and Peter, and Cheri and Tom, thank you for the unrelenting support and confidence in my success. Thank you to all my past teachers who challenged me and encouraged me to succeed.

Sharon and George, thank you for the love, the support, and all the delicious home cooked meals over the last year. My new sisters, brothers, nieces, nephews, aunts, uncles, and cousins, thank you making me feel welcome, for the support when my brother died, and for the encouragement to complete this Ph.D. adventure.

Finally, Julie, my love, thank you for the incredible support during the final year of my Ph.D. I would not have made it through this last year without your patience, understanding, and love. The next chapter of my life belongs to you.

Curriculum Vitæ

Bryce A. Boe

Education

2014	Doctor of Philosophy in Computer Science University of California, Santa Barbara
2013	Master of Science in Computer Science University of California, Santa Barbara
2008	Bachelor of Science in Computer Science University of California, Santa Barbara

Publications

July 2013	Hilary Dwyer, Bryce Boe, Charlotte Hill, Diana Franklin, and Danielle Harlow "Computational Thinking for Physics: Programming Models of Physics Phenomenon in Elementary School" In *Proceedings of the 2013 Physics Education Research Conference (PERC 2013)*, Portland, OR
March 2013	Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin "Hairball: Lint-inspired Static Analysis of Scratch Projects" In *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2013)*, Denver, CO

March 2013 — Diana Franklin, Phillip Conrad, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, Gerardo Aldana, Paulo Almeida-Tanaka, Brynn Kiefer, Chelsea Laird, Felicia Lopez, Christine Pham, Jessica Suarez, and Robert Waite "Assessment of Computer Science Learning in a Scratch-Based Outreach Program" In *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2013)*, Denver, CO

October 2011 — Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna "Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities" In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, Chicago, IL

July 2010 — Nick Childers, Bryce Boe, Lorenzo Cavallaro, Ludovico Cavedon, Marco Cova, Manuel Egele, and Giovanni Vigna "Organizing Large Scale Hacking Competitions" In *Proceedings of the 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2010)*, Bonn, Germany

April 2009 — Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao "User Interactions in Social Networks and their Implications" In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys 2009)*, Nuremberg, Germany

August 2008	Gayatri Swamynathan, Christo Wilson, Bryce Boe, Kevin C. Almeroth, and Ben Y. Zhao "Do Social Networks Improve e-Commerce?: A Study on Social Marketplaces" In *Proceedings of 1st Workshop on Online Social Networks (WOSN 2008)*, Seattle, WA

Fields of Study

2011 – 2014	Computer Science Education with Dr. Diana Franklin, LSOE
2009 – 2011	Computer Security with Professor Giovanni Vigna and Professor Christopher Kruegel
2008	Computer and Social Networking with Professor Ben Y. Zhao

Awards

2012	Outstanding Computer Science Teaching Assistant, College of Engineering, UCSB
2011	Outstanding Teaching Assistant, UCSB Academic Senate
2009	Outstanding Computer Science Teaching Assistant, College of Engineering, UCSB
Spring 2009	Outstanding Teaching Assistant, Computer Science Department, UCSB

Winter 2009	Outstanding Teaching Assistant,
	Computer Science Department, UCSB

Teaching Experience

Winter 2014	Teaching Assistant, CS24 Problem Solving with Computers II,
	Dr. Diana Franklin, LSOE
Fall 2013	Instructor, CS24 Problem Solving with Computers II
Summer 2013	Instructor, CS24 Problem Solving with Computers II
Summer 2012	Instructor, CS32 Object Oriented Design and Implementation
Winter 2012	Teaching Assistant, CS24 Problem Solving with Computers II,
	Dr. Diana Franklin, LSOE
Fall 2011	Instructor, CS501 Teaching Assistant Training
Spring 2011	Teaching Assistant, CS170 Operating Systems,
	Professor Ben Y. Zhao
Fall 2009	Instructor, CS501 Teaching Assistant Training
Spring 2009	Teaching Assistant, CS170 Operating Systems,
	Professor Christopher Kruegel
Winter 2009	Teaching Assistant, CS160 Compilers,
	Professor Timothy Sherwood

Professional Experience

2005 – 2014	Software Development and Networking Consultant, WorldViz, LLC.
Summer 2011	Software Engineering Intern, Appfolio, Inc.
Summer 2009	Software Engineering Intern, Appfolio, Inc.
Summer 2008	Software Engineering Intern, Google
2005 – 2006	Software Developer, VCEL, Inc.
2004	Computer Support Intern, Northrop Grumman

Abstract

Enabling Wide-Scale Computer Science Education through Improved Automated Assessment Tools

Bryce A. Boe

There is a proliferating demand for newly trained computer scientists as the number of computer science related jobs continues to increase. University programs will only be able to train enough new computer scientists to meet this demand when two things happen: when there are more primary and secondary school students interested in computer science, and when university departments have the resources to handle the resulting increase in enrollment. To meet these goals, significant effort is being made to both incorporate computational thinking into existing primary school education, and to support larger university computer science class sizes. We contribute to this effort through the creation and use of improved automated assessment tools.

To enable wide-scale computer science education we do two things. First, we create a framework called Hairball to support the static analysis of Scratch programs targeted for fourth, fifth, and sixth grade students. Scratch is a popular building-block language utilized to pique interest in and teach the basics of computer science. We observe that Hairball allows for rapid curriculum alterations and thus contributes to wide-scale deployment of computer science curriculum. Second, we create a real-time feedback

and assessment system utilized in university computer science classes to provide better feedback to students while reducing assessment time. Insights from our analysis of student submission data show that modifications to the system configuration support the way students learn and progress through course material, making it possible for instructors to tailor assignments to optimize learning in growing computer science classes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A recent report by code.org suggests that by 2020 there will be a one million person gap in the United States between the number of vacant computer science jobs and the number of computer scientists available to fill these jobs [9]. The lack of newly trained computer scientists is a twofold problem:

- There are not enough students interested in computer science coming out of high school, evidenced by the fact that of the 2.1 million students nationwide who took AP exams in 2013, only 31,000 (1.4%) students took the AP Computer Science exam [40]. The low numbers of high school students studying computer science could be due to the absence of availability of computer science-related instruction in areas such as computational thinking and elementary programming in primary through secondary curricula. The Computer Science Teachers Association acknowledges this problem in their 2011 K–12 Computer Science Standards where

they detail the path to resolution requiring the incorporation of computer science concepts beginning in primary school [18].

- At this point in time, the university system does not have the resources to adequately handle the rapid, yet insufficient, increase in numbers of students applying to computer science programs. The future does not look much more encouraging, as departments may not be able to support computer science enrollment growth through an increase in resources including faculty size. For example, Lazowska et al. report that while the student body of both Princeton and MIT comprises more than 10% computer science majors, it is unlikely that 10% of the total university faculty will ever be part of computer science [28]. The unfortunate result is that university computer science departments are turning away a significant number of qualified students from a discipline where they are severely needed.

This dissertation describes methods I developed along with my colleagues that have a positive impact on solving both the challenge to get more students interested in computer science coming out of high school, as well as the challenge to support the growth in number of university-level computer science students. My work focuses on ways to help increase student interest in computer science by introducing a 4^{th}–6^{th} grade Scratch-based computational thinking curriculum by supporting rapid curriculum de-

velopment via a design-based research approach. The upcoming wide-scale deployment of this curriculum in California will reach more than a thousand young students in the next year alone, with more reach in subsequent years. My work also investigates the important issue of how to increase the number of university-level computer science students through the incorporation of a real-time feedback and assessment system into existing computer science curriculum. My goal is for the system to reduce the amount of time that instructors currently devote to the labor-intensive assessment process, thereby making more time available to spend with students who need extra assistance. My research, in combination with future efforts, has the promise to enable the university system to produce more computer scientists.

1.1 Thesis Statement

The increase in interest in computer science has resulted in a need to scale computer science instruction from the primary school grades through undergraduate level university programs. These two ends of the spectrum, however, are in far different places in their development, with very little curriculum existing for primary schools and mature curriculum available at the university level. Assessment automation can greatly enhance both efforts by allowing us to understand certain important aspects of student learning and behavior. At the primary school level, assessment automation

through static analysis of student work can provide instructors with insight into student comprehension, enabling rapid curriculum changes that result in faster deployment of new curriculum. At the university level, automated feedback and assessment systems provide large numbers of students with immediate insight into their success with class assignments allowing them to iteratively achieve mastery of course topics, and reduce assessment time, permitting instructors to focus their efforts on students in need of assistance.

1.2 Dissertation Overview

The remainder of this dissertation is structured as follows. To promote young students' interest in computer science — and hopefully their continued interest through secondary levels — I investigate the use of static analysis in the curriculum development and assessment processes for 4^{th}–8^{th} grade students that focuses on computational thinking and introductory programming. Specifically, in Chapter 2 I look at the use of static analysis to assist with the *post assessment* of five Scratch assignments given in a two-week Scratch-based *summer camp* for 6^{th}–8^{th} grade students. In Chapter 3 I extend the use of static analysis of Scratch assignments to aid in the *development* of a 4^{th}–6^{th} grade *classroom-based computational thinking curriculum*. In an effort to support increasing numbers of computer science students in university level classes, in

Chapter 4 I look at *submission behaviors* of undergraduate computer science students in the presence of my real-time feedback and assessment system. Finally, in Chapter 5 I summarize my findings and discuss the impact of my research on computer science education at both the primary and university instructional levels.

Chapter 2

Using Static Analysis to Assist with the Post-Assessment of a Scratch-based 6^{th}–8^{th} Grade Summer Camp

In this chapter, I look at the significant role that static analysis plays in the evaluation of the overall effectiveness of our two-week Scratch-based 6^{th}–8^{th} grade summer camp.[1] Static analysis is a technique for automatically analyzing computer programs to gain insights into properties such as correctness, soundness, and simplicity. Scratch is a building-block programming language designed for kids which allows them to create programs in a manner similar to how they would construct physical structures with LEGO®. I apply static analysis to Scratch programs created by 6^{th}–8^{th} grade students in order to assess student success with camp assignments.

[1] The content of this chapter was published in SIGCSE 2013 under the title "Hairball: Lint-inspired Static Analysis of Scratch Projects". http://dx.doi.org/10.1145/2445196.2445265

2.1 Introduction

There is a movement toward both more interactive and more engaging assignments and languages for introductory and AP computer science courses. This movement includes the push for Python with Multimedia approaches, the various approaches to the AP Computer Science Principles course, as well as Alice and Scratch [1, 11, 19, 30, 35, 36].

One drawback of assignments written in building-block programming languages such as Alice and Scratch is that their evaluation can be more difficult than traditional text-based programming assignments. A common and straightforward practice in evaluating text-based assignments is to perform functional testing. That is, to write a script to run all submitted programs and compare their output with solution files [26]. More recently, unit-testing frameworks have been employed as part of automated assessment [15, 38]. When students are given creative freedom with a sensory assignment — an integral feature of languages such as Alice and Scratch — there is neither a text-based output file to compare to an expected output, nor a straightforward way to perform unit-testing. For example, Scratch evaluation typically requires that each Scratch program be individually opened and run. Inspection of Scratch program code requires many mouse clicks and navigation through a number of Scratch objects including the *stage* and all *sprites* as well as the associated *scripts* of each.

stage A single background object in Scratch that is otherwise nearly identical to a *sprite*.

sprite An object in Scratch. Any number of *sprites* can be added to a Scratch program each of which has its own set of attributes (e.g., position and orientation) and can be associated with any number of *scripts*.

script A series of one or more executable code statements (a *block*). Each script is associated with either a *sprite* or the *stage*.

To assist with assessment of Scratch programs, we propose a static analysis tool. Inspired by the Scratch mascot, a cat, and the concept of lint, a static analysis utility for C that looks for potential defects with program code, we call our system Hairball [27]. We propose two roles for Hairball:

formative assessment Black and Wiliam broadly define formative assessment as, "all those activities undertaken by teachers, and/or by their students, which provide information to be used as feedback to modify the teaching and learning activities in which they are engaged" [5]. Inspired by lint, we envision students will use Hairball as a form of formative assessment by receiving feedback on potential problems in the Scratch programs they are working on.

summative assessment Summative assessment generally refers to an overall assessment of a course or an assignment. In our context, researchers and instructors

can accelerate manual analysis of Scratch programs required for summative assessment by using Hairball to verify the presence and correct use of required computer science constructs within their students' Scratch programs.

We developed a plugin architecture so that, in Python, Hairball can be extended and adapted for evaluation of specific assignments, and tested Hairball on fifty-eight assignments created by 6^{th}–8^{th} grade students during our two-week Scratch-based summer camp in 2012.

The challenges we explore in this chapter relate to where the line should be drawn between what Hairball can do with static analysis, and where manual examination of the Scratch program is necessary. We find that each has its own strengths. Hairball can quickly differentiate between Scratch programs that do, or do not, contain certain targeted constructs. Hairball is also particularly helpful for identifying instances of various constructs and implementations that are not robust, but may not immediately cause obvious errors at runtime. Manual analysis, however, is still needed to evaluate the overall aesthetic effect and cohesion of a visual or auditory assignment.

We begin in Section 2.2 by providing a background on automated analysis in general and for Scratch in particular. We describe our Hairball framework in Section 2.3. The Hairball plugins we developed for our analysis are described in Section 2.4. We describe our methodology in Section 2.5, and results in Section 2.6. Finally, in Section 2.7 we conclude.

2.2 Related Work

Providing automation for analyzing traditional programs is not a new concept. Both ASSYST and Marmoset are automated assessment systems that perform end-to-end, or input/output, type testing of submissions [26, 38]. Web-CAT performs testing of code using student written unit tests [15]. All of the aforementioned tools supplement the feedback students receive with code coverage analysis and feedback from static analysis tools such as FindBugs by Cole et al. [10]. Douce et al. performed a more detailed analysis of existing automated assessment systems [14]. The problem with these existing systems is that they are not applicable to Scratch programs.

Scratch is a block-based programming language from MIT [30]. Scratch programs consist of two-dimensional interactive animations. Objects, or *sprites*, move on the screen as a result of either user input or the execution of scripts in a Scratch program. Sound and video can also be integrated into Scratch programs. Scratch was designed to allow students to learn computer science programming while employing great creativity in their work. This creative freedom is one of the reasons that Scratch programs are challenging to analyze.

An additional challenge in Scratch analysis compared to typical programming language analysis is that Scratch programs are developed and run within a graphical user interface. Rather than producing an easy to analyze text file, independent segments of

code, known as *scripts*, are associated with Scratch *sprites*, e.g., the Scratch Cat, and tied to a triggering event. There is no central *main* point of execution. Instead, Scratch programs might begin when a parallel set of *scripts* beginning with a *when green flag clicked* hat block are triggered.

Little prior work has looked at automated Scratch analysis. Adams and Webster describe using scripts and custom modifications to the Squeak source code of Scratch to perform their quantitative analysis of Scratch programs from the Imaginary Worlds summer camp [1]. Additionally, Burke and Kafai developed Scrape as a visualization tool to aid humans in understanding patterns across Scratch programs [42]. Scrape was used to assess Scratch programs produced in a middle school writing workshop [8]. Scrape is useful in answering questions such as:

- How many Scratch programs use loops?
- How many loops are present in each Scratch program?
- What level of nesting does the Scratch program use?

Hairball has two purposes — it quantifies the appearance of computer science constructs (e.g., loops, conditionals, nesting) like Scrape, and it also permits instructors to gain insight how students understand these constructs. We want to answer questions not just about the use of computer science constructs, but about the competence demon-

strated for different computer science concepts. Hairball can answer questions similar to those mentioned above and additionally can be used to answer questions such as:

- Which Scratch programs contain unmatched *broadcast EVENT* and *when I receive EVENT* blocks?
- Which Scratch programs contain broadcast and receive events that result in infinite loops?
- Which Scratch programs do not properly initialize the start state?
- Which Scratch programs do not properly implement complex animations (requiring the application of timing, costume change, motion, and loops)?

2.3 Design

We have two goals in designing Hairball. Our first goal is to perform analysis on a set of Scratch programs automatically. Without automated analysis, inspection and execution requires opening each Scratch program manually. This manual process is time-consuming and error-prone. Our second goal is that Hairball is easily extendable so that new Scratch analysis plugins can be created with only a basic amount of Python experience. Moreover, anyone should be able to make use of available plugins.

2.3.1 Plugin Architecture

We used the object-oriented features of Python to develop a base class from which Hairball plugins can be derived. Python was chosen due to the authors' experience with Python and its increased adoption in introductory computer science classes. A strong contributing factor to this decision is the open source Python package *Kurt* that provides simple access to all the elements contained within a Scratch program, i.e., the images, sounds, stages (backgrounds), sprites and most importantly the scripts [33].[2]

Implementing a Hairball plugin simply requires extending the base class and overloading a single method. The method's sole parameter is a handle to the Scratch program from *Kurt*. The method should return a dictionary containing the results of the desired static analysis. In principle, any type of static analysis of a Scratch program that can be described algorithmically can be implemented as a Hairball plugin in a straightforward manner by anyone with basic Python programming skills. The following code provides an example of a simple Hairball plugin that counts the number of times each Scratch block is used in a Scratch program.

```
class BlockCounts(HairballPlugin):
  def analyze(self, scratch):
    blocks = Counter()
    for block, _, _ in iter_blocks(scratch):
      blocks.update({block: 1})
    return blocks
```

[2] As part of our work, we made a few contributions that are now a core part of the *Kurt* Python package.

2.4 Hairball Plugins

In this section, we describe four Hairball plugins written to perform Scratch program static analysis. The plugins were designed to analyze Scratch programs submitted as part of our two-week interdisciplinary Animal Tlatoque summer camp [21]. The plugins target the computer science concepts used in the camp's cumulative assignment, an interactive movie about an animal. For this assignment, students were to demonstrate state initialization, use of *broadcast EVENT* and *when I receive EVENT* blocks, synchronization between *say MESSAGE* and *play SOUND* blocks, and creation of complex animation. While these plugins were developed for our summer camp, each provides valuable feedback that is generally useful both as a lint-like tool for individual developers of Scratch programs and for others who are tasked with analyzing numerous Scratch programs.

Each Hairball plugin for the camp was designed to evaluate whether, or to what extent, the Scratch program demonstrated competence in an area. More precisely, these plugins were designed to discover instances of the aforementioned concepts contained within a Scratch program and label each instance as *correct*, *semantically incorrect*, *incorrect*, or *incomplete*. Instances labeled *correct* should indicate that the concept was implemented correctly. Instances labeled *semantically incorrect* should indicate that the concept was implemented in a way that may not always work when executed.

Instances labeled *incorrect* should indicate the concept was implemented incorrectly. Finally, instances labeled *incomplete* should indicate that only a subset of the required blocks for a concept was discovered. A single Scratch program may contain multiple instances of a concept distributed across any or all of the labels. Ideally, instances labeled *correct* should not require manual analysis, whereas instances with any other label should be inspected manually.

Initial State

In any program, correctly setting the initial state is important. In Scratch programs, the significance is different. Scratch programs are comprised of animations, and in the runtime environment, they may run from start to finish and be restarted again. Alternatively, they may be stopped in the middle and restarted again. We want to determine via static analysis whether the code runs the same way in these two events.

The first problem is where to start the analysis. In traditional programs, execution starts at *main*. However, Scratch programs have no such globally defined starting point. Therefore, we taught our students to start their Scratch program using the green flag button. Thus the starting point for our evaluation is the *when green flag clicked* block.

The most complex problem, and the problem that introduces the possibility of error into our analysis, is that sprites are placed on the stage during implementation thus giving them an implicit set of attributes, which we will refer to as the base attributes.

Explicit initialization for a particular attribute, e.g., position or orientation, is only required when that attribute is modified by a script of the Scratch program. Thus, the challenge is distinguishing segments of scripts that perform initialization from those that perform general modification. To discover instances of initialization, we first determine the set of blocks that can be considered initialization blocks and then we restrict the location within the scripts that we search for these blocks. We call this location the *initialization zone*.

Attribute modifying Scratch blocks can be labeled as *relative* or *absolute*. *Relative* Scratch blocks alter the attribute based upon its current value, whereas *absolute* Scratch blocks directly set the attribute. As such, only *absolute* blocks can be considered initialization blocks. Table 2.1 shows our categorization for a subset of attribute modifying Scratch blocks.

For an *absolute* block to be considered an initialization block, it must appear in the *initialization zone*. We define the *initialization zone* only for scripts beginning with a *when green flag clicked* block. The *initialization zone* begins at the start of the script and ends when either a *relative* block or a *broadcast EVENT* block is encountered. We take a conservative approach when encountering blocks contained within loops or conditionals—*absolute* blocks are ignored due to the possibility that the block is not executed, and *relative* blocks continue to signify the end of the *initialization zone* due to the possibility that the block is executed.

Category	Relative	Absolute
Costume	next costume	switch to costume x
Visibility		show/hide
Orientation	turn clockwise x degrees	point in direction x
Position	move x steps	go to x,y
Size	change size by x%	set size to x%
Background	next background	switch to background x

Table 2.1: Lists the five categories of initial state, and provides a subset of the *relative* and *absolute* modification blocks for each category.

The initialization plugin labels a modified attribute of a sprite as *correct* when an *absolute* block for the same attribute exists in the *initialization zone*. Instances are labeled as *incorrect* otherwise. Non-modified attributes are ignored. Finally, despite this plugin's ability to detect unnecessary initialization, we did not include it as part of our analysis.

Say and Sound Synchronization

Synchronization between a speech bubble (*say MESSAGE* block) and sound file (*play SOUND* block) is not straightforward in Scratch. The desired behavior is that whenever a speech bubble appears with a message, a sound file of a voice speaking the message plays. When the sound is complete, the speech bubble disappears.

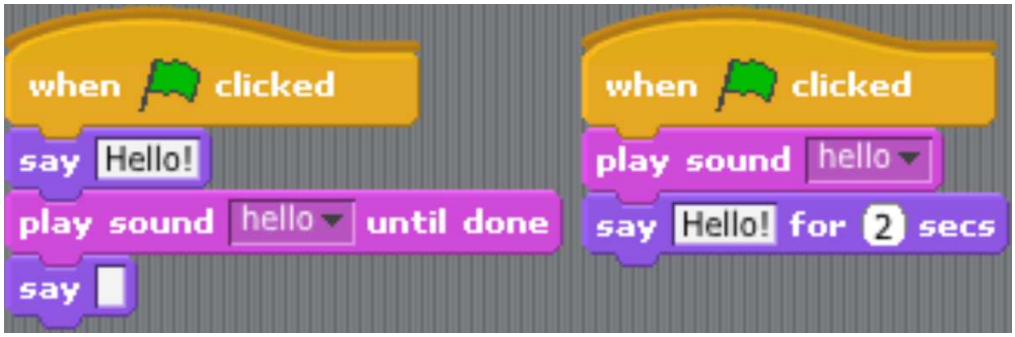

Figure 2.1: Shows the two methods for synchronizing messages from the *say MESSAGE* blocks with sound files played through the *play SOUND* blocks. While both methods can produce the desired effect, the method on the right requires manually setting an appropriate duration in the *say MESSAGE for SECONDS* block and thus is not as robust to modifications, whereas the method on the left guarantees synchronization between the play of the sound file and the display of the message.

Achieving this effect is complicated by the timing semantics of the two forms of the *say MESSAGE* block, and the two forms of the *play SOUND* block in Scratch. One form of the *say MESSAGE* block places the speech bubble on the screen indefinitely (until replaced by another *say MESSAGE* block, or *erased* with an empty *say MESSAGE* block), while the other, *say MESSAGE for SECONDS*, puts a speech bubble on the screen for n seconds and, as a side-effect, delays execution of the script for n seconds. Similarly, there are two forms of the block for playing a sound clip: *play SOUND until done* plays the entire sound file before continuing execution of the script, while *play SOUND* starts playing the sound and immediately continues with the script execution.

Figure 2.1 depicts the two methods to produce the desired effect. The first, displayed on the right, is to asynchronously play the sound via the *play SOUND* block

followed by a *say MESSAGE for SECONDS* block with duration equal to the elapsed time of the sound. Unfortunately, the timing must be manually determined and needs to be updated whenever the sound file changes. The second, displayed on the left, is to use a *say MESSAGE* block to display the message, followed by a *play SOUND until done* block, ending with an empty *say MESSAGE* block to remove the speech bubble. The campers were taught the latter method as the correct approach because it is robust to modifications of both the sound file, and to the message in the *say MESSAGE* block.

