NuzzleBug: Debugging Block-Based Programs in Scratch

Adina Deiner
University of Passau
Germany

Gordon Fraser
University of Passau
Germany

ABSTRACT

While professional integrated programming environments support developers with advanced debugging functionality, block-based programming environments for young learners often provide no support for debugging at all, thus inhibiting debugging and preventing debugging education. In this paper we introduce NuzzleBug, an extension of the popular block-based programming environment Scratch that provides the missing debugging support. NuzzleBug allows controlling the executions of Scratch programs with classical debugging functionality such as stepping and breakpoints, and it is an omniscient debugger that also allows reverse stepping. To support learners in deriving hypotheses that guide debugging, NuzzleBug is an interrogative debugger that enables to ask questions about executions and provides answers explaining the behavior in question. In order to evaluate NuzzleBug, we survey the opinions of teachers, and study the effects on learners in terms of debugging effectiveness and efficiency. We find that teachers consider NuzzleBug to be useful, and children can use it to debug faulty programs effectively. However, systematic debugging requires dedicated training, and even when NuzzleBug can provide correct answers learners may require further help to comprehend faults and necessary fixes, thus calling for further research on improving debugging techniques and the information they provide.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; **Software testing and debugging**.

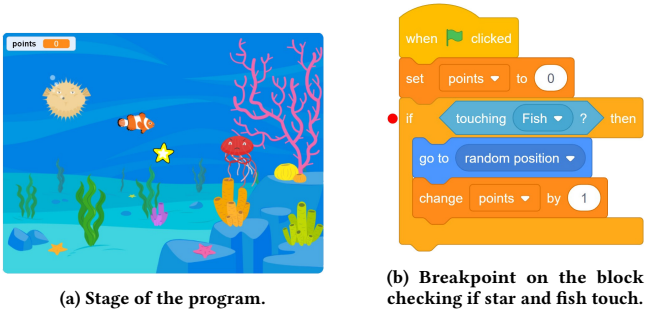
KEYWORDS

Debugging Tools, Omniscient Debugging, Interrogative Debugging, Scratch, Computer Science Education

ACM Reference Format:
Adina Deiner and Gordon Fraser. 2024. NuzzleBug: Debugging Block-Based Programs in Scratch. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623331

1 INTRODUCTION

Debugging is one of the most time consuming activities during software development [27, 31, 57]. Debugging is also a frequent activity when programming with the block-based programming language



(a) Stage of the program.

(b) Breakpoint on the block checking if star and fish touch.

Figure 1: The Scratch program *"Collect the Stars"*, containing a typical fault: The check for whether the fish touches the star should be contained in a loop.

Scratch [35], one of the most popular languages for introductory coding.[1] Although the visual composition of blocks prevents syntactical errors and enables learners to quickly create games and animations, programs can nevertheless implement wrong functionality. Finding the cause of a failure can be difficult, especially for programming beginners [17, 24, 41, 52, 61] as well as for teachers trying to support their learners [26, 59].

Consider the simple example Scratch program *"Collect the Stars"* shown in Fig. 1(a), where the clownfish controlled by the user has to collect the stars in the underwater world. Every time the fish touches the star, the star should change its position and increment the number of points by one. When executing the program, nothing happens when the fish touches the star. How can the programmer of this game find out what caused the failure?

Professional programming environments provide debuggers facilitating the debugging activity [64], but Scratch does not provide any debugging support. In this paper we therefore introduce NuzzleBug, which contributes debugging functionality to Scratch, such as the ability to pause the execution of a program, or to set breakpoints and execute a program step by step, even backwards in time. Figure 1(b) shows a breakpoint on the block checking if the fish is touching the star. The red dot visualizes the breakpoint and results in pausing the execution of the program every time the block is executed, which allows to investigate the program state used to evaluate the condition. The breakpoint reveals that the block is only executed once at the start of the program and the condition is not checked afterwards. The cause of the failure is found—the if-condition should be contained in a loop, the omission of which is a common mistake made by beginners [19].

Setting the breakpoint requires a hypothesis about the cause of the failure, but deriving such hypotheses is difficult [30]. Therefore, NuzzleBug uses interrogative debugging [28], which helps deriving hypotheses by allowing to pose questions one naturally would

[1] https://scratch.mit.edu/statistics, last accessed March 2023

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623331

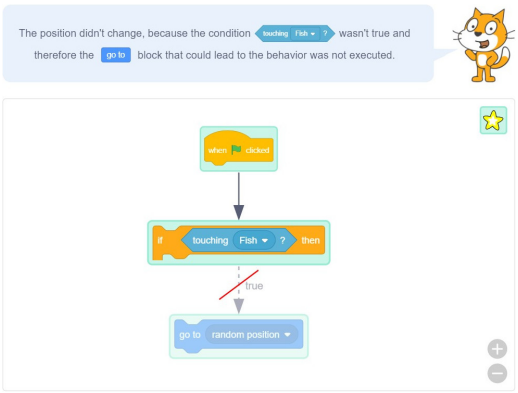


Figure 2: Answer for the question *"Why didn't the position of sprite Star change?"*

Figure 3: Answer for the question *"Why didn't the condition* touching Fish ? *evaluate to true?"*

want to ask about the execution of a program. Once a question is selected, the debugger calculates an answer explaining the behavior and directing the programmer to the fault. For our example, the debugger makes it possible to ask the question *"Why didn't the position of sprite Star change?"* and generates the answer visualized in Fig. 2, from which the programmer can conclude that the condition touching Fish ? never evaluated to true. Furthermore, it is possible to ask the question *"Why didn't the condition* touching Fish ? *evaluate to true?"* and the debugger provides the answer shown in Fig. 3. Since this evaluation to false is the only time the condition was checked, this helps understanding there is a missing loop.

In detail, the contributions of this paper are as follows:

- We introduce NuzzleBug, the first debugger for Scratch, and the first approach of omniscient and interrogative debugging for a block-based programming environment.
- We collate questions for Scratch-like programs, and provide novel answer types for block-based programs.
- We empirically evaluate NuzzleBug using a survey with teachers and a controlled study with pupils.

Overall, our evaluation demonstrates that NuzzleBug is intuitive and effective, and brings systematic debugging to the world of block-based programming. The teachers we survey confirm that

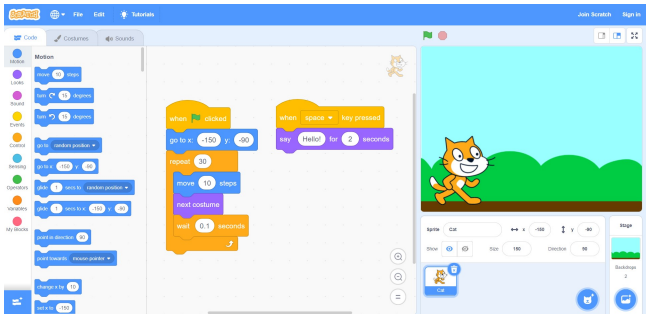


Figure 4: The Scratch GUI consisting of a block palette containing available blocks on the left, the cat's source code in the center, the stage on the upper right and the target editor on the bottom right.

NuzzleBug is useful, and pupils taught debugging are able to fix bugs using NuzzleBug, often even more effectively so than without debugger. However, we also find that maintaining the simplicity of block-based programming is challenging when presenting textual questions and answers, and young programming learners need to be explicitly educated on systematic debugging.

2 BACKGROUND

2.1 The Scratch Programming Environment

Scratch is a block-based programming language primarily aimed at children and is increasingly used in schools and coding clubs to introduce programming [35]. Program statements in Scratch are *blocks* that can be stacked to create *scripts*. Blocks can also contain holes of different shapes into which other blocks of matching shapes can be placed, for example to report the values of attributes or variables (*reporter blocks*) or conditions (*Boolean blocks*). Block shapes ensure that resulting programs are syntactically valid [46].

Scratch programs are structured into *targets*: The *stage* is the application window containing the background image, and *sprites* are rendered on top of the stage. Each target contains scripts that control its behavior. The first block of a script is always an event handler (*hat block*), and the execution of a script is triggered every time the event represented by the hat block occurs. One particular such event is the green flag, which the user can click to start the program execution. Scripts are executed in separate threads by the Scratch Virtual Machine (VM), and thread switching occurs when scripts encounter a waiting state, reach the end of a loop, or have no more blocks to execute. To avoid that learners have to deal with confusing error messages, Scratch uses a 'failsoft' approach where runtime errors are swallowed and execution is resumed without informing the user [23].

The Scratch's Graphical User Interface (GUI) is divided into four main sections (Fig. 4): The *block palette* on the left contains all available blocks divided into different color-coded categories. Blocks can be dragged out of the block palette and then dropped and snapped together in the *coding area* in the middle to define the behavior of the currently selected target. The upper right contains the *stage*, which renders the programmed behavior. The bottom right area enables selecting and editing targets.

