Hairball: Lint-inspired Static Analysis of Scratch Projects

Bryce Boe
UC Santa Barbara
bboe@cs.ucsb.edu

Charlotte Hill
UC Santa Barbara
charlottehill@cs.ucsb.edu

Michelle Len
UC Santa Barbara
mlen@cs.ucsb.edu

Greg Dreschler
UC Santa Barbara
gdreschler@umail.ucsb.edu

Phillip Conrad
UC Santa Barbara
pconrad@cs.ucsb.edu

Diana Franklin
UC Santa Barbara
franklin@cs.ucsb.edu

ABSTRACT

Scratch programming has risen in prominence, not only as a potential language for K-12 computer science, but also in introductory college courses. Unfortunately, grading Scratch programs is time-consuming, requiring manual execution of each program. Automation of this process is greatly complicated by the very reason Scratch is an attractive introductory language—the projects are multimedia in nature, requiring eyes and ears to fully appreciate.

We propose Hairball, an automated system that can be used both by a student to point out potential errors or unsafe practices, and by a grader to assist in inspecting the implementation of Scratch programs. Because automatic analysis will not be able to determine the sensory effect, Hairball focuses instead on the implementation, including safe/robust programming practices, providing a "lint-like" tool for Scratch.

In this case study, we have created an initial set of Hairball plugins that detect and label instances of initialization of Scratch state, synchronization between say and sound blocks, synchronization between broadcast and receive blocks, and use of timing and loops for complex animation. Our evaluation shows that Hairball is very useful in conjunction with manual analysis. Overall, Hairball was actually slightly more accurate than manual analysis at labeling these instances. Specifically for broadcast/receive, Hairball's analysis correctly classified 99% of the 432 instances, manual analysis only 81%. Overall, if Hairball was only used to identify correctly implemented instances, with manual analysis for the remainder, it would remove 76% of the instances for the manual analysis and assist in the rest, with a false positive rate of less than 0.5%.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: General

General Terms

Languages, Measurement

Keywords

Scratch; automated assessment; static analysis

1. INTRODUCTION

There is a movement toward less industrial and more engaging projects and languages for introductory and AP computer science courses. This movement includes the push for Python with Multimedia approaches [1, 7, 13], the various approaches to the AP CS Principles course [14], as well as Alice [4] and Scratch [11].

One drawback of visual and auditory projects 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 [9]. More recently, unit testing frameworks have been employed as part of automated assessment [6,15]. When students are given creative freedom with a sensory project—which is an integral feature of languages such as Alice and Scratch—there is neither a text-based output file with which to diff, nor a straightforward way to perform unit testing. For Scratch, for example, evaluation typically requires that each project be individually opened and run. Inspection of the code requires many mouse clicks and navigation through the stage, sprites, and all of their associated scripts.

To assist with assessment of Scratch projects, 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 [10]), we call our system Hairball. We propose two roles for Hairball: (1) formative assessment—inspired by lint, we envision students using Hairball to check their Scratch programs for potential problems, and (2) summative assessment—accelerating manual analysis of assignments by verifying the presence and correctness of certain programming constructs as well as directing the manual analysis toward potential problems. We have developed a plugin architecture so that, in Python, Hairball can be extended and adapted for evaluation of specific assignments.

The challenges we explore in this paper 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, and is particularly helpful for identifying instances of particular 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 project.

We begin in Section 2 by giving background on automated analysis in general and Scratch in particular. We then describe our Hairball framework in Section 3. Section 4 describes the Hairball plugins we developed for our analysis. We describe our methodology and results in Sections 5 and 6. Finally, in Section 7 we conclude.

