Testing Scratch Programs Automatically

Andreas Stahlbauer
University of Passau
Germany

Marvin Kreis
University of Passau
Germany

Gordon Fraser
University of Passau
Germany

ABSTRACT

Block-based programming environments like Scratch foster engagement with computer programming and are used by millions of young learners. Scratch allows learners to quickly create entertaining programs and games, while eliminating syntactical program errors that could interfere with progress. However, functional programming errors may still lead to incorrect programs, and learners and their teachers need to identify and understand these errors. This is currently an entirely manual process. In this paper, we introduce a formal testing framework that describes the problem of Scratch testing in detail. We instantiate this formal framework with the Whisker tool, which provides automated and property-based testing functionality for Scratch programs. Empirical evaluation on real student and teacher programs demonstrates that Whisker can successfully test Scratch programs, and automatically achieves an average of 95.25 % code coverage. Although well-known testing problems such as test flakiness also exist in the scenario of Scratch testing, we show that automated and property-based testing can accurately reproduce and replace the manually and laboriously produced grading efforts of a teacher, and opens up new possibilities to support learners of programming in their struggles.

CCS CONCEPTS

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

KEYWORDS

Automated Testing, Scratch, Education, Property-based Testing

ACM Reference Format:
Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing Scratch Programs Automatically. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338910

1 INTRODUCTION

Block-based programming languages [4] are intended to engage young learners with computer programming, and programming environments like Scratch [31] are hugely successful at doing so. At the time of this writing, more than 37 million Scratch projects have been shared by their creators, and many more programs are written without sharing, offline, or using some of the many derivative and related programming environments for learners that follow similar principles. A particular advantage of the block-based paradigm contributing to this success is that it eliminates syntactical programming errors [4]: In Scratch, programs are assembled by dragging and dropping command blocks from a toolbox; these blocks can only be assembled in syntactically valid ways to programs. However, while being syntactically correct, a program can be functionally incorrect, and there is nothing about block-based programming languages that would make identifying and correcting functional errors easier than in any other programming language.

Functional correctness is usually assessed with automated tests, which are a prerequisite for many other program analysis techniques ranging from debugging to repair. While Scratch programmers might be unlikely to write tests for their educational and fun programs, this does not mean there is no demand for automated analysis: In particular *because* users are learners, they would particularly benefit from tool support to overcome programming problems. This demand extends from learners to their teachers, who have to provide feedback, debug programs, and grade them. However, even though Scratch programs are constructed from standard program statements, what defines a program itself is different from standard programming languages. Without a notion of functions, tests have to focus on the user interface, which consists of objects (sprites) interacting with each other on a graphical stage. The properties of these objects may be fuzzy such that pixel-perfect precision is often not realistic to expect, and the nature of typical Scratch projects means that randomness is often involved. Currently, there are no means for automatically testing Scratch programs.

In this paper, we explore the problem of automatically testing Scratch programs. Since Scratch was created mainly with pedagogical goals in mind, it differs fundamentally from other types of programs. From a technical perspective, a Scratch program runs instances of its functional units (scripts) highly concurrently by dynamic process creation, and its behavior is determined in an event-driven manner; events can stem from interactions with the user, such as inputs from the keyboard or the mouse, can be triggered by different functional units of the Scratch program itself, which can send and receive custom messages, or can originate from peripheral hardware the program is supposed to interact with. We describe the desired behavior with a set of temporal properties, which constitute the program specification. Since Scratch programs heavily rely on timers (for example, to measure elapsed or remaining time, and to trigger events or choose from different control-flow branches to proceed in), the logic needs to provide a means to reason about real time besides being able to reason about temporal relationships of different behaviors and states.

These properties are checked by executing Scratch tests, which include sequences of events that are sent to the Scratch program. In order to test whether a program violates specified properties, we

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338910

Figure 1: A perspective on Scratch programs

have developed a framework to execute user-specified automated tests that exercise these properties. Since the combinatorial explosion of possible sequences of events that can be sent to a Scratch program, and its concurrent execution, are challenging to check with traditional testing, our framework provides functionality based on property-based testing [8, 14]: Automated input generation is used to exercise a Scratch program, while properties (safety and bounded liveness) are monitored for violations.

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

- We introduce a formal framework that defines Scratch programs and the problem of Scratch testing.
- We introduce Whisker, a concrete instantiation of our formal Scratch testing framework, which supports manually written tests as well as automated property-based testing.
- We empirically evaluate the feasibility of Scratch testing and the effectiveness of Whisker on Scratch projects in an educational context where tests are used for auto-grading.

Our study on 37 student-written and teacher-marked Scratch programs reveals a strong correlation (Pearson's correlation of up to 0.9) between automated tests and the grades determined by a teacher, demonstrating that automated Scratch testing with both, manually written tests as well as automated property-based test generation, is feasible. Experiments on 24 common educational projects further demonstrate that automated test generation is able to achieve an average of 95.25 % code coverage on Scratch projects, and can thus relieve users (teachers) from the need to provide test inputs manually. However, our experiments also reveal that problems known from the wider field of software testing, such as test flakiness, also exist in the domain of testing Scratch programs.

2 TESTING SCRATCH PROGRAMS

In the following, we provide a formalization for discussing automated testing for the Scratch programming language. Sets are denoted by *uppercase* letters: $A, B, \ldots, Z$. Elements of sets are denoted by *lowercase* letters: $a, b, \ldots, z$. Lists, vectors, sequences, or tuples (or sets thereof) are denoted by adding a *bar*: $\bar{a}, \bar{B}, \bar{Z}$. The set of all words over a set S is denoted by S^*. We consider lists, words, and sequences to be dual to each other. Sets get enumerated in curly brackets: $\{e_1, \ldots, e_n\}$. Sequences are enumerated in angle brackets: $\langle e_1, \ldots, e_n \rangle$. Tuples are shown in round brackets: $(e_1, \ldots, e_n)$.

2.1 The Scratch Programming Language

We describe and formalize three perspectives on Scratch: The user (programmer) perspective, its syntactic model, and its semantic model. These perspectives aid in (1) getting a deep understanding of the expressiveness and design of the language, and in (2) providing a precise description of our automatic testing approach.

User Perspective. Scratch programs are created by combining visual *blocks* that correspond to syntactical elements of the language, such as control-flow structures. Only blocks that lead to syntactically valid programs can be combined (no syntactical coding errors). From the user-perspective, a Scratch program consists of the *stage*, which represents the application window with its background image, a collection of *sprites* that are rendered as different images on the stage, which the user can interact with and manipulate. Thus conceptually (see Fig. 1) the user interface is composed of a list of *layers* $\bar{v} = \langle \text{Stage}, \text{Sprite}_1, \ldots \rangle$, where the stage and its sprites map to exactly one layer each. We abstract from possible additional layers representing canvases on which the developer can paint programmatically or on which movies can be shown without loss of generality. Each layer has a collection of *scripts* that define the actual program logic. In the development environment, the execution of programs is started by clicking on a *green flag* symbol. These *concurrent* programs are interpreted and executed in the Scratch virtual machine, which then makes use of techniques like WebGL, Web Workers, and Web Audio to provide a user interface.

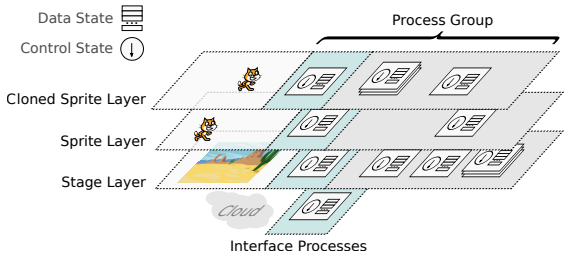


Figure 2: The semantic Scratch program model.

