Verified from Scratch: Program Analysis for Learners' Programs

Andreas Stahlbauer
University of Passau
Germany

Christoph Frädrich
University of Passau
Germany

Gordon Fraser
University of Passau
Germany

ABSTRACT

Block-based programming languages like Scratch support learners by providing high-level constructs that hide details and by preventing syntactically incorrect programs. Questions nevertheless frequently arise: Is this program satisfying the given task? Why is my program not working? To support learners and educators, automated program analysis is needed for answering such questions. While adapting existing analyses to process blocks instead of textual statements is straightforward, the domain of programs controlled by block-based languages like Scratch is very different from traditional programs: In Scratch multiple actors, represented as highly concurrent programs, interact on a graphical stage, controlled by user inputs, and while the block-based program statements look playful, they hide complex mathematical operations that determine visual aspects and movement. Analyzing such programs is further hampered by the absence of clearly defined semantics, often resulting from ad-hoc decisions made by the implementers of the programming environment. To enable program analysis, we define the semantics of Scratch using an intermediate language. Based on this intermediate language, we implement the Bastet program analysis framework for Scratch programs, using concepts from abstract interpretation and software model checking. Like Scratch, Bastet is based on Web technologies, written in TypeScript, and can be executed using NodeJS or even directly in a browser. Evaluation on 279 programs written by children suggests that Bastet offers a practical solution for analysis of Scratch programs, thus enabling applications such as automated hint generation, automated evaluation of learner progress, or automated grading.

CCS CONCEPTS

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

KEYWORDS

Software Model Checking, Scratch, Education

ACM Reference Format:
Andreas Stahlbauer, Christoph Frädrich, and Gordon Fraser. 2020. Verified from Scratch: Program Analysis for Learners' Programs. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416554