2. BACKGROUND AND RELATED WORK

Providing automation for analyzing programs is not a new concept. ASSYST is an automated assessment system that performs end-to-end, or input/output, type testing of submissions [9]. Both the Marmoset system by Spacco et al., and the Web-CAT system, by Edwards, perform testing of code using student written unit tests [6, 15]. All of the aforementioned tools supplement the feedback the student receives with code coverage analysis and feedback from static analysis tools such as FindBugs by Cole et al. [3]. Douce et al. performed a more detailed analysis of existing automated assessment systems [5]. The problem with existing assessment systems is that they are not applicable to Scratch programs.

Scratch is a block-based programming language from MIT [11]. Programs consist of 2-dimensional interactive animations. Objects, or sprites, move on the screen as a result of user input or scripts in the program. Sound and video can also be integrated into Scratch programs. Scratch was designed to allow students to learn computer science projects while employing great creativity in their work. This creative freedom is one of the reasons that Scratch projects are challenging to analyze.

An additional challenge in Scratch analysis is that Scratch programs are developed and run within a graphical user interface (GUI). 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, programs might begin when a parallel set of scripts beginning with a "when green flag clicked" hat block are triggered.

At least two other projects have 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 programs from the Imaginary Worlds summer camp [1]. Additionally, Burke and Kafai developed Scrape [16] as a visualization tool to aid humans in understanding patterns across Scratch files. Scrape was used to assess Scratch projects produced in a middle school writing workshop [2]. Scrape is useful in answering questions such as:

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

Hairball has two purposes. Like Scrape, Hairball can help verify the use of the required constructs of an assignment. The main contribution of Hairball, however, is the framework and set of available plugins that support more sophisticated analysis. We want to answer questions not just about the use of Computer Science (CS) constructs, but about the competence demonstrated for different CS concepts. Hairball can be used to answer questions such as:

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

3. DESIGN CONSIDERATIONS

We have two goals in designing Hairball. Our first goal is to perform analysis on a set of Scratch programs automatically. Without automated analysis, each Scratch program must be opened manually in order to be inspected and executed. 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, and anyone can make use of available plugins.

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 CS classes. However, Python was mainly chosen due to 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 [12].[1]

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*) and 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 program.

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

4. HAIRBALL PLUGINS

In this section we describe four Hairball plugins written to perform Scratch static analysis. The plugins were designed to analyze projects submitted as part of our two-week interdisciplinary Animal Tlatoque summer camp [8]. Notice that some of the traditional topics, i.e., variables and conditionals, are not represented, and that loops are represented in a very specific way (defined as animation). The plugins target the CS concepts used in the camp's cumulative project, an interactive movie about an animal. For this project, students were to demonstrate state initialization, use of broadcast and receive blocks, synchronization between say and 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 strives to evaluate whether, or to what extent, the program demonstrated competence in an area. These plugins attempt 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

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

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, glide z secs to x,y, etc.	go to x,y go to x,y
Size	change size by x%	set size to x%
Background	next background	switch to background x

Table 1: Five categories of initial state, along with example relative and absolute modification blocks

was implemented correctly. Instances labeled "semantically incorrect" should indicate that the concept was implemented in a way that may not always work when executed. Intuitively, 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 were discovered. A single 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 statically whether the code runs the same way in these two events. For reasons described below, the environment is different than in traditional programming.

The first problem is where to start the analysis. In traditional programs, execution starts at "main". Scratch programs have no such globally-defined starting point. We taught our students to start their programs using the green flag button, so 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 one of the program's scripts modifies the attribute. 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 such blocks can appear in a script. We call this location the "initilization 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 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 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.

The initialization plugin considers a modified attribute of a sprite as correctly initialized 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/Sound Synchronization

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

Achieving this effect is complicated by the timing semantics of the two forms of the say block, and the two forms of the play sound block in Scratch. One form of the say block places the speech bubble on the screen indefinitely (until replaced by another say block, or "erased" with an empty say block), while the other 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 sound synchronously, while "play sound" plays the sound asynchronously.

