SnapCheck: Automated Testing for Snap*!* Programs

Wengran Wang
North Carolina State University
Raleigh, USA

Chenhao Zhang
Northwestern University
Evanston, USA

Andreas Stahlbauer
University of Passau
Passau, Germany

Gordon Fraser
University of Passau
Passau, Germany

Thomas Price
North Carolina State University
Raleigh, USA

ABSTRACT

Programming environments such as Snap*!*, Scratch, and Processing engage learners by allowing them to create programming artifacts such as apps and games, with visual and interactive output. Learning programming with such a media-focused context has been shown to increase retention and success rate. However, assessing these visual, interactive projects requires time and laborious manual effort, and it is therefore difficult to offer automated or real-time feedback to students as they work. In this paper, we introduce SnapCheck, a dynamic testing framework for Snap*!* that enables instructors to author test cases with Condition-Action templates. The goal of SnapCheck is to allow instructors or researchers to author property-based test cases that can automatically assess students' interactive programs with high accuracy. Our evaluation of SnapCheck on 162 code snapshots from a Pong game assignment in an introductory programming course shows that our automated testing framework achieves at least 98 % accuracy over all rubric items, showing potentials to use SnapCheck for auto-grading and providing formative feedback to students.

ACM Reference Format:
Wengran Wang, Chenhao Zhang, Andreas Stahlbauer, Gordon Fraser, and Thomas Price. 2021. SnapCheck: Automated Testing for Snap*!* Programs. In *26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2021), June 26-July 1, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3430665.3456367

1 INTRODUCTION

Visual, interactive programming projects, such as creating student-designed apps and games, are widely used in many introductory programming courses (e.g., [10, 13]). They encourage students to pursue projects that produce a computational artifact that they can interact with, and to express their ideas creatively, which has been shown to motivate students [13]. Additionally, many popular block-based programming environments, such as Scratch [16] and Snap*!* [19], are specifically designed to create such visual, interactive programs.

One challenge to using visual, interactive programs in the classroom is that the very properties that make them engaging also make them difficult to assess automatically. While traditional introductory programming tasks can usually be tested using simple input/output pairs, visual, interactive programs are controlled by sequences of user inputs (e.g., key presses and mouse clicks), and their output is based on the visual composition of sprites on a screen[1], making it difficult to write automated test cases. Not only does this make it time consuming for instructors to assess these programs, limiting the ability to scale up courses, it also precludes the use of the automated, formative feedback found in many introductory courses. Prior work addressed these problems by developing automated, functional tests for Scratch [29]. However, it has not been extended to automated assessment for Snap*!* programs.

In this work, we introduce SnapCheck, an automated testing framework for visual, interactive programs. We define how to author SnapCheck test cases with a domain-specific-language (DSL), and describe how an instructor can use SnapCheck's user interface to author test cases with Condition-Action templates that simulate the interaction rules of a human tester when running a Snap*!* program. We evaluated SnapCheck on 162 program snapshots and final submissions from 42 students working on a programming assignment. Our results suggest that SnapCheck can accurately test these programs by providing at least 98 % accuracy on all rubric items, showing that SnapCheck can be used by instructors to automatically assess students' visual, interactive programs in Snap*!*.

2 RELATED WORK

Automated Assessment. Effective feedback should be timely and specific [25, 26]. However, in today's growing-size CS classrooms [1], traditional, manual assessment is usually insufficient for offering timely feedback to all students; many K-12 in-service programming instructors are in the process of transitioning from non-programming backgrounds, and therefore lack proficiency in grading students' assignments manually [9].

Automated assessment not only reduces instructors' grading efforts, but also allows students to receive automated feedback on where they were correct or made mistakes on [8] in the middle of programming. Unlike instructor feedback, automated feedback can be easily propagated to all students having the same mistake, allowing a larger group of students to benefit from receiving feedback. This feedback can take the form of success or failure of test cases [8, 14, 30], highlighting erroneous code [7, 22], or identifying

[1]A sprite in Snap*!* is an object (such as in object-oriented programming) and can have its own code (scripts), and variables.

likely misconceptions [11]. Providing such immediate feedback has been shown to engage and motivate students [17], improve their learning outcomes [4], and does not pose social threat to students [23].

To build automated assessment tools, researchers used two prevalent methods to analyze students' programs: syntax-based static analysis (e.g., [17, 21, 22, 30]), and functional program analysis, (e.g., [8, 14, 15, 20]), generated from dynamically running the program rather than inspecting the structure of its code.