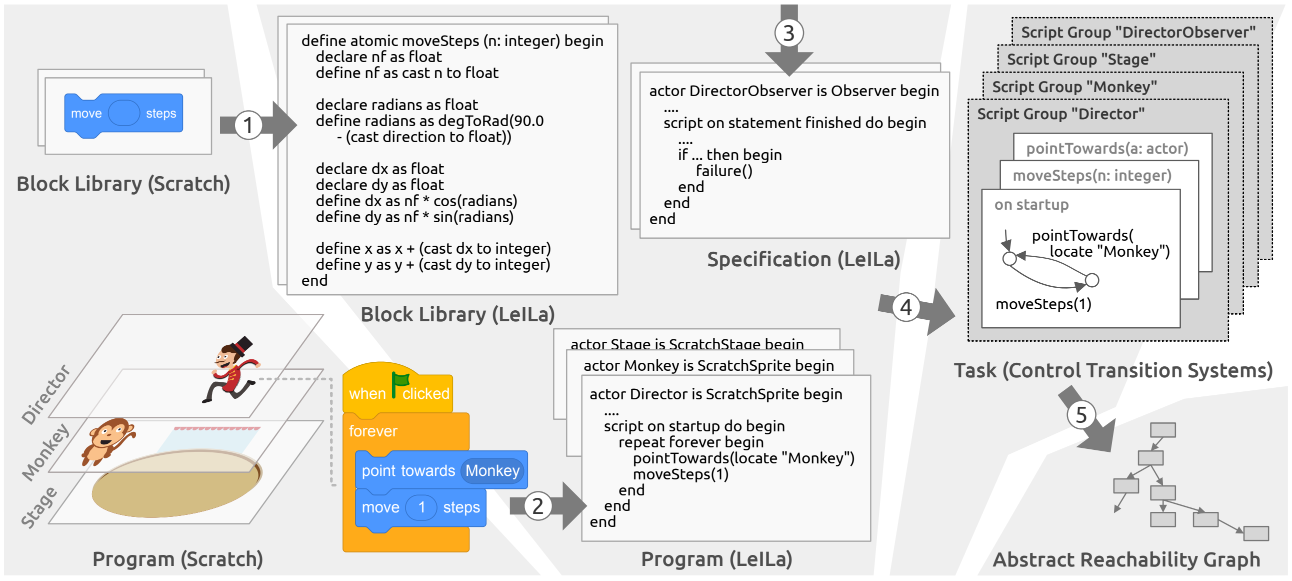
1 INTRODUCTION

Block-based programming is increasingly popular for introducing learners to programming [75] as well as for simplifying the challenges of programming for domain experts with limited programming skills [74]. In block-based programming languages, program statements are represented visually as blocks which users drag and drop from a toolbox of available commands to visually arrange programs. This paradigm is implemented by many popular programming environments such as, for example, Alice [23], Snap [35], PencilCode [9], or 28 other drag-and-drop programming environments surveyed recently [28]. Block-based programming increasingly has applications outside of education, for example, in domains such as robotics [72, 74] or internet-of-things [58]. The recent success of the block-based approach, however, can be attributed to a large extent to the popularity of Scratch [46]. At the time of this writing, the popular Scratch programming environment had more than 54 million registered users who have publicly shared more than 53 million programs[1] and written more programs not shared.

Although block-based programming languages like Scratch are successful at making programming easier, they do not simplify program analysis: Even though at first glimpse Scratch programs look small and easy, they tend to consist of many highly concurrent scripts, which are composed of blocks that hide complex mathematical functions that control the program's visual representation. To lower the entry barriers and to foster widespread adoption, most block-based programming environments are designed as Web applications, and the interpreters for these blocks tend to be built into the Web applications, and are often implemented in an ad-hoc way without clearly defined semantics. Since Scratch programs do not have an intermediate language that they are translated to, statically analyzing a Scratch program with its full semantics would therefore require to conduct an analysis of the Scratch VM together with the Scratch program to be analyzed loaded into it. Although challenging, the demand for automated program analysis has never been higher, for example for automated hint generation [57], supporting program improvement [71], evaluating learner progress [52], or automated grading [73].

In this paper we introduce Bastet[2], a general program analysis framework for Scratch 3. Figure 1 illustrates the overall workflow of Bastet using a primary school programming task, in which the aim is to make the circus director move to the monkey sprite. To analyze Scratch programs, Bastet uses a textual intermediate language (LeILa, described in Section 3). Bastet provides a library describing the functionality of all Scratch-blocks in LeILa ①. As an example, the LeILa implementation of the $\boxed{\text{move (n) steps}}$ block is shown. The Scratch program itself is translated to LeILa ②, and the same intermediate language is used for formal specification ③. The program is then interpreted using the semantics of

[1] https://scratch.mit.edu/statistics/ [2] Bastet is the ancient Egyptian goddess of cats; the Scratch logo is a cat.



```
define atomic moveSteps (n: integer) begin
    declare nf as float
    define nf as cast n to float

    declare radians as float
    define radians as degToRad(90.0
        - (cast direction to float))

    declare dx as float
    declare dy as float
    define dx as nf * cos(radians)
    define dy as nf * sin(radians)

    define x as x + (cast dx to integer)
    define y as y + (cast dy to integer)
end
```

```
actor DirectorObserver is Observer begin
    ....
    script on statement finished do begin
        ....
        if ... then begin
            failure()
        end
    end
end
```

```
actor Stage is ScratchStage begin
actor Monkey is ScratchSprite begin
actor Director is ScratchSprite begin
    ....
    script on startup do begin
        repeat forever begin
            pointTowards(locate "Monkey")
            moveSteps(1)
        end
    end
end
```

Figure 1: Bastet Overview: Scratch programs are translated to the LeILa intermediate language, which Bastet analyzes using concepts from abstract interpretation and software model checking with respect to a LeILa specification.

LeILa ④ (Section 3.3), thus enabling the application of various program analysis configurations ⑤ using concepts from abstract interpretation and software model checking (Section 4). Like Scratch, Bastet itself is based on Web technologies and written in TypeScript, and can be executed in NodeJS or even directly in a Web browser. In detail, the contributions of this paper are as follows:

(1) We define the LeILa intermediate language for Scratch programs and their formal requirements specification (Section 3). We discuss the central parts of its semantics, as well as approximations to handle the complexity of Scratch.
(2) We introduce the Bastet program analysis framework (Section 4) and release it as an open source project.
(3) We empirically demonstrate that Bastet is practically applicable to Scratch programs written by children (Section 5).