Thus the say and sound synchronization plugin detects instances of this concept by looking for sequential *say MESSAGE* and *play SOUND* blocks and verifies the instances are implemented using the appropriate method. A *correct* instance contains the previously described three blocks in the proper order. Instances following the method requiring manual timing are labeled *semantically incorrect*. Instances that have both *say MESSAGE* and *play SOUND* blocks, but do not match either of these methods are labeled *incorrect*, and isolated uses of *say MESSAGE* or *play SOUND* blocks are labeled *incomplete*.

Broadcast and Receive

One use of Scratch's *broadcast EVENT* blocks is to trigger the execution of other sprites' scripts beginning with the appropriate *when I receive EVENT* block. We taught our campers the broadcast and receive concept in the context of two animal sprites

conversing, where each sprite would signal the other's turn via an event broadcast. In the camp's cumulative assignment, the campers demonstrated an understanding of the broadcast and receive concept by transferring this idea to the new context of triggering scene changes in their interactive movie.

The broadcast and receive plugin verifies that for each broadcast or receive event, there is a *broadcast EVENT* block and at least one corresponding *when I receive EVENT* block. Such instances are labeled *correct*. All instances with a *broadcast EVENT* block appearing in the same script with another instance's *broadcast EVENT* block are labeled as *semantically incorrect*. All other instances are labeled *incomplete*. Note that this plugin does not use the *incorrect* label.

Complex Animation

We have a very specific definition of the term *complex animation* for the purpose of our assessment. We use this term to refer to animation involving integration of costumes, motion, timing, and repetition control structures such as loops. This definition of complex animation is to distinguish from, for example, the *glide to SPRITE* block built into Scratch. One example of complex animation is realistic motion of sprites representing people and animals, e.g., people walking, birds flying, and snakes slithering. Creating these complex animations requires the correct integration of several computer

science concepts. For example, creating an animation sequence where a sprite spins around, requires integration of loops, rotation, and timing.

A necessary component of complex animation instances is the pairing of costume change blocks with either rotation blocks, or motion blocks. We define a complex animation instance as either a loop containing these necessary components or a repeated sequence of these necessary components, since a repeated sequence can be considered an unrolled loop. In order to be labeled *correct*, an instance must also make use of a Scratch block that introduces a delay; otherwise the instance is labeled *semantically incorrect*. The plugin additionally labels instances that use repeated sequences instead of loops as *semantically incorrect* because the student did not demonstrate competence in the computer science concept of loops. Finally, if the Scratch program is missing any critical element, e.g., repetition, it is labeled *incomplete*.

2.5 Methodology

In the remainder of this chapter, we will use the term Hairball to refer to both the Hairball framework and its set of plugins as described in Section 2.4.

We tested Hairball on the Scratch programs submitted during our two-week summer camp. There were five assignments total, with a distribution of concept requirements. For example, complex animation was taught toward the end of the camp, thus instances

of this concept were only present in the last two assignments, whereas initialization was present in all assignments [21].

We first performed a manual analysis on all fifty-eight of the submitted Scratch programs. Three members of our staff independently analyzed the first five Scratch programs submitted for a given assignment using a common rubric. We discussed any discrepancies in our scores, and after coming to a consensus, we analyzed the remaining Scratch programs. Once again, any score discrepancies were reconciled.

Hairball was then programmed to match the methodology agreed upon by the staff members when classifying the concepts, and subsequently used to statically analyze all of the Scratch programs. We define our ground truth data set as all instances that were labeled identically by both manual analysis and Hairball. We performed a second manual analysis for each instance that Hairball and the manual analysis labeled differently to determine which was correct, Hairball or the initial manual analysis. The results of this second manual analysis were added to our ground truth data set. In Section 2.6, we compare both Hairball and our initial manual analysis to our ground truth data set.

Because the assignments are sensory in nature (auditory, visual), we are not attempting to create Hairball to replace manual analysis. Instead, we are automating the identification of the *easy* cases in order to accelerate manual analysis of the remaining cases. As the results in Section 2.6 show, Hairball did an excellent job of identifying issues that all three of our staff members missed.

2.6 Results

In this section, we present the results of using Hairball to assist in determining the level of competence demonstrated by students' Scratch programs for several computer science concepts. For each concept, we will compare the labels Hairball assigned to instances of the concept with those assigned via manual analysis. We look at both the false positive and the false negative rates for Hairball and the manual analysis based on comparison to the ground truth. Although our results include the labels *semantically incorrect*, *incorrect*, and *incomplete* to demonstrate that Hairball can be used for more than binary labeling, our assessment focuses on instances that are either labeled *correct* or not. Thus, we consider a false positive to be an instance that was labeled *correct*, when in fact it is not, and a false negative to be an instance that is actually *correct*, but was not labeled as such. For manual analysis, both false positives and false negatives represent the inaccuracies of manual assessment. For Hairball, false negatives can be considered warnings, i.e., they are used to indicate the need for additional manual analysis. However, any false positives produced by Hairball are cause for concern.

2.6.1 Initialization

We begin with initialization. Recall from Section 2.4 that Hairball looks for attributes that are modified, and expects to find a corresponding *absolute* block in the

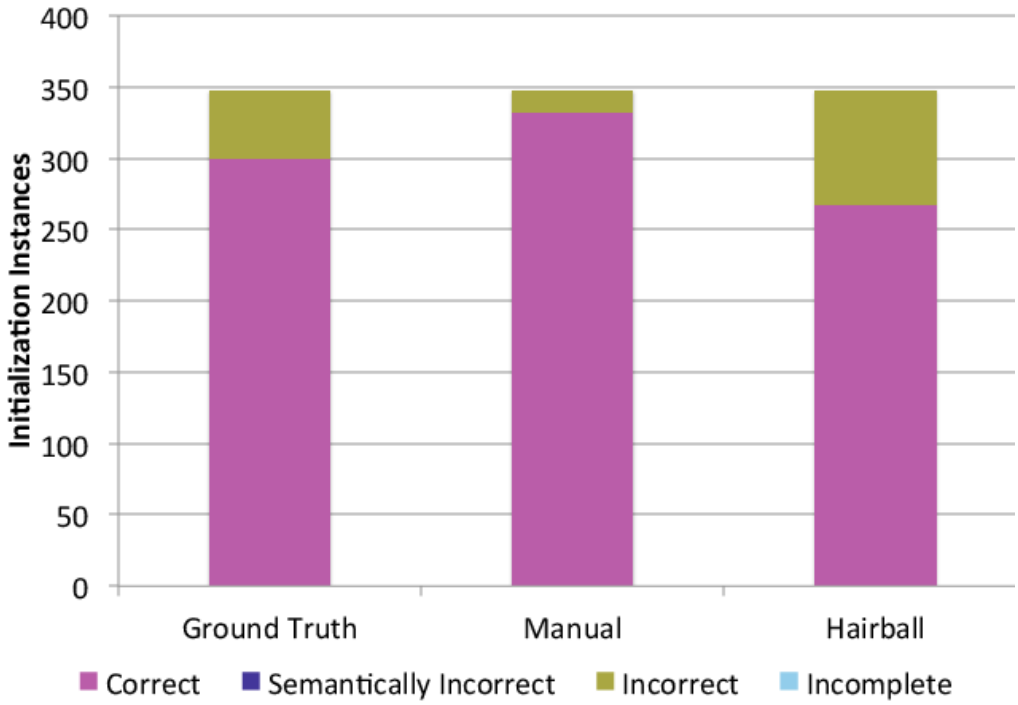

Figure 2.2: Compares the initialization instance labels. Note that this analysis used only the *correct* and *incorrect* labels. Manual analysis resulted in thirty-two false positives, and Hairball resulted in thirty-three false negatives. Note that for this concept there are exactly 348 possible instances as each of the fifty-eight Scratch programs have six attributes that require initialization if modified.

initialization zone in order to consider an instance *correct*. The manual analysis, on the other hand, only involved running the Scratch program twice, and confirming that the two executions matched.

Figure 2.2 provides the classification of the 348 initialization instances discovered across the fifty-eight Scratch programs. Of the sixty-five instances that Hairball and the initial manual analysis labeled differently, Hairball was accurate for thirty-two of the instances based on the ground truth, i.e., our second manual analysis. Many of

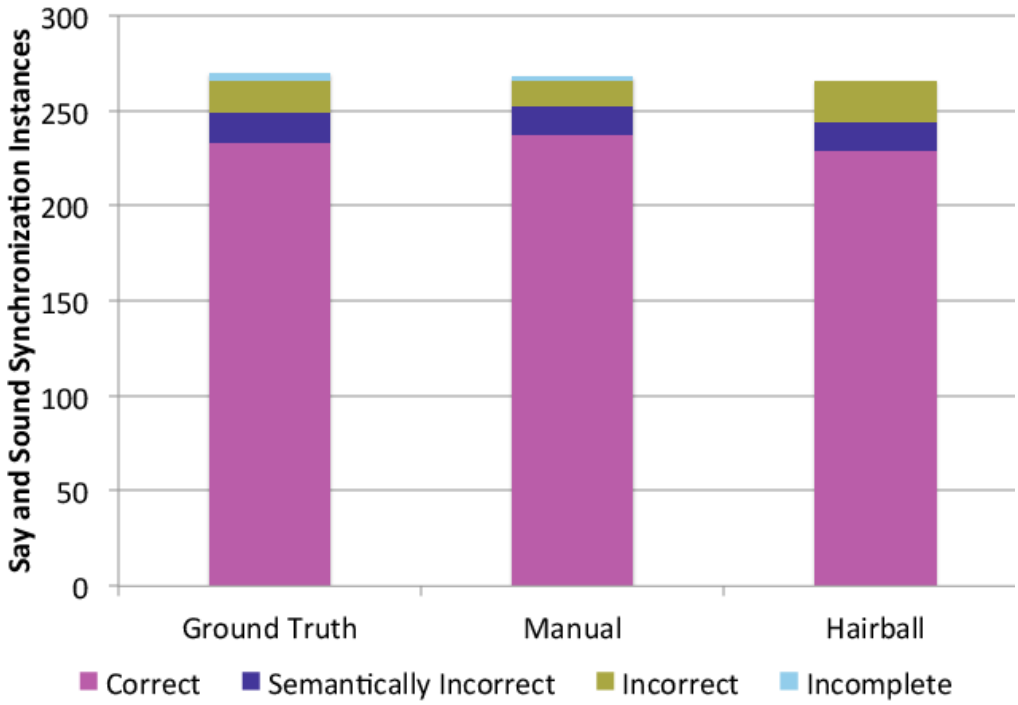

Figure 2.3: Compares the say and sound synchronization instance labels. Manual analysis and Hairball failed to detect two and four instances respectively and manual analysis resulted in four false positives.

the remaining thirty-three instances were not possible for Hairball to label as *correct* due to initialization taking place outside of the *initialization zone*. For example, an initially hidden sprite can correctly have its position initialized just before the sprite becomes visible. In spite of this discrepancy, these results overall indicate that Hairball is successful at pointing out problems in initialization.

2.6.2 Say and Sound Synchronization

Figure 2.3 shows the results of identifying and labeling instances of synchronization between speech bubbles and sound files. Manual analysis identified 237 *correct* instances, and a total of thirty-one other instances. Hairball identified 229 *correct* instances and thirty-seven others. Manual analysis and Hairball failed to find two and four instances respectively.

Comparison with the ground truth results in four false positives for manual analysis. Hairball labeled its instances with 100% accuracy. Two of the four instances undetected by Hairball were labeled *incomplete* by manual analysis. Hairball failed to detect these instances due to a separation of the *say MESSAGE* and *play SOUND* blocks with a *broadcast EVENT* block. To detect such instances, Hairball would need to additionally inspect all scripts triggered by the broadcast event to ensure none of them interfered with either the display of the speech bubble or the playing of the sound file.

2.6.3 Broadcast and Receive

Figure 2.4 shows the results of detecting and labeling instances of broadcast and receive. Here, the manual analysis differed from Hairball by additionally verifying that the intended action is performed for *correct* instances. Hairball is limited to static analysis, thus it is unable to perform this additional step.

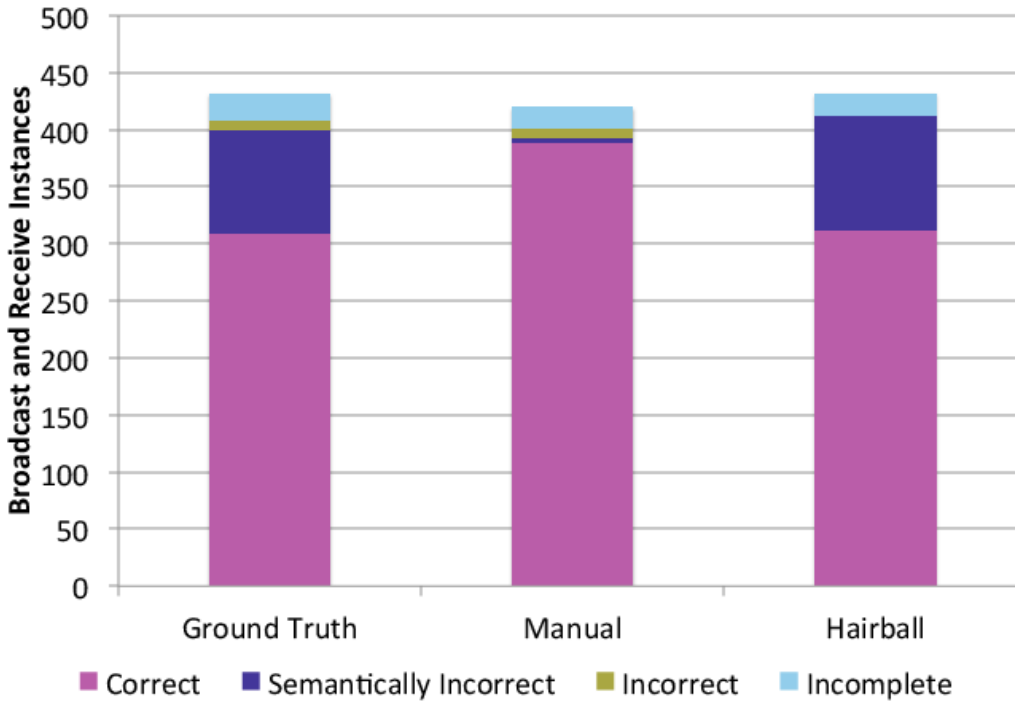

Figure 2.4: Compares the broadcast and receive instance labels. Manual analysis failed to discover twelve instances, and resulted in seventy-nine false positives. Hairball detected 100% of the instances with three false positives.

Overall, manual analysis failed to discover twelve instances, and identified 388 *correct* instances, of which, seventy-nine were false positives. Hairball discovered 100% of the instances with zero false negatives. However, three of the 312 instances Hairball labeled as *correct* were false positives. Although these three instances technically represent *correct* usage of the *broadcast EVENT* and *when I receive EVENT* blocks, our staff members labeled these instances *incorrect* in our ground truth set because the code in each case did not produce the intended behavior upon execution.

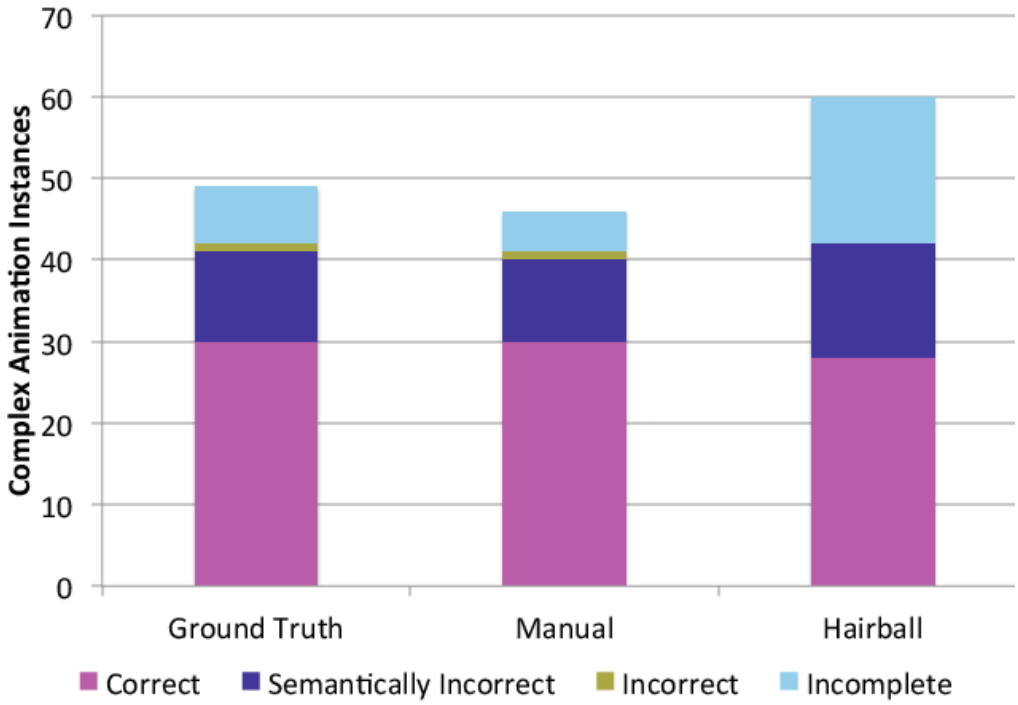

Figure 2.5: Compares the complex animation instance labels. Manual analysis failed to detect three instances whereas Hairball found eleven items that were determined to not be instances of complex animation. Hairball resulted in two false negatives.

2.6.4 Complex Animation

Complex animation was especially difficult for Hairball to detect. As Figure 2.5 shows, manual analysis was 100% accurate at labeling the forty-six instances found, and only failed to detect three instances. Hairball, on the other hand, labeled eleven items as *incomplete* that the ground truth analysis determined to not be instances at all. Excluding these instances, Hairball identified twenty-eight *correct* instances, and twenty-one others with only two false negatives.

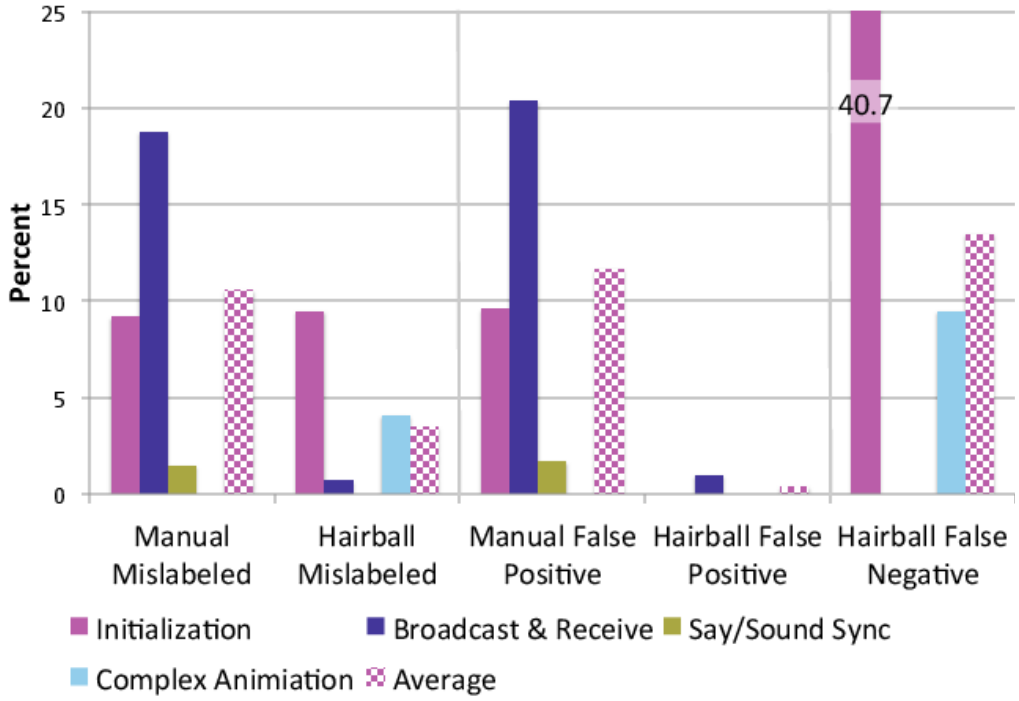

Figure 2.6: Provides a summary of the percent of mislabeled, false positive, and false negative instances resulting from manual analysis and Hairball for each of the four computer science concepts and the average. The *Manual False Negative* category was omitted as manual analysis resulted in zero false negatives. The y-axis is truncated for the smaller values, thus the tallest bar should extend to 40.7%. Missing bars represent 0%.

Hairball identified too many instances of complex animation due to the subjective nature of what is considered an animation. For example, Hairball detects an animation according to where the loops and repetition are located. Several times, Hairball detected two separate animations, when manual analysis determined that those two actions were working together to create a single larger animation. Additionally, Hairball considered a move, wait, and change in appearance as an *incomplete* animation instance. In such cases, manual analysis recorded nothing.

Summary Results

Figure 2.6 shows three sets of results across all four computer science concepts and the overall average. The first is the mislabel rate of manual analysis and Hairball. Percentages closer to zero indicate higher accuracy. We see that Hairball is actually slightly more accurate overall than manual analysis; largely due to Hairball's accuracy in labeling broadcast and receive instances.

The second set of results is the rate of false positives. Manual analysis's overall false positive rate of 11.7% indicates that manual analysis is quite error prone. On the other hand, Hairball's false positive rate of 0.4% strongly indicates Hairball is accurate at labeling *correct* instances.

Finally, the third set of results is the rate of false negatives. The lack of false negatives for manual analysis makes sense, considering Hairball was created according to the first manual analysis. Although Hairball has an overall false negative rate of 13.5%, we believe this rate to be acceptable due to the fact that four out of five instances in our ground truth set were labeled *correct*, meaning the use of Hairball on a similar corpus of Scratch programs would reduce the set of instances requiring manual analysis by 80%.

2.7 Conclusion

We presented a case study showing Hairball, a new static analysis tool that provides an extendable framework for automatically analyzing Scratch programs. In addition, we provide an initial set of plugins that analyze the implementation of Scratch programs for competence in four areas: initialization, broadcast and receive, say and sound synchronization, and animation. Our evaluation shows that Hairball is extremely useful in identifying correctly implemented instances, with a false positive rate of less than 0.5%. Overall, the mislabel rate of Hairball is less than half that of manual analysis. Therefore, we propose Hairball as an addition to, not replacement of, manual analysis.

We have made the complete Hairball source code available under the open source simplified BSD license. The source is hosted on github at `https://github.com/ucsb-cs-education/hairball`.

Our future work entails writing Hairball plugins suitable for widespread summative assessment in both AP Computer Science Principles courses, and other summer camps. Finally, we want to launch a web service that provides a convenient way to utilize Hairball for formative assessment of individual Scratch programs.

Chapter 3

Using Static Analysis to Assist with the Development of a Scratch-based 4^{th}–6^{th} Grade Classroom Curriculum

In the previous chapter, we detailed the role that static analysis plays in the post-assessment of Scratch-based 6^{th}–8^{th} grade summer camp assignments. We showed that the use of static analysis increased both the speed and accuracy of assessment. In this chapter, we describe our work that extends the application of static analysis from a summer camp context to the context of 4^{th}–6^{th} grade classroom curriculum development informed by design-based research. Using feedback from static analysis of student-created Scratch programs we direct changes to both the Scratch interface and the assignments that make up our curriculum. We found that static analysis helps reduce the time to complete the analysis portion of the curriculum development cycle.

3.1 Introduction

Computer science is becoming one of the most ubiquitous areas of education at the University level due to the importance of its core concepts within numerous other fields of study, including materials science, biology, medicine, and economics. The need for individuals to solve simple and complex problems using computers continues to grow. While there is great demand in the work-force for people with computer science experience, little effort has gone into classroom curricula to prepare young students for careers involving computer science skills. In fact, high school graduates have little, if any, idea of what computer science entails.