Syntax-Based Program Analysis. In block-based languages, many students complete programming assignments by making apps, games and simulations [10]. Because of the visual and interactive output of these programs, it is challenging to perform functional tests that checks correctness of procedures based on input-output pairs in these programs (we discuss *why* in Section 3). Therefore, many programming analysis approaches made use of syntax-based program analysis to check students' program structures based on abstract syntax tree (AST) rules [2, 17, 30].

For example, Marwan et al. developed a data-driven syntax-based autograder in Snap*!*, to provide real-time adaptive feedback, and found that it increased students' engagement with the programming environment and intentions to persist in CS [17]. Crescendo is a Snap*!*-based self-paced learning tool used syntax-based rules to automatically grade rubric-like assignment requirements (e.g., draw a triangle) during programming. This rule-based syntax-based analysis checks AST substructures to determine whether the code satisfied specific programming requirements (e.g., is a "move" block inside of a "do repeat" loop?). In the evaluation, they found that students benefited from receiving immediate feedback, as shown by high completion rates and fast task completion. However, such syntax-based rules can be brittle and difficult to author, and may lead to assessment errors when not aligning with a student's solution strategy [30]. This suggests that checking the syntactic structure is insufficient to correctly assess student projects.

Functional Program Analysis. In text-based programming languages such as Java and Python, instructors and learning systems commonly use functional testing to automatically assess students' programming assignments. For example, CloudCoder [14], Problets [15] and CodeWorkout [8] included test-case-based program assessment in its built-in programming assignments, enabling students to receive immediate feedback upon completion of a programming assignment. Marmoset [27] allows students and instructors to author and use test cases to grade their assignments, and provides instructors with overviews of students' performance on instructors' tests. These systems make use of **functional tests**, where certain test cases that include input-output pairs were executed, to examine the correctness of a certain procedure (i.e., usually a function with a fixed name) inside student programs. Prior work used functional tests to provide automated feedback to students, which has been shown to improve their learning outcomes [4, 12, 18]. For example, Gusukuma et al. used functional analysis to find students' erroneous programs in Python, and generate misconception-driven feedback to those programs, which has improved students' performance. However, these traditional, input/output-based testing may not apply to visual, interactive programs.

Whisker [29] is an automated testing framework for visual, interactive programs written in Scratch. Similar to Snap*!* programs, Scratch programs are highly concurrent, rely heavily on timers, and are driven by events such as user interactions using keyboard and mouse inputs [29]. To address the challenges of visual, interactive programs, Whisker allows educators to write and to automatically generate [5] test cases in JavaScript which simulate user inputs to a Scratch program and observe the program's resulting behavior. The authors evaluated Whisker on 37 students' programming assignments on their completion of 28 properties, such as "only one apple must fall down at a time." They found the results produced by Whisker to be strongly correlated with the instructors' grading, showing that it is possible to use functional tests to grade student projects in Scratch, although the actual test-authoring may still be challenging for instructors [29]. SnapCheck is inspired by Whisker, but targets Snap*!*. More elaborated steps towards checking Scratch programs are taken by Bastet [28], which, for example, implements an exhaustive state-space exploration to check for violations of requirements.

3 FORMATIVE STUDY & DESIGN CHALLENGES

Our goal is to allow programming instructors to automatically assess Snap*!* programs. Snap*!* allows students to create complex games and apps using visual and block-based programming; it is used by many students in the AP CS Principles and university course each year as part of the BJC curriculum [10]. To understand the design opportunities and challenges when grading students' visual, interactive assignments, we started by manually inspecting 42 student submissions, taken from a classroom assignment. In this assignment, students were required to implement a one-player Pong game, where there is one paddle on the left / right side of the stage [2], which must bounce a ball against a wall. The instructor required students to implement the game with the following 10 requirements:

1. *key_up*: The paddle moves up with an up arrow key.
2. *key_down*: The paddle moves down with a down arrow key.
3. *upper_bound*: When touching the upper bound, the paddle does not move upwards when the up arrow key is pressed.
4. *lower_bound*: When touching the lower bound, the paddle does not move downwards even when the lower key is pressed.
5. *space_start*: The ball starts movement when space key is pressed.
6. *edge_bounce*: The ball moves and bounce on edge unless touching the back wall.
7. *paddle_bounce*: The ball bounces back to stage when touching paddle.
8. *paddle_score*: If the ball touches the paddle, increase score.
9. *reset_score*: If the ball touches the back wall behind the paddle, reset score to 0.
10. *reset_ball*: If the ball touches the back wall behind the paddle, reset ball to the center of the stage.