Syntactic Model. On the syntactical perspective, we formally represent a Scratch program as a sequence $\text{App} = \langle \bar{s}_1 \ldots \bar{s}_n \rangle$ of groups of scripts, where a *script group* $\bar{s}_i = \langle s_{i1} \ldots s_{in} \rangle \in S^*$ binds scripts logically and maps either to the stage or one of the sprites. By convention, the first script of a script group always stores the actual images (*costumes*) for drawing the sprite. The collection of all scripts is denoted by the symbol S. A *script* $s = (L, X, G, l_0) \in S$ is a tuple that represents a control-flow automaton [20], with the set of *control locations* L, the set of *data locations* X, the *control transition relation* $G \subseteq L \times Ops \times L$, and the *initial control location* l_0. The set of program operations Ops is defined by a grammar, which is constituted from a set of communication primitives (send, receive, wait), operations to control processes (clone, kill), possibilities to determine the current time (now), and expressions on the types that are supported by Scratch (number, string, bool, and lists thereof). The semantics of the operations corresponds to similar operations found in programming languages such as C or Go.

Semantic Model. Running a Scratch program results in the creation of a collection of anonymous *processes*—see Fig. 2 for an illustration. Each process $p \in P$ is the instance of a script $s = (L, X, G, l_0) \in S$ which belongs to a script group $\bar{s}$. The processes of a script group are instantiated to form a *process group* which may contain several instances of the same script. By convention, the first process p_1 in a process group $\bar{w} = \langle p_1, \ldots \rangle$ is the *interface process* and corresponds to the first script in the script group. The interface process provides an interface to the environment, such as the underlying virtual machine, or peripheral hardware components. We use interface processes, for example, to trigger the initialization of the graphical

environment, and providing the possibility of querying and manipulating its attributes to other processes. Beside providing access to the graphical user interface, we also use interface processes to store and read data on remote processes (services)—which reflects the Scratch feature to store variables in the Cloud.

A *concrete state* $c = \langle p_1, \ldots, p_n \rangle \in C$ of a Scratch program is modeled as a list of concrete process states. A concrete process state maps to each data location $x \in X_p$ a concrete value—which is dependent on the type of the data location. The set of data locations $X_p = X \cup \{\text{pc}, \text{pstate}, \text{pgroup}, \text{pwaitfor}, \text{ptime}\}$ extends the set X of data locations from the script by five process-related entries: The program counter $\text{pc} \in L$ that stores the current position in the control flow of the corresponding script, the *process mode* $\text{pstate} \in \{\text{Done}, \text{Running}, \text{Wait}, \text{Yield}\}$, the process group id $\text{pgroup} \in \mathbb{N}_0$, a bounded number of awaited messages $\text{pwaitfor} \in \mathbb{N}_0^k$ (at most k many), and the system time $\text{pwaitfor} \in \mathbb{N}_0$ in milliseconds since 1970—which we assume to be synchronized among all processes. The function $\bar{l} : C \to L^*$ maps a concrete state c to a sequence $\bar{l}(c) = \langle l_1, \ldots \rangle$ of locations that reflects the current position in the control flow of the scripts.

The set of all feasible executions of a Scratch program is described as a collection of *concrete execution traces* $\overline{C} \subseteq C^*$, where the set C^* consists of all words over the set of concrete states C. Each *execution trace* $\bar{c}_i = \langle c_1, \ldots, c_n \rangle \in \overline{C}$ starts in an *initial concrete state* c_1 for which all processes are on the initial control location l_0 of the corresponding scripts. An execution trace $\bar{c}_i$ describes an interleaving of processes that are executed in parallel—as studied in trace theory [33]. We also refer to the set of all feasible execution traces of a Scratch program App by its denotation $[\![\text{App}]\!] \subseteq C^*$.

The semantics of Scratch can be described using a memory model that is based on *message passing*, where processes can only communicate by passing messages and without using globally shared variables for this purpose. Several arguments support this: (1) Scratch allows to store *variables in the Cloud* and the message passing paradigm can model the degree of reliability of processes [9], (2) Scratch allows to connect other sources of information such as hardware sensors that are connected in different, more or less reliable, ways, and (3) shared memory access can be modeled using message passing [3, 42].

2.2 Scratch Testing

Building on our formal definition of Scratch programs, we now (1) define our understanding of testing, (2) characterize the type of properties that we test, (3) describe how tests for Scratch are operationalized (4) how these operationalized tests are executed to actually check the program, and (5) provide a notion of error witness for failed Scratch tests. Please note that we test fully integrated Scratch programs, that is, on the level of *system tests*.

We assume that testing is based on the following *conceptual workflow*, which breaks down the specification and testing process into small tasks. Please note that we assume this to be the *optimal* process; in practice, step (2) and step (3) might be combined; in ongoing work we provide a more user-oriented way of specifying Scratch programs in a Scratch-like manner.

(1) *Formulation.* An *informal requirements specification* is written or available in natural language—which in a learning context would typically be presented to the learners as assignment.
(2) *Formalization.* Conceptually, the requirement specification is translated into a *formal language* (e.g., temporal logic) where the actual implementation techniques are not yet defined. The result is a formal specification, that is, a set Φ_{App} of properties that the program must satisfy.
(3) *Operationalization.* The properties are *operationalized* by translating them into a representation for actually comparing them to the semantic model of the program (submission). The result is a set $S_{\Phi_{\text{App}}}$ of operationalized properties.
(4) *Testing.* The implementation is *tested* on whether it satisfies the operationalized set of properties or not—also known as *conformance testing* [14]. For this purpose, the program has to be executed with appropriate test inputs such that the expected behavior is observed or an error can be witnessed.

Test, Test Input, Property. A *test* is a means to show the presence of a bug, or show the absence of a bug for a specific set of inputs [10]. The notion of a test is inseparable from a notion of correctness, which specifies the correct and expected behavior—which we denote as the desired *property* to test. A *test input* is an input—which can be a sequence of input symbols or triggered events—that is passed to the system under test; one test input can be applicable to conduct different tests, that is, to check different properties—which is known as collateral coverage. The process of *testing* is conducted by running a set of tests, that is, stimulating the system with test inputs and checking whether the properties are satisfied or not.

More formally, we define a *test* as a pair (ϕ, τ) of a *temporal property* ϕ to check and a *test input* τ. The property $\phi \in \Phi$ defines the set of execution traces of a program that exhibit the desired behavior. The denotation of a property $[\![\cdot]\!] : \Phi \to 2^{C^*}$ maps a given property ϕ to the set of execution traces $[\![\phi]\!] \subseteq C^*$ that satisfy the property. The test input $\tau \in T$ determines which execution trace of a program to check. That is, a test input is a function $\tau : 2^{C^*} \to 2^{C^*}$ that guides and restricts the set of execution traces that a program execution can take; given a set of execution traces $\overline{C} \subseteq C^*$, a test input is said to be *effective* if and only if $\tau(\overline{C}) \subset \overline{C}$.

Timed Temporal Properties. Different logics are available [25] (as well as corresponding algorithms and tools [5]) to specify and reason about desired properties of a program formally and on an abstract level. These logics are different in their levels of abstractions and their expressiveness. The actual choice of a specification logic is often a matter of taste and should align to the users' needs. The behavior (control flow) of Scratch programs is heavily influenced by (discrete) timers, which express how much time has elapsed since they have been started, and the calculus that can be done on this points in time. That is, a temporal logic that is used to specify Scratch programs should also be able to express properties based on real time. One example is MITL (metric temporal logic with intervals) [39], which extends LTL (linear temporal logic) and modifies the temporal operators to also reason about intervals of time and the relationship of this intervals. Table 1 provides examples for specifications in natural language and their MITL counterparts.

We use temporal logic in this paper to provide a clean formalisation, but it is *not necessary in practice for users* to express requirements in temporal logic. There would, however, be several benefits of doing so: (1) going from the abstract to the concrete

Table 1: An example specification. See the legend below.