The Computer Science Teachers Association (CSTA) has created a set of standards in an effort to introduce students to computer science and to prepare them to meet the demands of the future. These standards detail how to incorporate computer science concepts into existing primary and secondary school curricula [18]. The standards created by the CSTA are a tremendous step in the right direction, shifting the focus of computer science education from existing after-school outreach programs and summer camps to required learning in schools [2–4, 12, 20, 24, 30].

The incorporation of computer science into 4^{th}–6^{th} grade curricula poses a challenge, as few research studies have focused on formal instruction of computer science for this age group. We focus our attention toward this effort. Using a design-based

research approach, we begin with a simplified version of our summer camp curriculum and deploy this curriculum in a number of 4th–6th grade classes in California [21]. Through analysis of field notes collected by education researchers who observed many of the students in these classrooms in combination with analysis of the students in-progress work, we are able to identify computer science concepts that are difficult for these students to understand, as well as identify other issues with both the content of our curriculum, and our modified Scratch programming environment. We use this knowledge to improve our curriculum, and then we repeat this procedure with each wave of classes.

In this chapter, we focus on the use of static analysis to assist with computer science curriculum development. We concentrate our analysis on a single Scratch assignment that requires students to demonstrate the concept of sequential execution by programming a **Net** to *catch* three other sprites through a sequence of actions.

The remainder of this chapter is organized as follows. We provide a brief summary of related work in Section 3.2. In Section 3.3, we present the methodologies used in our study. We then present our results in Section 3.4. Finally, Section 3.5 contains our conclusion.

3.2 Related Work

Significant work has focused on teaching young students the basics of computer science in the context of outreach programs such as summer camps and after school programs. In these programs, young students learn the basics of computer science using languages such as *Alice* or *Scratch*. *Computer Science Unplugged* is another approach to help students learn computer science concepts without the use of a computer. Along with the curricula for these outreach programs, researchers have also developed improved ways of evaluating student success with computer programming in these languages.

Early research pertaining to assessment focused on student surveys to assess student attitudes about the concepts they were taught in these outreach programs. More recent work, however, moves toward detailed assessment of the computer science concepts applied in students' completed assignments to discover how these assignments reflect what the students have learned. An example of this type of study was completed by Maloney et al., where they analyzed 536 completed Scratch programs created by young students over an 18-month period in an after school program. They found that students demonstrated an ability to use key programming concepts with help only from inexperienced mentors [31]. Wilson et al. adapts the coding scheme of Denner et al. to identify the most frequently used programming concepts by children who created

games in Scratch [13,41]. Using their *Progress of Early Computational Thinking Model* on an existing set of Scratch programs, Seiter and Foreman worked to identify differences in computational thinking comprehension between students in 1st–6th grade [34]. While their work provides an excellent example of extracting student understanding from completed Scratch programs, their results depend on the inclusion of a computer science concept in a Scratch program to determine whether a student appears to understand the concept. Brennan and Resnick argue, however, that the mere inclusion of a concept in a student's Scratch program is not indicative of the student's understanding of the concept, especially when encouraged to modify existing Scratch programs [7].

Through a combination of field notes collected during observation of student work along with both manual and automated analysis of student Scratch programs, Franklin et al. attempt to more precisely determine the specific computer science concepts middle school students learned during a two-week summer camp. Franklin et al. also measure their students' ability to apply taught concepts to the camp's final assignment [6, 21]. However, Piech et al. assert that student understanding of computer science concepts is not entirely reflected by the student's final version of an assignment. By using machine learning on the sequence of student in-progress programs, i.e, snapshots, Piech et al. report a correlation between success in the class and the path students took to solve an assignment [32]. This correlation appears to be to be more significant

than the final score on an assignment, suggesting that future analysis should look at multiple snapshots in order to more accurately evaluate student understanding.

Our work uses a combination of field notes along with both manual and automated assessment of student's in-progress work to provide a depth of knowledge about student understanding of our curriculum concepts.

3.3 Methodology

Table 3.1 details the eight 4th grade, one 5th grade, and one 6th grade classes from across California included in this study. In all of these classes, we collected snapshots of the student created Scratch programs. Additionally, in the first five classes, education researchers both observed teacher instruction, and conducted student interviews.

In what we refer to as *wave 0*, classes *S0A*, *S0B*, and *S0C* piloted the curriculum. This wave was used exclusively to test our snapshot creation and collection procedure, as well as to test our ability to disseminate the curriculum to instructors. *Wave 0's* field notes and snapshots are therefore not useful in comparison to classes of the subsequent waves, and thus the data from *wave 0* is not included in this chapter.

All classes, save for *S2A*, were local. Education researchers accompanied these local classes in order to assist teachers with the instruction of our curriculum, and to help students with issues they experienced while completing our assignments. The

Class	Grade	Students	Snapshots	Notes	Location
S0A	4th	18	74	Yes	Local
S0B	4th	24	94	Yes	Local
S0C	5th	39	219	Yes	Local
S1A	4th	17	208	Yes	Local
S1B	4th	12	89	Yes	Local
S2A	6th	31	268	No	Remote
S2B	4th	20	69	No	Local
S2C	4th	23	117	No	Local
S2D	4th	21	67	No	Local
S2E	4th	25	117	No	Local

Table 3.1: Lists the participating classes prefixed by assignment iteration (i.e., *S0*, *S1*, or *S2*), including the grade level of each class, the number of students with consent, the number of snapshots collected, whether education researchers took field notes, and whether the class was local to the Santa Barbara area or remote.

education researchers' presence in the classroom, along with the field notes recorded following each visit, served as a form of participant observation enabling us to become aware of concepts that were difficult for the students to learn [39]. In addition, for all classes, we automatically generated and captured multiple snapshots of the students' Scratch programs. We did this because prior work reported that the final result of a student's development process does not accurately represent the student's understanding of the material [7, 32]. The snapshot generation, collection, and verification process is described in Section 3.3.3.

Once obtained, an analysis of the field notes resulted in an initial set of student issues on the assignment. We manually inspected a sample of the snapshots in order to both gauge the prevalence of each issue, and begin defining a model for the detection of each issue. Additionally, we looked for other unexpected student behavior that was not mentioned in the field notes, such as the addition of new sprites, or the addition of scripts to sprites that were not intended to be modified. Once we had a grasp on the issues encountered by the students and how to detect them, Hairball plugins were written in order to quantify the number of snapshots and/or students affected by each issue. Finally, we met with the education researchers to discuss both the cause of these issues, and what modifications could be made to our assignment, and our Scratch programming environment in order to minimize occurrences of these issues in future

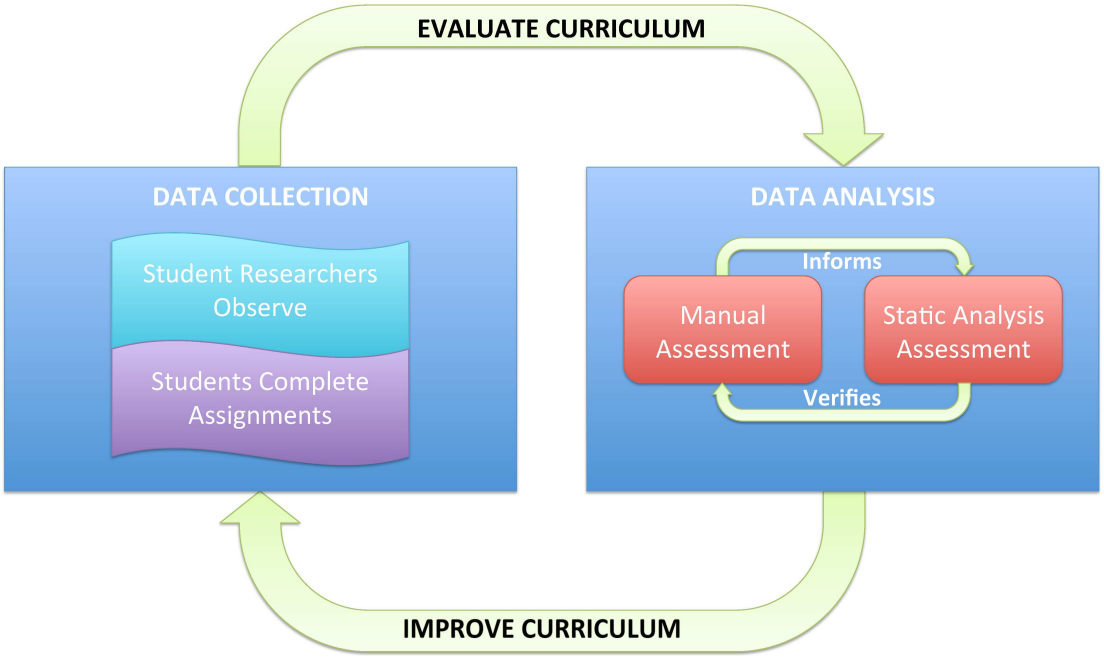

Figure 3.1: Visualizes the iterative process of evaluating and improving both our curriculum and our static analysis of Scratch programs.

iterations of the assignment. This process is visualized in Figure 3.1, and was repeated after both waves of classes.

3.3.1 Our Scratch Interface

Prior work by Lewis showed that students using Scratch, ages 10–12, had less confidence in their response to the statement "I am good at writing computer programs" than students using Logo, a text-based language. Lewis hypothesized that the students had not used all the blocks available within Scratch, thus distorting the students' view of their abilities [29]. As part of our prior work, we ran a two-week Scratch-based

summer camp, during which we observed that students of similar age to Lewis's were often distracted or overwhelmed by the plethora of blocks available when the assignment only required use of a small subset of them [21]. Furthermore, Halgren et al. reported that children, ages 5–14, have a curiosity to explore available functionality in their movie-making program:

> Our kids quickly got themselves trapped in advanced paint and movie-making modes which surpassed their expertise. They also loved clicking on the character buttons at the bottom of the screen. Each click of a character button placed a new character on the center of the screen [22].

While young students' curiosity to explore is amazing, it can be a hindrance in the classroom. Thus, in hope of minimizing both student confusion and student distraction, and in attempt to focus students' attention and maximize their programming confidence, we removed all unnecessary blocks from our Scratch interface. For this assignment, the only blocks remaining for students to use are:

- *glide NUM steps* (and a variation that adjusted speed)
- *turn clockwise NUM degrees*
- *turn counterclockwise NUM degrees*
- *point in direction X* (where X is *left*, *right*, *up*, or *down*)

3.3.2 The Sequential Execution Assignment

In this study, we consider only the first assignment given to the three *waves* of classes. The high-level goal of this assignment was for students to demonstrate competency programming a sequence of instructions that accomplish the provided task. More specifically, the goals were for students to:

- have confidence using the interface
- recognize additional blocks are to be added to the provided base script
- understand the importance of block ordering within a script
- understand that execution occurs *when NET clicked*

While the goals of the assignment in *wave 0* were consistent with those in the latter waves, the collected snapshots are inconsistent among *wave 0* students. Therefore, we will not discuss that iteration of the assignment. However, the latter two waves of classes had their own iterations of the sequential execution assignment, *Sequential1* and *Sequential2*, which, we discuss below.

Iteration 1: *Sequential1*

The first iteration of the sequential execution assignment, referred to as *Sequential1*, presents students with the *stage* and five sprites arranged as shown in Figure 3.2. In this

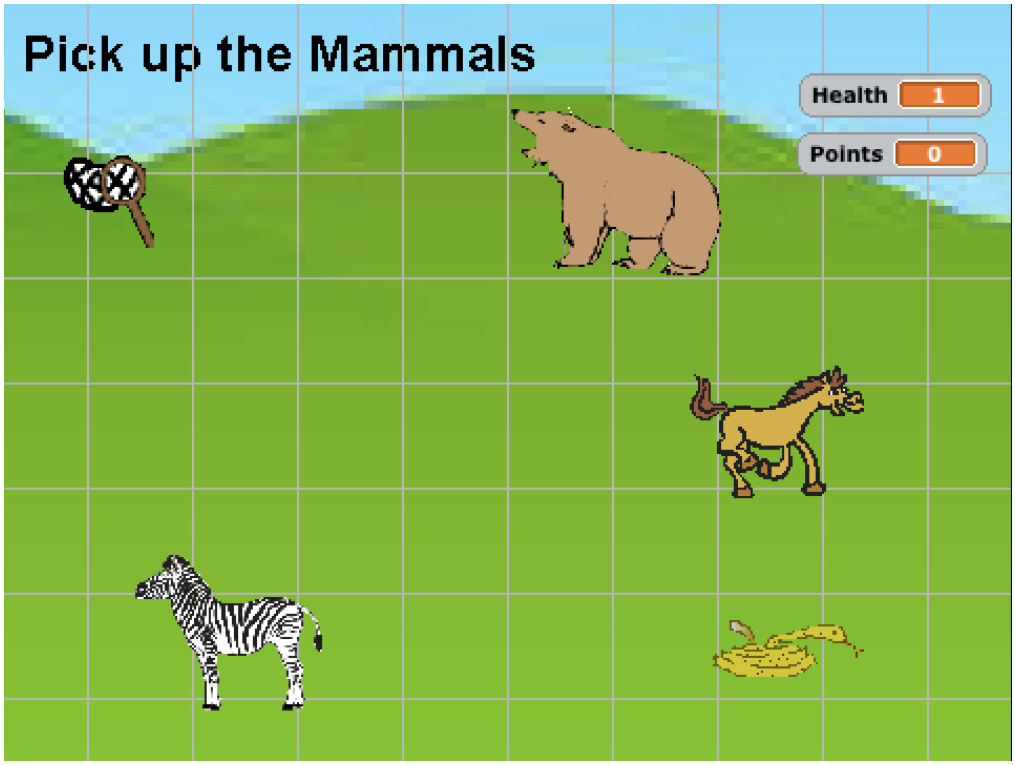

Figure 3.2: Depicts the initial screen for *Sequential1* including five sprites. Each student's task was to move the **Net** to *catch* the **Bear**, the **Horse**, and the **Zebra** in any order while avoiding the **Snake**. *Sequential2* visually differs only by the absence of the **Snake**.

assignment, the students are to program the navigation for the **Net**, located in the upper left, such that it *catches* the **Bear**, located in the upper right, the **Horse**, located in the middle right, and the **Zebra**, located in the lower left. The students are permitted to program the **Net** to *catch* these sprites in any order they desire. However, in *Sequential1*, the students are presented with an obstacle to avoid, the **Snake**, located in the lower right.

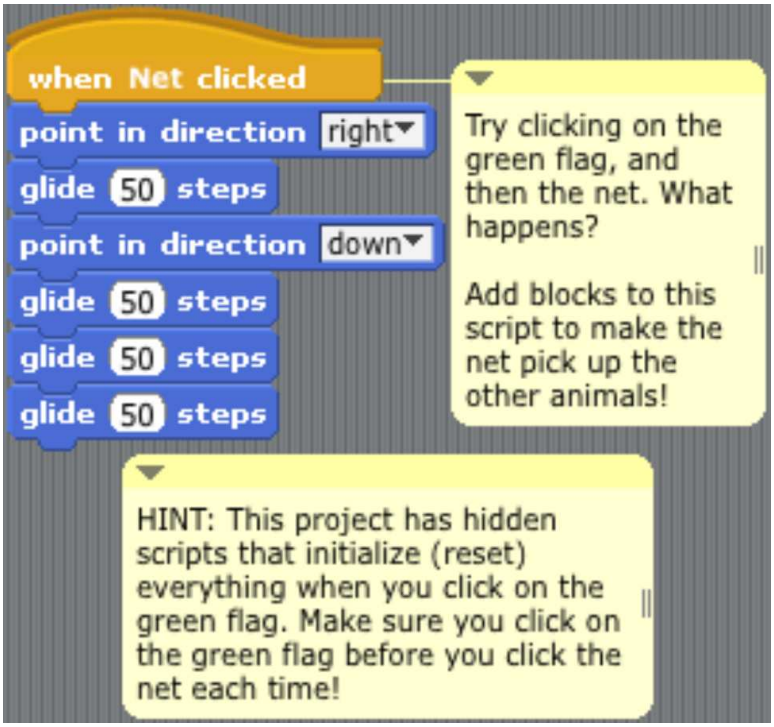

Figure 3.3: Depicts the script and comments provided in the base-project to the students in *Sequential2*. The script was the same in *Sequential1*'s base-project, however, the comments were not included.

The students were provided with a base-project that moves the **Net** such that it already *catches* the **Zebra** using a combination of *glide NUM steps* and *point in direction X* blocks as depicted in Figure 3.3. In Section 3.4.3 we classify this combination of blocks as an **orient and glide** approach utilizing *absolute* orientation.

Iteration 2: *Sequential2*

The second iteration, referred to as *Sequential2*, presents students with a screen similar to that of *Sequential1* (Figure 3.2). The only visual difference is the removal

of the **Snake**. Aside from needing to avoid the **Snake**, both the students' objective and provided base-project remained the same. However, in addition to the blocks included in *Sequential1*, we added two other block choices:

- *point towards SPRITE*
- *glide to SPRITE*

The *glide to SPRITE* block is a custom block we added to Scratch after observing student difficultly during *Sequential1* with the two blocks required to successfully move the **Net**. The students were not prompted to use either of these blocks. A comparison of student block choices is provided in Section 3.4.3. One other notable change made to the Scratch interface for *Sequential2* was the removal of the *double click to execute* functionality provided by Scratch. See Section 3.4.5 for a discussion of why we removed this functionality.

3.3.3 Capturing, Collecting, and Verifying the Accuracy of Snapshot Generation

Scratch, like many computer programs, only saves the most recent version of a Scratch program when explicitly directed to do so via a *save* action. In order to obtain snapshots of students' in-progress Scratch programs, we modified Scratch to automatically generate a snapshot when two conditions are met: the *green flag* button is clicked,

and at least four modifications to the Scratch program have occurred since the last snapshot. We create snapshots on clicks of the *green flag* button because these clicks are likely to occur only when students have made incremental progress on their Scratch programs as students were taught to test their Scratch programs in this manner. Snapshots are also created whenever the student explicitly invokes the *save* action. All snapshots for a student are labeled by time, and combined in one zip archive to reduce both the size and number of files we needed to collect.

We did not create a snapshot on every *green flag* button click due to a network issue we experienced during teacher training. The computers that we trained the teachers on were configured such that Scratch program files were stored on a remote server, thus each save required transferring data over the network. While the creation of a single snapshot is not an issue, the concurrent creation of many is. Because many school networks are configured similarly, we introduced the four-modification snapshot creation condition in hopes of preventing similar network issues; it worked.

Students were asked to submit their single snapshot archive at the end of each work period by uploading the file through a web service we wrote for collection purposes. This web service associated students with their uploaded snapshots. While this process was very effective, and much less error prone than the researchers manually fetching files from computers, it was not without issue; there were three:

1. A few students accidentally submitted the snapshots of a student who had used the same computer in a previous class due to the web browser's recall of the directory of the last uploaded file.

2. Instead of starting with the provided base-project, a few students accidentally started with a snapshot of a student who had used the same computer in a previous class.

3. Some students submitted only the most recent snapshot due to selecting the final version file rather than the snapshot archive file.

Fortunately, we were able to utilize the save-log contained within each Scratch program file (i.e., our snapshots) to identify students who experienced any of these three issues. The save-log in a Scratch program file records the history of every *save* action with a timestamp and the name of the file saved to, providing us with an expected number of snapshots for each student. Additionally, from this information we identified students sharing unexpected common starting points. That is, students who share more than one consecutive entry in their Scratch programs' save-logs that immediately succeeds the save-log of the Scratch program we provided the students to start with. Once identified, we disassociated these snapshots from the student in the chronologically later class. This procedure was used to resolve issue #1 and issue #2. Furthermore, we retroactively obtained the snapshots for some of the students affected by issue #1. We

resolved issue #3 by comparing the number of snapshots collected for a student to the actual number of snapshots created as indicated by the save-log. Students for whom we did not have all snapshots were removed from our dataset. Due to the fact that we did not analyze any of the *wave 0* classes, we did not validate the data for those classes. Thus the *student* and *snapshot* values for the first three columns in Table 3.1 are an upper bound.

3.4 Results

In this section, we describe the results of our analysis of the *Sequential1* and *Sequential2* data. In general we wanted to gain insight into the following questions:

- How successful were students in creating a Scratch program that completed the assignment?
- Did the changes we make after *Sequential1* improve student completion rates?
- How pervasive were the challenges identified by education researchers via direct student observation?

To answer these questions we (Section 3.4.1) look at the completion rate of students by class, (Section 3.4.2) compare the difficulty of *Sequential1* and *Sequential2* based on the number of snapshots to completion, (Section 3.4.3) analyze the approach students

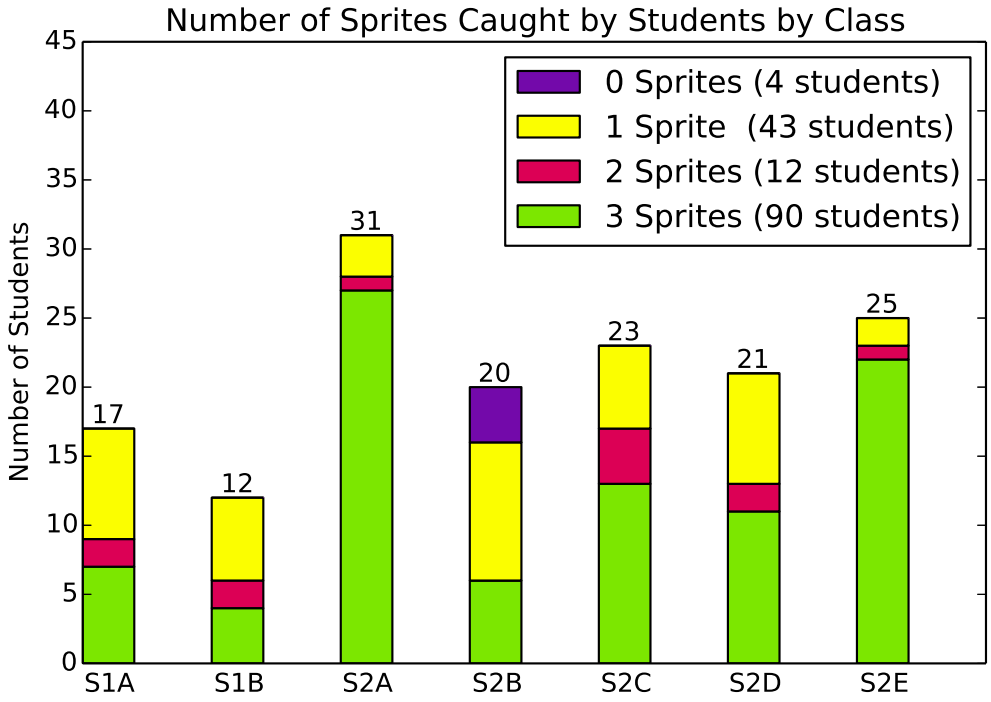

Figure 3.4: Compares the maximum number of sprites *caught* by student by class. A student is considered *complete* if any of their snapshots *catches* two or more sprites.

used in solving the assignment, (Section 3.4.4) quantify the number of students who may have experienced a race condition in Scratch, and (Section 3.4.5) quantify the number of students who may have utilized the *double click to execute* approach when initially working on the assignment.

3.4.1 Students by Class

We analyzed data from seven of the ten classes listed in Table 3.1: two for *Sequential1*, and five for *Sequential2*. This data include a total of 297 snapshots, twenty-nine

students for *Sequential1*, and 638 snapshots, 120 students for *Sequential2*. We have more data for *Sequential2* due to having more participating classes, all of which, contained more students for whom we had consent.

Figure 3.4 compares student completion of the assignment for each class where the total height of each bar indicates the number of students by class; this number is enumerated above each bar. The different colored portions of each bar groups students by the maximum number of sprites *caught*. Green, pink, and yellow respectively indicate that all, two, or only one of the three sprites were *caught*. Purple indicates none of those students' snapshots result in the **Net** *catching* a sprite upon execution. The four students in the *0 Sprites* group are interesting because the base-project provided to all students already *catches* one sprite, the **Zebra**. Thus, these four students made changes resulting in negative progress toward the goal. Also, it is noteworthy that of the 102 students who *caught* at least two sprites, only twelve (11.8%) did not *catch* the final sprite.