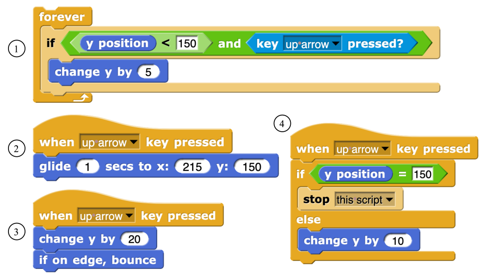
We manually analyzed these 42 student programs, and identified 5 design challenges of creating automated testing framework for these visual, interactive programs, described below.

[2]A stage is where Snap*!* displays its sprites and actions

Dynamic User Inputs. For non-interactive programming problems, students' programs usually consist of one or more individual functions. A standard testing approach is to use test cases that specify input-output pairs, checking if the output of the function matches the expected output for each input. For example, in an integer sorting problem, the students write a function (such as `sort()`) that takes a list of integers as the input parameter. To test students' programs, the instructor prepares test cases which call the students' `sort()` functions with a pre-defined list of integers, and then check whether the values returned are identical to the corresponding sorted list.

For visual, interactive programs like Pong, however, the input to the program is a constant stream of signals from input devices like the keyboard and mouse. Users observe the changes of graphical elements and send different inputs according to what they observe over time. Such inputs are dynamic and dependent on the program state. For example, in the Pong program, a user may press the up arrow to move the paddle up when they see the ball go up. Such input stream can be challenging to encode as standard input data.
Visual Outputs. In a non-interactive programming problem (e.g., an integer sorting problem), the instructor may define the outputs as a list of integers and check whether the student program returns such output. However, it is less clear how to define output as a single, "correct" value for visual, interactive programs. These programs include multiple elements (e.g. the sprites in Snap*!* or Scratch), each having its own properties (e.g., direction, position).
Delayed Responses or Outputs. For visual, interactive programs, specifying just the input and output is insufficient—the output sometimes happen after certain delays, and this time difference may be different across student programs. For example, when testing the *key_up* behavior, with different implementation approaches, the paddle movement and the keyboard pressing happen at different timestamps: a student's program may move the paddle with key continuously: the paddle may move smoothly upward, or in larger bursts. The delays between inputs and outputs are different in these two scenarios, and caused difficulties to specify the delay in a test program.
Requirements with Temporal Constraints. When specifying test cases for non-interactive programs, instructors do not need to distinguish between which specifications are "forever true", and which are "sometimes true". For example, in the integer sorting program, the program finishes execution in milliseconds, and instructors do not need to specify intermediate invariants during the program execution, such as the order of the array after the third iteration of the algorithm.

However, requirements on interactive programs include different temporal constraints: some requirements should be *always* satisfied (e.g., *edge_bounce*); some should be checked only once (e.g., *space_start*); some should be checked after another (e.g., *upper_bound* should be checked after *key_up*)). Specifying such time constraints adds challenges.
Various Implementations. We have identified several challenges of doing functional testing in these interactive programs, such as challenges to specify input, output, and specify temporal restrictions. Prior work in automatic assessment of Snap*!* programs identified similar challenges, and instead applied syntax-based rules on the AST [2, 17, 30]. However, our analysis of students' Pong code shows that students included a variety of code patterns to implement the same behavior, causing challenges to apply rule-based approaches directly to check for AST subcomponents: For example, Figure 1 shows that when implementing the *upper_bound*, students may 1) stop sprite movement when its *y* position is larger than a certain value; 2) use Snap*!*'s built-in "glide" block to fix the target of sprite movement; 3) use Snap*!*'s built-in block "if on edge, bounce" to stop sprite movement when it hits edge; 4) stop the entire script started by a "when up arrow key pressed" hat block. Students may also include combinations of the above approaches, resulting in multiple unpredictable ways to construct code that implements the same observable behavior. This caused syntax-based analysis approaches to be insufficient to understand and analyze these programs.



```
① forever
     if < < (y position) < 150 > and < key up arrow pressed? > >
         change y by 5

② when up arrow key pressed
   glide 1 secs to x: 215 y: 150

③ when up arrow key pressed
   change y by 20
   if on edge, bounce

④ when up arrow key pressed
   if < (y position) = 150 >
       stop this script
   else
       change y by 10
```

Figure 1: Examples of ways students constructed code to allow paddle to not move with the upper arrow key when it hits the upper bound.