A pilot study of 279 children's implementations of four educational programs shows that the translation of Scratch programs into LeILa and the interpretation with Bastet maintains the program semantics. This demonstrates that Bastet provides the foundations for many different types of program analysis on Scratch. By releasing Bastet as open source, we hope to inspire many future automated analyses and support techniques, thus helping learners as well as their teachers or automated tutoring systems.

2 BACKGROUND

Before we present our framework, we introduce relevant background based on existing work [13, 25, 65]. We use upper case letters $A, B, \dots, Z$ or letters with a hat $\widehat{a}, \widehat{B}$ for sets, lower case letters $a, b, \dots, z$ for set elements, lists and sequence variables are indicated by adding a bar $\overline{a}, \overline{A}$, the set of all words over an alphabet A is denoted by A^*. Sets are enclosed in curly brackets $\{a_1, \dots\}$, lists in angle brackets $\langle a_1, \dots \rangle$, and tuples in round brackets $(a_1, \dots, a_n)$.

Scratch. Scratch programs [47] are developed visually and block-based in the corresponding development environment [47]. As in other block-oriented languages, the grammar of Scratch is defined implicitly by allowing or preventing (Scratch) blocks to be combined. A Scratch program *App* is composed of a set of visual entities consisting of the sprites and the stage—the program in Fig. 1 is composed of the sprites Director and Monkey, and the stage Stage. The visual entities are rendered on a canvas; each entity is rendered on a separate layer [65]. One visual entity is composed of a set of scripts, a set of custom blocks, and sound and image resources. Each script handles an event, and is composed of a set of blocks to execute. Scratch programs are controlled by events, typically triggered by mouse or keyboard inputs. A Scratch program is executed in the Scratch virtual machine.

Concrete States and Behaviours. The semantics $[\![App]\!]$ of a Scratch program *App* is defined [65] by the set of concrete execution traces it exhibits, that is, $[\![App]\!] \subseteq C^\infty$. The set of all possible *concrete execution traces* $C^\infty = C^\omega \cup C^*$ consists of the set of finite traces C^* and the set of infinite traces C^ω. One *program trace* $\overline{c} = \langle c_0, \dots \rangle \in C^\infty$ is a sequence of concrete states and always starts in the *initial concrete state* c_0 of a program. A *concrete state* $c \in C$ (*configuration*) of a Scratch program [65] is a list $c = \langle p_1, \dots, p_n \rangle$ of concrete process states $p_i : X \to V$. Each *concrete process state* p_i is a mapping from a data location $x \in X$ (a variable) to a data value $v \in V$. We call an instance of a Scratch script a *process*. A concrete process state describes the state of a process $\lambda \in \Lambda$ at one point in time; the set of all processes is denoted by Λ. Processes are organized into *process groups* that correspond to visual entities of a Scratch program—we also use the term *actor* for such an entity.

The set of data values is typed, that is, it is the union $V = V_{int} \cup V_{float} \cup V_{string} \cup V_{list}$ of data values of different types, a list value is a sequence $\overline{v} \in V_{list}$ of data values. A set of special data locations $X_{ctrl} = \{\text{pid}, \text{pgroup}, \text{pc}, \text{pstate}, \text{pwaitfor}, \text{ptime}\} \subset X$ stores information about the computation (control) state of the processes and the environment: pid is the process identifier, pgroup identifies the process group, pc is the program counter which denotes the position in a script to execute, pstate $\in$ {Wait, Running, Yield, Done} is the computation state of the process, pwaitfor is the set of processes the process waits for, ptime denotes the time that has elapsed since the application has been started. Other typical elements in the set X are variables that describe the sprite attributes such as position, size, and the orientation of a sprite—{x, y, size, direction} $\subset X$.

3 INTERMEDIATE LANGUAGE

Analyzing Scratch programs *statically* is hard since it would require to analyze the full Scratch VM along with the Scratch program, because the VM defines the functionality of blocks: the semantics of Scratch are implemented in the Scratch VM[3]. The semantics of Scratch blocks is defined *informally* on the Scratch wiki[4]—a workaround[5], which defines how to mimic a block‘s semantics without using that block, is given for each of the blocks.