#	Natural Language ↙ Temporal Logic (MITL)
1	Sprite A moves downwards only. $\mathcal{G}\ \neg(A.y' > A.y)$
2	The countdown ticks down once a second. $\mathcal{G}\ \mathcal{F}_{[0,1\,\mathrm{s}]}(countdown' = countdown - 1)$
3	Points get increased by 7 whenever sprite A touches sprite B. $\mathcal{G}\ (\neg\mathrm{touches}(A,B) \vee \mathcal{F}_{[0,100\,\mathrm{ms}]}(points' = points + 7))$

Legend. We use MITL with the temporal operators: eventually $\mathcal{F}$, next $\mathcal{X}$, globally $\mathcal{G}$, until $\mathcal{U}$, and release $\mathcal{R}$. A subscript $\bar{t} = [a, b]$ can be added to each operator such that the condition is restricted to hold in the given time interval. The given predicates can be translated into propositions [27].

specifying desired properties of a program helps in *dividing the problem*, it (2) provides a less ambiguous [32] description of the expected behavior while *not anticipating a solution* upfront, and (3) this would provide a tool for Scratch to *introduce people to temporal logics* and the process of formal specification.

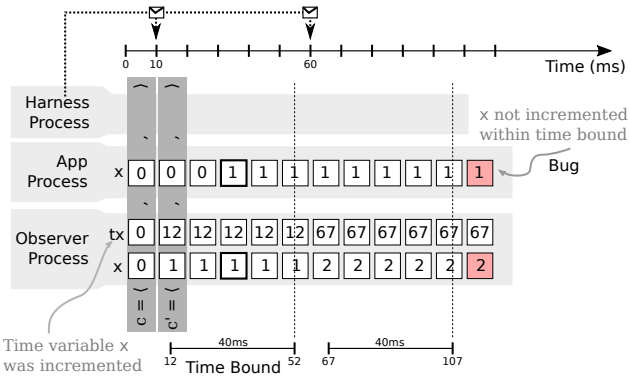
Test Operationalization. To model how a given test $t = (\phi, \tau) \in T$ is executed on the Scratch program App $= \langle \bar{s}_1, \ldots, \bar{s}_n \rangle$, we operationalize tests by bringing them to the same abstraction level the program is also formalized in, and instantiate both the test input and the property to check as Scratch scripts in script groups:

(1) *Scratch Observer.* A group of scripts—which might consist of a single script—that monitors whether or not the Scratch program under analysis satisfies a property ϕ is called the *Scratch observer*. More precisely, it monitors whether there is a state that *violates* the specification: That is, a *Scratch observer* is a script group $\bar{s}_{\neg\phi} \in S^*$ with at least one script $s_{\neg\phi} = (L, X, G, l_0, l_e) \in \bar{s}_{\neg\phi}$ with an additional *failing target location* $l_e \in L$—which must *not be reached* to satisfy the property.

The denotation $[\![\bar{s}_{\neg\phi}]\!]$ of the Scratch observer consists of the set of all execution traces for which the property is violated, i.e., that reach a violating location l_e. That is, each execution trace $\bar{c} = \langle c_1, \ldots, c_n \rangle \in [\![s_{\neg\phi}]\!]$ terminates with $l_e \in L(c_n)$. A Scratch observer $\bar{s}_{\neg\phi}$ is equivalent to the negation of the property to check, that is, $[\![\bar{s}_{\neg\phi}]\!] = [\![\neg\phi]\!]$.

The Scratch observer does not modify the state of the other processes that are executed along with it while running the test. Nevertheless, it is stateful and can introduce and modify its own variables to keep track of the system state and changes to it. That is, it describes an observable aspect which is known [24] to not affect temporal properties (except next-state properties, which we do not have) of the system to check. The operationalization as Scratch observers provides sufficient expressiveness for safety properties and bounded liveness properties [26, 40].

(2) *Scratch Harness.* A harness is used stimulate and control a subject. We operationalized the test input in a *Scratch harness*, which takes the role of stimulating the Scratch program under test with inputs, which would otherwise be provided by the environment—in our case, by the user—and restricts the state space of the program to analyze. The Scratch harness is a Scratch script group $\bar{s}_\tau \in S^*$ that stimulates the program with a sequence of inputs, each input for a different point in time.



Legend. A variable x must store the number of messages received within a time bound of 40 ms. The Scratch program App to check declares a variable x and initializes it with 0; so does the observer. The observer models the expected behavior and checks whether it corresponds to the actual behavior. The harness process produces the test inputs (messages).

Figure 3: Test execution example

The harness can also reset the random number generator of the Scratch virtual machine based on a seed to ensure deterministic and reproducible test execution. We require that $[\![\bar{s}_\tau]\!] \subseteq [\![\tau]\!]$.

Our operationalization of a given test $t = (\tau, \phi)$ results in a pair $(\bar{s}_\tau, \bar{s}_{\neg\phi}) \in S^* \times S^*$ of script groups.

Test Execution. To check whether a test leads to an undesired state or behavior on a given program, we check whether the intersection of the execution traces denoted by $[\![\mathrm{App}]\!] \cap [\![\tau]\!] \cap [\![\neg\phi]\!]$ is empty, that is, if the program App does not have an execution trace that violates the property ϕ for all given test inputs. Formally, we describe the possible executions of a program by a transition system $\mathrm{TS} = (C, c_1, \rightarrow)$, where $c_1 \in C$ is the initial (concrete) program state and the transition relation $\rightarrow \subseteq C \times C$ denotes all possible transitions between program states. Please note that this transition relation also implicitly encodes information about the scheduling of processes which were instantiated from the Scratch scripts.

Given the program App $= \langle \bar{s}_1, \ldots, \bar{s}_n \rangle$ to test, we add the operationalized test as additional script groups $\bar{s}_{\neg\phi}$ and $\bar{s}_\tau \in S^*$ and create the instrumented program variant $\mathrm{App}' = \langle \bar{s}_\tau, \bar{s}_{\neg\phi}, \bar{s}_1, \ldots, \bar{s}_n \rangle$. The program App′ is then executed by a virtual machine (VM) which is aware of the role of the script groups $\bar{s}_\tau$ and $\bar{s}_{\neg\phi}$, and schedules them such that they can take their roles. One step of the VM, which is represented by the transition relation of the transition system, conducts a special *scheduling* that allows for controlling and observing the execution behavior in a sound manner: (1) The processes of the Scratch harness can make their transitions first, (2) the actual processes of the Scratch program App make their steps afterwards, and finally, (3) the Scratch observer is invoked to check if the resulting state conforms to the specification. That is, one transition $(c_i, c_i''') \in \rightarrow$ of the transition relation for the program App′ consists of following steps:

$$c_i \xrightarrow{\bar{s}_\tau} c_{i'} \xrightarrow{\bar{s}_1, \ldots, \bar{s}_n} c_{i''} \xrightarrow{\bar{s}_{\neg\phi}} c_{i'''}$$

From this scheduling arises a requirement: Certain variables must only be compared by the observer but not assigned to observer-local variables to determine state transitions. The process under

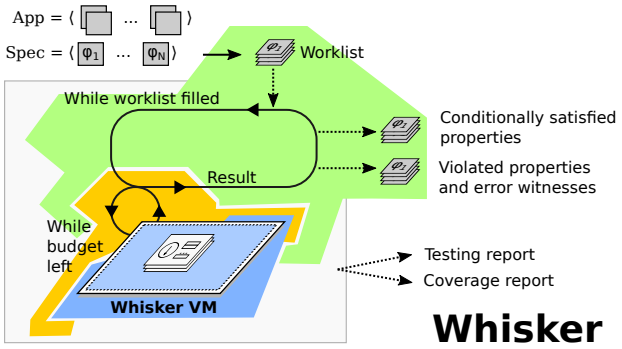


Figure 4: Our automatic test generation framework

observation might or might not have handled a message, which the observer might have already anticipated, which the observer itself could already have done. We assume that the harness defines all information that is provided by the environment, such as mouse positions, pressed keys, and even numbers from random number generators. We assume that the observer receives the same information to model the *expected* behavior.