These design challenges are in line with the challenges addressed by Whisker [29] in the context of testing Scratch programs. We next discuss the design of SnapCheck to address these goals.

4 AUTOMATED TESTING WITH SNAPCHECK

Based on the challenges uncovered by the formative analysis, we formulate the design goal for SnapCheck as to provide automated testing framework for Snap*!*, with user-friendly specification of desired properties, including complex inputs, outputs, and temporal restrictions. We implemented SnapCheck in Snap*!*, but the testing model of SnapCheck can also be applied to interactive sprite-based visual programs on other systems, such as Scratch [24]. This would require representing the relevant objects (e.g., Sprites) and properties (e.g., x coordinate) of those systems. A similar model may be able to support 3D interactive graphical programs such as Alice [3]. SnapCheck is an open-source software that can be downloaded at github.com/emmableu/SnapCheck.

4.1 Authoring Test Cases in SnapCheck

To test programs using SnapCheck, an instructor needs to write a list of *test cases*, each with a name, and its rules to programmatically imitate what a human tester does when running a Snap*!* program, described below. The test cases can be defined either through an selection-based User interface (shown in Figure 4), or in JavaScript by using APIs provided by SnapCheck. For example, to check *paddle_bounce* rubric item, an instructor may define a test case using the same name — shown in Figure 4.

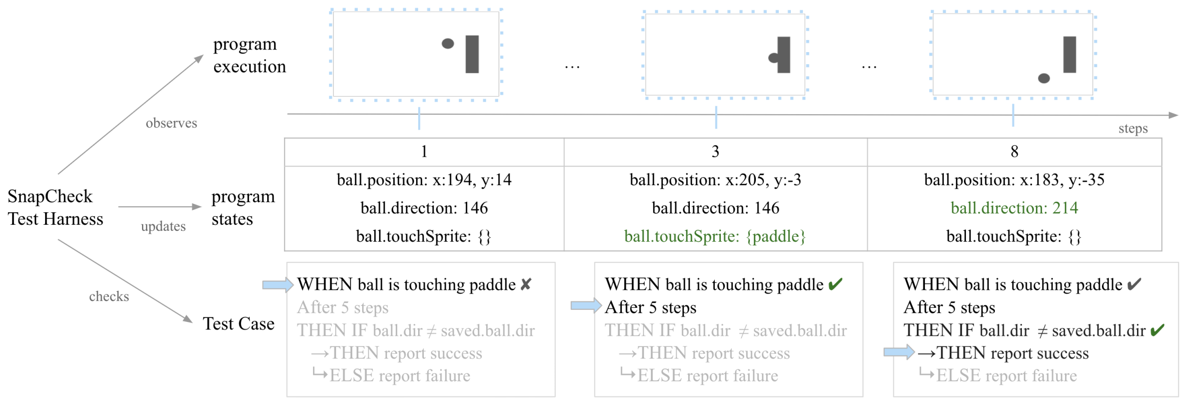


Figure 2: Automated Testing for *paddle_bounce*: the SnapCheck Test Harness keeps updating program states. When the *WHEN* condition evaluates to True (Step 3), it checks for the *THEN-IF* condition after the defined delay (Step 8).

(*always*:); *WHEN* ball is touching paddle; *After* 5 steps; *THEN IF* the direction of paddle is different from its saved direction; *THEN* report test success; *ELSE* report test failure.

We next define how such requirements can be specified step by step, and illustrate the SnapCheck grammar using a railroad syntax diagram, shown in Figure 3.

Step 1: Define a *WHEN* condition: *WHEN Ball touches paddle.*

The first step is to define *when* the rubric item should be evaluated, which is done using a WHEN condition.

A *condition* is a predicate function [3] that keeps track of the program state (e.g., sprite location, variable values) during the program execution. It evaluates to true if a program state satisfies a required property. SnapCheck allows multiple options for specifying a condition, including 1) sprite location and directions; 2) relation between sprites, such as one sprite touches another or a sprite touching the edge of the stage; 3) variable values; 4) comparison with a cached state saved from from the last step of execution, such as x coordinate smaller than that from 1 step ago, meaning the sprite moves to the right.