2.2 Debugging Tools

Debugging is the activity of detecting, locating and eliminating program errors [18]. A typical *debugger* provides three functionalities [64]: First, it supports tracing by pausing the execution of a program at any point in time or on specified conditions. The most common available pausing condition is a *breakpoint*, which is a specific location in the program or condition on the execution that results in pausing the execution when reached. Second, once the execution is paused, classical debuggers provide context information, such as the values of variables or the current stack trace. Third, it allows resuming the execution, either until the next pause, or the next statement, which enables stepping through a program.

One of the reasons why debugging is difficult is the temporal or spatial chasm between the cause and the symptom of an error [15]. Although programs are executed forward in time, debugging requires thinking backwards from the failure to discover the cause. *Omniscient debuggers* [4, 5] record executions to enable users to explore the execution history [33], going back and forth to arbitrary moments instead of having to restart programs multiple times [43].

Debugging is triggered by a question about the behavior and methods like breakpoints or code stepping require coming up with a hypothesis about the cause of the behavior, which may be difficult [29]. False assumptions may lead to a time-consuming investigation of unrelated code and in the worst case prevent detecting the error. Ko and Myers observed that programmers naturally would like to ask *"Why did ...?"* questions about unexpected output that did occur and *"Why didn't ...?"* questions about expected output that did not occur [28]. *Interrogative debuggers* such as Whyline for Alice [28] and Java [29] allow to directly ask such questions via a "why"-menu, in which *"Why did ...?"* and *"Why didn't ...?"* questions can be selected for available objects. Answers point out the cause of the queried behavior as graphs explaining causality in terms of data and control flow, derived from execution traces, dynamic slices, and code analysis.

2.3 Debugging in Block-Based Programming

Since block-based programming environments generally follow a preventive approach that avoids exposing learners to errors [23], debugging tools are usually missing. Scratch offers no support to trace executions, and to inspect states one needs to resort to  blocks to have sprites communicate values at runtime. A recent survey of block-based programming environments [51] confirms there are only few exceptions: Snap! [22] allows pausing executions in addition to starting and stopping the entire execution, and Microsoft's MakeCode [3] supports breakpoints. An experimental extension of the hybrid (text and block-based) PencilCode [6] programming environment highlights executed blocks using arrows and supports step-wise execution and variable tracking [7]. Stepping, breakpoints and watches are also part of a proposed visual debugger [50] for Google's Blockly[2]. Alice [12], which is not strictly block-based but similarly constrains the syntactic validity of programs, provides no dedicated debugging support, but the Whyline interrogative debugger was originally implemented for Alice [28]. The general lack of debugging support in block-based environments is concerning: A recent study [36] found that students found and fixed fewer bugs in block-based environments than in hybrid ones.

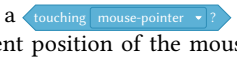
There is a growing awareness of the importance of teaching debugging [39] and the lack of debugging in education [38]. Several studies on the processes applied by young learners [58, 63] in block-based programming revealed that they tend to debug unconsciously without a systematic but with a tinkering-based approach [13]. To counter this, research has resulted in instructional materials [2, 10], learning trajectories [47], teaching strategies [20, 54], serious games [32, 34, 40], and unplugged [1] and tangible [53] environments for teaching debugging of block-based programs. Debugging tools, however, are generally not included. Debugging tools would furthermore not only be important for learners, but their educators also require skills and confidence in debugging to teach how to debug block-based programs, and to support students in the classroom. While teachers have been reported to struggle finding and fixing bugs in block-based programs [26, 59], tools have been shown to be helpful particularly for teachers [21].

3 THE NUZZLEBUG SCRATCH DEBUGGER

NuzzleBug is the first debugger for Scratch, and provides all the debugging functionality described in Section 2.2. In order to provide this debugging functionality, NuzzleBug extends the Scratch VM to trace executions (Section 3.1), which results in collecting all the information that is necessary for debugging [48]. NuzzleBug also extends the Scratch user interface for controlling the debugger, allowing users to pause executions, set breakpoints, or step through program executions forward and backward (Section 3.2). Execution traces are also prerequisite for creating questions and answers during debugger interrogation (Section 4).

3.1 Tracing Executions

NuzzleBug instruments the VM with a custom tracer, which records which blocks have been executed in which order. Every time a block is executed by the VM, the tracer appends a new trace entry to a trace, storing the executed block and the program state resulting from the execution of the block. The trace is automatically cleared every time the green flag is clicked, because then a new program run starts, and the trace is also reset whenever the code is changed.

A trace entry records the execution of a block and the resulting program state. It stores (1) the identifier of the block, (2) the type of the block, (3) the names of input parameters and (4) variables, (5) all evaluated parameter values, as well as (6) a list of trace entries recording the execution of reporter and Boolean blocks used by the block. For non-parameter blocks the trace entry also stores (7) the execution time of the block, and (8) the id of the target instance executing the block. Finally, the trace entry records (9) the program state after the execution of the block. For each target, this state consists of the target's costume or backdrop, visual effects, volume, sound state and variable values. If the target is a sprite, it also contains the position, direction, rotation style, size, visibility, draggability, layer and state of its speech or think bubble. If the executed block contains a 

condition, the tracer also saves the current position of the mouse-pointer. Finally, to enable reverse execution the execution state information

[2] https://developers.google.com/blockly

(a) Original. (b) Extended by NuzzleBug.

Figure 5: Scratch control panel.

includes the currently active threads, the id of the next thread to execute, the number of executed steps and the last executed blocks.

To reduce the size of execution traces and the computation time of their analysis, the tracer applies two optimizations. First, Scratch allows displaying the values of variables or attributes on the stage, which internally is treated like block executions. Since these have no effects on the state, the tracer ignores them. Second, blocks that halt the execution of their thread waiting for a timeout or condition are internally processed like repeated block executions in the VM. For such blocks (glide time seconds to ..., say message for time seconds, think message for time seconds, play sound sound until done, wait time seconds, and wait until condition), only the initial and final execution states are stored.

The tracer can be enabled or disabled by the user with a new button in the control panel above the stage. If tracing is active the button has a blue background color and otherwise it has a transparent background color. Toggling the value also clears the trace to avoid gaps between trace entries.

3.2 Execution Control

Scratch allows starting and stopping executions using the control panel above the stage shown in Fig. 5(a). NuzzleBug extends this with debugging functionality (Fig. 5(b)). In particular, NuzzleBug provides functionality to pause an execution and to resume it again once paused. Moreover, we enable the user to execute a program step by step and to rewind the execution to any traced program state. In addition, it is possible to add breakpoints to blocks such that the program pauses every time the block would be executed.

3.2.1 Pausing. The *"Pause / Resume"* button (⏸ if the execution can be paused, and ▶ if the execution can be resumed) allows halting executions. If the program is not running, the button is disabled. When the execution is paused, a white arrow with a red border is shown next to the last executed block of each active script, and a red arrow is shown next to the overall last executed block (Fig. 6). In addition, the selected target is changed to the one containing the overall last executed block, which is blinked.

3.2.2 Stepping. If the execution is paused, the user can execute a single block at a time by clicking the *"Step Over"* button in the control panel (Fig. 5(b)), which informs the Scratch VM's runtime about stepping, such that the modified Scratch pauses after one executed block, adhering to the thread execution order. During the execution of a block, the target instance executing the block is highlighted with a blue overlay (Fig. 7). This is especially useful when multiple instances of the same target (i.e., *clones*) exist and it is unclear which one executed the block. While stepping, the arrows emphasizing the last executed blocks are shown. Finally, the button (Fig. 5(b)) allows executing the first block of a program, by activating stepping and then invoking the program like when the green flag is clicked. This causes the runtime to start the program execution, execute the first block and then pause.

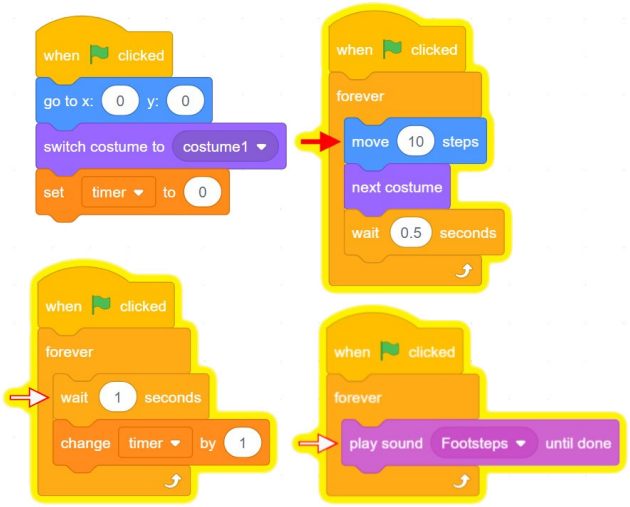
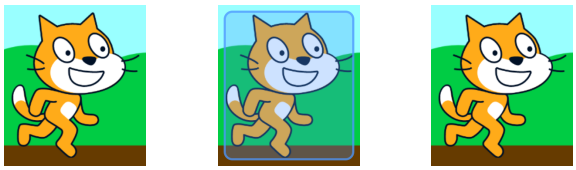
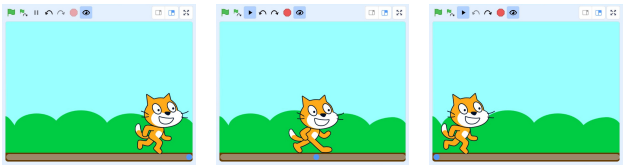


Figure 6: Arrows emphasizing the last executed block of each active script and the overall last executed block.