There are two methods to produce the desired effect. The first is to asynchronously play the sound via the "play sound block" followed by a "say for" block with a 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, is to use a "say" block to display the message, followed by a "play sound until done" block, ending with an empty "say" block to clear the previous speech bubble. The campers were taught the latter method as the correct approach.

This plugin detects instances of this concept by looking for sequential say and 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 and sound blocks but do not match either of these methods are labeled incorrect, and isolated uses of say or sound blocks are labeled incomplete.

Broadcast and Receive

One use of Scratch's broadcast blocks is to trigger the execution of other sprites' scripts beginning with the appropriate receive 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 broadcasting an event. In the camp's cumulative project, campers demonstrated an understanding of the broadcast and receive concept by triggering scene changes in their interactive movie.

The broadcast and receive plugin verifies that for each broadcast or receive event, there is a broadcast block and at least one corresponding receive block. Such instances are labeled correct. All instances with a broadcast block appearing in the same script with another instance's broadcast 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 purposes of 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" block built into Scratch. One example of complex animation is realistic motion of sprites that represent people and animals. E.g., people walking, birds flying and snakes slithering. Creating such animations requires the correct integration of several Scratch and CS concepts. For example, creating an animation sequence where a sprite spins around requires integration of loops, rotation, and timing.

A necessary component of a complex animation instance is either rotation blocks, or motion blocks paired with costume change blocks. We define a complex animation instance as either a loop containing these necessary components or a sequence of these necessary components, since a 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 sequences instead of loops as semantically incorrect because the student did not demonstrate competence in the CS concept of loops. Finally, if the program is missing any critical element, e.g., repetition, it is labeled incomplete.

5. METHODOLOGY

In the remainder of this paper, we will use Hairball to refer to both the framework and the set of plugins described in section 4.

We tested Hairball on the projects submitted during our two week summer camp. There were five assignments total, with a distribution of concept requirements. For example, complex animations were taught toward the end of the camp, so they were only present in the last two assignments whereas initialization was present in all [8].

We first perfomed a manual analysis on all 58 of the submitted Scratch projects. Three members of our project staff independently analyzed the first five projects submitted for a given assignment using a common rubric. We discussed any discrepancies in our scores and after coming to agreement, we analyzed the remaining projects. 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. Hairball was run on all of the projects. When there were discrepancies between Hairball and the manual analysis, there was a second manual analysis to determine which was correct—Hairball or the manual analysis. In the results section, we compare the results between Hairball and the reconciled manual analysis using the second analysis results as a ground truth.

Because the projects 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 analysis. The methodology is not perfect because Hairball was informed by the manual analysis - sort of like a fourth entity whose answers needed to be reconciled in the group. As the results show, Hairball did an excellent job of identifying issues that all three of our staff members missed.

6. RESULTS

In this section, we present the results of using Hairball to assist in determining the level of competence demonstrated by students' Scratch projects for several CS concepts. For each concept, we will compare the labels Hairball assigned to instances of the concept

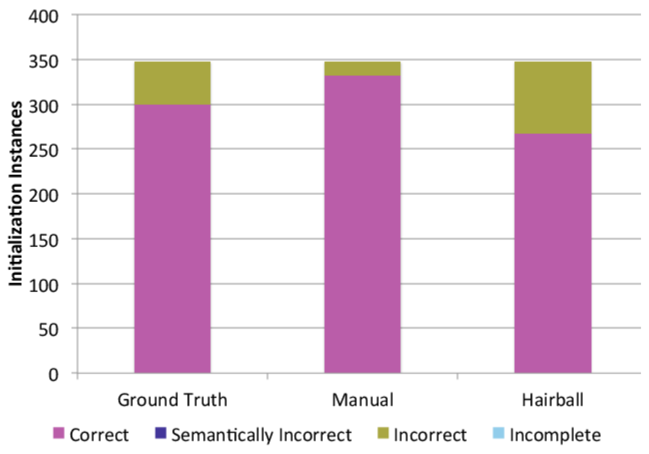

Figure 1: Compares the initialization instance labels. Note that, this analysis used only the "correct" and "incorrect" labels. Manual analysis resulted in 32 false positives, and Hairball resulted in 33 false negatives.

with those assigned via manual analysis. We will use the second manual analysis as the ground truth, and look at both false positive and false negative rates for Hairball and the manual analysis. 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.