We allow for *composing* the functionality of harness and observer into one logical unit under the following condition: The result must not restrict the state space of the program under analysis besides modeling an environment, which can produce and consume inputs. This allows for writing tests in a fashion that established tools, such as Selenium, propagate. Figure 3 illustrates a test execution.

Error Witness. To make bugs (property violations) reproducible and to reduce the chance of a false alarm—which could lead to wrong grading in case of an automated grading scenario—an *error witness* [6] is provided. In case of a Scratch program, an error witness is a sequence of inputs $\tau \in T$ for which a given property is violated; that is, one of the states along the execution trace $\bar{c} \in [\![\text{App}]\!]$ that results from replaying the error witness violates the property. We consider only error witnesses of finite length to prove the violation of safety properties and bounded liveness properties [26, 30].

3 WHISKER FRAMEWORK

We present Whisker, a tool and framework for *automatically* testing Scratch programs in a *property-based* [8, 14] manner. The framework maintains a collection of properties to check and provides means for automatically exploring states and behaviors of a given program by generating effective test inputs. Tests are executed on fully-integrated programs, that is, they are system tests which also interact with the environment, for example, the connected output device. Please note that the current version of the framework does not yet provide means to explicitly mock other connected hardware components such as micro:bit [16] or Lego [12] components—for now, we assume that we can simulate these components by providing a corresponding interface process that mocks components by sending and receiving expected messages.

3.1 Framework

Whisker consists of three main components—see Fig. 4: (1) a worklist algorithm, (2) a testing procedure, (3) the Whisker virtual machine; these components are controlled using a user interface.

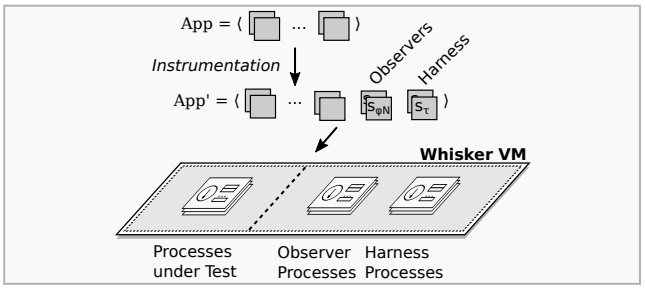


Figure 5: Instantiated processes for testing in Whisker

Input and Output. Whisker takes as input a set of programs to test—for example, submissions for a programming exercise, or different variants of a program in general—and the specification, that is, a set of properties to check. For now, we assume that only one program is tested, but the approach extends to sets of programs naturally. Whisker produces two sets as output: the violated properties, where each property is then paired with an error witness for reproducing the bug, and a set of conditionally satisfied properties—they are tested but not verified. Optionally, Whisker produces a testing report—in the TAP13 format [28]—which summarizes all tests that have been conducted, as well as a coverage report.

Whisker Worklist Algorithm. Whisker executes a worklist algorithm, which maintains three sets: worklist, violated, and tested; the algorithm is illustrated in Fig. 4. The set worklist contains the properties that have not been tested so far. The set violated contains the violated properties, paired with corresponding error witnesses. The set tested contains the set of properties that have been tested and for which Whisker was not able to identify any violations (i.e., they are conditionally satisfied). In each iteration of the algorithm, which loops until the set worklist is empty, a set of properties to test is removed from the set worklist by an operator choose, and then handed over the Whisker test loop for testing.

Whisker Test Loop. Test execution is controlled by the *Whisker test loop*, which is called by the Whisker worklist algorithm and wraps the Whisker virtual machine. This algorithm is responsible for initializing the Whisker virtual machine with the program to test, and to instantiate corresponding processes for the Scratch observers and the Scratch harness (see Fig. 5). That is, Whisker loads and starts the program before each test starts. The test loop repeats with testing the program with new test inputs until the resource budget that is assigned to the set of properties to test is exhausted. To avoid flaky tests [29] due to randomness, the random number generator of the Whisker virtual machine can be initialized with a seed. For testing with randomly generated inputs, programs often need to be reset multiple times inside of one test execution, because the test could get stuck in some state of the program.

Whisker Virtual Machine. The actual execution of tests is conducted in the Whisker virtual machine (VM) in which the Scratch program, the observers, and the harness are instantiated as a set of processes. The Whisker VM wraps the original, and unmodified, Scratch VM and makes use of its step-function. The function step implements the operational semantics of the Scratch programming language and is responsible for making the transition from one concrete state $c \in C$ of the machine to its successor state $c' \in C$. The

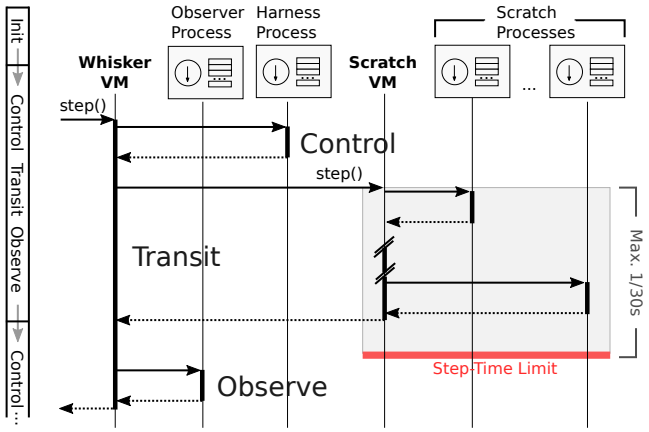


Figure 6: The Whisker step function and its scheduling

function step of the Whisker VM consists of the following smaller steps—Fig. 6 provides a sequence diagram that illustrates how this results in the *preemptive scheduling* of the different processes (the Scratch VM executes processes as Green Threads [44]):

(1) *Control.* The *Scratch harness* is invoked to make its transitions, which produce inputs that are sent as messages/events to the Scratch program under test.

(2) *Transit.* The *Scratch VM step* function is invoked to make the actual transition to the next state of the Scratch program, for example, by handling the messages that were added to the queue previously. To ensure responsiveness of the user interface, the number of control-flow transitions that are taken within one call of step is limited to a fixed amount of time (at most 1/30 s), after which a repaint is conducted.

(3) *Observe.* The *observer* of each property is invoked. We check whether the Scratch VM is still in a state that satisfies a given property; if not, the information is propagated back to the test loop, which takes note of the violation and also decides whether to continue the state space traversal or not—resources might be left to also check other properties.

Both the Whisker VM and the wrapped Scratch VM advance several of the processes (instantiated scripts) in one step of the VM, that is, the processes are executed concurrently. The VM's scheduling strategy allows that one process can take several control-flow transitions in one step. Whenever the program state or *behavior that is observable* from the perspective of the user changes, execution of the scripts is stopped and the step function of the Scratch VM returns. This ensures that the Scratch observer scripts can take note of (observe) all relevant state transitions.

Whisker Web UI. Currently, Whisker provides its own web GUI—see Figure 7. This Web UI displays Scratch's stage, a table of tests loaded, and a test report in TAP13 format. Furthermore, it supports batch testing more than one program with the same test suite.

Whisker and Scratch 3.0 are written in JavaScript. Our formalization, in form of concurrently executed sequential processes, provides an abstraction that masks out the details of the implementation, and provides a clear mental model of Scratch programs and the Whisker framework. Together with Whisker, we ship the Whisker testing API for automated testing of Scratch programs.