We reverse-engineered the semantics of Scratch and its blocks and describe it based on our own intermediate language LeILa (LEarners Intermediate LAnguage), similar to the use of Promela in the Spin model checker for C programs [40]. We replicate the functionality of the blocks available in Scratch in a *Scratch block library* written in LeILa. Only control structures, data types, expressions, and statements that are relevant to mimic the behavior of Scratch in this library are part of the LeILa language definition.

3.1 Language Features

LeILa is designed to aid in program analysis, and also to be easily readable for people that are familiar with block-based programming languages like Scratch—also, because LeILa is intended to be used as a specification language by teachers. LeILa allows for inheritance and event-driven programming, and has distinct keywords for program analysis and verification. Data is exchanged either via message passing or via global memory access.

Syntax. The syntax of LeILa is oriented on languages [43] that are used to introduce people to programming early, for example, Pascal [76, 77], Grail [48], or Logo [29]—and influenced the design of Scratch. Evidence shows [67] that these languages use syntactic elements that are often more intuitive for novices, compared to those in widely used general purpose languages. Figure 2 shows central parts of LeILa's grammar. The full ANTLR [54] grammar is shipped with our framework. An example for a program written in LeILa can be found in Fig. 1.

Actors and Roles. We use the term *actor*—corresponding to the keyword **actor**—to denote an entity that is visible to the user or the environment—also for interaction using inputs or by passing messages—that provides functionality to act in a particular role. LeILa supports inheritance by providing the keyword **is** and uses the keyword **role** to define abstract actors, that is, collections of pre-defined methods and attributes (data) that can be instantiated in an actor. We consider this notion of actors and roles to be closer to the intuition that is promoted by Scratch. Examples for typical roles to realize Scratch programs are Sprite and Stage.

```
⟨program⟩ ::= program ⟨id⟩ ⟨group⟩*
⟨group⟩ ::= ( actor | role ) ⟨id⟩ [is ⟨id⟩ ( , ⟨id⟩ )* ] begin ⟨comps⟩ end
⟨comps⟩ ::= ⟨ressource⟩* ⟨attribute⟩* ⟨method⟩* ⟨script⟩*
⟨ressource⟩ ::= ( image | sound ) ⟨id⟩ ⟨uri⟩
⟨attribute⟩ ::= declare ⟨id⟩ as ⟨type⟩
⟨type⟩ ::= integer | float | boolean | string | list of ⟨type⟩ | actor
⟨script⟩ ::= script [⟨id⟩] on ⟨event⟩ do [restart] ⟨stmts⟩
⟨method⟩ ::= define [atomic] ⟨id⟩ ⟨params⟩ ⟨stmts⟩
        [returns ⟨id⟩ : ⟨type⟩]
    |   extern ⟨id⟩ ⟨params⟩ ⟨stmts⟩ [returns ⟨type⟩]
⟨params⟩ ::= ( ( ⟨param⟩ ( , ⟨param⟩ )* | ϵ ) )
⟨param⟩ ::= ⟨id⟩ : ⟨type⟩
⟨stmts⟩ ::= begin ⟨stmt⟩* end
⟨event⟩ ::= bootstrap
    |   startup
    |   started as clone
    |   message ⟨string⟩ [in ⟨string⟩]
    |   ⟨specEvent⟩
⟨specEvent⟩ ::= bootstrap finished | statement finished
```

Figure 2: A fraction of the LeILa grammar. The full grammar is defined based on ANTLR.

Each Scratch sprite, each clone of a sprite (corresponding to the fork of a process), and the stage correspond to one actor; another actor is added for communicating with the environment (mouse and keyboard inputs). On the technical level, an actor is formed by the list of processes that are executed concurrently—dual to a process group in related work [65]. Note that our notion of actor is slightly different from the notion used to describe actor models [2, 45]: In that work, an actor is not composed of several processes, that is, an actor corresponds to a single process only.