A *WHEN condition* is a predicate function that triggers the start of a test case when it is true. For example, if an instructor defines a *WHEN condition*: *WHEN ball touches paddle*, SnapCheck will constantly track whether the ball is touching paddle, and when this function returns to true, it starts executing the next statement defined by the user, described in Step 2.

Implementation detail: How does SnapCheck define and track its *WHEN conditions* internally? As shown in Figure 2, SnapCheck uses a *Test Harness* module to monitor the program state at every internal execution step of Snap*!* (i.e., a block of code). At every step, the *Test Harness* executes the *WHEN condition* function, and when it returns true, it starts executing the test cases following the *WHEN condition*, which we describe next.

Step 2: Define a *THEN-IF* condition: *THEN IF ball changes direction.*

After defining the *WHEN condition*, the instructor can use the keyword *THEN-IF* to specify correct (or incorrect) behavior of the program when a rubric item is evaluated, as determined by the WHEN condition. This property is defined using the same set of predicate functions as defined in Step 1.

Here, if the instructor wants to check whether the ball changes direction, they can define the THEN-IF condition to check whether the ball's direction has changed from its "saved direction" - In *THEN-IF* conditions, all properties have a *saved* variant, which is by default the value of that property (e.g., direction) saved from Step 1 in the *WHEN* condition. Based on whether the *THEN-IF* condition is satisfied, we may next run Test Case executes actions, defined in Step 4.

Implementation detail: The keyword *THEN* starts a callback function, which begins executing after the previous condition is satisfied. It may optionally take the saved ball direction as an argument. When called, the callback function compares the current ball direction with the argument, and records the success and failure respectively. In addition, because *THEN* starts a callback function, it is easily to add multiple *THEN* conditions after one, to afford more expressive test authorizations.

Step 3: Define a delay: *After 5 steps.*

The *WHEN condition* and the *THEN-IF condition* do not always happen simultaneously: there can be delays between two conditions, explained in Section 3. This delay happens because it takes time for the student program to execute from one line (e.g., detection of touching) to another (e.g., change the ball direction), especially when they are not directly adjacent.

An instructor may define the delay between the *WHEN condition* and the *THEN-IF condition* based on how many steps will be executed between these two conditions. Here, a "step" is a time interval, usually in milliseconds. Program environments such as Snap*!* and Scratch make use of a "step" to update sprite properties atomically on the stage [29], which by default takes place at every frame of the display. For example, Figure 2, SnapCheck executes "After 5 steps" after the *WHEN condition*, meaning to check the *THEN-IF condition* after 5 steps of program execution.

Implementation detail: SnapCheck uses a countdown to track how many steps has passed between the *WHEN condition* and the *THEN-IF condition*: After the *WHEN condition* evaluates to true, the paddle direction referred to in the callback function is saved at that

[3]A predicate function is a function that returns True or False.

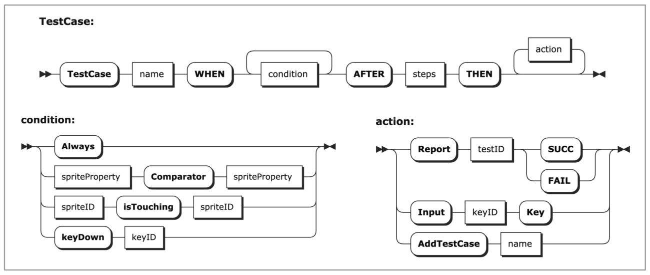


Figure 3: Grammar of the SnapCheck DSL

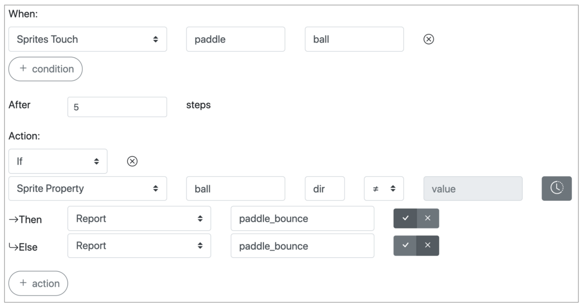

When:
Sprites Touch | paddle | ball
+ condition
After 5 steps
Action:
If
Sprite Property | ball | dir | ≠ | value
→Then Report | paddle_bounce
↳Else Report | paddle_bounce
+ action

Figure 4: Using SnapCheck UI to specify *paddle_bounce*

step. The test case then waits for the delay parameter number of steps and executes the callback defined in Step 2.