(a) Before. (b) During. (c) After.

Figure 7: Highlighted target instance during the execution of a block.

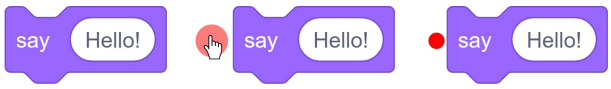


(a) Final state. (b) Intermediate state. (c) Initial state.

Figure 8: Execution states during rewinding of an execution.

3.2.3 Reverse Execution. In order to jump to arbitrary points in a trace, the user interface provides a slider to select the trace position (see bottom of Fig. 8). In addition, a *"Step back"* button above the stage allows to rewind the execution by one executed block (if there is a predecessor). In either case the recorded program state is restored in terms of the clones existing in that state, attributes of all target instances, and thread status. If the selected trace entry is not the last element in the trace, the *"Step over"* button does not execute the next block, but instead increments the index of the selected trace entry by one and restores the state of this trace entry. Clicking the *"Resume"* button does not resume the execution as usual, but deletes all trace entries succeeding the selected one and resumes the execution at the corresponding program state.

3.2.4 Breakpoints. NuzzleBug provides functionality to set a breakpoint on a block, which results in halting the execution every time it is executed. Breakpoints are supported by all blocks except reporter, Boolean and hat blocks. Reporter and Boolean blocks are implicitly



(a) Unset breakpoint. (b) Hovering over a breakpoint. (c) Set breakpoint.

Figure 9: Possible states of a breakpoint.

evaluated when the block they are contained in is executed. Hat blocks would interfere with the event triggering the hat block.

Each block stores whether its breakpoint is set. By default, a block's breakpoint is not set and the block is rendered as usual (Fig. 9(a)). While hovering the mouse pointer over the left margin of a block, a red dot visualizing the breakpoint appears (Fig. 9(b)). Clicking on it sets the breakpoint, which leads to the red dot being shown (Fig. 9(c)) permanently. Another click unsets the breakpoint and removes the red dot again. It is also possible to toggle the breakpoint by clicking the context menu option *"Add Breakpoint"* or *"Remove Breakpoint"*. In order to pause executions at the correct time, the VM is instrumented to check for activated breakpoints before executing blocks. If a breakpoint is encountered, the execution is paused and the breakpoint's block is highlighted in red.

4 SCRATCH INTERROGATIVE DEBUGGING

In addition to regular and omniscient debugging functionality (Section 3), NuzzleBug implements interrogative debugging. It generates questions the user might want to ask about the execution of a program (Section 4.1), as well as visual and textual answers for these (Section 4.2). Figure 10 shows the full interrogative debugger in action, with categorized questions to the left, textual answer on the top, and an answer visualized as a graph beneath.

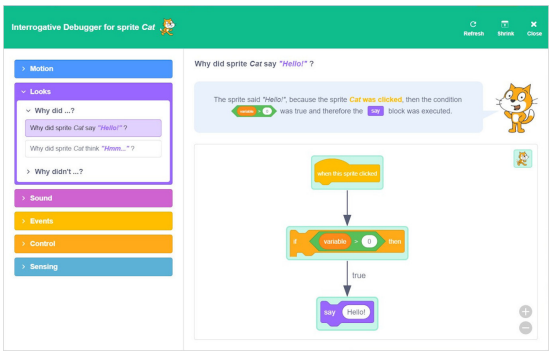


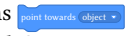
Figure 10: The dialog of the interrogative debugger.

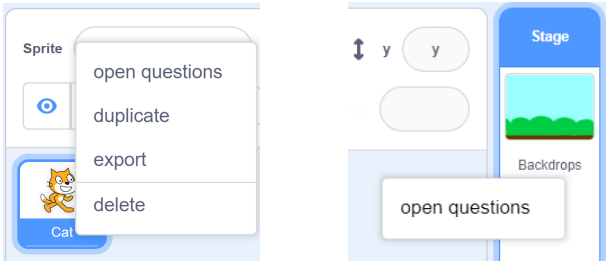
The principle of interrogative debugging was originally introduced in Whyline [28]. NuzzleBug is inspired by Whyline, but differs in a number of ways: First, the user interface for asking questions is different and designed specifically to blend in seamlessly in the Scratch UI, and allows navigation of a new set of questions tailored specifically for Scratch programs and blocks. Second, NuzzleBug uses analyses tailored for Scratch and a different approach to represent answers, based on custom answer graphs as well as visualizations of the Scratch stage.

4.1 Questions

Like Whyline [28], at the highest level we distinguish between positive (*"Why did ...?"*) and negative (*"Why didn't ...?"*) questions, but also add a third category (*"When did ...?"*). NuzzleBug then provides comprehensive questions covering all attributes (e.g., size, rotation, position, ...) and behaviors of sprites (e.g., movement, rotation, saying, ...), as well as control-flow questions (e.g., why blocks were or were not executed) and data-flow questions (e.g., regarding the values of attributes, variables, or conditions).

Questions are generated based on the code of the program that is being debugged, such that only questions about relevant hypotheses are shown. To further avoid overwhelming users with too many questions, they first have to select an object of interest, which can be a target or a block. For each possible object the context menu is extended with the option *"open questions"*, which pauses the execution if running, opens the interrogative debugger and triggers the generation of questions related to the selected object. The questions are categorized according to the category of the corresponding block: Motion, Looks, Sound, Events, Control, Sensing, Operators, Variables and Lists. Additionally, Execution contains general questions about the execution of a block. Within each category questions are further grouped into *"Why did ...?"*, *"Why didn't ...?"* and *"When did ...?"* questions, resulting in the hierarchy located in the left of the dialog (Fig. 10).

4.1.1 Target behavior questions. The context menu option shown in Fig. 11 can be used to ask questions about the behavior of targets (i.e., sprites or the stage). Since a sprite may have clones, users can choose which instance they want to ask questions about using a drop down menu in the dialog's header (Fig. 12). An overview of all target questions can be found in Table 1. Each of these questions relates to either a certain type of block, or a change in attribute. For example, if the code contains  blocks, NuzzleBug generates the question *"Why didn't <sprite> point towards <object>?"* for each target object (i.e., another sprite or the mouse-pointer). The question is positive (*"Why did ...?"*) if at least one of the related blocks was executed, and negative (*"Why didn't ...?"*) otherwise.



(a) Context menu of a sprite. (b) Context menu of the stage.

Figure 11: The new context menu option *"open questions"*.

4.1.2 Block questions. A context menu option *"Open questions"* on blocks (Fig. 13) can be used to ask about their execution and contained reporter and Boolean blocks. As a block may be executed multiple times, all its executions are numbered and selectable in a drop down menu similar to Fig. 12. Block questions are generated by iterating over the blocks of a selected target, and instantiating all applicable questions of those in Table 2 based on whether each block is contained in the execution trace and its value for Boolean and reporter blocks.

Table 1: Target questions.

MOTION
Why didn't the position of <sprite> change?
Why did <sprite> move right- / left- / up- / down-wards?
Why didn't the direction of <sprite> change?
Why didn't the direction of <sprite> change to <direction>°?
Why didn't <sprite> point towards <object>?
Why didn't <sprite> turn clockwise / counterclockwise?
Why didn't <sprite> turn to the right / left?
LOOKS
Why didn't <sprite> say / think <message>?
Why didn't the size of <sprite> change?
Why did the size of <sprite> increase / decrease?
Why didn't <sprite> show / hide itself?
Why didn't the costume of <sprite> change?
Why didn't the costume of <sprite> change to <costume>?
Why didn't <sprite> change the backdrop?
Why didn't <sprite> change the backdrop to <backdrop>?
Why didn't the backdrop change?
Why didn't the backdrop change to <backdrop>?
SOUND
Why didn't <target> play sound <sound>?
Why didn't <target> stop all sounds?
EVENTS
Why didn't <target> broadcast the message <message>?
Why didn't <target> receive the message <message>?
CONTROL
Why didn't <sprite> start as a clone?
Why didn't <target> create a clone of <sprite>?
SENSING
Why didn't <target> ask <message>?

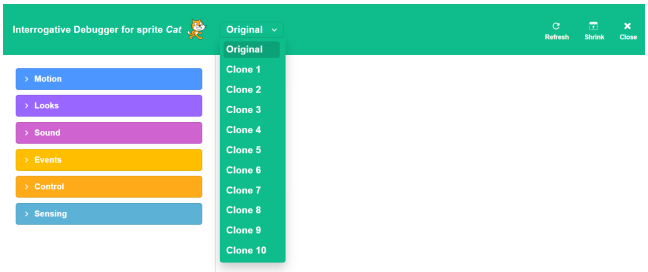


Figure 12: Drop down menu to select the sprite instance of interest.