We determine the success of a snapshot by running it through a Hairball plugin that emulates the **Net's** movement according to the **Net** *script* beginning with the *when NET clicked* block. A snapshot is considered *complete* if the emulated movement of the **Net** results in intersection with any two or more sprites corresponding to the **Bear**, **Horse**, and **Zebra**. In the event intersection with the **Snake** occurs (only valid for *Sequential1*), the snapshot is considered *incomplete*. Of the 297 *Sequential1* snapshots, only one was

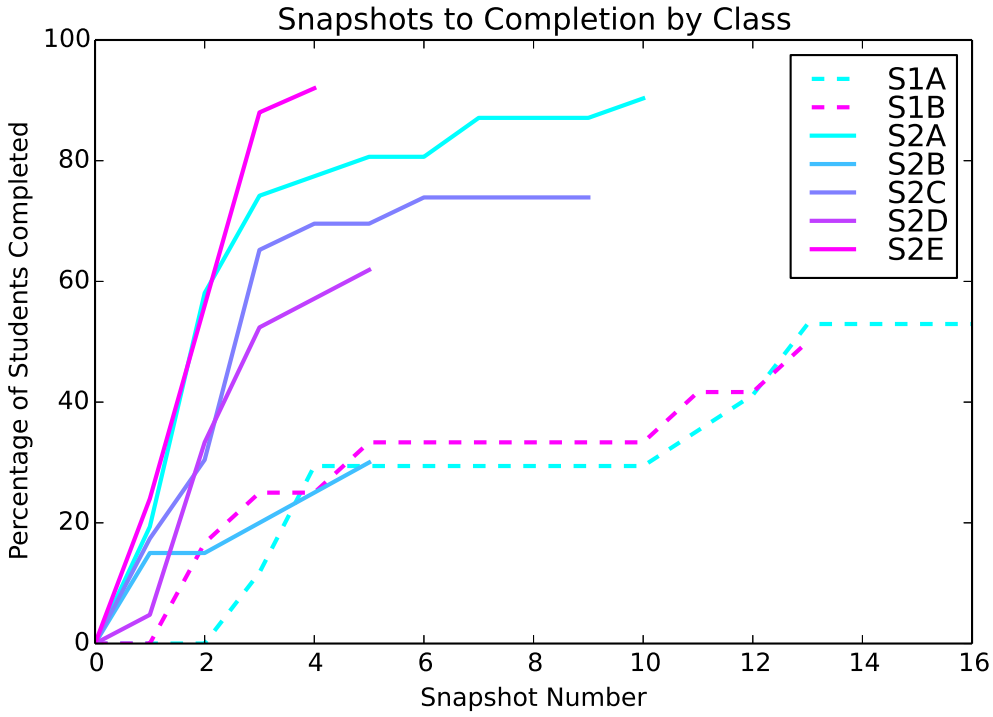

Figure 3.5: Depicts the percentage of students by class that completed the assignment by the number of snapshots indicated on the x-axis. The dashed cyan line representing *S1A* was truncated. It would otherwise extend horizontally out to the twenty-first snapshot.

incomplete due to intersection with the **Snake**. A student is considered *complete* if they have at least one *complete* snapshot.

3.4.2 Number of Snapshots to Completion

In the previous section, we looked at the total number of sprites *caught* by students in the seven classes analyzed. While this information provides us with the overall completion rate, it does not provide any insight related to the difficulty of the assignment.

We approximate the assignment difficulty for a student as the number of snapshots saved up to their first *complete* snapshot. Recall from Section 3.3.3 that we consider each snapshot to be a unit of work.

Figure 3.5 plots the number of snapshots saved for students in each class on their path to completion. An increase in the *y-value* for a line indicates what percent more students were able to *complete* the assignment after the corresponding number of snapshots. The end of a line indicates the maximum number of snapshots generated on the path to completion for students of that class. This figure clearly conveys two discrepancies between *Sequential1*, indicated by dashed-lines, and *Sequential2*, indicated by solid-lines:

- All *Sequential2* classes, save for *S2B*, had a higher completion rate than the two *Sequential1* classes.
- More importantly, this figure shows that *Sequential2* was considerably less difficult to complete than *Sequential1* based on the strictly fewer number of snapshots to completion for all *Sequential2* classes, again save for *S2B*.

Over 50% of *Sequential2* students completed by snapshot three, whereas fewer than 25% of *Sequential1* students completed by their third snapshot. Furthermore, approximately 20% of *Sequential1 complete* students required more than ten snapshots to complete the assignment. *Sequential2* was less challenging to the students due in part

to the addition of the *glide to SPRITE* block. We look specifically at the impact of the *glide to SPRITE* block in the next section.

3.4.3 Approach to Solving the Assignment

As previously described, this assignment asks students to program a set of directions to navigate the **Net** to *catch* the **Bear**, the **Horse**, and the **Zebra**. This set of directions can be constructed in a number of ways. At the highest level, there are two approaches:

Glide to SPRITE: With a single *glide to SPRITE* block the **Net** will glide on a direct path to the target sprite resulting in an intersection between two. The simplest *complete* solution requires only three of these blocks, one for each of the **Bear**, the **Horse**, and the **Zebra**. This approach was only available in *Sequential2*.

Orient and Glide: The other high-level approach is to modify the **Net's** orientation via one of three classes of orientation changing blocks, and then to glide an appropriate number of steps via a *glide NUM steps* block in order to reach the desired target or waypoint. The three classes of orientation changing blocks are:

Absolute orientation: This orientation change is accomplished via a *point in direction X* block where *X* can be selected as *up (0)*, *right (90)*, *down (180)*, or *left (-90)*. Alternatively, any number can be manually entered for a more precise orientation. These orientations are absolute with respect to the *stage* meaning *up* always orients toward the top of the *stage*, *right*, toward the right of the *stage*, etc.

Relative orientation: This orientation change is accomplished via either a *turn clockwise NUM degrees*, or a *turn counterclockwise NUM degrees* block. The use of one of these blocks results in a modification to the current orientation of the **Net**. That is, if the **Net** is oriented toward the right of the *stage*, a *turn clockwise 90 degrees* block will result in the **Net** being oriented toward the bottom of the *stage*; in this particular case, the *absolute* orientation block *point in direction down* would have the same effect.

Sprite orientation: The third type of orientation change is accomplished via a *point towards SPRITE* block. When invoked as *point towards **Zebra***, the **Net** will orient itself toward the **Zebra**. This block was only made available in *Sequential2*.

A student may utilize a combination of these high-level approaches to complete the assignment. For instance, in a single snapshot a student may use the *orient and glide* approach via a *relative* orientation block to *catch* the **Bear**, and subsequently use the *glide to SPRITE* approach to *catch* the **Horse**. Alternatively, students may utilize several different classes of orientation blocks. We wanted to see which combination of approaches was most preferred among students who completed the assignment.

As mentioned in Section 3.3.2, the base-project used as the starting point for all students utilized an *absolute* orientation approach. In order to accurately assess what approach combination the students explicitly utilized, the code provided in the base-project was excluded from our approach combination analysis.

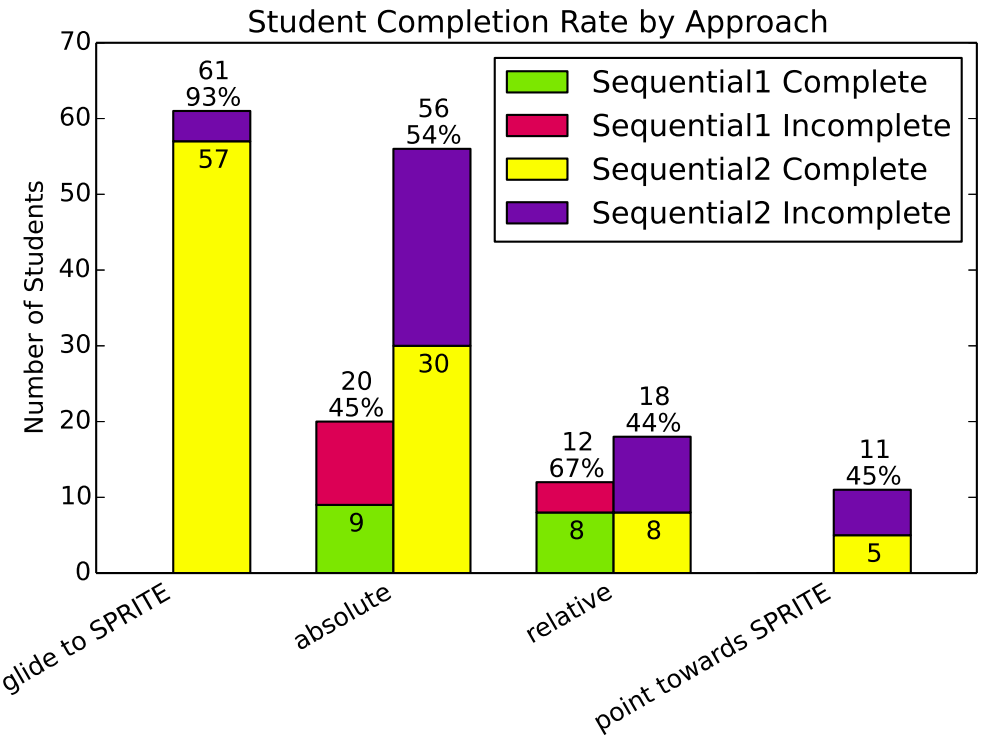

Figure 3.6: Shows the completion rate of each approach by student grouped by *Sequential1* and *Sequential2*. An approach for a student is *complete* if any of the student's *complete* snapshots utilizes that approach. An approach for a student is *incomplete* if they utilize the approach in any *incomplete* snapshot and the approach is not found in any of the student's *complete* snapshots.

Figure 3.6 shows a comparison of the overall completion rate by student of each approach by assignment iteration. Only snapshots up to a student's first *complete* snapshot are included in this analysis as some teachers provided additional challenges to students who had completed the assignment. The height of each bar indicates the total number of students who had at least one snapshot that utilized the indicated approach. This value is provided as the upper-most number above the bar. The lower number is

the completion rate as a percentage, and the number within the lower segment of the bar quantifies the number of *complete* students for the approach. An approach is *complete* for a student if they utilized that approach in their first *complete* snapshot, otherwise, an approach is *incomplete* for a student. An approach is counted even when used in combination with another approach. When comparing the fifteen *Sequential1 complete* students to the eighty-seven *Sequential2 complete* students, only two and fifteen respective *complete* students (13.3% and 17.2%) utilized a combination of approaches in their first *complete* snapshot.

This figure shows overwhelming evidence that students understood how to use *glide to SPRITE* as the approach was *complete* for all but four (93.4%) students who attempted the approach. Conceptually, this observation makes sense as the approach requires only a single block per *catch*, rather than two or more blocks as required by other approaches.

The *absolute* approach had around a 50% completion rate for both *Sequential1* and *Sequential2*. Considering that all students were provided with a base-project utilizing the *absolute* approach to *catch* the **Zebra**, this result indicates students struggled with the *absolute* approach.

While there are not many students for *Sequential1*, the figure does not convey that all of the *complete* snapshots for the *relative* approach belong to students in the *S1A* class. In fact, only one *S1B* student attempted a *relative* approach, whereas all but

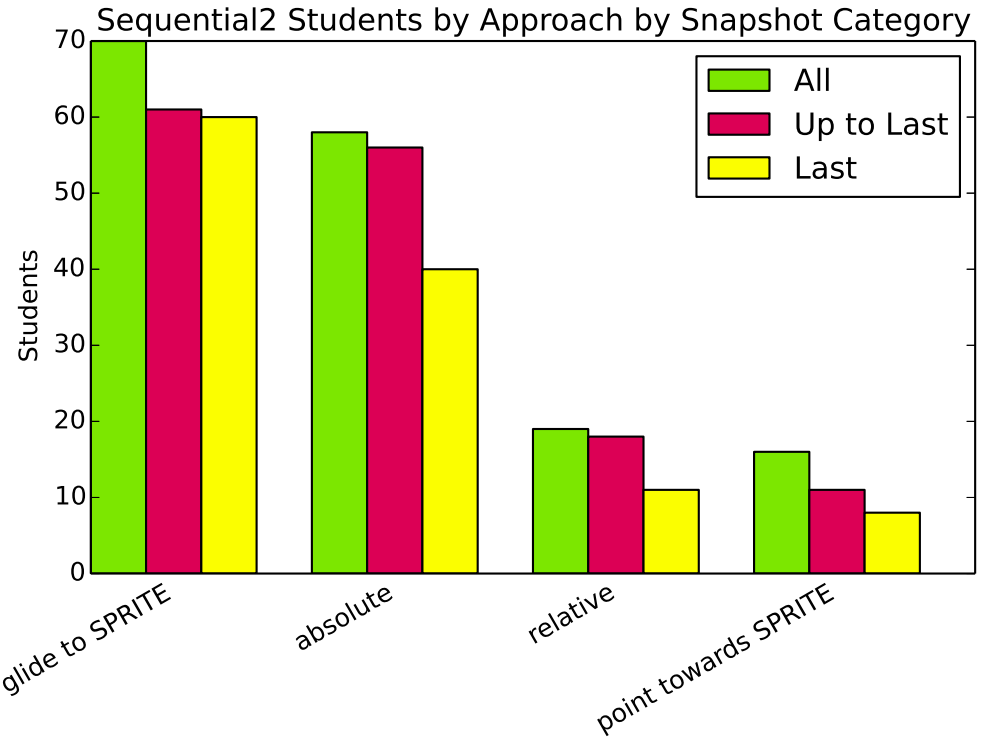

Figure 3.7: Shows how many students utilized each approach in *Sequential2* snapshots for three categories: *all* snapshots, snapshots up to the first *complete* or last *incomplete*, and the *last* snapshot.

four (76.5%) *S1A* students attempted an *absolute* approach. None of our field notes provide any insight as to why the *relative* approach was so prominent with the *S1A* class, especially when compared to the insignificance of the *relative* approach with *Sequential2*.

Finally, we look at students' usage of an approach across snapshots in three categories:

all the student utilized the approach in at least one snapshot, including snapshots made following a *complete* snapshot

up to last the student utilized the approach in at least one snapshot up to and including their first *complete* snapshot (includes all snapshots for students who had no *complete* snapshot)

last the student utilized the approach in either their first *complete* snapshot, or their last snapshot if all of their snapshots were *incomplete*

Figure 3.7 quantifies the number of students who utilize each approach in *Sequential2* snapshots for each of the aforementioned categories. There are two primary observations: The first is that the difference in height between the pink and yellow bars show the number of students who abandoned an approach. The minuscule difference for the *glide to SPRITE* approach provides additional evidence for the ease-of-use of that approach. We make no claims about the abandonment of other approaches due to the low number of students utilizing those approaches. The second observation pertains to the difference in height between the pink and green bars for each category. This difference indicates students who, only after making their first *complete* snapshot, attempted a new approach. These additional attempts made by students after a *complete* snapshot are likely due to additional challenges posed by instructors. The figure shows very little growth in the *absolute* and *relative* approaches, but a nearly 15% increase in

the *glide to SPRITE* approach; once again providing support for the ease-of-use of the *glide to SPRITE* approach. A figure for *Sequential1* is not provided as there are only two possible approaches, and none of the students switched approaches after their first *complete* snapshot.

3.4.4 Quantifying Students Affected by a Scratch Race Condition

Our curriculum development and testing process, as described in Section 3.3 and visualized in Figure 3.1, allowed us to focus analysis on issues we became aware of due to the in-class researchers' field notes. However, the field notes did not always capture the relevant information. During manual analysis of the students' snapshots we noticed a number of snapshots that produced inconsistent results across multiple executions. These snapshots should have consistently *caught* the **Zebra**, however, only did so approximately 50% of the time. We discovered the problem to be a race condition within Scratch where the detection of the intersection between two sprites may not occur in the brief period of time that the sprites intersect. Instead, the next block in the script, always a rotation block, would execute and the rotation would result in the separation of the two sprites; i.e., the two sprites were no longer intersecting. In-class education researchers confirmed having observed this issue, however, their field notes did not quantify the number of students affected. We hypothesized that students affected by this issue may have struggled completing the assignment.

In order to quantify the students affected, we wrote a Hairball plugin to track the **Net's** execution sequence, i.e., the sequence of blocks beginning with *when NET clicked*. We wanted to discover snapshots where the **Net's** execution sequence matches that of one of the execution sequences we manually verified as exhibiting the race condition. We manually verified execution sequences by programming them in Scratch, and executing the Scratch program up to twenty times. If we observed inconsistency in the *catching* of the **Zebra** within these twenty executions, then the execution sequence was labeled as exhibiting the race condition; otherwise it was not. While it is possible for a race condition to emerge at a lessor frequency, we assume that few, if any, students were affected by these cases. No race condition exhibiting execution sequence required more than eight executions to detect.

What resulted was a Hairball plugin with a state machine that handled all execution sequences of the **Net** shared by more than any ten snapshots. We only handled execution sequences up to the point that we could label them as exhibiting the race condition or not. Of the 297 and 638 snapshots for *Sequential1* and *Sequential2* respectively, only thirty-nine and thirty-eight respective snapshots (13% and 6%) contained an execution sequence not explicitly handled by our plugin.

In addition to labeling execution sequences exhibiting the race condition, we labeled those resulting in a consistent intersection with the **Zebra**. Students with such a snapshot subsequent to a snapshot exhibiting the race condition are likely to have

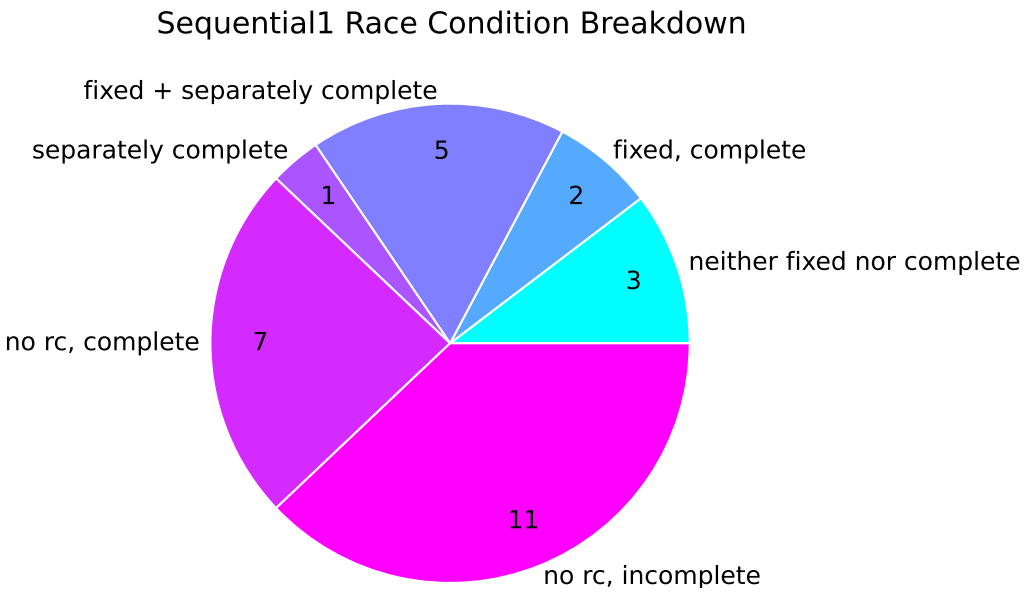

Figure 3.8: Shows the breakdown of students affected by the race condition issue in Scratch for *Sequential1*.

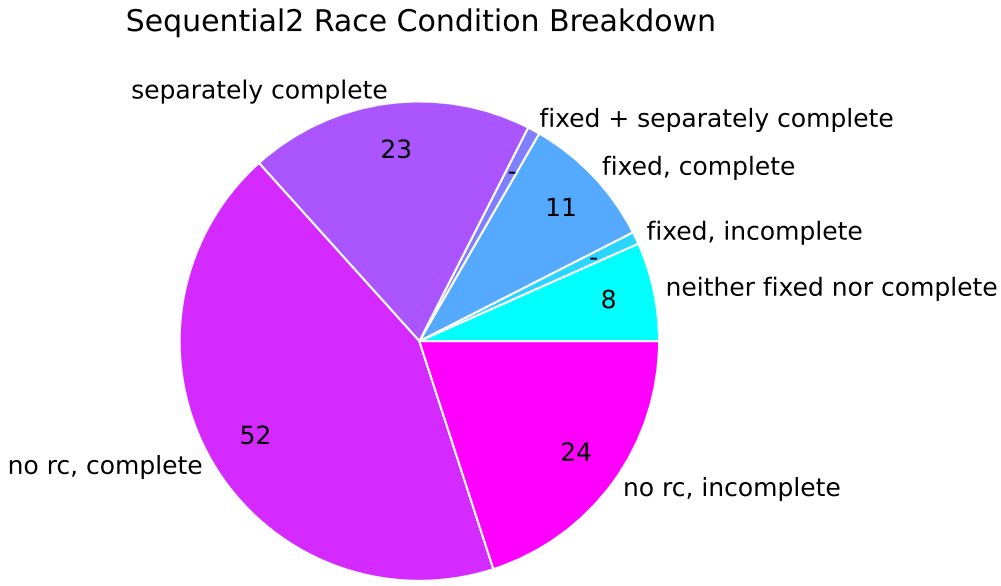

Figure 3.9: Shows the breakdown of students affected by the race condition issue in Scratch for *Sequential2*.

expended effort to resolve the race condition. We label these snapshots as *fixed*. Additionally, for each student with one or more snapshots exhibiting the race condition, we consider whether or not they completed the assignment.

Figure 3.8 and Figure 3.9 show the breakdown of all students grouped by whether or not they have at least one snapshot exhibiting the race condition. In total, eleven and forty-four students (38% and 37%) have snapshots exhibiting the race condition for *Sequential1* and *Sequential2* respectively. Of those, three and nine students (27% and 20%) were unable to complete the assignment. Six and twenty-four students (55% and 55%) took a completely separate approach to completing the assignment after experiencing the race condition, whereas only two and eleven students (18% and 25%) solved the assignment by the addition of one or more blocks that *fixed* the race condition.

Fortunately, a majority of students were able to avoid the race condition. We sampled the snapshots of a few of these students and discovered the following three approaches that students used to avoid encountering the race condition:

1. The student immediately removed some or all of the provided code, thus starting with a modified base-project.

2. In *Sequential2*, the student simply appended *glide to SPRITE* blocks to the code.

3. The student immediately added an additional *glide NUM steps* block resulting in consistent intersection between the **Net** and the **Zebra**.

Students were also able to *fix* the race condition using the above approaches. Of the thirteen students who first encountered, and then resolved the race condition, two students (15.4%) *fixed* the race condition using approach #2, and the remaining eleven students (84.6%) *fixed* using approach #3. Note that while students may have used the same approach to avoid the race condition, only students for whom we have a prior snapshot exhibiting the race condition are labeled as *fixed* in Figure 3.8 and Figure 3.9.

The data show that over one third of students experienced the race condition. Interestingly, the students who experienced the race condition were statistically significantly more likely to complete the assignment: 73% and 82% compared to 39% and 68% (chi square, $p < 0.028$). This result was unexpected, nevertheless, the labels provided by our static analysis indirectly allowed us to discover the primary reason why students who did not experience the race condition did not complete the assignment. Manual inspection of these labeled snapshots revealed that a large majority of these students either removed, or significantly altered the provided code in their first snapshot. The only other reason we discovered was due to race condition avoidance approach #3 where, in each of these cases, it was apparent that the avoidance of the race condition was unintentional based on the subsequent erratic modifications made by these students.

Overall, the effect of these results is that we are now able to adjust our assignment so that students are less likely to encounter a Scratch race condition. Moreover, given

the negative impact of students removing the provided code, we learned that preventing students from doing so may have a positive effect on learning.

3.4.5 Snapshots Exhibiting the *Double Click to Execute* Behavior

Scratch is built such that students can double click on any script, i.e., one or more connected blocks, in order to execute that script. During *Sequential1*, in-class education researchers described in their field notes that some students took advantage of this behavior in order to execute scripts they created. While manually executing disjoint scripts in this manner may trigger the success screen we built into the assignment, the education researchers found that students exhibiting this behavior did not understand the concept of a script. Instead, these students viewed the blocks as independent entities not sequentially triggered by an event (e.g., *when NET clicked*), and therefore these students did not exhibit the conceptual understanding of sequential execution we had intended. Furthermore, the education researchers noted that students would *double click to execute* a script in order to move from the start location to the first sprite, and then alter that script to perform the next step of the sequence. Thus, as another use of instructional scaffolding in the assignment, we disabled the *double click to execute* feature in *Sequential2* in attempt to prevent students from going down an unintended path while completing the assignment.