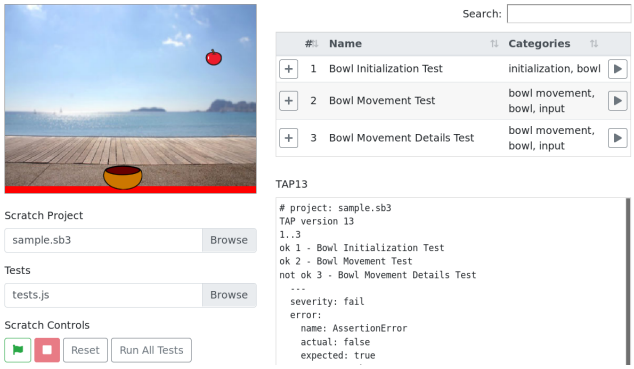


Figure 7: Screenshot of Whisker's web GUI

3.2 Dynamic Test Harness

Testing is conducted by stimulating a test subject with inputs and observing the behavior, which is then checked for conformance with the property to check. The task of stimulating the subject with inputs is conducted by the test harness. We consider two types of harnesses: (1) *static test harnesses* that stimulate the system with an upfront fixed sequence of inputs, and (2) *dynamic test harnesses* that stimulate the system with a dynamically (possibly at random) determined sequence of inputs—a repeated execution of the same dynamic test harness should eventually lead to different sequences of messages if they are produced non-deterministically (randomly).

Informed Stimulation. Generally, the sequence of inputs, and the alphabet thereof, that is produced by a dynamic test harness can be determined based on (1) the history of states of the virtual machine that it has observed already, (2) earlier instantiations of the machine and its states, (3) a persistent storage (catalog or database) as used for search-based test generation [19], or (4) by some form of static analysis (for mining or inference) of the Scratch program.

A large portion of information for input generation can be gained by looking into the state of the VM, which also reflects the messages different processes are waiting for: Given a concrete state $c = \langle p_1, \ldots, p_n \rangle \in C$, which consists of n concrete process states: Each concrete process state p_i can be in a different mode, at a different location of the control flow of the corresponding script, some of the processes can be waiting for events to happen; the set of messages that are awaited is denoted by $\text{wf}(c) = \bigcup \{m_i \mid p_i \in c \wedge m_i \in p_i(\text{pwaitfor})\}$. This information can be used to choose inputs to steer the execution to certain regions of the state space.

We leave elaborated strategies to derive (more) effective sequences of inputs as options for extending our framework. Ideally, also the distribution of messages and their payloads is taken into account for optimizations and to achieve the chosen coverage goal.

Scratch Inputs. Test harnesses stimulate and control the test subject by sending messages and triggering corresponding events in the processes. The sequence of states that a program can be observed in is determined by a sequence of messages it receives. Scratch programs, as discussed in this work, are controlled by the user using *mouse* and *keyboard*, that is, the program reacts to corresponding inputs: Possible inputs include, for example, mouse movement, mouse button presses, keyboard key presses and entering answers

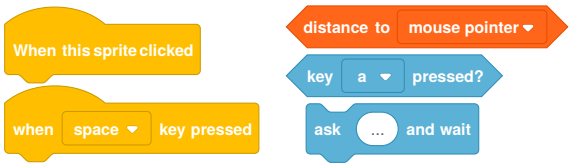


Figure 8: Some of Scratch's event and sensing blocks

to ask blocks—see Fig. 8 for examples of Scratch blocks that handle inputs. This is reflected in the following *input grammar*, which defines how a message can be produced:

$$\begin{aligned} input &= \text{KeyDown } key \mid \text{KeyUp } key \mid \text{MouseDown } pos \mid \\ &\quad \text{MouseUp } pos \mid \text{MouseMove } pos \mid \text{TextInput } text \\ key &= \text{keycode } integerliteral \\ pos &= \text{xpos } integerliteral \text{ ypos } integerliteral \\ text &= \text{txt } stringliteral \end{aligned}$$

Grammar Strengthening. While typical Scratch programs can be controlled by inputs that were produced based on above grammar, not all productions might be *effective* for a given (specific) Scratch program. To increase the effectiveness of the resulting test inputs, we add constraints that restrict the possible products of the grammar; this information is obtained by statically analyzing the syntactic representation of the Scratch programs:

$$\begin{aligned} keycode &\in [\text{<keys checked by the program>}] \\ txt &\in [\text{<strings used in the program>}] \\ xpos &\in [-240..240] \wedge ypos \in [-180..180] \end{aligned}$$

The grammar is strengthened with the following constraints: (1) only key codes are produced for which the program implements corresponding handlers, (2) only strings are produced that can match in the corresponding string comparisons in the program, and (3) the mouse coordinates are limited to be within the screen.

Stochastic Test Harness. Similar to other tools for property-based [8, 14] testing, we generate inputs at random and use a dynamic test harness for this purpose. Each time our stochastic test harness is invoked to control the execution of the program, it uses the strengthened grammar to produce an input message to send to the program by choosing a random production: We conduct a simple form of grammar-based fuzzing [18] to generate inputs. While it would be possible to assign the productions different probabilities, we assume a uniform distribution of different message types.

4 EMPIRICAL STUDY

To study the feasibility of Scratch testing and the Whisker framework, we empirically answer the following research questions:
RQ1: How frequent are *flaky tests* in Scratch programs?
RQ2: Can automated Scratch tests be used for *automated grading*?
RQ3: What *block coverage* can automated test generation achieve?
RQ4: Can property-based testing be used for *automated grading*?

An artifact that contains all data and software to reproduce our study is available online: github.com/se2p/artifact-esecfse2019.

4.1 Study Objects

We use two sets of Scratch programs for our experiments. The first set is based on two Scratch workshops with a sixth and a seventh grade class, respectively, conducted at a local school in Passau. In these workshops, the students were taught important

Table 2: Project specification in natural language

	#	Property
Init	1	Timer and score start at 30 seconds and 0 points, respectively
	2	Bowl starts at $X = 0$ / $Y = -145$
	3	Fruits have a size of 50%
Bowl	4	Bowl moves left/right when corresponding arrow key is pressed
	5	Bowl can only move horizontally with a speed of 10
Fruit Falling	6	Apples fall down
	7	Apples fall in a straight line with a speed of -5
	8	Bananas fall down
	9	Bananas fall in a straight line with a speed of -7
Fruit Spawn	10	Apples spawn again at the top of the screen after touching the bowl
	11	Apples spawn at random X position
	12	Apples spawn at $Y = 170$
	13	Bananas spawn again at the top of the screen after touching the bowl
	14	Bananas spawn at random X position
	15	Bananas spawn at $Y = 170$
	16	Only one apple must fall down at a time
	17	Only one banana must fall down at a time
	18	Banana must wait for a second before falling down in the beginning
	19	Banana must wait for a second before falling down after displaying "-8"
Fruit Interaction	20	Apple gives 5 points when it touches the bowl
	21	Game over when the apple touches the ground
	22	Apple displays "Game Over!" message when it touches the ground
	23	Banana gives 8 points when it touches the bowl
	24	Banana subtracts 8 points when it touches the ground
	25	Banana displays "-8" message when it touches the ground
Timer	26	Timer is decremented by one once a second
	27	Game stops after 30 seconds elapsed
	28	Bowl must display "End!" after 30 seconds elapsed

Scratch concepts. At the end of the workshops, all students were given a textual description of a game that they had to independently implement. In this game (embedded in the Whisker GUI in Fig. 7), the player controls a bowl with the left/right arrow keys on the keyboard, and has to catch fruit falling down from the top of the stage. The duration of the game is determined by a timer, and different numbers of points are awarded for catching two different types of fruit (apples and bananas). The game is lost when dropping an apple. The solution consists of three sprites and two variables (for time and points) and one script for each sprite as well as the stage. After the workshop, all student solutions were manually graded using a point system by the school teacher who defined the task, resulting in a total of 37 graded implementations.