4.2 Answers

In contrast to WHYLINE [28], which visualizes answers using a graph structure representing data and control flow causality over time, NUZZLEBUG simplifies answers by separating causality and temporality: Each answer explains the program state selected with the slider (Section 3.2) or the occurrence of a specific event or block execution. After selecting a question, the answer is visualized on the right side of the dialog using the structure depicted in Fig. 14, with a textual answer in a speech bubble explained by a cat (the SCRATCH mascot), and a visual answer in the form of an answer graph or visualization of the program state.

Table 2: Block questions.

EXECUTION
Why didn't the block execute?
When did the block execute?
MOTION
Why did (x position) *have the value <value>?*
Why did (y position) *have the value <value>?*
Why did (direction) *have the value <value>?*
LOOKS
Why did (size) *have the value <value>?*
Why did (costume ...) *have the value <value>?*
Why did (backdrop ...) *have the value <value>?*
OPERATORS
Why didn't the condition (< / = / >) *evaluate to true?*
Why didn't the condition (contains ?) *evaluate to true?*
SENSING
Why didn't the condition (touching object ?) *evaluate to true?*
Why didn't the condition (touching color ?) *evaluate to true?*
Why didn't the condition (is touching ?) *evaluate to true?*
VARIABLES
Why didn't <variable> have the value <value>?
LISTS
Why did <list> have the value <value>?
Why didn't <list> contain <item>?

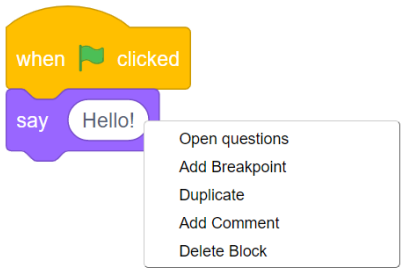


Figure 13: Extended context menu of a block.

4.2.1 Block execution answers. The (non-)execution of a block is based on its control dependencies, which in SCRATCH can be control blocks, hat blocks triggered by the user, and hat blocks triggered programmatically. For an executed block NUZZLEBUG shows a graph of all control dependencies causing the execution of the block; for unexecuted blocks the graph visualizes every possible way the block would execute and why it did not occur. Thus, the answer graph is a subgraph of the Control Dependence Graph (CDG) containing the block of interest and some or all of its transitive predecessors. The answer graph is created by first removing all circular dependencies in the CDG, since they can easily be confusing and are not necessary to understand why a block was executed or not. Then, starting

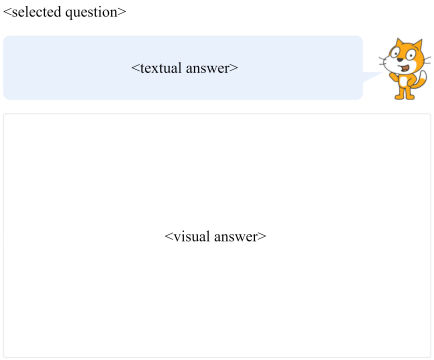
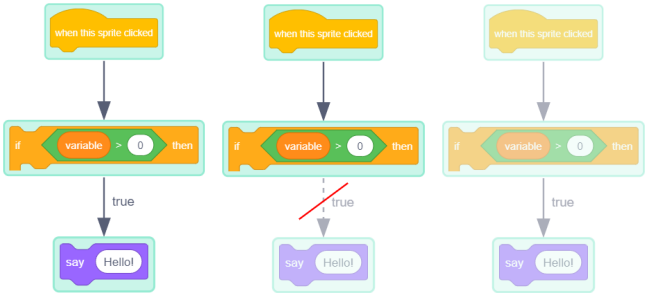


Figure 14: Answer structure; examples are shown in Fig. 2 and Fig. 3.

from the target block the CDG is traversed backwards for all transitive dependencies that do not represent user events, copying each traversed node and edge to the answer graph.

The resulting graph is rendered using Scratch blocks as nodes, and each block has a colored margin indicating its associated target. If a node represents a block that was never executed the block has a reduced opacity of 0.5. The same holds for edges, which count as executed if their source and target nodes were executed. Figure 15 shows the same graph with all blocks executed, some blocks executed and none executed. If the source of an edge is a control block containing a condition, the condition's value required to flow this direction is used as label for the edge. If the control block was executed, the visualization depends on the condition's traced value: If it is not the required one, then the edge is dashed and crossed out using a red line (Fig. 15(b)). If the execution should have caused the execution of another block, but that block was not executed, the execution was interrupted, which is indicated with a crossed out edge labeled with *"execution paused / stopped"*.



(a) All blocks executed. **(b) Partially executed.** **(c) No block executed.**

Figure 15: Different answer graphs for the say block depending on whether the contained blocks were executed.

Hovering the mouse cursor over a reporter block (Fig. 16(a)) or a Boolean block (Fig. 16(b)) displays the value this block had during execution. Clicking a node in the answer graph opens the interrogative debugger for that block, and if it was executed multiple times, the relevant execution is selected.

Textual answers are generated using the answer graph. For positive questions this answer explains every control dependency responsible for a block's execution. Starting with the entry node of the

(a) Tooltip for a reporter block (variable). **(b) Tooltip for a Boolean block (>).**

Figure 16: Tooltips showing the traced value of parameter blocks.

graph, a message is generated by adding $\langle reason_1 \rangle$ to describe the control dependency, and then iteratively appending explanations $\langle reason_i \rangle$ of the node's successors with appropriate conjunctions. For this, NuzzleBug defines positive and negative parametrizable text templates for each block. For example, for the graph in Fig. 15(a) the answer is *"The block was executed, because the sprite Cat was clicked and afterwards the condition* variable > 0 *was true."*

The textual explanation why a block was not executed is based on the reason in the answer graph that prevented the execution from flowing towards the block of interest, i.e., the edge where the source block was executed and the target was not. The answer then consists of *"The block wasn't executed, because <reason>."* If the execution to the target node flowed in the correct direction but was interrupted, *<reason>* is *"the execution was stopped / paused"*. Otherwise, if the execution did not flow in the correct direction due to a condition not having the required value, *<reason>* is *"the condition <condition> wasn't <requiredValue>"*. Since events can depend on multiple blocks or on user input, the *<reason>* for nodes depending on events is set to a generic message specified for each type of event. If the node that interrupted the execution from flowing towards the target block is not its immediate predecessor, the template for the answer is *"The block wasn't executed, because <reason> and therefore all subsequent blocks that could lead to the execution of the block were not executed."*, since resolving *<reason>* might still not guarantee the execution of the target block.

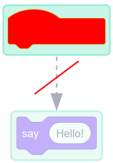


Figure 17: Answer graph for an unreachable say-block.

Blocks that are not executed because they are unreachable (dead code) result in *"The block wasn't executed because it is not reachable!"*, and the visualization shown in Fig. 17.

4.2.2 Block execution time answers. To answer the question *"When did the block execute?"* NuzzleBug extracts the timestamps of the first trace entry and the selected block execution, and calculates their difference in seconds. If the first trace entry recorded the execution of the when clicked block, the answer message is *"The block was executed <elapsedTime> seconds after clicking the green flag."* and otherwise it is *"The block was executed <elapsedTime> seconds after starting the recording."* The visual part of the answer is a picture of the stage rendering the state of the program when the selected block execution took place (Fig. 18). Below the picture a slider with a fixed value indicates the time of the block execution, with the calculated temporal difference shown as a red label.

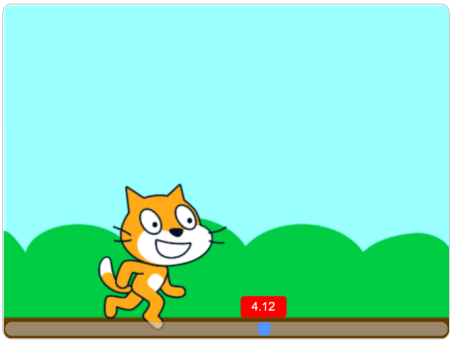


Figure 18: Visual answer showing the execution time of a block.

4.2.3 Target behavior answers. Questions about target behavior refer to the execution of one or multiple blocks. For example, a question like *"Why didn't <sprite> say <message>?"* can be transformed to *"Why didn't a say <message> block of <sprite> execute?"* The general procedure to answer positive questions about target behavior is to extract all blocks that caused the behavior and then answer why each of these was executed. If there are multiple graphs (e.g., for ambiguous control flow), then they are merged. Starting at the end of the trace causes the graph nodes to visualize values of the latest relevant execution.