Double Click to Execute Filters

Once aware of the problem, we sought to retroactively identify students who may have utilized this *double click to execute* approach. Ideally, we wanted a plugin that would positively identify these students based on their snapshots. However, with the information provided in the snapshots, we could only incrementally filter out snapshots matching models that we verified do not demonstrate the *double click to execute* behavior. The following paragraphs detail, in order, the static analysis filters we created in attempt to approximate the students affected by the *double click to execute* behavior.

***Complete* Snapshots** Any snapshots that when emulated by our plugin result in the **Net** *catching* any two or more sprites are filtered. Furthermore, any chronologically subsequent snapshots by the same student are filtered. The subsequent snapshots are filtered because, once a student demonstrates success, we are not concerned about their *double click to execute* use.

Motionless Snapshots Any snapshots that result in no movement due to either having zero scripts or having only a single *when NET clicked* script with no movement blocks are filtered. These snapshots do not result in any motion and thus are not indicative of *double click to execute* behavior that we are concerned with.

Net Ends in Expected Location Each Scratch program, i.e., each snapshot, stores the last location of all its sprites. We compare the stored location of the **Net** to its

final location as computed by our **Net** emulation Hairball plugin. Snapshots containing movement whose emulated **Net** location matches the **Net's** stored location are filtered. These snapshots are filtered because our emulation plugin does not support the *double click to execute* behavior, thus it is not possible for these locations to match when the *double click to execute* behavior is used.

Multiple Net Clicks We expect students' scripts to execute only once upon *when NET clicked* following a reset of the environment via a click on the *green flag*. However, it is still possible to click on the **Net** multiple times resulting in an execution for each of these clicks. In such cases, the **Net** will initially be in an invalid state for all but the first **Net** click. By performing the expected location test multiple times, we are able to both identify and filter snapshots exhibiting this multiple **Net** click behavior. These snapshots are filtered for the same reason as those filtered due to ending in an expected location.

Double Click to Execute Snapshots

After applying all the filters, we counted the number of remaining snapshots per student that may exhibit the *double click to execute* behavior. Figure 3.10 shows the number of potential snapshots by student for both *Sequential1* and *Sequential2*. Recall that the *double click to execute* functionality was disabled completely in *Sequential2*, thus we hoped that this filtering would result in nearly zero *Sequential2* snapshots and a

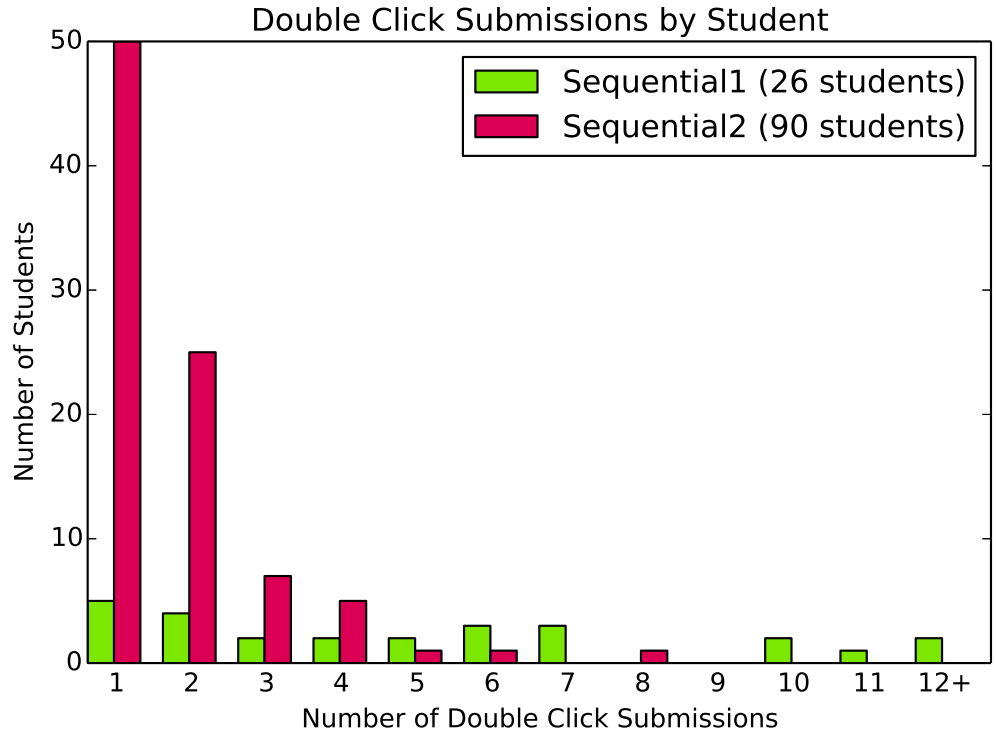

Figure 3.10: Depicts the number of *double click to execute* snapshots we identified for each student.

significant decrease in the number of *Sequential1* students whose snapshots we would manually inspect. However, that was not the case. In reality, this filtering only removed three of twenty-nine and thirty of 120 students (10.3% and 25.0%) respectively. Unfortunately, without more precise field notes, we cannot quantify the number of students affected because there is not sufficient information in the snapshots for us to even manually identify students exhibiting the *double click to execute* behavior.

It is important to note that while we could not proceed, it was not due to a limitation of static analysis. The Hairball plugin we wrote considerably helped us come to the

conclusion that we simply had not gathered enough information to either manually or automatically determine the students affected by the *double click to execute* behavior. Based on this experience with a lack of data, we are altering our data collection to capture every change made by students.

3.5 Conclusion

This chapter detailed our continued use of Hairball for assessment of Scratch-based assignments. We described our modifications to Scratch in order to apply instructional scaffolding, and presented the results of two iterations of our sequential execution assignment. Two goals of our assignment were that students would recognize the need to add additional blocks to the base-project, and understand the importance of block ordering in order to demonstrate proficiency of sequential execution in Scratch. In this section, we state our conclusions regarding improvements made to our curriculum, and to the use of static analysis as a curriculum development tool.

3.5.1 Curriculum Improvements

We created and utilized Hairball plugins to help answer the three questions we sought to answer (Section 3.4). In total, 102 of the 149 students for whom we had consent completed the assignment. While an overall 68% is not impressive, the per-

centage increased from 52% to 73% due to modifications we made to both our Scratch interface and our curriculum after *Sequential1*. While there is more room for improvement with respect to student success on the assignment, we consider this increase to be a success of our modifications between the two iterations of the assignment.

The single most important change we made was the addition of the *glide to SPRITE* block as it enabled the use of only a single block for each of the three essential *pick up* actions. Recall that our goal was not for students to understand position and orientation changes, but simply for them to program sequential code that *picks up* all the objects. Based on these results, we are introducing additional instructional scaffolding to our curriculum. For instance, students will first be asked to solve the challenge using only *glide to SPRITE*, and once mastering that task, will then be challenged with a similar task using only one of the *Orient and Glide* approaches. In both cases, only the necessary blocks will be available for students to use.

3.5.2 Static Analysis

Hairball plugins were written to quantify students challenged with issues identified by education researchers' field notes. We described one such plugin identifying that 40% of all students experienced a Scratch race condition. Interestingly, 78% of those students completed the assignment indicating a statistically significant correlation between experiencing the race condition and completing the assignment. While we were

successful in writing a Hairball plugin for the race condition issue, we could not do the same for the *double click to execute* issue. The problem was that even with manual analysis we could not precisely differentiate between snapshots exhibiting this behavior and normal behavior due to the lack of information contained in the snapshots. In this regard, we consider the use of static analysis a success as it helped us swiftly determine the large subset of submissions that may exhibit the *double click to execute* behavior. This information, in turn, permitted us to come to the aforementioned conclusion.

The two most significant benefits of using static analysis in assignment assessment are the speed of assessment, and the accuracy of assessment. While there is overhead involved in creating static analysis, it is a one-time overhead with essentially infinite scaling capabilities. The overhead for training a human, on the other hand, may require less time, but does not scale. Furthermore, static analysis will consistently produce the same results, whereas humans are significantly less likely to do so.

Another significant advantage of incorporating static analysis in assignment assessment is due to the dramatic reduction in overhead required with each iteration of assessment criteria; of which, we had many. With only the addition of a short amount of time required to adapt our static analysis to updated assessment criteria, we were otherwise able to rerun the entire modified assignment assessment across all snapshots in a matter of minutes. Conversely, a human could at best assess six snapshots in a minute. Assuming that is feasible, each assessment criteria iteration would have required 2.6 hours

for analysis of our 935 snapshots. While, in general, the number of assessment criteria iterations can be reduced with more in-depth up-front preparation, using static analysis permits a flexibility in assignment assessment that is not limited by human factors.

Finally, the plugins written for our assessment will be used in future iterations of the assignment to validate additional interface and curriculum changes. The use of Hairball plugins in our assessment shows the usefulness of static analysis tools in the development of 4th–6th grade curriculum. In the future, we hope to incorporate these plugins in an automated snapshot collection and feedback system in order to provide real-time feedback and assessment to students and instructors as students progress through an assignment.

Chapter 4

Analyzing Undergraduate Student Submission Patterns in the Presence of a Real-Time Feedback and Assessment System

In this chapter, we move from using static analysis in the assessment of 4^{th}–6^{th} grade Scratch programs to investigating the submission behavior of university students in the presence of a real-time feedback and assessment system. Because these systems dramatically decrease the required assessment time, many computer science departments are using them to handle the work related to the growing number of computer science undergraduates. However, little research is available that has looked specifically at how students interact with these systems, or how these systems impact student learning. We focus our investigation on these two neglected areas of research, paying particular attention to differences in student submission behavior in response to changes in feedback timing.

4.1 Introduction

The growing demand for computer science education is resulting in a shift toward larger class sizes. In order to accommodate these larger classes, many computer science instructors have begun to utilize automated assessment technology where their students submit assignments electronically, and a significant portion of the assessment is performed by pre-written test cases and static analysis. Furthermore, a subset of these automated assessment systems provide students with real-time feedback and unlimited submission attempts up to the deadline making it possible for students to iteratively achieve mastery on their assignments. While these feedback and assessment systems support scaling class sizes with a minimal increase in human resources, little is known about the impact of such systems on student learning.

We created and deployed a real-time feedback and assessment system for the purposes of supporting scale in University of California, Santa Barbara (UCSB) computer science lower division courses. From others' prior work, and our own previous experience with real-time feedback and assessment systems, we knew that the use of our new system would significantly reduce assignment assessment time, permitting instructors and teaching assistants more time to work one-on-one with students in need of additional help. Our system was tested with 289 consent-giving students in a total of seven

instances of two UCSB computer science courses from Winter Quarter 2013 through Spring Quarter 2014.

Anecdotal evidence suggested that while the system permitted students to achieve success on assignments, the students were observed to rely on the system rather than develop their own testing and debugging skills — skills they were previously forced to develop in order to succeed in the absence of real-time feedback. We hypothesized that students who are able to receive significant feedback in any given period of time will take advantage of the system to the detriment of their testing and debugging skill development. Although our system does not evaluate these skills, it measures the effect that changes in feedback timing, referred to as feedback delay, have on student assignment progress. Therefore, we sought to measure this effect on students in attempt to discourage reliance upon the real-time feedback and assessment system, and thus support students' continued self-development of testing and debugging skills.

In this chapter, we present the results obtained by an analysis of 20,777 submissions made by 289 consent-giving students as previously described. We provide a general overview of student submission behavior in the presence of a real-time feedback and assessment system, and provide an analysis of the feedback delay's effect on student submission behavior.

The remainder of this chapter is organized as follows. We provide a brief summary of related work in Section 4.2. In Section 4.3 we describe the methodology of our study. We then present our results in Section 4.4, and finally, conclude in Section 4.5.

4.2 Related Work

A number of educators have designed and built automated feedback and assessment systems for programming assignments going back in time as far as 1960. A survey of the history and application of these systems was performed by Douce et al. in 1995 [14]. Ihantola et al. picks up where Douce et al. left off with a review of computer science related automated feedback and assessment system literature published between 2006 and 2010 [25]. Despite the plethora of related publications, little has been reported on student submission behavior in the presence of these systems. Only recently have researchers begun to look at the behavior of students utilizing automated feedback and assessment systems in order to gain insight into behaviors that are more likely to contribute to successful programming assignment completion.

Spacco et al. analyzed over 37,000 snapshots from ninty-six students collected using their Marmoset automated feedback and assessment system in Spring 2006. They correlated both starting early with better final scores, and the length of a work session with score improvement. In attempt to encourage students to start assignments earlier

than they would normally, Marmoset was designed with a renewable token component that would permit students to receive feedback from additional assignment test cases at most three times in a one-day period. Spacco et al. reported, however, that their data do not show significant evidence of students starting earlier in order to be able to utilize additional tokens [37, 38].

Edwards et al. analyzed nearly 90,000 assignment submissions from 1,101 students collected over a five year period beginning in Spring 2004 using Edwards's Web-CAT automated feedback and assessment system [15]. Edwards et al. found that, among students who did not score consistently across assignments, these students both started and finished earlier when receiving an *A* or *B* score, than when receiving a *C*, *D*, or *F* score. Furthermore, they also showed a general correlation between starting earlier and assignment score [16].

Helminen et al. reported on student programming and testing behaviors collected by their online code editor and execution environment in Fall 2012. While students were only required to submit their assignments through the environment, many used it for development and testing. With this environment, Helminen et al. were able to capture detailed student activity including when students started and stopped working, edits made to their code and associated tests, commands issued for testing, and when students made submissions. They found that few students took advantage of the automated

feedback provided by their system. Helminen et al. speculated this result was due to the already significant test coverage from test cases provided with the assignments [23].

Most recently, Falkner et al. looked at the impact of the granularity of assignment scores on student submission behavior. They found that student scores improved with an increase in assignment score granularity [17]. While it may seem intuitive that assignments with more precise scoring will generally improve student scores, their research provides evidence supporting this claim.

Overall, a side effect of these studies is an ever growing corpus of student submission behavior embedded within an extraordinary number of assignment submissions. These submissions and the student behavior they represent contain a surfeit of knowledge that computer science education researchers have only just begun to understand. We hope to reveal some of this knowledge by comparing analysis results from our study with previous results, and by looking at the impact of a delay in feedback on student submission behavior.

4.3 Methodology

In this section, we describe the classes in our study, explain the concept of a feedback delay, and briefly provide an overview of our system architecture.

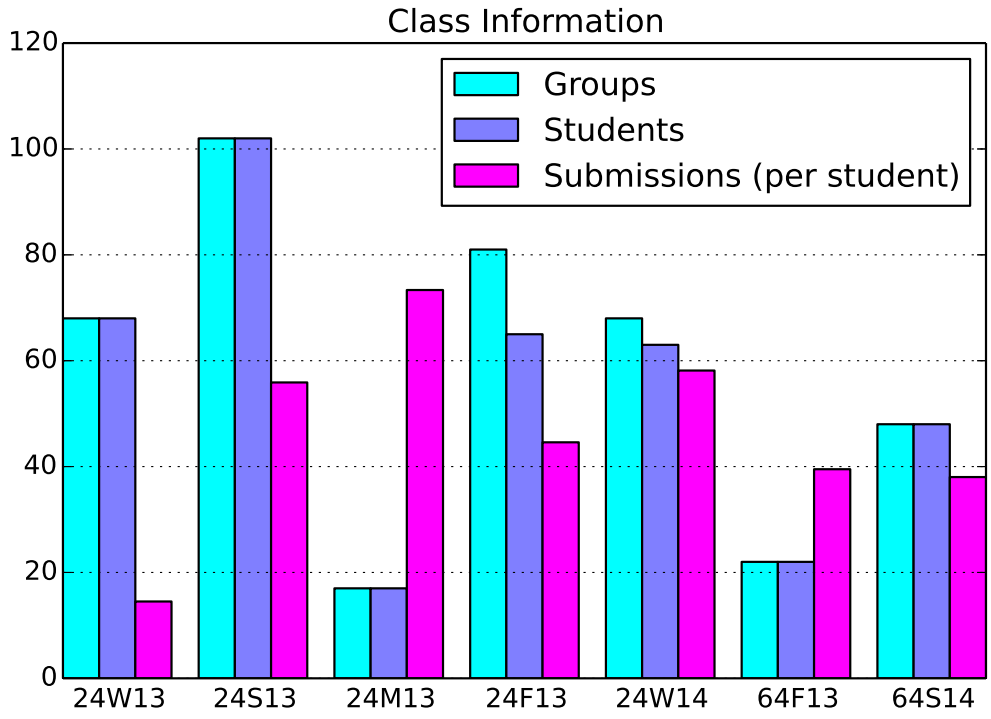

Figure 4.1: Visualizes the number of groups, students, and average number of submissions by student for each of the seven classes in our study.

4.3.1 Classes

Our study involves seven instances of two UCSB computer science courses from Winter Quarter 2013 through Spring Quarter 2014. The first course, *CS24*, is the second required course in UCSB's lower division computer science curriculum and builds upon students' prior knowledge of *C* in order to introduce them to data structures, and object oriented programming in *C++*. The second course, *CS64*, is a lower division computer architecture course that educates students on assembly programming and the basics of computer architecture, including digital design. In total, there are five instances

of *CS24*, and two instances of *CS64* represented. A single instructor taught *24M13* and *24F13*, and all others were taught by another individual instructor. All classes were taught during a ten-week quarter including *24M13*, the only summer instance represented.

While our feedback and assessment system supports both programming assignments and *fill in the blank*-type assignments, we only consider programming assignments as students were expected to make relatively many more submissions to *fill in the blank*-type assignments in order to reach the correct answers. Figure 4.1 provides useful information for each class in this study. The purple bars indicate the number of consent-giving students that made at least one submission, and the pink bars indicate the average number of submissions made by each student. Finally, the cyan bars indicate the number of unique groups that made at least one submission to any of a class's assignments. While most students formed the same groups across assignments, that was not a requirement. Thus, some students changed groups across assignments, and a few chose to work independently on some assignments. The latter are treated as single-student groups.

We distinguish between students and groups because our feedback and assessment system enforces an instructor-defined maximum group size per assignment. When the maximum group size is more than one, students are able to join into groups with other students up to the maximum group size. With regard to consent-giving students, we

only include submissions in this study if we have consent from all group members. This grouping functionality was introduced in September 2013, therefore it was only utilized by classes *24F13* and *24W14*.

Two classes have an average number of submissions which stand out. First, the average number of submissions is low for *24W13* due to only using the feedback and assessment system for half of the quarter. In the first half of the quarter, the students made submissions using an archaic submission system that provides no feedback. Second, the average number of submissions per student for *24M13* is relatively high due to students making post-deadline submissions as discussed in Section 4.4.4.

4.3.2 Feedback Delay

The primary educational purpose of a real-time feedback and assessment system is to provide feedback to students so that they may iteratively achieve mastery on their assignments. The use of these systems has positive side effects for instructors including reducing assessment time while increasing assessment equitability. As many instructors have previously observed, however, student usage of real-time feedback and assessment systems may result in dependency upon the system. This dependency could inhibit students from expanding their knowledge of compilation, execution, testing, and debugging processes.

Researchers have made various attempts to solve this dependency problem. Web-CAT ensures students develop testing skills by requiring students to submit test cases along with their assignment code [15]. While this approach appears successful to help students develop testing skills, it is not suitable for our purposes because it would require a change to our lower-division curriculum in order to emphasize testing. In another attempt, Marmoset restricts the frequency of running a subset of assignment test cases, called *release tests*, through a limited number of *release tokens*. While this notion of feedback reduction was also meant to encourage students to start assignments earlier, Spacco et al. indicated that the release tokens were seldom used [37]. This result suggests that the standard assignment test cases, which students could always get feedback from, provided sufficient coverage for students to complete their assignments. Furthermore, this observation may be indicative of a problem with requiring instructors to properly partition their test cases into *standard* and *release* tests.

We built our system in order to take an alternative approach that can be transparently utilized by UCSB's existing computer science curriculum, and requires minimal assignment configuration by instructors. Our approach is to introduce a configurable per-assignment feedback delay for submissions that occur within a short period of time to each other. For example, if the feedback delay is configured as five minutes, then students will receive immediate feedback from only one new submission in any five-minute window. Alternatively, if students make submissions exactly five minutes or

more apart, they will never experience any delay in receiving feedback. Our hypothesis was that as the feedback delay increased, students would spend more time testing their assignments prior to submission, with the result of both lengthening the time between submissions, and inflating the improvement in score between submissions.

In attempt to measure the effects of the feedback delay on student submission behavior, we increased the feedback delay in five-minute increments for each subsequent assignment in three of our classes: *24S13*, *24M13*, and *24F13*. In all other classes, the feedback delay was not intentionally altered between assignments. The impact of the feedback delay is detailed in Section 4.4.5 and Section 4.4.6.

4.3.3 The Feedback and Assessment System

In this section, we describe our rationale for creating a new real-time feedback and assessment system, as well as provide a high-level overview of the system's architecture.

Rationale

We chose to design and build our own system for a number of reasons. First, and foremost, we wanted to have a system that was easy to adopt into existing curriculum in order to encourage more instructors to use the system in their classes. We specifically

designed our system to match existing department submission and assessment work flows.

In a similar vein, we designed the *workers* — processes that execute student code as described in detail in Section **Architecture Overview** — such that they would run on existing lab machines in order to provide a consistent test and development environment without requiring additional resources from the technical support department. Running the workers in this consistent environment also contributes to instructor adoption of the system due to minimizing the distinct components instructors would require modifications to in the common event that they need specific software or libraries for one of their assignments. Additionally, by utilizing existing machines we are able to provide significant *worker* redundancy making it possible to have zero issues with the most volatile part of the system at no additional cost to our department.[1]

Finally, by building our own system we could ensure that we had total knowledge of all components of the feedback and assessment system. This knowledge allows us to easily adjust and control the various aspects of the submission, feedback, and assessment processes as necessary for both current and future research efforts.

In designing the feedback and assessment system we had two primary goals:

- Students should be able to make assignment submissions from either the web interface or lab machine terminals with little or no instruction.

[1] A redesign and implementation of this component was required in order to achieve this result. The system has since run with a peak activity for three months without a single issue.

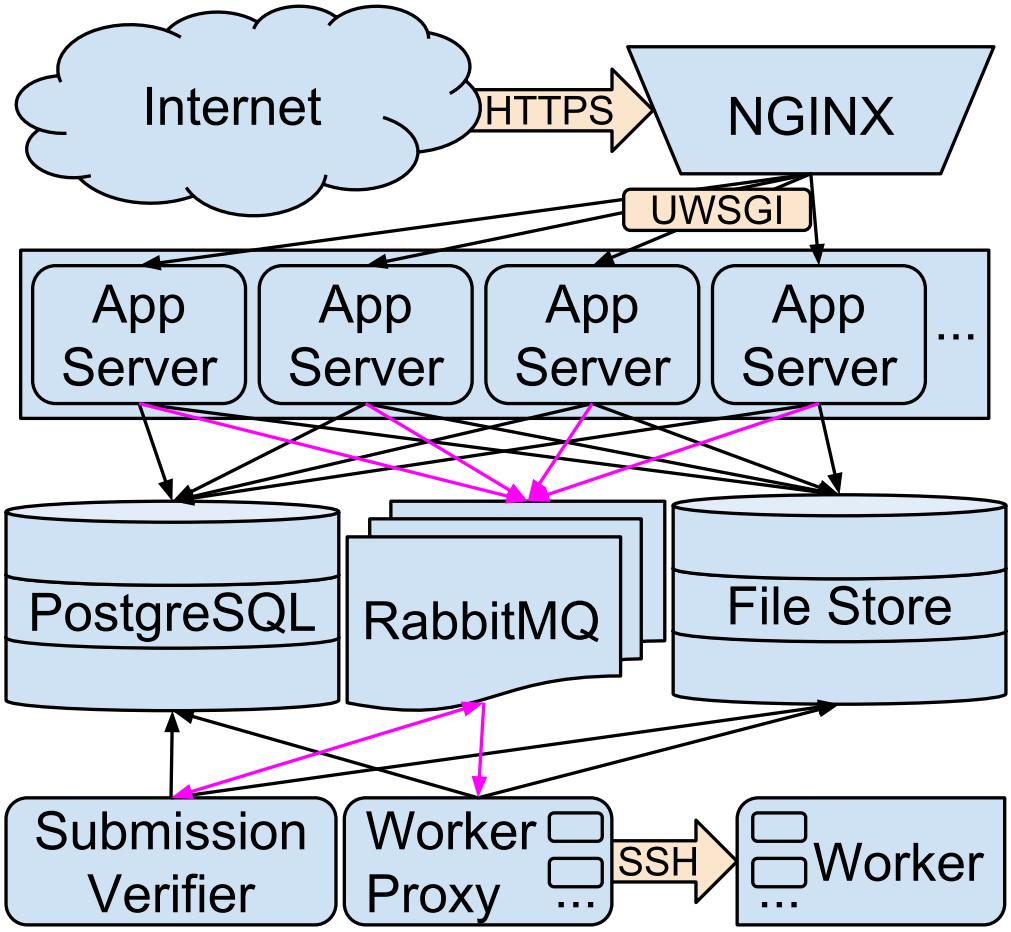

Figure 4.2: Provides an overview of the system architecture and how components interact. Pink lines indicate messages being passed to and from the RabbitMQ service. Note that each *worker* runs in a separate isolated environment.