6.1 Initialization

We begin with initialization. Recall from section 4 that Hairball looks for attributes that are modified, and expects to find a corresponding absolute block in the initialization zone in order to consider an instance correct. The manual analysis, on the other hand, only involved running the program twice, and confirming that the two executions matched.

Figure 1 provides the classification of the 348 initialization instances discovered across the 58 projects. Of the 65 instances that Hairball and the manual analysis labeled differently, Hairball was correct for 32 of the instances. Many of the remaining 33 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.

6.2 Sound/Say Synchronization

Figure 2 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 31 other instances. Hairball, identified 229 correct instances and 37 others. Manual analysis and Hairball failed to find two and four instances respectively.

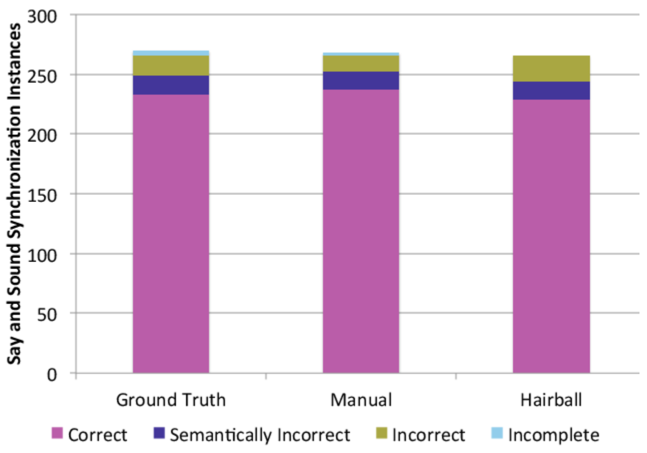

Figure 2: Compares the say and sound synchronization instance labels. Manual analysis and Hairball failed to detect two and four instances respectively and manual analysis resulted in 4 false positives.

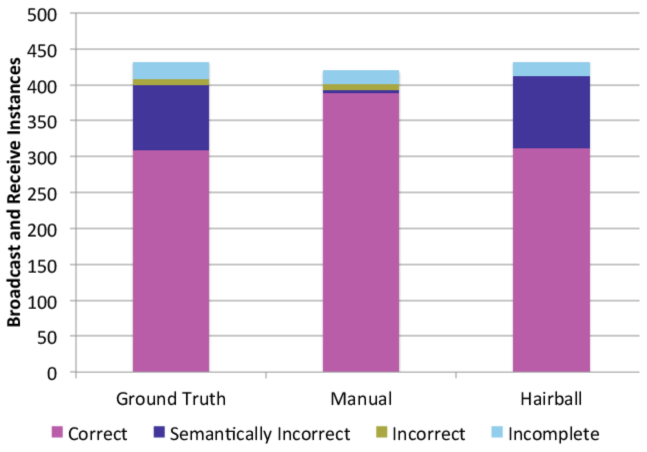

Figure 3: Compares the broadcast and receive instance labels. Manual analysis failed to discover 12 instances, and resulted in 79 false positives. Hairball detected 100% of the instances with three false positives.

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 and sound blocks with a broadcast block. To detect such instances, Hairball would need to additionally inspect all scripts triggered by the broadcast message to ensure none of them interfered with either the speech bubble or the playing of the sound file.

6.3 Broadcast/Receive

Figure 3 shows the results of detecting and labeling broadcast and receive instances. 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.

Overall, Manual analysis failed to discover 12 instances, and identified 388 correct instances, of which 79 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 represent "correct" usage of broadcast and receive blocks, the ground truth analysis determined the evaluation behavior to be incorrect.