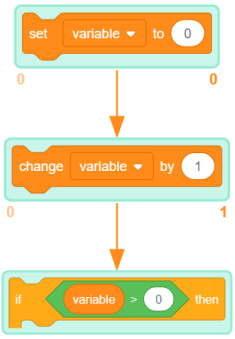
Depending on the question and related block, relevant attribute values before and after the execution of the block are added to the graph's node, for example the position of a sprite before (left) and after (right) the block's execution (Fig. 19). The same holds for blocks changing the attributes direction, size, costume and backdrop.

Figure 19: Graph node visualizing the position of a sprite before and after the relevant execution of the node's block.

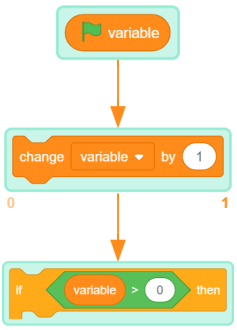
The textual answer is of the form *"<behavior>, because <reason$_1$>, then <reason$_2$>, next <reason$_3$>, then <reason$_4$>, ..., next <reason$_x$> and therefore the <type> block was executed."*, where *<behavior>* is a text describing the target's behavior the question asks about, *<type>* is the type of the block causing the behavior, and *<reason$_i$>* are based on the control dependencies. If more than one block's execution caused the question's behavior, then there are multiple graphs, in which case a question mark button is shown next to each leaf node. Initially, the general answer message *"<behavior>, because the execution of <count> <type> blocks caused this behavior. Do you need an explanation for a block? Then click on the ? next to it!"* is displayed. Clicking a question mark button next to a node highlights the node and displays its answer message.

For negative questions the strategy is to find all blocks that could lead to this behavior, and then explain why each of these did not cause it. If a block was not executed, the answer graph is used to show why not and to determine the answer message of the form *"<behavior>, because <reason> and therefore the <type> block was not executed."*, where *<behavior>* is a text describing the queried unobserved behavior, and *<type>* is the type of the block that could lead to the behavior; *<reason>* is created as described in Section 4.2.1. Unobserved target behavior can also be explained by (1) blocks attached to a  , which is never executed by an original sprite; (2) blocks attached to the block  but queried for a clone, which can never be executed since clicking the green flag deletes all clones; and (3) events that occurred outside the lifetime of a clone. All of these cases have dedicated answers.

4.2.4 Reporter block answers. Questions about reporter blocks are only available if the object of interest is an executed block containing them. The answer graph visualizes blocks causing or preventing the reporter block from having a certain value. To answer the positive question *"Why did <variable> have the value <value>?"* this graph contains the data dependencies responsible for the variable's recorded value *<value>*. In order to distinguish between data and control dependence edges, data dependence edges are colored using the relevant value's color (e.g., orange for variables); values before and after each block are shown beneath the node (Fig. 20(a)). If the graph's entry node represents the initial value, this node is visualized using  if the first traced block is  (Fig. 20(b)), or else using  . An answer message is generated by traversing the graph from the entry node, appending a pre-defined message *<reason$_i$>* for each block's effects, and thus generating the answer *"<variable> had the value <value>, because <reason$_1$>, then <reason$_2$>, then <reason$_3$>, ... and afterwards <reason$_x$>."*



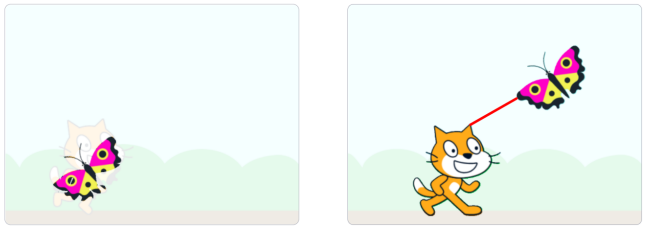
(a) Graph explaining the data dependencies starting at the latest block overwriting the value.



(b) Graph explaining the data dependencies starting at the initial value.

Figure 20: Visual answers for a positive question asking about the value of a variable.

The negative question *"Why didn't <variable> have the value <value>?"* is generated, if the program contains at least one block setting the variable's value to *<value>*, but the value traced during the selected block execution is different. First, all relevant blocks setting the variable to *<value>* are determined. If none of these was executed, then the corresponding answer graphs are rendered side by side, together with a general message (with detailed messages accessible with the question mark button.) If at least one set block was executed and the variable had the value at some point but was changed before the selected block execution, the answer is *"The variable was set to <value>, but changed afterwards."*, with a graph visualizing all block executions changing the variable's value.



(a) Cat touching the butterfly. (b) Cat not touching the butterfly.

Figure 21: Visual answers explaining the values of a Boolean touching block evaluating to true and false.

4.2.5 Boolean operator block answers. The question *"Why didn't the condition* <operator> *evaluate to true?"* is answered with the values of all reporter blocks within the Boolean operator block. For example, for *"Why did the condition* A < B *evaluate to true?"* the values of A and B are extracted from the trace. Assuming A = 0 and B = 1, the answer is: *"The condition evaluated to true, because* A *had the value* 0, B *had the value* 1 *and therefore* 0 < 1 *is true."* If there are no reporter blocks, the answer is *"The condition is always true / false, because none of the blocks is a* variable*"*.

In order to answer *"Why didn't the condition* touching ? *evaluate to true?"* questions, a picture of the stage in the relevant state is used to visualize the reason for the condition's value. Each existing different kind of touching ? block checks if a set of positions P_A is touching another set of positions P_B. For touching object ? blocks P_A are the positions of the sprite containing the block and P_B are the positions of the object selected in the drop down menu, which could be another sprite (i.e., all non-transparent pixels within that sprite), the stage's edges (i.e., positions having an x-value of ± 240 or an y-value of ± 180) or the mouse-pointer position. touching color ? blocks compare the positions P_A of the sprite containing the block with all positions P_B on the stage having the selected color. For is touching ? blocks P_A are all positions of the first selected color and P_B are all positions of the second one. If the positions P_A and P_B are touching each other, all overlapping positions $P_A \cap P_B$ are highlighted with decreased opacity of all other positions $P_{\text{Stage}} \setminus P_A \cap P_B$. Figure 21(a) shows an example of the touching Butterfly ? evaluating to true; other types of touching ? blocks are visualized similarly. In addition, the answer message *"When the block was executed, <A> touched <B> as shown in the picture."* is shown.

For false conditions, the distance of each position in P_A to each position in P_B is calculated and $p_A \in P_A$ and $p_B \in P_B$ are selected, such that the distance between p_A and p_B is minimal. The distance is visualized in the picture by drawing a red line between p_A and p_B and highlighting the positions P_A and P_B by decreasing the opacity of all other positions $P_{\text{Stage}} \setminus P_A \cup P_B$. Figure 21(b) shows an example of the touching Butterfly ? evaluating to false. Then, the answer message *"The distance from <A> to <B> was <distance> when the block was executed."* is generated and displayed in the speech bubble. It can happen that a list of positions is empty because a sprite is invisible or a color does not exist on the stage. In this case no distance can be calculated, and the answer given is: *"<sprite> could not be touched, because it was invisible when the block was executed!"* or *"The color <color> did not occur when the block was executed! Try to select the desired color with the color picker."*

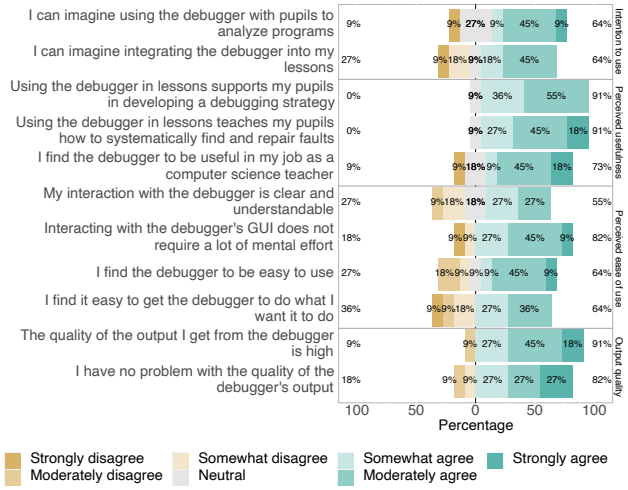


Figure 22: Teacher responses to *TAM* questions on intention to use, perceived usefulness, perceived ease of use, and output quality. Percentages aggregate all levels of agreement or disagreement.

5 EVALUATION

In order to provide initial insights on the usability of NuzzleBug, our evaluation aims to answer the following research questions:
RQ1: Do teachers consider NuzzleBug to be useful?
RQ2: Can children find and fix faults using NuzzleBug?

5.1 RQ1: Usability Study with Teachers

5.1.1 Experimental Setup. We surveyed 11 secondary school computer science teachers during a teacher training at the University of *blinded*. All 11 teachers already had prior experience with Scratch, and the teacher training included a Scratch programming activity prior to the survey. For the survey itself, we first demonstrated the debugger's functionality for 30 minutes, then let the teachers use NuzzleBug to debug nine faulty programs for one hour, and finally asked them to complete a survey based on the *Technology Acceptance Model (TAM)*, a common instrument to predict the acceptance and usage of a technology [60]. The model specifies 26 survey items of which we used 22, because the others were not relevant or applicable. All items were measured on a 7-point Likert scale ranging from strongly disagree to agree. At the end, the survey contained an optional open question for textual feedback. The full survey and the responses are available in the replication package[3].