- To reduce the overall assessment time for instructors and teaching assistants, including the time to prepare assignment test cases.

We believe the first goal was met due to the absence of complaints regarding usability of our system by the more than 300 students who have used it. We confirmed we met

the second goal when, on more than one occasion, an instructor had to find additional work for their teaching assistants due to a significant reduction in assessment time.

Architecture Overview

Figure 4.2 provides a diagram of the system architecture. In a nutshell, the primary interface to the system for instructors, teaching assistants, and students is their web browser. An *NGINX* web server distributes Internet HTTPS requests across a number of *app servers* that run the actual web service code. The system data is stored either in a *PostgresSQL* database, or deduplicated via an on-disk *file store*. A *submission verifier* process exists that checks new submissions for proper files prior to triggering one or more relatively resource expensive build and test jobs. A one-to-one mapping exists between a *worker proxy* and a *worker* where the *worker proxy* is responsible for selecting a machine for the *worker* to run on, initiating the build and test processes via the *worker*, and comparing the results generated by the *worker* to the assignment's expected results. *RabbitMQ* is used to pass messages that trigger the jobs run by the *submission verifier* and the *worker proxies*.

All of the components, save for the *workers*, run on a single machine as we have yet to experience any web service related performance issues. While there is a single point of failure at that machine, a manual failover to the development machine requires only minutes, with at most an hour of data loss. Moreover, providing redundancy on

these components is trivial because the system was designed to support this expansion pending available hardware.

In addition to the primary web interface, any number of additional interfaces can be created that communicate through the system's REST API. For instance, two such interfaces exist which both simplify a distinct task for command-line savvy users of the system:

- A submission creation program was written that creates a submission for a student by uploading the specified submission files. This program was written to provide transparency with the archaic non-feedback submission process.

- An assignment test case synchronization program was written that allows an instructor or teaching assistant to quickly synchronize an assignment on the system with the contents of a directory on their local machine. This program dramatically decreases the time to configure an assignment because, while it is easy to add test cases through the web interface, it can be tedious if there are more than a handful of them.

4.4 Results

While our initial motivation for this study was to look at the effect of the feedback delay on student submission behavior, the data we collected also allow us to offer new

insights into questions previously investigated by other researchers. The analysis of our submission data lends support to some existing answers to these questions, and contradicts others. In particular, we compare our results to those of Spacco et al. collected in Spring 2006. [37]. Where our results differ, we draw new conclusions using information from both our results, and theirs.

In this section, we present the results obtained from an analysis of the 20,777 submissions collected from seven UCSB computer science classes from Winter Quarter 2013 through Spring Quarter 2014. We seek to answer the following questions:

- Section 4.4.1: Does Starting Early Help?
- Section 4.4.2: Does Time Pressure Affect Behavior?
- Section 4.4.3: Does Time Pressure Affect Efficiency?
- Section 4.4.4: Why Do Students Submit Well After an Assignment's Deadline?
- Section 4.4.5: Does Delaying Feedback Impact Student Submission Behavior?
- Section 4.4.6: Does Delaying Feedback Impact Student Work Sessions?

4.4.1 Does Starting Early Help?

Many educators encourage their students to start early on assignments. Intuitively, starting early gives students more time to receive feedback from their instructor and

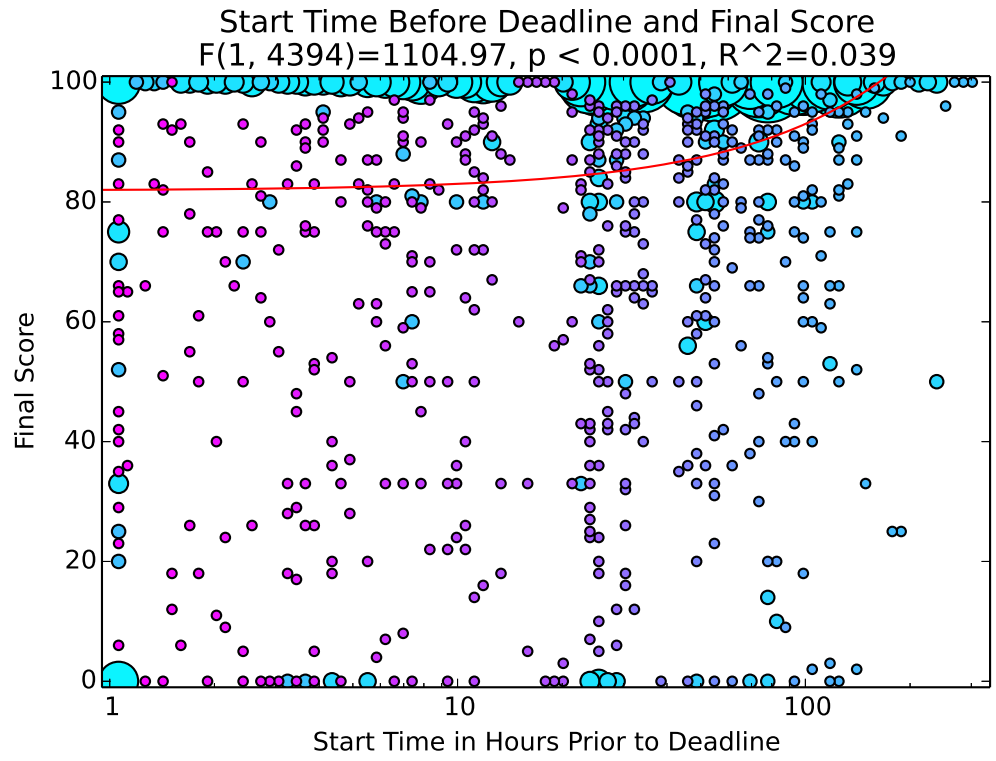

Figure 4.3: Compares the number of hours groups started an assignment before its deadlines to the final score they received. Both the size and color of each circle correspond to the number of groups represented at that position. The circles are plotted such that smaller circles are strictly in front of larger circles. The red line represents a best-fit trend-line of the data.

teaching assistants in order to make improvements to the work they submit by the deadline. However, with traditional assessment, is it unlikely for an instructor to offer multiple early assessment iterations to all students. With a real-time feedback and assessment system, on the other hand, starting early additionally offers all students multiple opportunities for assignment feedback and assessment. Furthermore, the feedback

provided by these systems can then be leveraged by the student and instructor in office hours.

Prior work in this field has shown that a positive correlation does in fact exist between starting early and assignment score [16, 37]. We sought to verify that our results are consistent with those of the prior work.

Figure 4.3 plots the number of hours between a group's first on-time assignment submission and the corresponding assignment deadline against the final score the group receives. Our data statistically significantly correlate earlier assignment start times with higher scores. While it may seem odd that groups can receive 100% on an assignment having started only an hour or less prior to the deadline, this result is merely an artifact of our active data collection. Our data collection methodology only provides a lower bound to how long prior to assignment deadline a group began working. We verified that a small number of groups, distributed uniformly across assignments, would make their first submission in the last hour. This behavior indicates that some groups mostly worked without using the system in order to receive feedback.

While Figure 4.3 shows a correlation with start time and final score, we can break down relative start times even further to a per-assignment basis. Figure 4.4 compares the average score of the first 10% of groups to make a submission on an assignment, to that of all groups, and to the last 10% of groups to make their first submission. The assignments are sorted according to the first 10% value, and then the last 10% value.

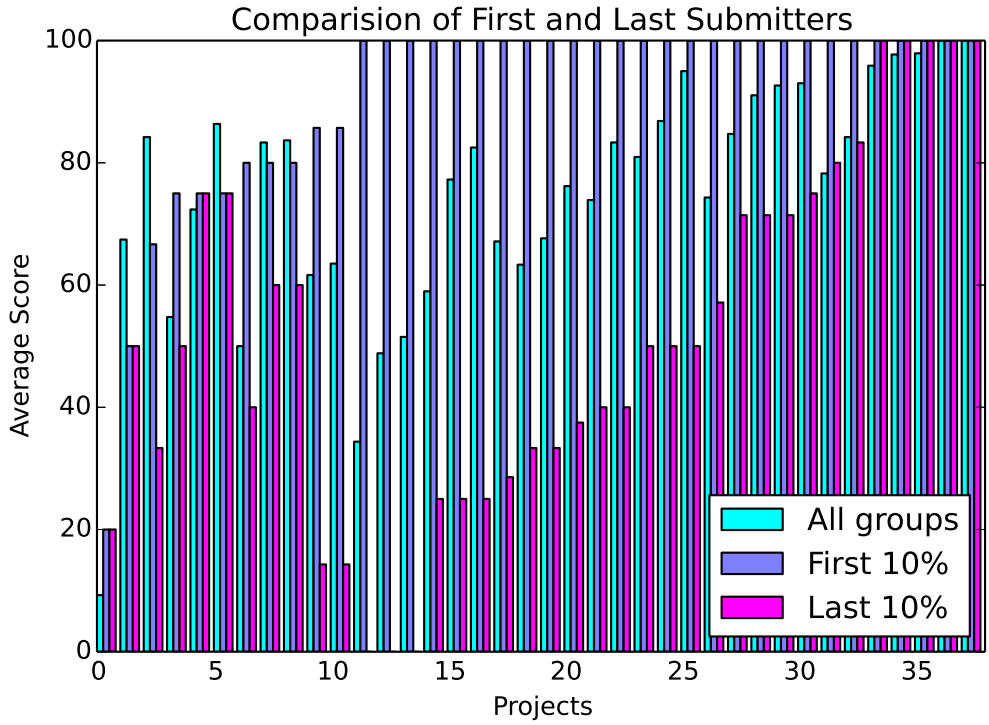

Figure 4.4: Compares the average final score of the first 10% of groups to submit to the average and to the last 10% of groups to submit by assignment. The first 10% of groups to submit had perfect scores on twenty-seven of the thirty-eight assignments.

Assignments with fewer than thirty groups are excluded so that at least three groups make up each of the 10% categories. Of the thirty-seven assignments that meet the criteria, the first 10% of groups all received 100% on twenty-six (70%) assignments, only five (20%) of which, the last 10% of groups also all received 100%. In the other twenty-one, the last 10% scored significantly worse than the average.

In the cases where the first 10% of groups did not receive 100%, there are a few outliers where the average is higher than the first 10% of groups to submit. All of those five cases were lab assignments where each student attended one of many lab sessions.

It is likely that a subset of the first 10% of groups to submit all emanate from an early lab session where they were directed to make their first submission at a point in their development process that they would not otherwise have made it.

4.4.2 Does Time Pressure Affect Behavior?

Motivated by the work of Spacco et al., we wanted to see if our real-time feedback and assessment system, which required students to actively make assignment submissions, measured similar student working behavior to their passively collected snapshots. Spacco et al. discovered their students produced the most work in days prior to the deadline at 4PM. The amount of work significantly dropped at 6PM and remained nearly consistent until 1AM. Although all of their assignment deadlines were at 6PM, they attributed the peak in work between 4PM and 6PM as the time that students preferred to work and thus suggested that "setting the deadline a couple of hours later might allow students to work at their preferred time without the added pressure of an impending deadline" [37]. Our results indicate otherwise.

Figure 4.5 depicts the number of submissions made at each time of day for submissions made more than a day from their deadline. We observe a steady increase in the number of submissions beginning at 9AM and peaking at 4PM. This peak is followed by a decrease in submissions through 6PM. Our results are nearly identical to that of Spacco et al., however, rather than observing a consistent amount of work for the re-

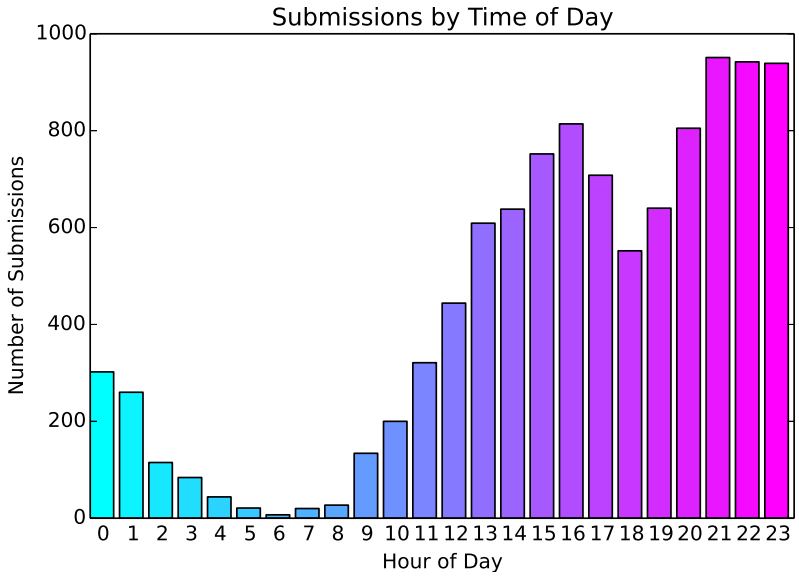

Figure 4.5: Visualizes the time of day submissions were made excluding submissions within a day of their deadline. Note the 4PM peak and the larger peak starting at 9PM that continues through midnight.

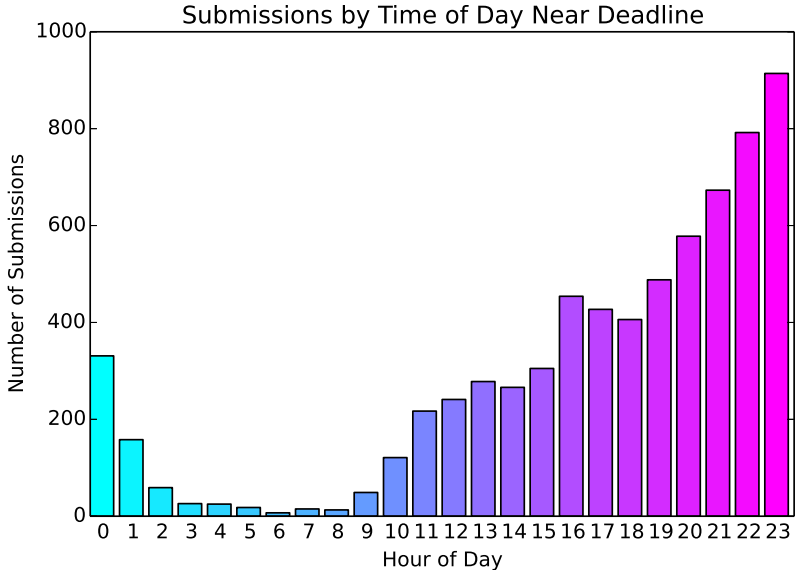

Figure 4.6: Visualizes the time of day submissions were made including only submissions within a day of their deadline. The 11PM peak corresponds to the hour prior to the deadline for most assignments.

mainder of the night, we instead observe an even larger increase in work until 9PM where the amount of work then remains nearly consistent until midnight. Thus, while our students are also productive between 4PM and 6PM, they are even more productive in the three hours before midnight. Coincidentally, seventy-one out of seventy-six (93%) of our assignments had a midnight deadline. This observation combined with the results of Spacco et al. lead us to believe that students learn to work most efficiently in the hours just prior to the time of day of an expected deadline, regardless of the proximity in days to the deadline.

As previously indicated, we excluded submissions from Figure 4.5 that were made fewer than twenty-four hours from their respective assignment deadline. Our hypothesis was that a more significant majority of the submissions would be made in the hours just prior to their assignment deadline. Figure 4.6 confirms that hypothesis. While there is little difference in the figure shape prior to 11AM, there is only a slight increase in work during the 11AM to 3PM range. A sharp spike in submissions occurs at 4PM and has a gradual decrease until 6PM. This decrease in submissions occurs in both figures, and we suspect this decrease corresponds with the time students leave campus, head home, and eat dinner, prior to resuming work. Finally, we observe a consistent increase in work right up to midnight, the most common deadline.

It is important to note that this data include an insignificant amount of error. Prior to the introduction the feature enabling students to create groups using our feedback

and assessment system that reflect their actual assignment groups, it was common for multiple members of a group to make independent submissions to the system, often only submitting the final complete version of the assignment. We detected and excluded eighty-seven subsequent exact duplicate submissions (0.4%). Of these, seventeen (20%) occurred in the 11PM hour. However, only nine (10%) occurred in the hour prior to their deadline. While we detected and excluded subsequent identical submissions, we do not do the same for nearly identical submissions because the error they introduce is insignificant. We come to this conclusion by assuming there are a similar number of undetected non-exact duplicate submissions, and that these submissions have a similar hour-prior to deadline distribution.

4.4.3 Does Time Pressure Affect Efficiency?

The previous section describes how submission behavior is altered by assignment deadlines. In this section, we look at the effect of a pending deadline on submission efficiency. There are a number of ways we could define submission efficiency. One metric is to look at the amount of change in source code between submissions. Another is to look at the change in cyclomatic complexity between submissions. While each of these metrics may provide interesting insights into student behavior, we are more interested in correlations with changes in groups' scores between submissions. Thus, we consider efficiency by looking at improvements and regressions in subsequent submis-

sions based on the change in score between submissions. A change in score is a result of a subsequent submission passing more or fewer test cases. We quantify changes in submission efficiency by classifying subsequent submissions into one of four categories:

Improvement

The submission increases the group's maximum score on the assignment.

No Improvement

The submission has the same score as the group's maximum score.

No Improvement 2

The submission's score is less than the group's maximum score and is not lower than the local-minimum score.

Worse

The submission results in a local-minimum score. That is, it is the lowest score since the last *Improvement* submission.

Figure 4.7 depicts the total number of submissions made in the days prior to each submission's respective deadline according to their respective category. The figure only extends to seven days, as the number of submissions more than seven days prior to their deadline is insignificant. Our results show that a majority of submissions are made in

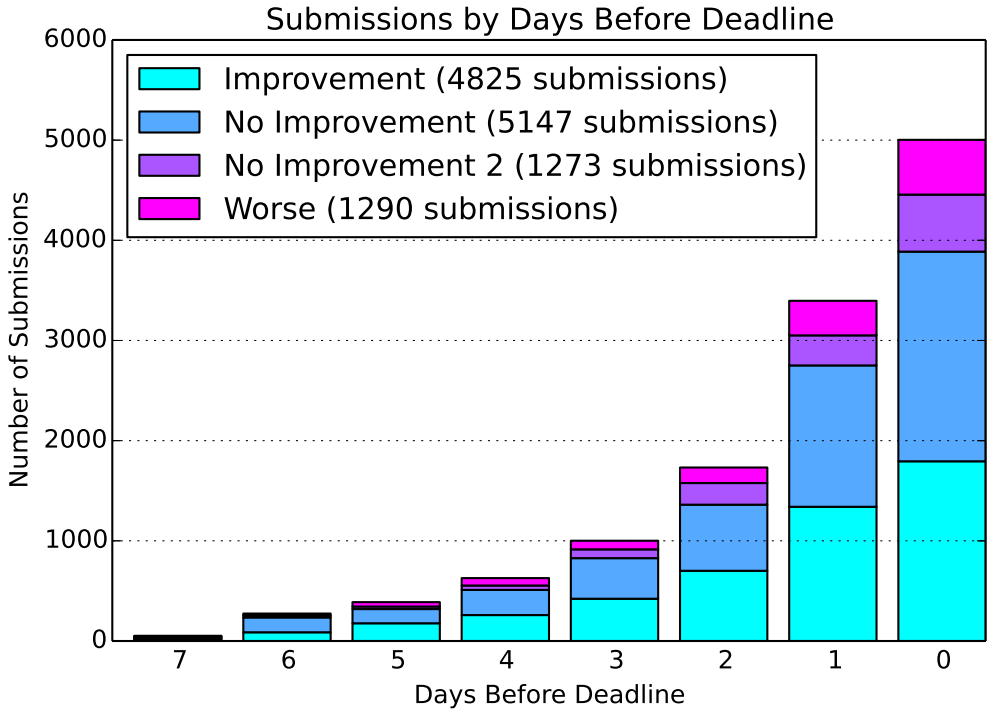

Figure 4.7: Shows the number of submissions by the number of days each submission was made prior to their deadline grouped by improvement category. Submissions that improve the group's maximum assignment score are labeled *Improvement*, and those that tie are labeled *No Improvement*. *Worse* submissions are those that result in a local minimum, and all submissions between the group's maximum assignment score and the local minimum are labeled *No Improvement 2*.

the two days prior to their deadline; this observation is consistent with Spacco et al. In contrast, however, we observe a much higher percentage of *Improvement* submissions when compared to Spacco et al.'s *positive* snapshots [37].

The likely reason for this discrepancy is the difference between their passively collected snapshots and our actively collected submissions. While a snapshot may not represent a complete unit of work, a submission often does because groups explicitly

make submissions in order to receive feedback. One other difference is our *Improvement* submissions are only submissions that improve upon a group's maximum score. It is unclear from Spacco et al.'s description if a snapshot that improves the score of a *negative* snapshot is considered *positive* even if its score does not improve the group's maximum score. If this is the case, then the number of *positive* snapshots is inflated in their results as compared to ours.

Regardless, we think a comparison of efficiency between submissions is more interesting than a comparison between snapshots, because, despite submissions representing a unit of work, there are still a significant number of submissions that are not *Improvement*. Figure 4.8 shows the relative percent of submissions in each category by the number of days prior to their deadline. Overall, the difference in submission efficiency is insignificant with respect to the number of days prior to assignment deadline. While our deadlines were distributed such that 24% and 25% of submissions were made to assignments with a Monday and Friday deadline respectively, there was no difference in submission efficiency with respect to the day of the week a submission was made.

An analysis of the hour of the day of each submission also resulted in no significant changes in submission efficiency. Thus, these results convince us that time pressure does not affect submission efficiency.