The second set consists of 24 of Code Club's [17] Scratch programs, listed in Table 3. Code Club offers these projects along with detailed instructions for programming beginners, and thus represents the target domain of Scratch programs we have in mind. The projects cover a range of different modes of interaction. We use the sample solutions to these projects provided by Code Club.

4.2 Experiment Procedure

RQ1. To answer RQ1, we used the textual description provided by the school teacher, and implemented tests in Whisker. To do so, we identified 28 properties of the specification listed in Table 2, and created one test for each. We executed these tests on all 37 student implementations 10 times [37] each to accommodate for potential randomness To determine whether flakiness is a problem when testing Scratch programs or not, we check for inconsistencies in the repetitions of the test executions. A test is considered flaky if

Table 3: Counts and input methods for Code Club projects

Project	# Sprites	# Scripts	# Blocks	Project	# Sprites	# Scripts	# Blocks
Archery	2	3	21	Lost In Space	6	4	24
Balloons	2	4	26	Memory	6	11	58
Beat The Goalie	3	6	30	Moonhack Scratch 2017	3	4	27
Boat Race	3	3	27	Paint Box	9	14	42
Brain Game	4	19	76	Poetry Generator	4	2	18
Catch The Dots	5	11	82	Rock Band	5	6	18
Chat Bot	2	2	26	Snowball Fight	4	7	37
Clone Wars	7	17	76	Space Junk	8	13	68
Create Y.O. World	13	26	165	Sprint	5	9	78
Dodgeball	5	10	78	Synchronised Swimming	2	7	23
Ghostbusters	5	11	58	Tech Toys	9	5	25
Green Your City	7	8	52	Username Generator	3	2	5

there exist both, pass and fail test outcomes of the test execution, within the 10 repetitions. We conduct the experiment twice: With and without a seeded random number generator.

RQ2. To answer RQ2, we use the test executions from RQ1 and correlate the average number of failing tests with the points determined manually by the teacher. We provide a seed for the random number generator of the Scratch VM such that flakyness of test executions is limited.

RQ3. To check if random testing of Scratch programs is feasible, we consider the Code Club dataset such that results generalize better. On each of these projects we executed Whisker for 600 s, each with random input generation, with 10 repetitions. We measured how many of the blocks of the Scratch programs were covered (i.e., block coverage) throughout the execution and at the end.

RQ4. To determine if property-based testing is feasible and can be used in a Scratch teaching context, we ran Whisker with automated input generation 300 s on each of the 37 student implementations of the fruit catching game. We chose to reset the program every 10s during random testing since a correct implementation stops once the game is over. We chose to emit random inputs with a duration between 50 ms and 100 ms every 150 ms.

4.3 Threats to Validity

Internal Validity. One threat to validity arises from our use of the manual scores as ground truth to assess the quality of test results. The manual scores were assigned by the school teacher immediately after the course was held and the grading scheme was withheld for this paper to avoid bias; only the textual specification given to students was used to create test cases in our study. However, even if the grading scheme had been known, our experiment would still demonstrate that the grading task can be automated. Since many Scratch programs use randomness, test outcomes may be inconsistent. To eliminate the possibility of random chance affecting our results as much as possible, we repeated all experiments and test executions ten times, considered the average results, and explicitly evaluated flakiness. Test executions may be influenced by how Whisker invokes Scratch's step function and interleaves its own executions. While we omit details, we performed experiments to measure the overhead: On average, execution of a single step in the fruit catching game takes 1.97 ms, of which Whisker consumes only 0.12 ms, and thus it is unlikely to interfere with test executions.

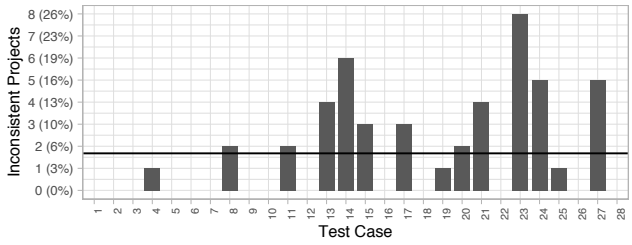


Figure 9: Inconsistencies per test, without seeding the random number generator; all except for test 4 on one project are avoided by seeding the random number generator

External Validity. We used 25 different Scratch programs in our experiments, and results may not generalize to other Scratch programs. However, we aimed to choose programs that are representative of the intended target usage in the field of education.

Construct Validity. We used a version of statement coverage to measure whether Whisker can exercise Scratch programs sufficiently. While our results to RQ4 suggest that this is an appropriate choice, it might be that more rigorous coverage criteria would reveal more limitations of the automated test generation approach.

4.4 RQ1: How frequent are flaky tests in Scratch programs?

An important issue in automated testing is the problem of flaky tests, that is, test cases that can yield different results for repeated test executions on the same version of a program. To evaluate how frequent this problem is in context of Scratch programs, we consider the inconsistency of test results for the 10 repetitions of the test execution. To isolate the role of controlling the random number generator, we run the tests once with a seed, and once without seeding. We use the term *inconsistency* to denote if a test case produces a flaky result within these 10 executions on the same version of the program.

Figure 9 shows the number of such inconsistencies for each of the individual tests, with an unseeded random number generator. For this experiment, overall only 4.15 % of all test-project pairs showed flakiness. In total 14 out of 28 tests showed some degree of flakiness at some point, which is similar to the degree of flakiness observed in general software engineering [29], but it nevertheless a cause for concern.

We manually investigated all cases of flakiness. In most cases, the root cause for the flakiness is the underlying non-deterministic nature of the programs, mainly caused by the reliance on some form of random numbers as is common in game-like programs. We therefore used Whisker's functionality to seed the random number generator, which removed all flakiness except for one test on one project. An analysis revealed that this flakyness was due to a bug in the student submission: The current time was checked to decide if a control-flow transition should be conducted or not instead of relying on a less sensitive timer variable.

Summary (*RQ 1*) In our experiments, 4.15 % of the combinations of test/project showed some flakiness if the random number generator was not seeded. Seeding nearly eliminated flakyness for all analyzed properties and projects.

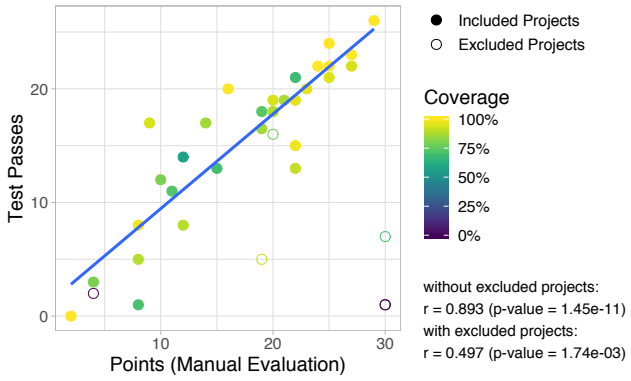


Figure 10: Automatic vs. manually assigned scores

4.5 RQ2: Can automated Scratch tests be used for automated grading?

The Scratch programs were graded by the school teacher using a point-based grading scheme in the range of [0, 30], where a higher score represents a better solution. To make automated grading possible, we would expect a larger number of tests to pass on better solutions. Figure 10 shows the correlation between manually assigned point score and the number of tests passed. Overall, the Pearson's correlation between the points and number of passing tests is strong with a value of 0.497 (p-value < 0.002), which confirms that better solutions lead to more passing tests, and the number of passing tests is moderately correlated to manually assigned scores.