5.1.2 Threats to validity. The survey is small and subject to threats to *external validity* concerning the generalization of results. Threats to *internal validity* arise as we introduced NuzzleBug within 30 minutes and afterwards the teachers used it for one hour only, which may not be sufficient to identify all strengths or weaknesses. While the teachers debugged programs using NuzzleBug, they did not experience it while *teaching* debugging, thus implying a threat to *construct validity*. Nevertheless, all participants were teachers in practice with experience in teaching, and the results give a good first impression on the usability and usefulness of NuzzleBug.

[3] https://figshare.com/s/68281791e01be5179699

5.1.3 Results. TAM contains questions in different categories; for space reasons, Fig. 22 only shows the questions of the categories relevant for RQ1 (intention to use, perceived usefulness, perceived ease of use, and output quality). The responses indicate that the majority of teachers considers NuzzleBug to be useful, especially in supporting pupils in developing a debugging strategy (91% agree somewhat or moderately). The output of NuzzleBug is understandable for almost all teachers (91% agree the quality of the output is high, and 82% had no problems with the output), and more than half (64%) can imagine integrating it into their lessons. Six out of the 11 teachers provided free-text feedback, which was positive in five cases, for example, "*The debugger is a very nice tool*" or "*That's a nice idea*". They also commented that the experience of teaching using the debugger would be helpful to provide better feedback.

However, the study also revealed drawbacks of NuzzleBug. Questions regarding ease of use are rated comparatively low (Fig. 22), and four of the teachers commented on the user interface in their free-text answers, such as "*Sometimes it was difficult to understand where the block shown in the answer graph is located in the program.*" and "*The questions are slightly hidden, because they are only accessible via the context menu.*" We plan to improve usability, for example by adding direct navigation from answers to corresponding scripts, or a questionmark-cursor to click directly on objects, even on the rendered stage. One teacher rated almost all items negatively, and justified this with a general aversion of Scratch independently of the debugger. Another teacher praised the debugger in the free-text response but at the same time explained low ratings with being unsure whether to use Scratch in lessons at all. More generally, Fig. 22 shows that even though the perceived usefulness is high, the intention to use is visibly lower, which confirms issues in how debugging is perceived in education [38].

Summary (*RQ 1*): Teachers believe that NuzzleBug is useful and understandable, but suggest improving the user interface.

5.2 RQ2: Usability Study with Children

5.2.1 Study participants. We conducted the study at a secondary school with six school classes (two year 6 aged 12, one year 7 aged 13, two year 9 aged 15, and one year 10 aged 16) with a total of 125 pupils. Prior to our study, the computer science teacher of these classes implemented the same Scratch introduction in regular lessons for all classes, in which all programming basics required for our study (e.g., loops, branches, message passing, sprite cloning) were first explained, and then practiced by the pupils using a detailed task of implementing a game covering all these concepts.

5.2.2 Experiment tasks. A debugging task in our study consists of a faulty Scratch project with a textual description of (1) expected behavior of the program and (2) the incorrect behavior, and (3) their task, which is to "fix the faulty behavior". Table 3 shows details of the individual tasks in terms of the difficulty estimated using the number of programming concepts (e.g., loops, conditions, message passing, sprite cloning, ...) included in a program (easy = 2, medium = 3, hard > 3), the number of blocks and scripts, and the inter-procedural cyclomatic complexity. Each task was created to contain exactly one bug, and the bugs are based on common bug patterns in Scratch [19]. In addition we created four introductory debugging tasks, two of which introduce bugs into the program created by the pupils in their Scratch preparation lesson, and two easy bugs similar to the main experiment tasks. To increase the number of data points, for each task there are two versions with identical code and bug, but slightly changed theme and storyline.

Table 3: Debugging task statistics

Difficulty counts programming concepts (easy = 2; medium = 3; hard = > 3); B = Blocks; S = Scripts; ICC = inter-procedural cyclomatic complexity; Bugs refer to common Scratch bug patterns [19].

Task	Difficulty	B	S	ICC	Tests	Bug	Data points
1	easy	23	4	7	3	Type Error	125
2	easy	27	4	9	3	Missing loop sensing	125
3	med.	65	11	26	6	Message never sent	123
4	med.	37	4	11	3	Position equals check	122
5	med.	35	6	13	1	Forever inside loop	110
6	med.	48	9	20	6	Message never received	103
7	hard	104	9	35	2	Interrupted loop sensing	95
8	hard	48	6	16	3	Missing clone call	79

5.2.3 Experiment procedure. The study itself covered two two-hour lessons, each consisting of an (1) introduction with explanations, followed by (2) two introductory tasks, and then (3) 40 minutes dedicated to four debugging tasks. In the first week we taught the basics of debugging without a debugger using teaching material adapted from existing didactic approaches for systematic debugging with text-based languages [39]. The introductory tasks include diagrams explaining the steps involved in systematic debugging. After the introduction the pupils practiced debugging using four debugging tasks. In the second week we first demonstrated systematic debugging using NuzzleBug, and pupils then practiced with two introductory tasks, where the task description included modified diagrams of the steps involved in systematic debugging with a debugger [39]. Afterwards, they had to solve four debugging tasks within 40 minutes. Their instructions were to fix the bugs, but they were free to choose whether to use NuzzleBug. We used a modified Scratch version [9] with a button for indicating when a task is completed, collecting the corresponding .sb3 file and time, and resulting in the data points per task shown in Table 3.

Each session consisted of one easy, two medium, and one hard task. To collect data for all eight tasks in the second session, we used tasks 1, 3, 4, 7 for half the classes in the first session and tasks 2, 5, 6, 8 in the second session, and vice versa for the other half. To avoid that information about tasks was exchanged between groups, we used the alternative task versions depending on the order.

5.2.4 Experiment analysis. The analysis is based on the eight debugging tasks used during the second session. For each pupil and task a modification in the Scratch UI logged whether they used NuzzleBug, and we compare those who used NuzzleBug with those who did not. To determine whether a bug was correctly fixed we created automated tests for each task using the Whisker [56] framework, which allows specifying UI tests for Scratch programs using JavaScript notation. The number of tests per task is shown in Table 3; note that the tests not only check if the bug was fixed, but also check that all other behavior of the program has not been broken. We executed the tests on each pupil solution, and count a bug as successfully fixed if all tests pass. We compare the fix ratios between pupils using and not using NuzzleBug with a Fisher

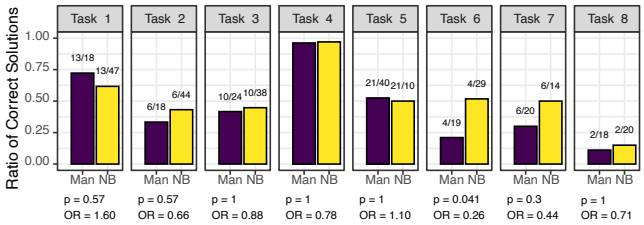


Figure 23: Ratio of correctly fixed programs with manual debugging (Man) and with NuzzleBug (NB).

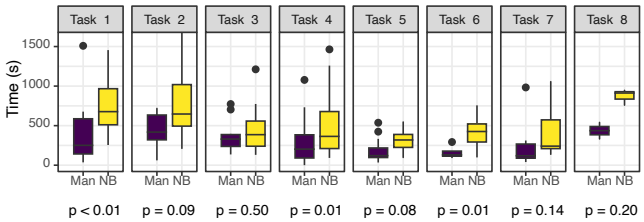


Figure 24: Time spent for correct fixes.

exact test at $\alpha = 0.05$ and the odds ratio as effect size measure. The modified version of Scratch [9] also keeps track of the time spent per completed task. We compare the times between pupils using and not using NuzzleBug with a Mann-Whitney U test at $\alpha = 0.05$.

5.2.5 Threats to Validity. Threats to *internal validity* result as we did not force pupils to use a debugger, but compare those who chose to use it with those who did not, which may be those who immediately spotted bugs or are overwhelmed by the debugger. To reduce the influence of the latter cause, we monitored the pupils during the study and provided feedback to pupils who were lost. Results may differ depending on how debugging is taught, therefore the tasks and experiment sessions were co-designed and conducted by a teacher. We also conducted a pilot study with another school class and refined the debugger and teaching material based on the experience. Results may further differ depending on programming knowledge, but all pupils received the same introduction to Scratch and were taught by the same computer science teacher. Threats to *construct validity* arise from measuring whether programs were fixed and how quickly; the debugger may have helped locating but not fixing the fault. Furthermore, in an educational setting the learning outcome may be more important than the raw fix ratio or performance. Threats to *external validity* arise from our sample of pupils and faulty programs, and results may not generalize.