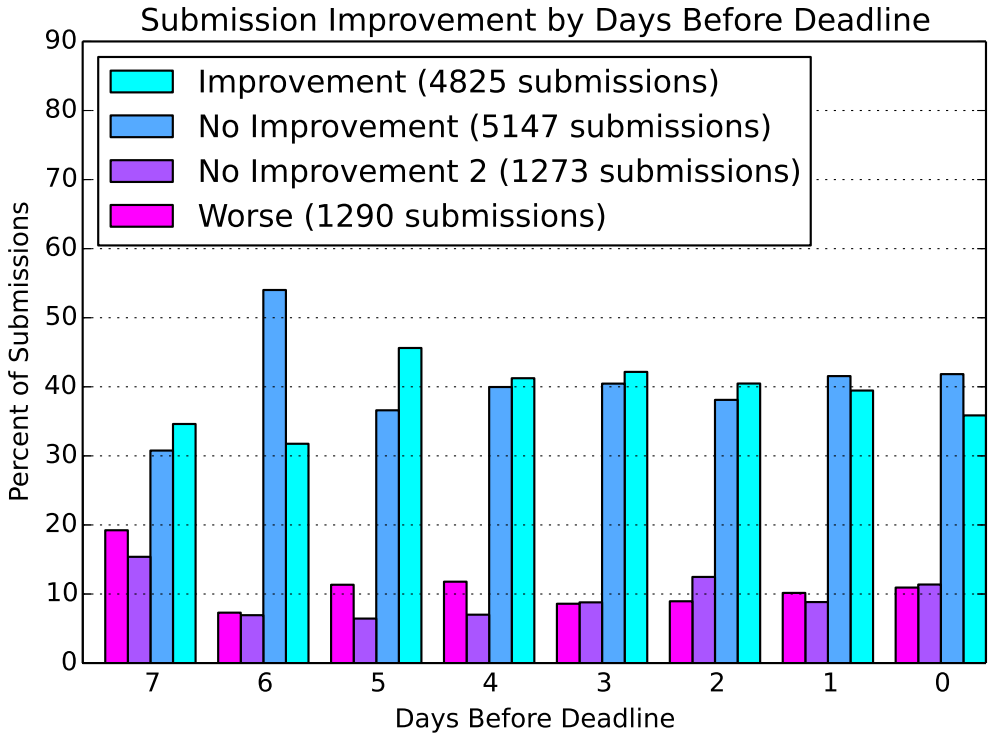

Figure 4.8: Depicts the percentage of submissions in each improvement category by the number of days each submission was made prior to its deadline.

4.4.4 Why Do Students Submit Well After an Assignment's Deadline?

An interesting aspect of real-time feedback and assessment systems that prior work has not touched upon, is student usage of these systems beyond an assignment's deadline in order to make improvements to their work and verify the correctness of those improvements. These systems inherently provide this functionality, and some students take advantage of it.

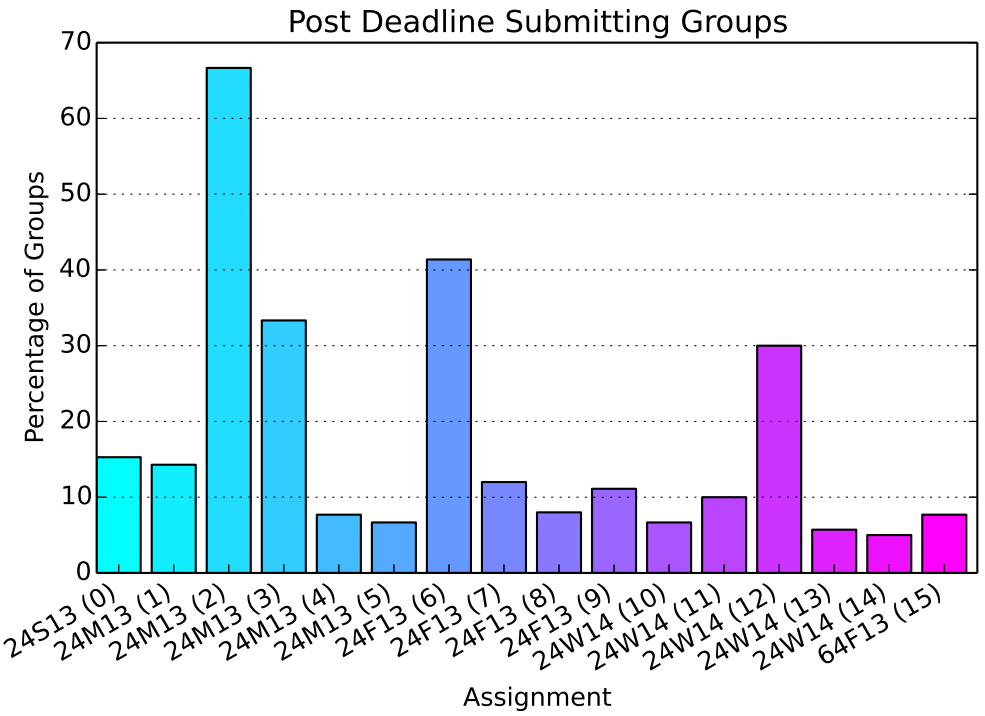

Figure 4.9: Shows the percentage of groups that submit more than two days following an assignment's deadline. The x-axis groups the assignments by class.

Figure 4.9 shows the percentage of groups by assignment that made one or more submissions more than two days following an assignment's deadline. Of the seventy-six assignments, only sixteen are shown in the figure. Five (6.6%) assignments were excluded for representing fewer than ten groups. A majority of these assignments were the first assignment of a class for which we had only received a portion of the consent forms. Twenty-nine assignments (38.2%) had zero post-deadline submitting groups and are therefore not shown. Finally, twenty-six (34.2%) assignments had only between 1%

and 4% (average 2.4%) of groups with post deadline submissions. These assignments are excluded from the figure for space purposes.

A minimum of two days following an assignment's deadline was chosen because by this time, all students were beyond any late-credit that may have been offered across all assignments. Except where otherwise noted, submissions occurring after this two-day period provided the student with no direct grade-benefits.

Five of the assignments in Figure 4.9 are from *24M13*, which comprised only seventeen consenting students whereas the next smallest class, *64S14*, comprised forty-eight consenting students. *24M13's* small class size is significant as a larger percentage of the students were able to receive assistance in office hours from both the teaching assistant and the instructor.

Additionally, five of the assignments are from *24W14*. In this class, half of the groups submitting post deadline did so for more than one assignment. We discovered that the majority of these post deadline submissions occurred just prior to one of *24W14's* course examinations. This discovery suggests that a number of students utilized the real-time feedback and assessment system as a tool for exam preparation. Messages on the course discussion group confirmed students utilized the feedback and assessment system to improve upon previous course assignments as a form of studying.

Overall, four of the assignments, *2*, *3*, *6*, and *12*, stand out from the remainder. The first two, *2*, and *3*, were assignments that subsequent assignments depended on. Thus,

as reported by the instructor, many students sought help during office hours to correct issues with the former assignment prior to moving on to the latter. The instructor reported that the real-time feedback and assessment system was invaluable during office hours for its capability to efficiently verify correctness of modifications to students' code without exposing the test cases, nor requiring the instructor to manually obtain and test the students' in-progress work. Assignment *6* presented students with the opportunity to make-up missed points after the deadline, thus not surprisingly, explaining its spike in post deadline submissions. Finally, assignment *12* had both a number of students revisit prior to a class exam, and was a dependency of a subsequent assignment.

In summary, our results show that there are three primary reasons why students continue to work on assignments well after the deadline:

- Intuitively, the most prominent reason we observed is to make up points lost on an assignment. While abusing this functionality may result in students not taking initial assignment deadlines seriously, the ability for instructors to easily reassess student work provides a paradigm of assignment assessment that has never before been feasible.

- The second most prominent reason we observed is due to inter-assignment dependency. When an assignment depends on the work of a former assignment,

a number of students found it useful to first verify correctness of improvements made to the former assignment before advancing to the latter.

- Finally, we observed a small number of students who made improvements to past assignments as part of studying for their examinations.

Overall, we consider any submissions made after the deadline to be a success of the real-time feedback and assessment system. Without lowering the barriers to additional feedback, these students may not have made any effort to improve their comprehension of the material through improvements to their past assignments.

4.4.5 Does Delaying Feedback Impact Student Submission Behavior?

As indicated in Section 4.3.2, we sought to measure the impact of altering assignment feedback delay on student submission behavior. Here, we first look at the impact of the feedback delay on the time between subsequent submissions made by the same group. Figure 4.10 plots the time between subsequent submissions grouped by assignments sharing the same feedback delay value to a five-minute precision and combining those with a feedback delay of more than thirty minutes. Note that the x-axis is shown in a log-scale, and the size and color of each circle represents the relative number of submission gaps represented by that circle. For instance, with a feedback delay of ten

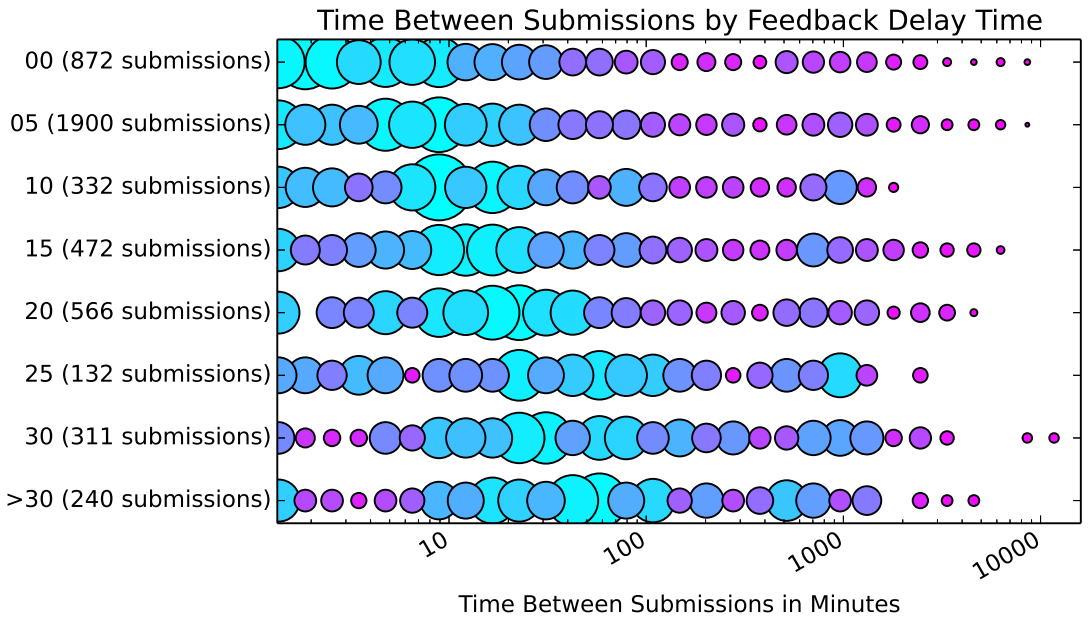

Figure 4.10: Plots the time between submissions grouped by assignment feedback delay. Note the shift to a longer time between submissions in the most significant portion of each row (indicated by the largest circles) as the feedback delay increases.

minutes, the most significant gap between subsequent submissions is around ten minutes as indicated by the bright cyan large circle in that position. This figure clearly shows by the shift in position of the bright cyan large circles in each row that as the feedback delay increases, so does the most significant grouping of subsequent submission gaps. This result indicates that delays in feedback affect student submission behavior.

Figure 4.11 shows the relative efficiency of submissions grouped by the feedback delay. This figure appears to indicate that there is a significant improvement in submission efficiency with delays of thirty minutes or more. Using Student's t-test, we

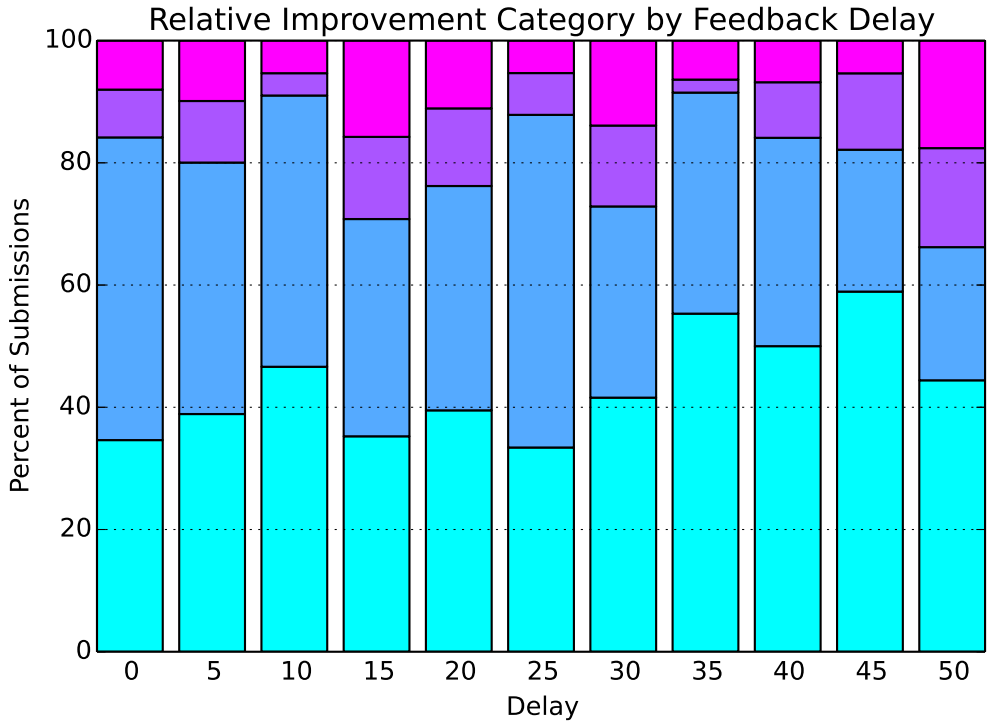

Figure 4.11: Plots the percent of submissions in each improvement category for each five-minute delay interval from zero to fifty. Refer to Figure 4.7 for the legend and its description.

compared the percent of *Improvement* submissions for each feedback delay fewer than thirty minutes, to those of each feedback delay thirty minutes or longer. The difference was statistically significant with P=0.0095.

For comparison, Figure 4.12 depicts the time between submissions for each of the improvement categories. The aggregate results from the figure indicate that the most common time between subsequent submissions is approximately ten minutes. Furthermore, there is a consistent spike in time between submissions just prior to the 1,000-minute mark. This time corresponds with a diurnal working pattern of our students.

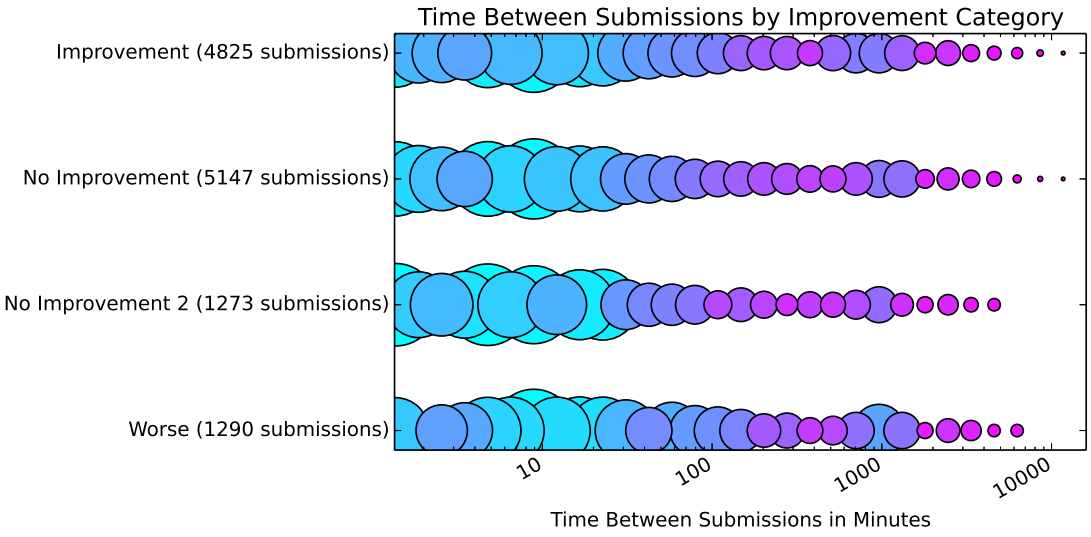

Figure 4.12: Plots the time between submissions by their improvement category.

Overall, there is not a significant difference in time between submissions with respect to improvement category. However, the shape of the individual scatter lines provides two insights:

- Relatively the most *No Improvement 2* submissions occur in the first few seconds as indicated by the size and color of the *No Improvement 2* category's first circle compared to the first circle of other categories. Recall that *No Improvement 2* submissions occur in the period after a *Worse* submission and prior to an *Improvement* submission. This short period of time between these submissions and their corresponding former submissions is too small for a group to have understood any feedback received, suggesting these groups did not independently test whatever changes they made prior to resubmission.

- Interestingly, with respect to *Worse* submissions, the spike around the 1,000-minute mark is the most significant compared to other categories. This data suggest that long breaks have an initially negative impact on assignment progress.

Our results confirm that changes in the feedback delay have an impact on student submission behavior. In general, as the feedback delay increases, students wait longer to submit, and when comparing delays of less than thirty minutes to those of thirty minutes or more, the students are more likely to improve upon their previous score with longer feedback delays.

4.4.6 Does Delaying Feedback Impact Student Work Sessions?

Section 4.4.5 showed that a delay in feedback has an impact on both the time between submissions and the likeliness for a group to make an improving subsequent submission. In this section, we attempt to group submissions into a work session. Conceptually, a work session is a continuous period of time that students are actively working on an assignment. Multiple work sessions are separated by periods of inactivity that may be due to sleep, distraction, other work, or some other form of break. Grouping multiple submissions into a work session provides another level of depth to insight on student behavior. We use these groupings to both look at the effect of the feedback delay on work sessions, and to compare our work session results to those of

prior work. While our data collection methodology does not allow us to express work sessions with great precision, we make an approximation.

We define work sessions similarly to Spacco et al. Specifically, we define a work session as a collection of submissions by the same group for the same assignment in which all subsequent submissions are made within some window of time to its prior submission. We refer to this window of time as the *window size*.

Determining an Appropriate Window Size

We investigate the ideal window size for which to discover work sessions. Spacco et al. arbitrarily chose a window size of twenty minutes [37]. While this window size may have been appropriate for their data, where snapshots were collected passively upon changes to students' code, it is not appropriate for our data due to the fact that students actively submitted only when they desired feedback. Furthermore, it would not make sense for us to define a window size fewer than fifty minutes due to inclusion of assignments with a feedback delay of fifty minutes, where groups regularly make no more than one submission in any fifty-minute period. Thus, the window size we select must be at least fifty minutes in length.

There are two forms of error that we must mitigate when selecting a window size:

- The first error is due to not being able to distinguish between work and non-work time occurring between submissions in a work session. While a student may

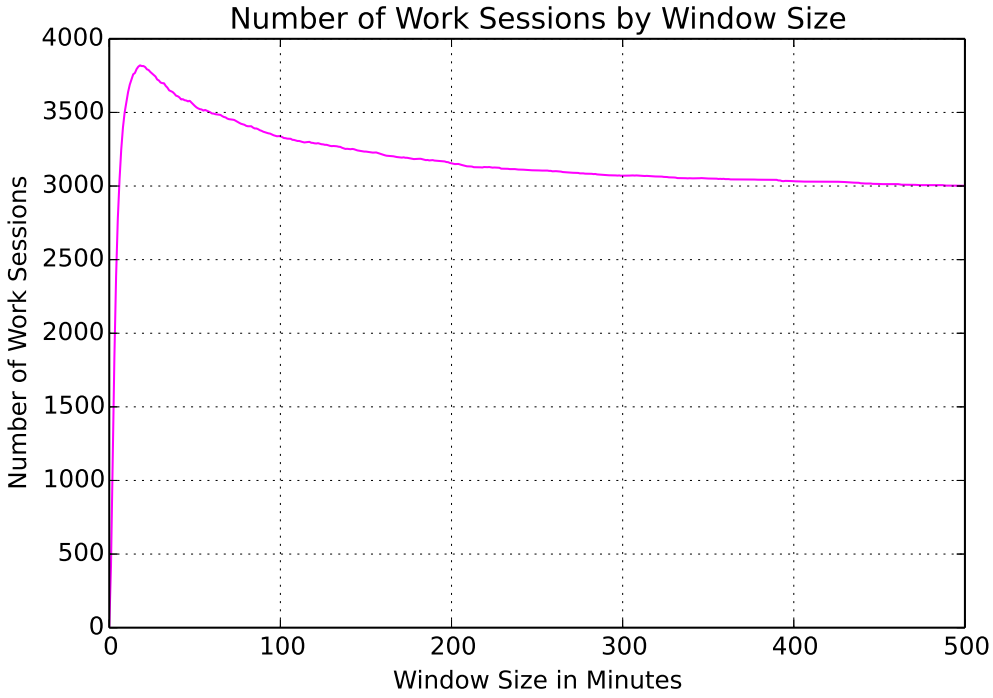

Figure 4.13: Plots the number of work sessions as the window size increases.

make two submissions in a period of time shorter than the window size, they may not have worked for the entire period between those two submissions. Intuitively, this error is reduced by minimizing the window size due to the fact that any non-work periods longer than the window size will not be included as part of the work session.

- The second error is a result of selecting a window size that is too small to encompass actual periods of student working time between two submissions. For instance, if we select the window size as sixty minutes, then this error corre-

sponds to the number of subsequent submissions representing more than sixty minutes of actual work.

Although we can measure the number of subsequent submissions over a given window size, we cannot measure either error due to a lack of information as to when students are actually working. We accept these errors, and assume that they equally effect working sessions independent of the feedback delay time.

Rather than arbitrarily choosing a value, we attempt to select an ideal window size based on features of our data. We use the maximum session length as a heuristic for limiting the window size, as it is unlikely that more than a handful of all sessions are longer than eight hours in addition to time to account for the error in work time between two submissions. Figure 4.13 shows the effect of increasing the window size on the number of work sessions created. This figure reveals that for our data the maximum number of work sessions occurs with a window size of approximately twenty minutes, after which the number of work sessions gradually decreases. This decrease indicates that as the window size grows, the number of work sessions merged together is more significant than the number of new work sessions created by the grouping of two independent submissions. Aside from the twenty-minute peak, there are no other points of interest in this figure.

Figure 4.14 plots a number of lines corresponding to various work session lengths as the window size increases. The four non-vertical lines correspond to:

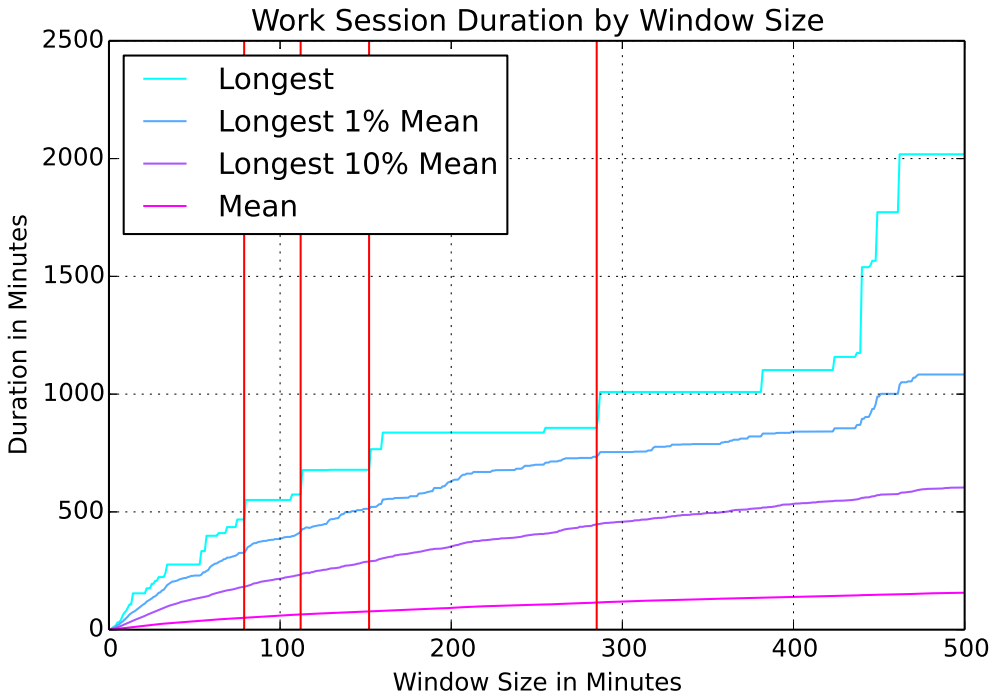

Figure 4.14: Shows the change in work session duration as the window size changes. The red vertical lines indicate points of interest due to significant changes in the longest duration work session. The red lines occur at window sizes seventy-nine, 112, 152, and 285.