Verification Additions. We added particular language features to LeILa that aid in analysis and verification, and allow for realizing different approximations of a programs behavior—see also Sect. 3.5. The keyword **assume** can be used to specify invariants that can be assumed to always hold at particular points in a program. In particular, this is relevant for restricting possible values of variables that have been initialized non-deterministically. Other features help to provide the formal specification of desired behaviors of a program as LeILa code (**on statement finished**, **on bootstrap finished**)—see Sect. 3.4, and help to determine which code fragments to consider as one atomic (keyword **atomic**) program operation (which must not be preempted by the scheduler; the specification must not be checked in-between; all non-control-flow blocks of Scratch are modeled as atomic methods).

3.2 Control Transition System

After translating a Scratch project to LeILa, we use its abstract syntax tree (AST) and translate it into collections of control transition systems—the result is a list of actor definitions with a control flow graph for each script and each method the actor defines. This is the foundation to define the *operational semantics* of LeILa programs—and their Scratch counterparts. The control flow semantics of LeILa are implemented in the process of translating LeILa programs into lists of actor definitions with their corresponding control transition relations.

[3] github.com/LLK/scratch-vm
[4] en.scratch-wiki.info
[5] en.scratch-wiki.info/wiki/List_of_Block_Workarounds

LeILa Program. We formally define a LeILa program as a list $\overline{a} \in \mathcal{A}^*$ of actor definitions. One *actor definition* $a = (\overline{s}, \widehat{m}, h) \in \overline{a}$ is a tuple consisting of a list $\overline{s}$ of scripts, a set $\widehat{m}$ of method definitions, and a concern h. Each actor contributes to a concern $h \in H$, where H denotes the set of all concerns. For this work, we restrict the set of concerns to $H = \{\mathsf{Prog}, \mathsf{Spec}, \mathsf{Env}\}$, that is, it can be either the *program concern* Prog, the *specification concern* Spec, or the *environment concern* Env. The analysis procedure treats actors that are instantiated from actor definitions with the specification concern in a special way—see Sec. 3.4.

Methods and Scripts. Both, methods and scripts, are defined based on transition systems. A *control transition system* $\gamma = (L, l_0, L_x, G)$ consists of a set of *control locations* L, an *entry location* $l_0 \in L$, a set of exit locations $L_x \subseteq L$, and a set of *control transitions* $G \subseteq L \times Op \times L$. One *control transition* $(l_1, op, l_2) \in G$ has a predecessor location l_1, a program operation to execute op, and a successor location l_2, that is, location l_2 is *syntactically reachable* via location l_1 after conducting program operation $op \in Op$.

A *method* $m = (id, \overline{\rho}, rt, \gamma) \in \widehat{m}$ is defined by a tuple consisting of an identifier id, a list of parameters $\overline{\rho} \in X^*$, a result variable $rt \in X$, and a transition relation γ. A *script* $s = (w, r, \gamma) \in \overline{s}$ is defined by a tuple consisting of an event $w \in \mathbb{S} \times \mathbb{S} \times X^*$, a flag $r \in \mathbb{B}$ (keyword **restart**), which indicates whether the execution of the script should be re-started from its transition relations entry location in case the event is triggered, and the transition relation γ. The event $w = (\mu, \kappa, \overline{x})$ consists of a message identifier string μ, a string κ that specifies the message channel, and a list of arguments $\overline{x}$ that is passed to the script if the event is triggered—the set $\mathbb{S}$ denotes all possible strings expressible in LeILa.

We assume that the inheritance relation of actor definitions is dissolved upfront in a preprocessing step—for example, all methods of the ancestor actors (or roles) become methods of the given actor definition itself. Methods are used to define the behavior of the blocks that can be composed visually to a Scratch program. A script can invoke all methods defined for the actor.

Program Operations. The *set of atomic program operations Op* consists of operations of various types, which can manipulate or check the set of data locations X (variables): For example, *assume operations*—such as **assume** a > b, constructed from **if** a > b **then**—are guarding statements [27] and express conditions under that the successor location is reachable, *assign operations*—such as **define** x **as** 42—assign new values to data locations. The *epsilon operation* $\epsilon \in Op$ is a special operation that does neither affect the state space nor the behavior of the program. The *termination operation* **halt** $\in Op$ signals the termination of the program. The set of *typed data locations* X is the union of the set of integer data locations X_{int}, float locations X_{float}, string locations X_{string}, and list locations X_{list}.