However, there are outliers in the plot. A closer investigation revealed an issue in the textual specification created by the school teacher: Although intended, it is not explicitly specified that programs have to be started by clicking Scratch's green flag. While the majority of students followed the pattern of using the green flag, four students start their game after pressing a key (space bar, up-key, a-key) instead. Since our tests also made the assumption that games are started with the green flag, most tests failed on these four projects. Arguably, this is an issue in the specification, not the tests. Two other projects revealed a second source of discrepancy: One project had omitted sprite initializations, such that sprites need to be manually repositioned after each execution—a common error in Scratch programs. A second project had placed the initialization code at the end of the game (when showing a "Game Over" message), such that the initialization is incorrect for the initial execution, but correct for successive executions. In both cases many tests failed as a result, while the teacher had apparently decided to be generous. It is debatable whether in these two cases the tests or the manual grading are correct. If we exclude these six cases and only consider projects that are not affected by these specification issues (i.e., only the filled dots in Fig. 10), the correlation is strong at 0.893 (p-value < 0.001). This demonstrates an essential aspect of all automated grading approaches: To be applicable, the programming task needs to be specified rather precisely. It is also important to keep in mind that programming beginners are less biased regarding possible implementations of a given requirement, which results in a larger variety of solutions that the operationalization of the specification must cover. As we will see in RQ4, property-based testing can relax this requirement somewhat.

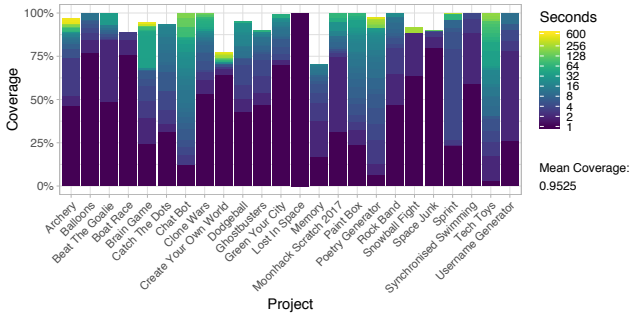


Figure 11: Block Coverage achieved by Whisker automatically on the Code Club projects.

Summary (*RQ 2*) We see a strong correlation between the manually assigned points and the number of passing tests, confirming that automated grading of Scratch programs is possible.

4.6 RQ3: How well can automated test generation cover Scratch programs?

A prerequisite for property-based testing is a test generator that can sufficiently cover the behavior of Scratch programs. To determine how well Whisker's test generation approach covers Scratch programs, we applied it to the 24 projects of the Code Club dataset. Figure 11 summarizes the block coverage achieved by Whisker using random inputs after 10 minutes; the time needed to achieve different levels of coverage is encoded in the shading of the bars. After ten minutes of run time, Whisker achieved an average coverage of 95.25 %, with the lowest coverage for a project being 71 % and the highest being 100 %.

There are two projects with clearly lower coverage than the others. "Create Your Own World" implements an adventure game with a player sprite, which starts on the left of the screen and is able to move through several rooms through a portal on the right side of the screen. This project is difficult to cover, because the sprite needs to move a long way in one direction to reach a screen transition. Due to the random nature of the inputs that Whisker selects, it takes a long time for the sprite to move longer distances. "Memory" revolves around four colored drums, for which in each round a random sequence is generated; the player then has to click the drums in that order. Clicking an incorrect drum leads to a game-over state. Random input selection is unlikely to select the correct sequence of drums, and thus coverage remains low. Both of these cases demonstrate general limitations of random testing, which are particularly relevant in the game-like nature of Scratch projects, suggesting that future work should consider more advanced test generation techniques (e.g., search-based testing [19]).

Summary (*RQ 3*) On average, Whisker achieved 95.25 % block coverage on the 24 Code Club projects.

4.7 RQ4: Property-based automated grading?

Figure 12 shows the correlation of the number of points resulting from manual grading and the number of properties determined to be violated, using property-based testing with random test inputs. The overall Pearson's correlation is 0.788 (p-value < 0.001); excluding the same projects that were discussed for RQ2 as related to the

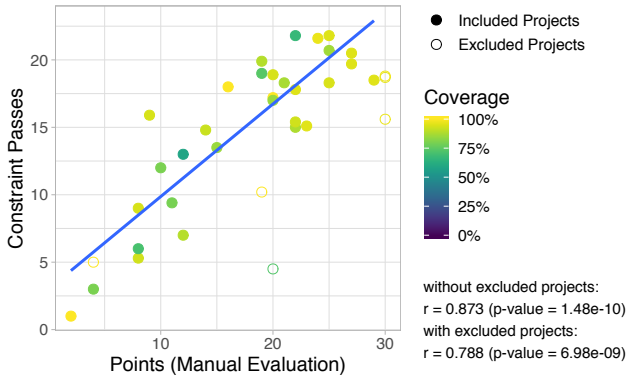


Figure 12: Automatic testing vs. manually assigned scores

unclear specification, the correlation is 0.873 (p-value < 0.001);. Interestingly, the correlation is thus higher overall than on the manually written test suite (RQ2); this is because the four projects that use other means than the green flag to start the project will receive the starting event they are waiting for from the random test generator. Consequently, automated test generation provides some additional flexibility compared to manually written tests.

Summary (*RQ 4*) The strong correlation between manual grading and property violations confirms that property-based testing of Scratch programs for grading is possible.

5 RELATED WORK

Analysis of Scratch Programs. Although there has been work on showing the prevalence [1, 41, 45, 46] and effects [21] of code smells in Scratch programs, only little work on analysis of Scratch programs has been presented. Hairball [7] is a basic static analysis tool that implements detectors for basic types of code smells (e.g., long scripts, duplicated code). It serves as basis for the web-based assessment tool Dr. Scratch [36], which measures and scores the complexity and quality of Scratch programs. These tools match patterns in the JSON-based Scratch data format and cannot judge functional correctness like testing could. The concept of testing Scratch programs was discussed in the context of the ITCH (Individual Testing of Computer Homework for Scratch Assignments) [23] tool, which converts Scratch programs to Python code and then applies tests on the Python code. However, it is limited to checking inputs and outputs in terms of the "ask" and "say" blocks, and thus only supports a small subset of Scratch functionality, whereas our approach covers the entire functionality of Scratch.

Automated Grading. Although we envision that our testing approach will enable many different types of dynamic analyses, an immediate application lies in automated grading. The idea of automated grading has been discussed since the early beginnings of programming education [15], and grading systems are available for many different programming languages and types of applications [22]. A central component of most automated grading systems is the use of functional tests (e.g., input/output pairs, or unit tests) to derive a score. Property-based testing as an alternative has been explored on basic Java programming exercises [11], providing further evidence for the feasibility of our proposed application scenario, and there is evidence that automated test generation can lead to increased grading accuracy [2] over manually written tests. While we compared the test scores directly with the teacher grades in our study, grading systems generally tend to combine functional correctness scores with other metrics on code complexity and quality, and in future work we consider combining Whisker tests with other quality metrics for grading. Many test-based automated grading systems focus on API-level tests with predefined interfaces, whereas GUI-level tools have to face the challenge of providing robust test harnesses for flexible GUIs. In-line with our approach, it has previously been shown that specification-driven automated testing can help to overcome such issues [13, 43].

Automated Testing of Event-Driven Programs. Our framework uses and generates GUI-centered system tests. For our initial implementation of Whisker, we used a basic random testing approach. There are other general GUI-testing approaches [38] that could be used to improve the Whisker test generator, for example, by explicitly modelling the event-flow of the programs [34], using search-based optimizations [19] or symbolic execution [35] to guide test generation towards covering all code. While the current random event generator already provided promising results, we will consider improving the Whisker test generator in future work.

6 CONCLUSIONS

Block-based programming environments like Scratch are highly popular and successful at engaging young learners, but are lacking means for automatically analyzing programs. In this paper, we have introduced a theoretical framework for testing Scratch programs, and we presented Whisker, a concrete instantiation that can automatically test Scratch programs. Our experiments on student and teacher-written Scratch programs demonstrate that Whisker tests can be used to automatically grade Scratch programs.