Step 4: Define a *THEN* action: *THEN report test success.*

An *action* defines programmatic actions SnapCheck can do to 1) report the success or failures of a test case, 2) supply input to the executing program, or 3) change how future tests are run. For example, an action can be reporting a test success; it can also be to used to remove or add another test case test case, which we describe in detail in Step 5.

After executing "*WHEN ..., After ..., THEN-IF ...*", The instructor can use the keyword *THEN* to define what to report, such as test success or failures. This creates a data point for the final test statistics, which writes to a spreadsheet that gets generated at the end of the tests.

Implementation detail: As in Step 2, the keyword *THEN* also starts a callback function that follows the previous callback functions (e.g., in Step 2), or a direct *WHEN condition* (e.g., in Step 1). Here, the callback reports the test to be a success.

Step 5: Specify temporal constraints: *Always.*

As explained in Section 3, some rubric items are satisfied when a student's program satisfied the requirements once, while some require that it satisfies the requirement the whole time. To allow specification for different time constraints, an instructor may use 3 ways to specify temporal constraints: (1) *Always.* v.s. *One-shot.*: The default is *Always.*, where SnapCheck runs the test case non-stop, even when it returns a success; *One-shot* test cases, on the other hand, only check the test case once. (2) *add-on-start*: the instructor may use the keyword *add-on-start* to define test cases that test programs immediately after the program starts. (3) *Add/remove test cases*: As explained in Section 3, some test cases have dependencies, and may benefit from specifying test cases with orders, such as testing *space_start* first, and *paddle_bounce* next. Therefore, in the action part of the program (e.g., in Step 4), an instructor may also define adding and removing test cases, to specify the sequential order of test cases.

4.2 Defining Test Inputs

When testing visual, interactive programs, instructors may need to send relevant inputs reacting to what is observed. These inputs can also be encoded as test cases using the *THEN input...* syntax, where "Input" is a type of *THEN action* defined in Step 4. For example, in the Pong game, a user needs to press the up/down arrow keys when they observe that the ball is in a higher/lower position than the paddle in order to make the paddle follow the ball, so that touching can be observed. We call this "follow ball' input, which can be directly encoded as a test case in SnapCheck:

> *WHEN* the y coordinate of the ball is greater than that of the paddle; *After* 1 step; *THEN* input 'up arrow' key for 4 steps.

5 EXPERIMENT

Dataset & Experiment Setting. To evaluate how well SnapCheck achieves its goal of offering accurate automated tests for students' visual, interactive programs, we used the same dataset as in our formative study. But in addition to the 42 students' final submissions in Pong, we also made use of their their intermediate snapshots, based on their trace data at 10 minutes, 20 minutes, and 30 minutes (42, 40, and 38 snapshots at each time, respectively). We manually graded all 162 programs based on the same 10 project requirements specified in Section 3. We selected Pong as a proper assignment to test the affordances of SnapCheck, since it includes a wide range of properties, can be implemented using a variety of different approaches, and its requirements for test automation embody the challenges that we designed SnapCheck to address. One researcher authored three different input series for SnapCheck: 1) up arrow key; 2) down arrow key; 3) "follow ball" (see Section 4.2, and when ball touches paddle, stops following. We authored automated test sequences to re-execute the program by clicking green flag to start game, and clicking space key to start ball movement. Because many students starts ball movement by pointing to a random direction, with each input sequence, we specify three different random seeds, and run program against all test cases using all combinations of random seeds and input series. At the end of testing, SnapCheck reports the #satisfied reports / #total reports (i.e., satisfaction rate) during each run. Because some test cases may have spurious failures (e.g., when *upper_bound* is satisfied, *key_up* is not satisfied), we used satisfaction rates that were lower than 0.1 to indicate a failure of a test case, and those higher than 0.1 being satisfied test cases [4].

Test Accuracy. Table 1 shows the accuracy, precision, and recall of SnapCheck's grading on 162 students' snapshots. Our results show that SnapCheck was able to provide accurate grading on all rubric items. The high accuracy of these graded assignments showed that teachers may benefit from SnapCheck for automatically grading students' project submissions. While instructors might ideally want

[4]We set this threshold a priori, but our final analysis found that a threshold of 0.05 also reached similar results, showing that SnapCheck catches the satisfaction of properties most of the times.