3.3 Concrete Semantics

So far, we have described the formalization of a LeILa program as a list of actor definitions $\overline{a} \in \mathcal{A}^*$ and their scripts' and methods' control transition relations, now, we are interested in its semantic denotation $[\![\overline{a}]\!] \subseteq C^\infty$: Which execution traces are *feasible* for a given program? The operations on the control transitions between the control locations give rise to the actual behaviors and states of a LeILa program—the operational semantics [37, 50, 55]. In the following, we describe an important subset of LeILa's semantics; a document with the full formal semantics is left for future work.

Initialization. The execution of a LeILa program $\overline{a} \in \mathcal{A}^*$ is bootstrapped by a special *bootstrapping actor*, which is instantiated from the actor definition $a_{boot} \in \overline{a}$ and orchestrates the initialization process. The bootstrapping actor is the first active actor in the system. It conducts the following three steps: The bootstrapper first triggers the **bootstrap** event using the statement **broadcast** "BOOTSTRAP" **to** "SYSTEM" **and wait**, which activates the **on bootstrap** event handler scripts. After all actors have handled this event, it activates all handlers for the event **bootstrap finished**, after which the system is considered initialized and all scripts that handle the event **startup** are activated—corresponding to Scratch's green flag event. Details on LeILa's event handling are explained later in this section.

A LeILa program is bootstrapped from the initial concrete state $c_0 = \langle p_1, \ldots, p_n \rangle$, which is a list of concrete process states. Note that one actor (see Sect. 3 for our notion of actors) is formed by a set of processes—also known [65] as a *process group.* Only the process that belongs to the bootstrapping actor is in the computation state Running, that is, supposed to make state transitions. In detail, the initial concrete state is defined as follows—the vertical bar corresponds to a set-theoretic *"such that"* assuming that the lists a and $\overline{s}_a$ are iterated from left to right:

$$\begin{aligned} c_0 = \langle\{&(\mathsf{pid}, s), (\mathsf{pgroup}, a), (\mathsf{pc}, l_0), (\mathsf{pstate}, r), (\mathsf{pwaitfor}, \langle\rangle), \\ &(\mathsf{ptime}, 0)\} \mid \gamma_s = (\cdot, l_0, \cdot, \cdot) \wedge s = (\cdot, \cdot, \gamma_s) \in \overline{s}_a \\ &\wedge a = (\overline{s}_a, \cdot, \cdot) \in \overline{a} \wedge r = \mathsf{Running}\ \mathit{if}\ a = a_{boot}\ \mathit{else}\ \mathsf{Wait}\rangle. \end{aligned}$$

The initial concrete state c_0 is the first element in all concrete execution traces of a given LeILa program $\overline{a}$. Note that variables declared by the user are added to the concrete state as soon as a corresponding variable declaration statement was executed.

Step. After we have defined the initial concrete state of a LeILa program, we define the prefix-closed [1] set of sequences of concrete states that are considered feasible for a given program. The *concrete state transition relation* $\rightarrow \subseteq C \times C$ of a given LeILa program defines the set of traces that are considered feasible. Given a concrete state c_i, a concrete transition $c_i \rightarrow c_{i+1}$ exists if the list of processes in c_i that are in the computation state pstate = Running and the control transitions $\widehat{g} \subset L \times Op \times L$ they conduct lead to the concrete state $c_{i+1} = \langle p_i', \ldots, p_n' \rangle$ by interpreting the program operations of the control transitions $\widehat{g}$. A trace $\overline{c} = \langle c_0, \ldots \rangle \in C^\infty$ is *feasible* if for all succeeding states $(c_i, c_{i+1}) \in \overline{c}$ holds that $(c_i, c_{i+1}) \in \rightarrow$. In this work, we restrict the number of processes that can concurrently conduct a state transition to one at a point in time, and realize an *interleaving concurrency.*

Given a concrete state $c \in C$ and a concrete process state $p \in P$ of a process $\lambda \in \Lambda$ to run, a program operation $op \in Op$ that leads to a successor control location $l \in L$ is always interpreted in context of the process $\lambda \in \Lambda$ and the actor a (process group) it belongs to. We therefore define the concrete successor function $csucc$: $C \times (P \times Op \times L) \rightarrow C$ such that a call $csucc(c, p, op, l)$ takes a concrete predecessor state c, a process p, to interpret the operation op in, to get to the successor location l. This call can also be written as $csucc_p(c, op, l)$. The function implements the actual semantics of given program operations.