- the duration of the single longest work session
- the mean duration of the top 1% of work sessions sorted by length
- the mean duration of the top 10% of work sessions sorted by length
- the mean duration of all work sessions

Of these four lines, we find only the duration of the single longest work session to be of interest as there are a number of distinct window sizes that result in increasing

the longest duration. The red vertical lines on the figure highlight the window sizes of interest, i.e., they are the window sizes just prior to a significant increase in maximum work session length. We excluded consideration of points of interest at larger window sizes due to the infeasibility for 10% of all work sessions to be over eight hours in length. Additionally, we exclude the point of interest that occurs just after fifty minutes due to its proximity with our longest feedback delay.

We select the left-most point of interest, seventy-nine minutes, as the window size we use in the remainder of the results section. Note, however, that where statistical significance is concerned, we verified that each highlighted window size in Figure 4.14 produces consistent results with those produced using the seventy-nine minute window size. This comparison shows that our analysis is unaffected by the specific choice of window size from the options we highlighted with red lines.

Properties of Work Session Lengths

Before considering the impact of the feedback delay on work sessions, we first compare a few general properties of our work sessions to those of the passively collected work sessions of Spacco et al.

Having defined a window size, we are able to group submissions into work sessions, and thus approximate the length of a work session as the time between the first and last submission in a work session. Rather than looking at the improvement between

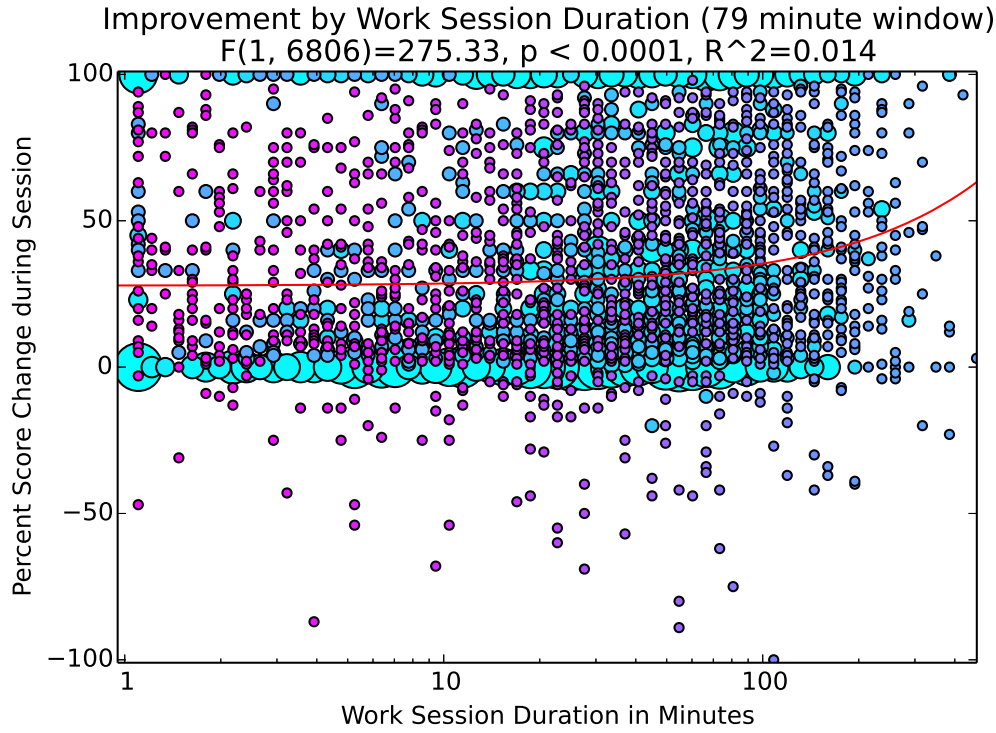

Figure 4.15: Depicts a positive correlation between work session duration and percent score change. The results are statistically significant according to an F-test.

individual submissions as we did in Figure 4.12, Figure 4.15 plots the percent score change made between the first and last submission in a work session against the length of the work session. In this figure, circles at 0% score change on the y-axis would be considered *No Improvement*, and a vast majority of changes in work sessions would be considered *Improvement* due to the significant imbalance between the number of sessions that improve the score when compared to those that reduce the score. The red line represents the trend-line and an F-test of the data confirms that there is a statistically significant positive correlation between the length of a work session and the percent

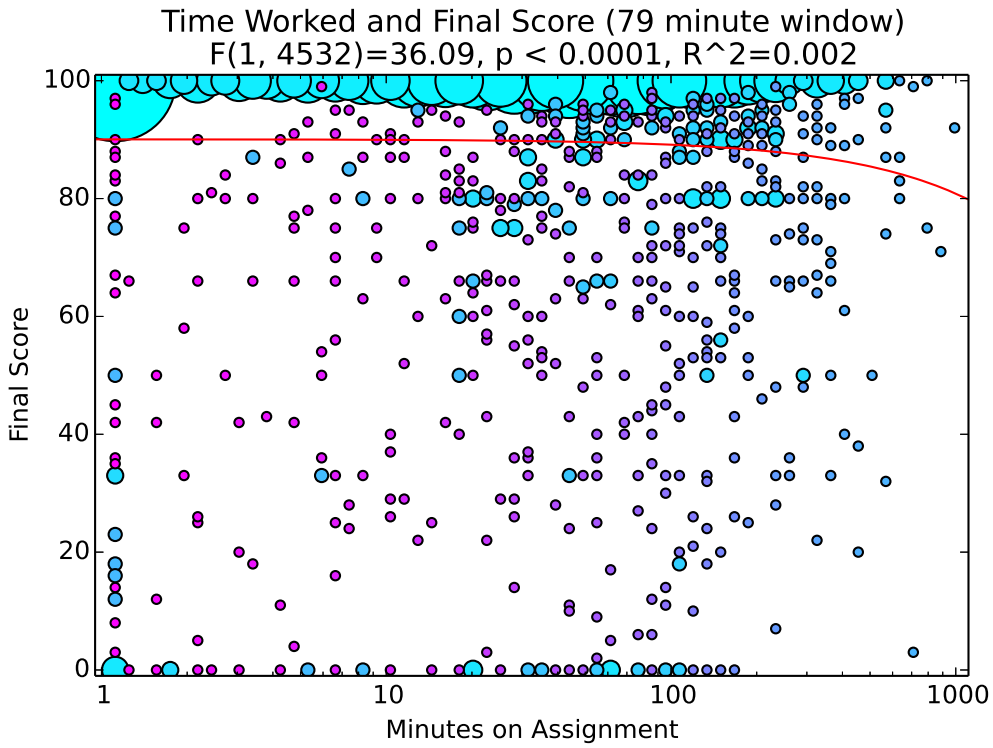

Figure 4.16: Depicts a negative correlation between the minutes spent on an assignment and the final score. The results are statistically significant according to an F-test.

change in score. This analysis indicates that longer work sessions are more likely to result in increased improvements in score. These results are consistent with those of Spacco et al. [37].

We approximate the total time spent on an assignment by summing the length of all the work sessions by group for an assignment. Figure 4.16 plots the final score compared to the total time spent on an assignment. The red trend-line shows there is a statistically significant negative correlation between the total work time and the final score. This negative correlation indicates that the longer a group works on an

Score	Count	Mean Work Time	stderr	No Progress	stderr
0%	142	54m	6	50%	4
0%–100%	616	119m	6	58%	2
100%	1509	60m	2	34%	1

Table 4.1: Lists properties of group work times (79 minute window) grouped by those that scored 0%, between 0% and 100%, and 100%. The *mean work time* is the mean time groups in each grouping spent working on the assignment along with the corresponding *stderr*. *No progress* represents the mean percent of time that groups in each grouping spent without improving their maximum score.

assignment, the less likely they are to score well. These results contradict the results of Spacco et al. where they found a statistically significant positive correlation between the two.

This result intuitively makes sense under the assumption that a majority of groups who do not complete an assignment do not do so due to a lack of effort. Although groups who complete an assignment are more likely to start earlier, these groups, on average, spend much less time working on an assignment. Table 4.1 confirms this intuition by showing that the mean work time of groups who receive 100% on an assignment is nearly half that of all groups who receive scores between 0% and 100%, with little error. The *No progress* column shows that groups who complete an assignment spend a larger percentage of their time improving upon their assignment score,

whereas groups, who do not, spend over 50% of their assignment time without making forward progress.

That begs the questions, why are groups who complete an assignment likely to spend less time on it, and why are these groups more efficient? While we cannot precisely answer these questions, we offer two possible explanations:

- In general, successful students simply may have a better understanding of assignment material, resulting in more productive work sessions, and overall reducing the amount of time to completion.

- Successful students are more likely to start earlier, thus providing them with more opportunity to attend office hours. Students who attend office hours gain useful insights to an assignment, resulting in more forward progress, and an overall reduction in the time required to complete an assignment.

From the opposite perspective of both explanations we can understand why unsuccessful students may spend more time on an assignment. Without a directed approach to completing an assignment, these students may figuratively spin their wheels requiring significant time with little, if any, progress. In such cases, if these students are not performing their own testing, and waiting for feedback from the system, delays in feedback may have a negative effect on student performance.

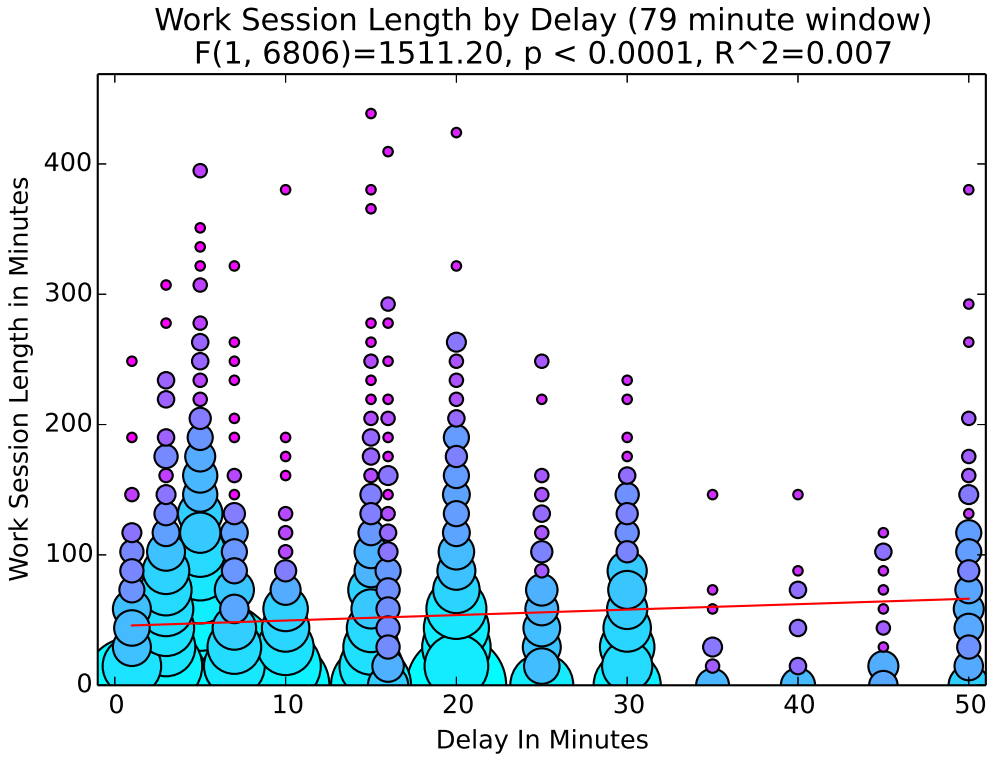

Figure 4.17: Plots work session length against feedback delay. There is a slight, nevertheless, statistically significant positive correlation between the two.

Impact of Feedback Delay on Work Sessions

Finally, we consider the impact of the feedback delay on work sessions. We first consider the effect a change in feedback delay has on the length of a work session. Figure 4.17 shows that there is a statistically significant, though slight, positive correlation between the feedback delay and work session length when using a seventy-nine minute window size to group submissions. The correlation is consistent, though more prominent when using the larger window sizes as shown by the red lines in Figure 4.14.

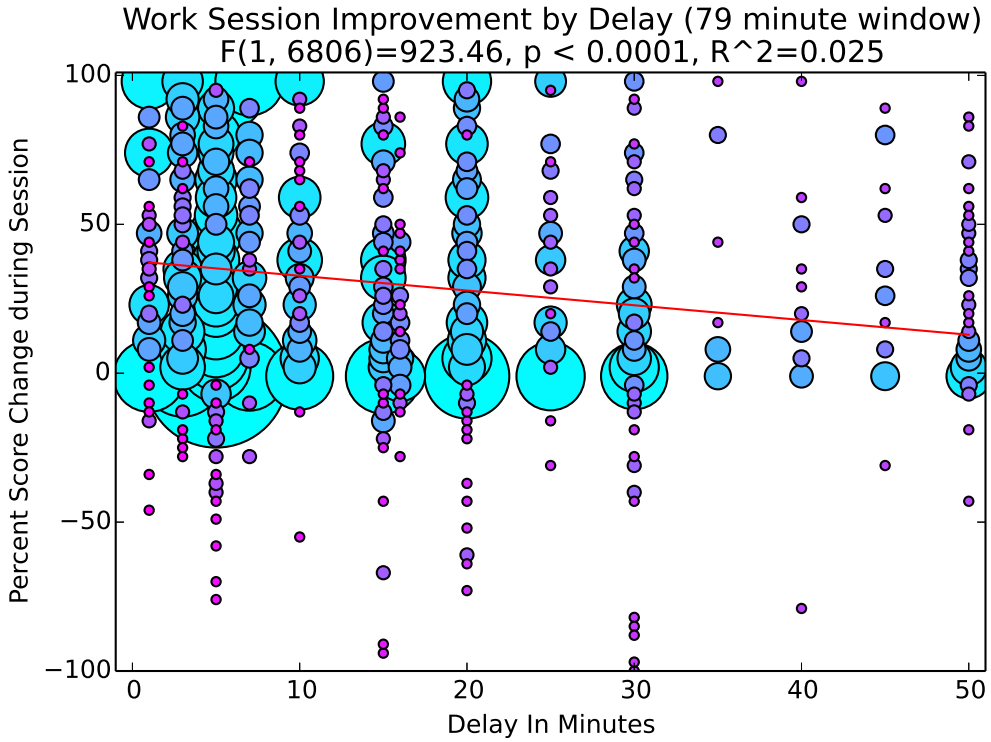

Figure 4.18: Plots work session improvement against feedback delay. There is a statistically significant negative correlation between the two.

Despite this correlation, we cannot absolutely attribute this increase in work session length to the feedback delay because assignments with delays over thirty minutes were only given in the latter half of the courses. It is common for these assignments to be more difficult than the initial assignments, and as a result require more time to complete.

Second, we look at the impact of the feedback delay on improvement between the first submission in a work session and the last submission in a work session when grouped by assignment feedback delay. Figure 4.18 shows that there is a statistically

significant negative correlation between work session improvement and assignment feedback delay. While this result may seem puzzling at first, it is logical. With a smaller feedback delay, groups have more opportunities to receive feedback and may therefore make more significant improvement within a single work session. Thus, while their submission efficiency may be lower, the net result is increased work session improvement. Conversely, with fewer opportunities for feedback, groups working on assignments with longer feedback delays have higher submission efficiency but the relative amount of improvement within work sessions is not as significant.

4.5 Conclusion

In this chapter, we discover student submission behaviors through the analysis of 20,777 submissions made by 289 students across seven classes. The data were collected by a real-time feedback and assessment system we created that allowed a per-assignment feedback delay to be configured. Our results show that delaying feedback impacts student submission behavior. Furthermore, delays of at least thirty minutes positively affect submission efficiency, as we defined it, when compared to smaller delays. Our results also suggest that delaying feedback impacts student work sessions in two ways:

- Increases in delay correlate with longer work sessions (Section 4.4.3).

- Increases in delay correlate with less improvement during work sessions (Section 4.4.6).

The aggregate result of the feedback delay suggests that assignments should be configured with a thirty minute feedback delay. Our results also provide an interesting comparison to prior work:

- We confirm that starting early correlates with higher assignment scores (Section 4.4.1).
- We confirm a high period (not peak) of student activity between 4PM and 6PM regardless of deadline (Section 4.4.2).
- We identify peak student activity occurring in the hours prior to the time of a deadline both on the day of the deadline and others. Due to our differences with prior work, we hypothesize that students adjust their peak working hours to align with pending deadlines (Section 4.4.2).
- We confirm that a majority of activity is completed in the two days prior to assignment deadline (Section 4.4.3).

In addition to our comparison with prior work we offer some other new insights:

- We discover that there is no difference in submission efficiency due to proximity with an assignment deadline (Section 4.4.3).

- We show that students take advantage of the ability to continue using the feedback and assessment system in order to receive feedback after the deadline (Section 4.4.4).

- We show that, within reason, the selection of the window size used to group submissions into work sessions is irrelevant (Section 4.4.6).

Overall, this study provided an insight into a few aspects of student submission behavior. While more research is required to fully understand student submission behavior, our results should help guide instructors toward ideal assignment configuration with respect to feedback delay and assignment deadlines in an effort to improve student success.

Chapter 5

Conclusion

As discussed in this dissertation, it is critical that we in the computer science community do whatever possible to increase the number of new computer scientists and prepare them for the challenges they must meet in industry, education, medicine, science, and so on. The increase in enrollments in response to job demand will impact all levels of the educational system. My research contributes to this effort. First, I demonstrate the effectiveness of static analysis in both the post-assessment of a Scratch-based 6^{th}–8^{th} grade summer camp and the development of a Scratch-based 4^{th}–6^{th} grade classroom curriculum. Second, I report on the submission behavior of university computer science students in the presence of a real-time feedback and assessment system. The significance of this collective research is to support the growth in number of students who seek computer science education, and to do so while maximizing student performance.

My work is but a single step on the journey to increasing the yearly number of new computer scientists by both increasing student interest in computer science and maximizing the learning potential of those studying computer science. Continued research across all levels of computer science curricula from primary school through university is necessary to complete this journey. In primary school, static analysis can be incorporated into the student feedback and assessment cycle much like I have done with university assignments. However, it is unknown how younger students will respond to such feedback, thus there is much to be done with respect to how best to provide feedback for students of various ages and topic mastery. In both areas, the continued application of machine learning across collected data sets can be used to understand how students best solve certain programming assignments in order to differentiate successful approaches from unsuccessful approaches to solving common programming problems.

I envision a future where years from now, many students with exposure to and successful completion of our primary school computer science related curriculum will ultimately choose a college major that involves some degree of computational thinking. Their university assignments will include electronic submission, and will be designed by their instructors to provide them with optimal feedback at the optimal time to maximize their understanding in the least amount of time. Students of the future will spend

less time to learn more and instructors will have more time to work with students requiring additional assistance.

This evolution in student learning is possible only through analysis and assessment of student in-progress work via studies similar to those I performed. The iterative application and subsequent measurement of new and altered techniques will not only advance computer science education at a fundamental level, it will also make it possible to educate increased numbers of computer science students without a proportional increase in instructional resources. The resulting cohort of well-educated computational thinkers with shared knowledge and concrete skill sets will be able to solve more of our real-world problems. I hope we will all see this future.

Bibliography

[1] J. C. Adams and A. R. Webster. What do students learn about programming from game, music video, and storytelling projects? In *SIGCSE '12*, pages 643–648, 2012.

[2] S. Alliance. The stars alliance: A southeastern partnership for diverse participation in computing. NSF STARS Alliance Proposal.

[3] I. Arroyo, R. Walles, C. Beal, and B. Woolf. Effects of web-based tutoring software on students' math achievement. In *AERA*, 2004.

[4] T. Bell, I. H. Witten, and M. Fellows. *Computer Science Unplugged*. 2006.

[5] P. Black and D. Wiliam. Assessment and classroom learning. *Assessment in education*, 5(1):7–74, 1998.

[6] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 215–220, New York, NY, USA, 2013. ACM.

[7] K. Brennan and M. Resnick. New frameworks for studying and assessing the development of computational thinking. In *AERA*, 2012.

[8] Q. Burke and Y. B. Kafai. The writers' workshop for youth programmers: digital storytelling with scratch in middle school classrooms. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 433–438. ACM, 2012.

[9] Code.org. What's wrong with this picture?, 2013 (accessed July 26, 2014). `http://code.org/stats`.

[10] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st*

ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, pages 673–674. ACM, 2006.

[11] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 191–195. ACM, 2003.

[12] W. Dann, S. Cooper, and R. Pausch. Making the connection: Programming with animated small world. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSEconference on Innovation and Technology in Computer Science Education*, ITiCSE '00, pages 41–44, New York, NY, USA, 2000. ACM.

[13] J. Denner, L. Werner, and E. Ortiz. Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Comput. Educ.*, 58(1):240–249, Jan. 2012.

[14] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), Sept. 2005.

[15] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 148–155. ACM, 2003.

[16] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 3–14, New York, NY, USA, 2009. ACM.

[17] N. Falkner, R. Vivian, D. Piper, and K. Falkner. Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 9–14, New York, NY, USA, 2014. ACM.

[18] C. S. T. Force. *CSTA K–12 Computer Science Standards*. Association for Computing Machinery, 2011.

[19] A. Forte and M. Guzdial. Computers for communication, not calculation: Media as a motivation and context for learning. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4 - Volume 4*, HICSS '04, pages 40096.1–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] D. Franklin, P. Conrad, G. Aldana, and S. Hough. Animal tlatoque: Attracting middle school students to computing through culturally-relevant themes. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 453–458, New York, NY, USA, 2011. ACM.

[21] D. Franklin, P. Conrad, B. Boe, K. Nilsen, C. Hill, M. Len, G. Dreschler, and G. Aldana. Assessment of computer science learning in a scratch-based outreach program. In *Proceedings of the 44th SIGCSE technical symposium on Computer science education*, SIGCSE '13. ACM, 2013.

[22] S. L. Halgren, T. Fernandes, and D. Thomas. Amazing animation: Movie making for kids design briefing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 519–525, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[23] J. Helminen, P. Ihantola, and V. Karavirta. Recording and analyzing in-browser programming sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 13–22, New York, NY, USA, 2013. ACM.

[24] C. S. Hood and D. J. Hood. Teaching programming and language concepts using legos®. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 19–23, New York, NY, USA, 2005. ACM.

[25] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[26] D. Jackson and M. Usher. Grading student programs using assyst. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, SIGCSE '97, pages 335–339. ACM, 1997.

[27] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[28] E. Lazowska, E. Roberts, and J. Kurose. Tsunami or sea change? responding to the explosion of student interest in computer science, July 2014. `http://lazowska.cs.washington.edu/NCWIT.pdf`.

[29] C. M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. In *Proceedings of the 41st ACM Technical Symposium*

on Computer Science Education, SIGCSE '10, pages 346–350, New York, NY, USA, 2010. ACM.

[30] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[31] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: Urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 367–371, New York, NY, USA, 2008. ACM.

[32] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[33] T. Radvan. Kurt. `https://github.com/blob8108/kurt`, September 2012.

[34] L. Seiter and B. Foreman. Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 59–66, New York, NY, USA, 2013. ACM.

[35] B. Simon, P. Kinnunen, L. Porter, and D. Zazkis. Experience report: Cs1 for majors with media computation. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, pages 214–218. ACM, 2010.

[36] L. Snyder, T. Barnes, D. Garcia, J. Paul, and B. Simon. The first five computer science principles pilots: summary and comparisons. *ACM Inroads*, 3(2):54–57, June 2012.

[37] J. Spacco, D. Fossati, J. Stamper, and K. Rivers. Towards improving programming habits to create better computer science course outcomes. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 243–248, New York, NY, USA, 2013. ACM.

[38] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th annual*

SIGCSE conference on Innovation and technology in computer science education, ITICSE '06, pages 13–17. ACM, 2006.

[39] J. P. Spradley. *Participant observation*. Holt, Rinehart and Winston, 1980.

[40] The College Board. Program summary report 2013, 2013. http://research.collegeboard.org/programs/ap/data/participation/2013.

[41] A. Wilson, T. Hainey, and T. Connolly. Evaluation of computer games developed by primary school children to gauge understanding of programming concepts. In *Proceedings of the 6th European Conference on Games-based Learning*, ECGBL '12, 2012.

[42] U. Wolz, C. Hallberg, and B. Taylor. Scrape: A tool for visualizing the code of scratch programs. Poster presented at the 42nd ACM Technical Symposium on Computer Science Education, Dallas, TX., March 2011.