Table 1: Accuracy, precision, and recall of SnapCheck 's functional tests in 162 intermediate and final programming snapshots, on 10 rubric items. As well as the prevalence of positive items within each rubric item.

rubric item	prevalence	accuracy	precision	recall	F1
key_up	0.76	0.99	0.99	1.00	1.00
key_down	0.73	0.99	0.98	1.00	0.99
upper_bound	0.55	1.00	1.00	1.00	1.00
lower_bound	0.56	0.99	0.99	1.00	0.99
space_start	0.50	0.98	0.98	0.99	0.98
edge_bounce	0.49	1.00	1.00	1.00	1.00
paddle_bounce	0.42	0.99	0.97	1.00	0.99
paddle_score	0.35	0.99	1.00	0.98	0.99
reset_score	0.23	1.00	1.00	1.00	1.00
reset_ball	0.31	0.99	0.98	1.00	0.99

100% accuracy, we argue that most teaching assistants (TAs) also do not grade students' work with perfect accuracy, and instructors rely on students to dispute incorrect grades due to oversights.

Our results on students' intermediate, incomplete programs suggest there may also be applications of this work to providing formative feedback as students are working. While auto-grading requires a set of test inputs and time to run them, a student may potentially use SnapCheck to run their program and view the test results presented by SnapCheck, such as "complete", "incomplete" or "not demonstrated by the test run". This may help students develop productive testing strategies, as prior work suggests that they often do not use systematic testing [6].

6 DISCUSSION

Our results show that SnapCheck can not only assess students' final projects accurately, but also their incomplete programs, which includes buggy and erroneous code. This suggests the potential of using SnapCheck to offer automated feedback to students on their progress *during* programming. Similar to prior work [29], our work identified challenges in automatically assessing students' visual, interactive programs. Additionally, we present a novel testing framework for Snap*!*. Going beyond prior work such as Whisker, we designed a selection-based user interface and presented a step-by-step procedure to author test cases.

In addition, our experience with SnapCheck also identified **key factors that influence the accuracy of performing automated tests for visual, interactive programs**. These are factors that any functional tests on different types of visual, interactive programs (e.g., Scratch) would also potentially rely heavily on, and therefore add important considerations for future work, discussed below:

High-Coverage Inputs. SnapCheck is only able to check for the presence of behaviors when they are *executed*. For example, a *paddle_bounce* behavior would not be present if the paddle does not catch the ball, potentially leading SnapCheck to make false negative predictions if this behavior is implemented. In this assignment, we designed the *follow ball* input to allow the paddle to follow the ball's movement and catch it, which requires expert knowledge of the program. However, in more open-ended assignments, one set of input series may not cover all possible student programs, and may cause low program coverage and lower testing accuracies.

Temporal Specifications. The SnapCheck DSL uses a *delay* to specify the time difference between the *WHEN* condition and the *THEN-IF* condition. A delay can take any number of steps of the program execution, defined by the user. Because each step executes in milliseconds, the difference between 1 step and 10 steps may be undetectable to the human eye, but it can make an important difference in the detection of behaviors. In our experiment, we generally used a higher number of steps in delays (e.g., for the *paddle_bounce* rubric item, even if the ball changes direction immediately when touching paddle, we detect the change of direction *after* 5 steps). However, this approach may not work if another event changes the ball's state in before 5 steps.

Complex Conditions. Unlike describing a rubric item in natural language, testing programs requires the author to specify details formally, including specifying multiple, complex conditions to handle different edge cases. In the Pong program, there are two examples:

1) The rubric item *space_start* says that the ball should not move before the space key is pressed (i.e., the ball movement is triggered by the user pressing the space key, not immediately when the game starts). Therefore, an instructor needs to define test cases to first check that the ball does not move when the game starts, and after key presses, start checking for ball movement. In addition to the complex conditions, this test case may still fail erroneously, e.g., when a student's program first resets from another position to the center when the game starts, the "not moving when game starts" check would fail, causing false negatives in the detection.

2) For the rubric item *paddle_bounce*, a student may fail to implement this behavior, so the ball passes the paddle but bounces off the wall behind the paddle. Distinguishing these two different bouncing behaviors requires a specification of the difference in delays, the x coordinates of the ball in relation to the paddle, and the change of ball y coordinates in between *WHEN* condition and *THEN* condition. Authoring these specifications require knowledge of the Pong program, and may easily cause overly strict / loose specifications. Going beyond Pong, such requirements of complex conditions may cause difficulties to specify more complex behaviors, such as shooting or jumping, which require authoring chains of multiple test cases.