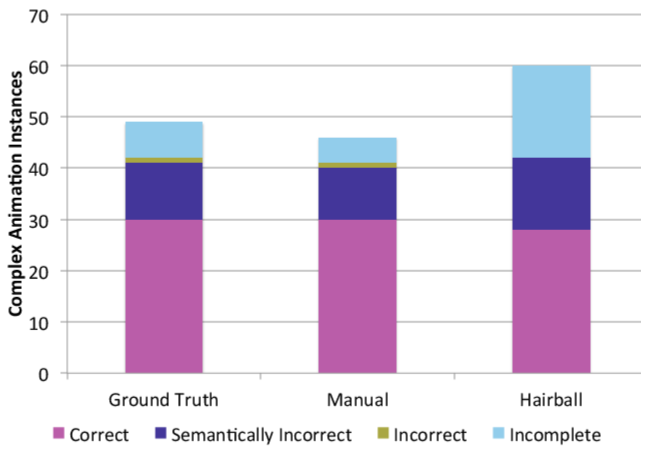

Figure 4: Compares the complex animation instance labels. Manual analysis failed to detect three instances whereas Hairball found 11 items that were determined to not be instances of complex animation. Hairball resulted in two false negatives.

6.4 Complex Animation

Complex animation was especially difficult for Hairball to detect. As Figure 4 shows, the manual analysis was 100% accurate at labeling the 46 instances found, and only failed to detect three instances. Hairball, on the other hand, discovered 11 incomplete instances that the ground truth analysis determined to not be instances at all. Excluding these instances, Hairball identified 28 correct instances, and 21 others with only two false negatives.

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.

6.4.1 Summary Results

Figure 5 shows three sets of results across all four CS concepts and the overall average. The first is the mislabel rate of manual analysis and Hairball. Percentages closer to zero indicate higher accuracy. We can see that Hairball is actually slightly more accurate overall than manual analysis, largely in part of its accuracy in labeling broadcast and receive instances.

The second set of results are the rate of false positives. Manual analysis, with an 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 that instances Hairball labels as "correct" can be trusted.

Finally, the third set of results are 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".

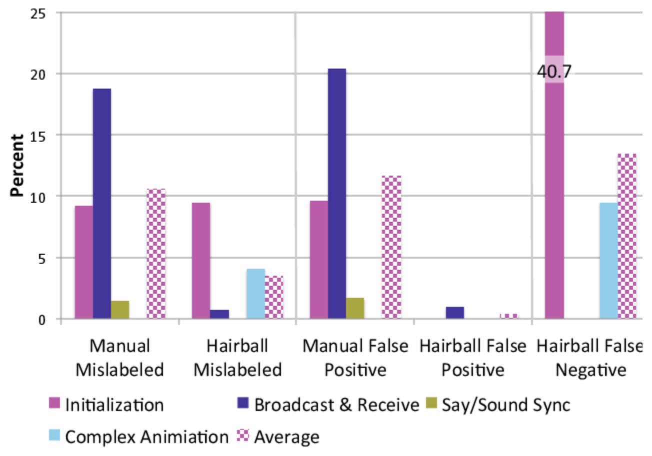

Figure 5: Provides a summary of the percent of mislabeled, false positive, and false negative instances from manual analysis and Hairball for each of the four CS concepts and the average. The "Manual False Negative" category was omitted as manual analysis resulted in zero false negatives. The y-axis is truncated for the smaller values, thus the tallest bar should extend to 40.7%. Missing bars represent 0%.

7. CONCLUSIONS AND FUTURE WORK

We presented a case studying showing a new static analysis tool, Hairball, 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 usful 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 `github.com/ucsb-cs-education/hairball`.

Our future work entails writing Hairball plugins suitable for widespread summative assessment in both AP CS 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 projects.

8. REFERENCES

[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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.