Specifying properties and creating tests are major challenges in automated testing in general. In the context of Scratch testing, the programmers (i.e., learners) are unlikely to write their own tests, and educators may not be best qualified for this task either. We are investigating ways to provide more convenient ways to represent fuzzy properties appropriate for graphical, game-like programs; for example, by expressing them with Scratch blocks, or by inferring them from executions on a model solution.

While our experiments confirmed that Whisker can already achieve high degrees of block coverage with its random testing approach, new notions of coverage and more systematic testing approaches (e.g., search-based testing) may further improve the effectiveness of Scratch testing.

Although we used Scratch testing for automated grading in this paper, we envision many further applications of automated tests. The educational nature of Scratch interactions would benefit from the dynamic analysis enabled by a testing framework like Whisker. Applications range from supporting learners with fault localization to producing hints and repairs automatically. To support this area of research, the latest version of Whisker is available as open source at github.com/se2p/whisker-main.

ACKNOWLEDGEMENTS

This work is supported by EPSRC project EP/N023978/2 and DFG project FR 2955/3-1 "TENDER-BLOCK: Testing, Debugging, and Repairing Blocks-based Programs".

REFERENCES

[1] E. Aivaloglou and F. Hermans. 2016. How kids code and how we know: An exploratory study on the Scratch repository. In *Proc. ICER*. ACM, 53–61.

[2] G. Anielak, G. Jakacki, and S. Lasota. 2015. Incremental test case generation using bounded model checking: an application to automatic rating. *STTT* 17, 3 (2015), 339–349.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (1995), 124–142.

[4] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. A. Turbak. 2017. Learnable programming: blocks and beyond. *Commun. ACM* 60, 6 (2017), 72–80.

[5] J. Bengtsson and W. Yi. 2003. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets (LNCS 3098)*. Springer, 87–124.

[6] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness validation and stepwise testification across software verifiers. In *Proc. ESEC/FSE*. ACM, 721–733.

[7] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin. 2013. Hairball: Lint-inspired Static Analysis of Scratch Projects. In *Proc. SIGCSE (SIGCSE '13)*. ACM, New York, NY, USA, 215–220.

[8] K. Claessen and J. Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (2011), 53–64.

[9] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. 2003. *Shared memory vs message passing*. Technical Report.

[10] E. W. Dijkstra. 1970. Notes on Structured Programming.

[11] C. Benac Earle, L.-A. Fredlund, and J. Hughes. 2016. Automatic Grading of Programming Exercises using Property-Based Testing. In *Proc. ITiCSE*. ACM, 47–52.

[12] LEGO Education. [n.d.]. LEGO Education. https://education.lego.com.

[13] M.Y. Feng and A. McAllister. 2006. A tool for automated GUI program grading. In *Proc. FIE*. IEEE, 7–12.

[14] J. -C. Fernandez, L. Mounier, and C. Pachon. 2003. Property Oriented Test Case Generation. In *Proc. FATES (LNCS 2931)*. Springer, 147–163.

[15] G. E. Forsythe and N. Wirth. 1965. Automatic grading programs. *Commun. ACM* 8, 5 (1965), 275–278.

[16] Micro:bit Educational Foundation. [n.d.]. Micro:bit. https://microbit.org/.

[17] Raspberry Pi Foundation. [n.d.]. Code Club. https://codeclubprojects.org/.

[18] P. Godefroid, A. Kiezun, and M. Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proc. PLDI*. ACM, 206–215.

[19] F. Gross, G. Fraser, and A. Zeller. 2012. Search-based System Testing: High Coverage, No False Alarms. In *Proc. ISSTA (ISSTA 2012)*. ACM, New York, NY, USA, 67–77.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. 2002. Temporal-Safety Proofs for Systems Code. In *Proc. CAV (LNCS 2404)*. Springer, 526–538.

[21] F. Hermans and E. Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *Proc. ICPC*. IEEE, 1–10.

[22] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Koli Calling*. ACM, 86–93.

[23] D. E. Johnson. 2016. ITCH: Individual Testing of Computer Homework for Scratch Assignments. In *Proc. SIGCSE*. ACM, New York, NY, USA, 223–227.

[24] S. Katz. 2006. Aspect Categories and Classes of Temporal Properties. (2006), 106–134.

[25] S. Konur. 2013. A survey on temporal logics for specifying and verifying real-time systems. *Frontiers Comput. Sci.* 7, 3 (2013), 370–403.

[26] O. Kupferman, N. Piterman, and M. Y. Vardi. 2009. From liveness to promptness. *Formal Methods in System Design* 34, 2 (2009), 83–103.

[27] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. 2006. SMT Techniques for Fast Predicate Abstraction. In *Proc. CAV (LNCS 4144)*. Springer, 424–437.

[28] A. Lester, M.Schwern, and A. Armstrong. [n.d.]. The Test Anything Protocol v13. https://testanything.org/tap-version-13-specification.html.

[29] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 643–653.

[30] O. Maler, D. Nickovic, and A. Pnueli. 2007. On Synthesizing Controllers from Bounded-Response Properties. In *Proc. CAV (LNCS 4590)*. Springer, 95–107.

[31] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. 2010. The Scratch Programming Language and Environment. *TOCE* 10, 4 (2010), 16:1–16:15.

[32] T. Matlock, M. Ramscar, and L. Boroditsky. 2005. On the Experiential Link Between Spatial and Temporal Language. *Cognitive Science* 29, 4 (2005), 655–664.

[33] A. W. Mazurkiewicz. 1986. Trace Theory. In *Advances in Petri Nets (LNCS 255)*. Springer, 279–324.

[34] A. M. Memon. 2007. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.* 17, 3 (2007), 137–157.

[35] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R.Mahmood. 2012. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[36] J. Moreno-León and G. Robles. 2015. Dr. Scratch: A Web Tool to Automatically Evaluate Scratch Projects *(Proc. WiPSCE)*. ACM, New York, NY, USA, 132–133.

[37] F. Palomba and A. Zaidman. 2017. Does Refactoring of Test Smells Induce Fixing Flaky Tests?. In *Proc. ICSME*. IEEE Computer Society, 1–12.

[38] M. Pezzè, P. Rondena, and D. Zuddas. 2018. Automatic GUI testing of desktop applications: an empirical assessment of the state of the art. In *ISSTA/ECOOP Workshops*. ACM, 54–62.

[39] A. Pnueli and A. Zaks. 2008. On the Merits of Temporal Testers. In *25 Years of Model Checking (LNCS 5000)*. Springer, 172–195.

[40] Y. S. Ramakrishna, P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kutty. 1993. Really visual temporal reasoning. In *RTSS*. IEEE Computer Society, 262–273.

[41] Gregorio Robles, Jesús Moreno-León, Efthimia Aivaloglou, and Felienne Hermans. 2017. Software clones in scratch projects: On the presence of copy-and-paste in computational thinking learning. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE, 1–7.

[42] A. W. Roscoe. 2001. Compiling shared variable programs into CSP. In *Proc. PROGRESS*, Vol. 2001.

[43] Y. Sun and E. L. Jones. 2004. Specification-driven automated testing of GUI-based Java programs. In *Proc. 42nd annual Southeast regional conference*. ACM, 140–145.

[44] M. Sung, S. Kim, S. Park, N. Chang, and H. Shin. 2002. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread. *Inf. Process. Lett.* 84, 4 (2002), 221–225.

[45] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How do Scratch programmers name variables and procedures?. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 51–60.

[46] P. Techapalokul and E. Tilevich. 2017. Understanding recurring quality problems and their impact on code sharing in block-based software. In *Proc. VL/HCC*. IEEE, 43–51.