7 CONCLUSION

We introduced SnapCheck, a novel, automated testing framework that tests visual, interactive programs in Snap*!*. We explained how to author test cases in SnapCheck using a domain-specific-language, and presented a novel UI for authoring these test cases. Our evaluations on 162 student projects show that instructors may use SnapCheck to accurately assess students' programs, and to enable offering formative feedback in the middle of programming.

8 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1917885.

REFERENCES

[1] Computing Research Association et al. 2017. Generation CS: Computer science undergraduate enrollments surge since 2006. *Retrieved March* 20 (2017), 2017.

[2] Michael Ball. 2018. Lambda: An Autograder for snap. *Masterscriptie. EECS Department, University of California, Berkeley* (2018).

[3] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, Vol. 15. Consortium for Computing Sciences in Colleges, 107–116.

[4] Albert T Corbett and John R Anderson. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 245–252.

[5] Adina Deiner, Christoph Frädrich, Gordon Fraser, Sophia Geserer, and Niklas Zantner. 2020. Search-Based Testing for Scratch Programs. In *International Symposium on Search Based Software Engineering*. Springer, 58–72.

[6] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany Barnes. 2019. Defining tinkering behavior in open-ended block-based programming assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 1204–1210.

[7] Bob Edmison, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. 2017. Using Spectrum-Based Fault Location and Heatmaps to Express Debugging Suggestions to Student Programmers *(ACE '17)*. Association for Computing Machinery, New York, NY, USA, 48–54. https://doi.org/10.1145/3013499.3013509

[8] Stephen H Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 188–193.

[9] Barbara J Ericson, Mark Guzdial, and Tom McKlin. 2014. Preparing secondary computer science teachers through an iterative development process. In *Proceedings of the 9th workshop in primary and secondary computing education*. 116–119.

[10] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The Beauty and Joy of Computing. *ACM Inroads* 6, 4 (2015), 71–79.

[11] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 160–168.

[12] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-Driven Feedback: Results from an Experimental Study. *Proceedings of the 2018 ACM Conference on International Computing Education Research - ICER '18* 1 (2018), 160–168. https://doi.org/10.1145/3230977.3231002

[13] Mark Guzdial and Elliot Soloway. 2003. Computer science is more important than calculus: The challenge of living up to our potential. *ACM SIGCSE Bulletin* 35, 2 (2003), 5–8.

[14] David Hovemeyer and Jaime Spacco. 2013. CloudCoder: a web-based programming exercise system. *Journal of Computing Sciences in Colleges* 28, 3 (2013), 30–30.

[15] Amruth N Kumar. 2006. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Technology, Instruction, Cognition and Learning (TICL) Journal* 4, 1 (2006).

[16] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[17] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W Price, and Tiffany Barnes. 2020. Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 194–203.

[18] Antonija Mitrovic. 2003. An intelligent SQL tutor on the web. *International Journal of Artificial Intelligence in Education* 13, 2-4 (2003), 173–197.

[19] J Moenig and B Harvey. 2012. BYOB Build your own blocks (a/k/a SNAP!). *URL: http://byob. berkeley. edu/, accessed Aug* (2012).

[20] Ashlesha Patil. 2010. Automatic grading of programming assignments. (2010).

[21] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *International Conference on Machine Learning*. 1093–1102.

[22] Thomas Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. *International Educational Data Mining Society* (2017).

[23] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. 2017. Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 127–135.

[24] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[25] Mary Catherine Scheeler, Kathy L Ruhl, and James K McAfee. 2004. Providing performance feedback to teachers: A review. *Teacher education and special education* 27, 4 (2004), 396–407.

[26] V. J. Shute. 2008. Focus on Formative Feedback. *Review of Educational Research* 78, 1 (2008), 153–189. https://doi.org/10.3102/0034654307313795

[27] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin* 38, 3 (2006), 13–17.

[28] Andreas Stahlbauer, Christoph Frädrich, and Gordon Fraser. 2020. Verified from Scratch: Program Analysis for Learners' Programs. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 150–162. https://ieeexplore.ieee.org/document/9286012

[29] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 165–175.

[30] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W Price. 2020. Crescendo : Engaging Students to Self-Paced Programming Practices. In *Proceedings of the ACM Technical Symposium on Computer Science Education*.