5.2.6 Results. Overall, 82 debugging tasks were addressed using NuzzleBug, and 65 without. Figure 23 shows the ratio of pupils who correctly fixed the bug. For six out of the eight tasks, the ratio is higher when using NuzzleBug, and for Task 6 the improvement is statistically significant (Fisher exact test at $\alpha = 0.05$). This bug consists of incorrectly nested forever-loops, which can be detected with stepping, or by asking why blocks or behavior outside the loops are not reached. A large improvement can also be seen for Task 7, which contains incorrect handling of clone generation, which can easily be understood by asking the right question. Both of these tasks represent more complex programs; the task where the debugger helped least (Task 1) is a very simple program, where sprite collision is incorrectly checked only once rather than continuously. Understanding this bug requires understanding timing visualizations, for which deriving a fix may be more challenging as no code is shown—which may be challenging for children, who have been shown to focus on code rather than deducing a causal model [26].

Figure 24 shows the time spent for the cases where the bug was successfully fixed. For all tasks the time is higher with NuzzleBug (significant for tasks 1, 4, and 6). On the one hand, this may be influenced by pupils ignoring the debugger if they spot a bug immediately. On the other hand, NuzzleBug provides substantially more information to process compared to the few blocks of most programs. Interestingly, the Whyline interrogative debugger was reported to reduce time [28]. However, we did not study students in higher education, but children, which leads to unique challenges, such as their focus on solutions themselves rather than how the program works [26, 61]. Furthermore, in a teaching context it might actually be more productive when pupils spend more time with debugging as this may consolidate knowledge.

Summary (*RQ 2*): The effectiveness of pupils at fixing faults increases when using NuzzleBug, but those who manage to fix the fault without the debugger tend to be quicker.

6 CONCLUSIONS

Programmers who are good at debugging are more likely to be good at programming [11], but being able to program does not immediately result in being able to debug [25]. Novice programmers in particular lack some of the necessary skills [14] for debugging, are often discouraged by errors in their code [42], and struggle with debugging [17, 24, 41, 52, 61]. Although the importance of debugging for novices has been known for decades [42, 55], most programming courses still do not cover it in detail [8] and debugging tools are lacking, in particular for Scratch. This inhibits novices, debugging education, and educators trying to support students.

To remedy this situation we introduced NuzzleBug, a debugger for Scratch providing classic, omniscient, and interrogative debugging functionality. Our initial experiments showed promising results, but also suggest future research on improving the techniques on which NuzzleBug is based, for example by narrowing down or abstracting questions, by integrating further automated debugging techniques such as spectrum-based fault localization [62], by improving answers with fix suggestions [16, 37, 44, 45, 66], by integrating generated code explanations [49], or by automatically generating breakpoints [65]. There is also further need for research on education of systematic debugging [39] since debugging is generally not taught sufficiently, and in particular not using debugging tools [38]. To support education and research on debugging, all code, tasks, and data are available at: https://figshare.com/s/68281791e01be5179699

The source code of NuzzleBug is also available here:

https://github.com/se2p/nuzzlebug

7 ACKNOWLEDGMENTS

We thank Stephan Gramüller and Edi Wasmeier for supporting the studies; Phil Werli, Sebastian Schweikl, and Patric Feldmeier for supporting the implementation; and Luisa Greifenstein and Ute Heuer for supporting the design of questions and answers. This work is supported by the DFG under grant FR 2955/3-1 "TENDER-BLOCK: Testing, Debugging, and Repairing Blocks-based Programs".

REFERENCES

[1] Junghyun Ahn, Woonhee Sung, and John B Black. 2022. Unplugged debugging activities for developing young learners' debugging skills. *Journal of Research in Childhood Education* 36, 3 (2022), 421–437.

[2] Jung-Hyun Ahn, Yaoli Mao, Woonhee Sung, and John B Black. 2017. Supporting Debugging Skills: Using Embodied Instructions in Children's Programming Education. In *Society for Information Technology & Teacher Education International Conference*. Association for the Advancement of Computing in Education (AACE), Waynesville, NC, USA, 19–26.

[3] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michal Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *SPLASH-E '19 - Proceedings of the 2019 ACM SIGPLAN Workshop on SPLASH-E, Athens, Greece, October 25, 2019*, Elisa L. A. Baniassad (Ed.). ACM, 7–12. https://doi.org/10.1145/3358711.3361630

[4] Robert M Balzer. 1969. Exdams: extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*. AFIPS Press, 567–580.

[5] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-travel debugging for JavaScript/Node.js. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 1003–1007. https://doi.org/10.1145/2950290.2983933

[6] David Bau, D. Anthony Bau, Matthew Dawson, and C. Sydney Pickens. 2015. Pencil code: block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children, IDC '15, Medford, MA, USA, June 21-25, 2015*, Marina Umaschi Bers and Glenda Revelle (Eds.). ACM, 445–448. https://doi.org/10.1145/2771839.2771875

[7] Amanda Boss, Cali Stenson, and Jeremy Ruten. 2015. Visual debugging technology with pencil code: Position paper. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 115–117.

[8] Elizabeth Carter. 2015. Its Debug: Practical Results. *Journal of Computing Sciences in Colleges* 30, 3 (2015), 9––15.

[9] Laura Caspari, Luisa Greifenstein, Ute Heuer, and Gordon Fraser. 2023. ScratchLog: Live Learning Analytics for Scratch. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, Turku, Finland, July 7-12, 2023*, Mikko-Jussi Laakso, Mattia Monga, Simon, and Judithe Sheard (Eds.). ACM, 403–409. https://doi.org/10.1145/3587102.3588836

[10] Chiung-Fang Chiu and Hsing-Yi Huang. 2015. Guided debugging practices of game based programming for novice programmers. *International Journal of Information and Education Technology* 5, 5 (2015), 343.

[11] Ryan Chmiel and Michael C. Loui. 2004. Debugging: From Novice to Expert. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2004, Norfolk, Virginia, USA, March 3-7, 2004*, Daniel T. Joyce, Deborah Knox, Wanda P. Dann, and Thomas L. Naps (Eds.). ACM, 17–21. https://doi.org/10.1145/971300.971310

[12] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges* 15, 5 (2000), 107–116.

[13] Yihuan Dong, Samiha Marwan, Veronica Cateté, Thomas W. 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, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, Elizabeth K. Hawthorne, Manuel A. Pérez-Quiñones, Sarah Heckman, and Jian Zhang (Eds.). ACM, 1204–1210. https://doi.org/10.1145/3287324.3287437

[14] Mireille Ducasse and Anna-Maria Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 11th International Conference on Software Engineering*. IEEE Computer Society Press, 162–171.

[15] Marc Eisenstadt. 1997. My Hairiest Bug War Stories. *Commun. ACM* 40, 4 (1997), 30–37. https://doi.org/10.1145/248448.248456

[16] Benedikt Fein, Florian Obermüller, and Gordon Fraser. 2022. CATNIP: An Automated Hint Generation Tool for Scratch. In *ITiCSE 2022: Innovation and Technology in Computer Science Education, Dublin, Ireland, July 8 - 13, 2022, Volume 1*, Brett A. Becker, Keith Quille, Mikko-Jussi Laakso, Erik Barendsen, and Simon (Eds.). ACM, 124–130. https://doi.org/10.1145/3502718.3524820

[17] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116. https://doi.org/10.1080/08993400802114508

[18] International Organization for Standardization. [n. d.]. *ISO/IEC/IEEE 24765:2017*.

[19] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common Bugs in Scratch Programs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020*, Michail N. Giannakos, Guttorm Sindre, Andrew Luxton-Reilly, and Monica Divitini (Eds.). ACM, 89–95. https://doi.org/10.1145/3341525.3387389

[20] Xuemin Gao and Khe Foon Hew. 2023. A flipped systematic debugging approach to enhance elementary students' program debugging performance and optimize cognitive load. *Journal of Educational Computing Research* (2023), 07356331221133560.

[21] Luisa Greifenstein, Florian Obermüller, Ewald Wasmeier, Ute Heuer, and Gordon Fraser. 2021. Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training. In *WiPSCE '21: The 16th Workshop in Primary and Secondary Computing Education, Virtual Event / Erlangen, Germany, October 18-20, 2021*, Marc Berges, Andraes Mühling, and Michal Armoni (Eds.). ACM, 3:1–3:10. https://doi.org/10.1145/3481312.3481344

[22] Brian Harvey and Jens Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? *Proc. Constructionism* (2010), 1–10.

[23] Juraj Hromkovic, Jacqueline Staub, et al. 2021. The Problem with Debugging in Current Block-based Programming Environments. *Bulletin of EATCS* 135, 3 (2021). http://bulletin.eatcs.org/index.php/beatcs/article/view/667

[24] Matthew C. Jadud. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40. https://doi.org/10.1080/08993400500056530

[25] Claudius M. Kessler and John R. Anderson. 1986. A Model of Novice Debugging in LISP. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., 198––212.

[26] ChanMin Kim, Jiangmei Yuan, Lucas Vasconcelos, Minyoung Shin, and Roger B Hill. 2018. Debugging during block-based programming. *Instructional Science* 46 (2018), 767–787.

[27] Amy J. Ko, Htet Htet Aung, and Brad A. Myers. 2005. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 126–135. https://doi.org/10.1145/1062455.1062492

[28] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*, Elizabeth Dykstra-Erickson and Manfred Tscheligi (Eds.). ACM, 151–158. https://doi.org/10.1145/985692.985712

[29] Amy J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg (Eds.). ACM, 1569–1578. https://doi.org/10.1145/1518701.1518942

[30] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Software Eng.* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[31] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 492–501. https://doi.org/10.1145/1134285.1134355

[32] Michael Jongseon Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, Sheridan Long, Margaret M. Burnett, and Amy J. Ko. 2014. Principles of a debugging-first puzzle game for computing education. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, Scott D. Fleming, Andrew Fish, and Christopher Scaffidi (Eds.). IEEE Computer Society, 57–64. https://doi.org/10.1109/VLHCC.2014.6883023

[33] Bil Lewis. 2003. Debugging backwards in time. *arXiv preprint cs/0310016* (2003).

[34] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan D. Tempero, and Paul Denny. 2018. Ladebug: an online tool to help novice programmers improve their debugging skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018, Larnaca, Cyprus, July 02-04, 2018*, Irene Polycarpou, Janet C. Read, Panayiotis Andreou, and Michal Armoni (Eds.). ACM, 159–164. https://doi.org/10.1145/3197091.3197098

[35] John H. Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4 (2010), 16:1–16:15. https://doi.org/10.1145/1868358.1868363

[36] Phoebe Martinez, John Lopez, Fernando J. Rodríguez, Joseph B. Wiggins, and Kristy Elizabeth Boyer. 2020. Novice Debugging in Block-Based and Hybrid Environments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, Portland, OR, USA, March 11-14, 2020*, Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter, and Alvaro E. Monge (Eds.). ACM, 1291. https://doi.org/10.1145/3328778.3372642

[37] Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2022. A Survey of Automated Programming Hint Generation: The HINTS Framework. *ACM Computing Surveys (CSUR)* 54, 8 (2022), 172:1–172:27. https://doi.org/10.1145/3469885

[38] Tilman Michaeli and Ralf Romeike. 2019. Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. In *IEEE Global Engineering*

Education Conference, EDUCON 2019, Dubai, United Arab Emirates, April 8-11, 2019, Alaa K. Ashmawy and Sebastian Schreiter (Eds.). IEEE, 1030–1038. https://doi.org/10.1109/EDUCON.2019.8725282

[39] Tilman Michaeli and Ralf Romeike. 2019. Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education, WiPSCE 2019, Glasgow, Scotland, UK, October 23-25, 2019*. ACM, 15:1–15:7. https://doi.org/10.1145/3361721.3361724

[40] Michael A. Miljanovic and Jeremy S. Bradbury. 2017. RoboBUG: A Serious Game for Learning Debugging Techniques. In *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER 2017, Tacoma, WA, USA, August 18-20, 2017*, Josh Tenenberg, Donald Chinn, Judy Sheard, and Lauri Malmi (Eds.). ACM, 93–100. https://doi.org/10.1145/3105726.3106173

[41] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. (2008), 163–167. https://doi.org/10.1145/1352135.1352191

[42] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., USA.

[43] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Software* 26, 6 (2009), 78–85. https://doi.org/10.1109/MS.2009.169

[44] Thomas W. Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating Data-driven Hints for Open-ended Programming. In *Proceedings of the 9th International Conference on Educational Data Mining, EDM 2016, Raleigh, North Carolina, USA, June 29 - July 2, 2016*, Tiffany Barnes, Min Chi, and Mingyu Feng (Eds.). International Educational Data Mining Society (IEDMS), 191–198.

[45] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2017, Seattle, WA, USA, March 8-11, 2017*, Michael E. Caspersen, Stephen H. Edwards, Tiffany Barnes, and Daniel D. Garcia (Eds.). ACM, 483–488. https://doi.org/10.1145/3017680.3017762

[46] Mitchel Resnick, John H. Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

[47] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, and Diana Franklin. 2019. A K-8 Debugging Learning Trajectory Derived from Research Literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, Elizabeth K. Hawthorne, Manuel A. Pérez-Quiñones, Sarah Heckman, and Jian Zhang (Eds.). ACM, 745–751. https://doi.org/10.1145/3287324.3287396

[48] Jonathan B. Rosenberg. 1996. *How debuggers work: algorithms, data structures, and architecture*. John Wiley & Sons, Inc.

[49] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *ICER 2022: ACM Conference on International Computing Education Research, Lugano and Virtual Event, Switzerland, August 7 - 11, 2022, Volume 1*, Jan Vahrenhold, Kathi Fisler, Matthias Hauswirth, and Diana Franklin (Eds.). ACM, 27–43. https://doi.org/10.1145/3501385.3543957

[50] Anthony Savidis and Crystalia Savaki. 2019. Complete Block-Level Visual Debugger for Blockly. In *Proceedings of the 2nd International Conference on Human Systems Engineering and Design: Future Trends and Applications, IHSED 2019, Munich, Germany, September 16-18, 2019 (Advances in Intelligent Systems and Computing, Vol. 1026)*, Tareq Z. Ahram, Waldemar Karwowski, Stefan Pickl, and Redha Taïar (Eds.). Springer, 286–292.

[51] Ben Selwyn-Smith, Craig Anslow, and Michael Homer. 2022. Blocks, Blocks, and More Blocks-Based Programming. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, PAINT 2022, Auckland, New Zealand, 5 December 2022*, Tom Beckmann, Robert Hirschfeld, Juan Pablo Sáenz, and Mauricio Verano Merino (Eds.). ACM, 35–47. https://doi.org/10.1145/3563836.3568726

[52] Beth Simon, Dennis J. Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. 2008. Common sense computing (episode 4): debugging. *Computer Science Education* 18, 2 (2008), 117–133. https://doi.org/10.1080/08993400802114698

[53] Arnan Sipitakiat and Nusarin Nusen. 2012. Robo-Blocks: designing debugging abilities in a tangible programming system for early primary school children. In *The 11th International Conference on Interaction Design and Children, IDC '12, Bremen, UNK, Germany, June 12 - 15, 2012*, Heidi Schelhowe (Ed.). ACM, 98–105. https://doi.org/10.1145/2307096.2307108

[54] Chrysanthos Socratous and Andri Ioannou. 2021. Structured or unstructured educational robotics curriculum? A study of debugging in block-based programming. *Educational Technology Research and Development* 69 (2021), 3081–3100.

[55] Elliot Soloway and James C. Spohrer. 1988. *Studying the Novice Programmer*. L. Erlbaum Associates Inc., USA.

[56] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 165–175. https://doi.org/10.1145/3338906.3338910

[57] Gregory Tassey. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology* (2002).

[58] Marilyn Tenorio Melenje María, María Alejandra Trujillo, Julio Ariel Hurtado Alegría, and Cesar Collazos. 2019. Debugging Block-Based Programs. In *Human-Computer Interaction: 4th Iberoamerican Workshop, HCI-Collab 2018, Popayán, Colombia, April 23–27, 2018, Revised Selected Papers 4*. Springer, 98–112.

[59] Jennifer Tsan, David Weintrop, and Diana Franklin. 2022. An Analysis of Middle Grade Teachers' Debugging Pedagogical Content Knowledge. In *ITiCSE 2022: Innovation and Technology in Computer Science Education, Dublin, Ireland, July 8 - 13, 2022, Volume 1*, Brett A. Becker, Keith Quille, Mikko-Jussi Laakso, Erik Barendsen, and Simon (Eds.). ACM, 533–539. https://doi.org/10.1145/3502718.3524770

[60] Viswanath Venkatesh and Fred Davis. 2000. A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies. *Management Science* 46 (2000), 186–204.

[61] Iris Vessey. 1989. Toward a Theory of Computer Program Bugs: An Empirical Test. *International Journal of Man-Machine Studies* 30, 1 (1989), 23–46. https://doi.org/10.1016/S0020-7373(89)80019-7

[62] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[63] Wei Yan, Maya Israel, Feiya Luo, and Ruohan Liu. 2021. Exploring Elementary Students' Debugging Behaviors in Puzzle-based Programming: A Learning Trajectory Approach. In *SIGCSE '21: The 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA, March 13-20, 2021*, Mark Sherriff, Laurence D. Merkle, Pamela A. Cutter, Alvaro E. Monge, and Judithe Sheard (Eds.). ACM, 1308. https://doi.org/10.1145/3408877.3439625

[64] Andreas Zeller. 2009. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press.

[65] Cheng Zhang, Dacong Yan, Jianjun Zhao, Yuting Chen, and Shengqian Yang. 2010. BPGen: an automated breakpoint generator for debugging. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 271–274. https://doi.org/10.1145/1810295.1810351

[66] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. *arXiv preprint arXiv:2209.14876* (2022).