We now provide a more formal definition of the concrete transition relation using the functions $\overline{step} : C \times P^* \rightarrow C$ and $step : C \times P \rightarrow C$. The function $\overline{step}$ takes a concrete state and a list of concrete process states as input and computes one concrete successor state for which all processes that were in the state Running conducted their state transitions. The processes in the state Running are processed deterministically from left to right. We define $(c, c') \in\rightarrow$ if and only if $\overline{step}(c, c) = c' \wedge c' \neq c$, where

$$\overline{step}(c, \overline{p} = \langle p_1, \ldots \rangle) = \begin{cases} c & \text{if } |\overline{p}| = 0 \\ \overline{step}(step(c, p_1), \langle p_2, \ldots \rangle) & \text{if } |\overline{p}| > 0 \end{cases}$$

The actual interpretation of the control transitions for a particular concrete process states to reach a successor location l' is initiated by the function *step*:

$$step(c{,}p = \{(\text{pc}, l), (\text{pstate}, r), \ldots\})$$
$$= \begin{cases} c & \text{if } r \neq \text{Running} \\ c' \in \bigcup_{(l, op, l') \in G} csucc_p(c, op, l') & \text{otherwise} \end{cases}$$

Note that both $\overline{step}$ and *step* are *deterministic* functions since they operate on concrete states, for which none of the data locations has non-deterministic values.

Scheduling. LeILa programs have an inherent notion of parallelism. A scheduler determines the next process to run—the process state to put in the Running state—after a state transition has been conducted. For LeILa, we propose a round robin scheduling strategy in combination with a sequentialization: Exactly one process conducts a state transition at one point in time; steps of different processes are interleaved. This corresponds to the Green threading [68] strategy that is implemented in the Scratch VM. This important design choice reduces the number of interleavings to consider by a model checker considerably, since there is always only one process (thread) to conduct the next state transition.

The round robin scheduling is paused as long as a process performs computation in a code block marked with the keyword **atomic**. This is important to mimic the scheduling of processes in environments like the Scratch VM as well as possible. For example, the operations that implement the $\boxed{\text{move (n) steps}}$ Scratch block (see Fig. 1) must not be interrupted by other computations and are handled as one atomic unit.

Another important deviation of round robin scheduling is made for observer processes with **statement finished** event handlers. These types of processes implement monitors that observe whether or not the specification is still satisfied, and signal a violation if not. These monitoring handler processes become activated each time a non-monitoring process finished an atomic state transition.

Event Handling. Scratch programs are driven by events, and so are LeILa programs. Most events boil down to handle incoming messages. For example, the event **on bootstrap** translates to **on message** "BOOTSTRAP" **to** "SYSTEM", and **on startup** translates to **on message** "STARTUP" **to** "SYSTEM". Messages can be explicitly qualified with a channel name, for example, the message "BROADCAST" **to** " SYSTEM" is qualified with the channel name "SYSTEM". In case no channel is provided, we use the default channel "USER".

```
actor ThingObserver is Observer begin
  define atomic checkBehaviorSatisfied () begin
    ...
    if ... then begin
      failure("The thing must not ...")
    end
  end
  script on bootstrap finished do begin
    ...
    checkBehaviorSatisfied()
  end
  script on statement finished do begin
    checkBehaviorSatisfied()
  end
end
```

Figure 3: Formal specification skeleton based on LeILa

After handling a broadcast statement, all processes that correspond to scripts which are handlers for the received message are activated in the computation mode Yield, in case a script has the flag **restart**, the process's program counter variable pc is set to the initial control location of the script. A broadcast with the postfix **and wait** pauses the sending process (computation state Wait) as long as there exists an active process that was triggered by sending the given message and has not reached an exit location of its script.

3.4 Specifying Programs

One goal of our work is to enable teachers, students, and others in writing formal programming task specifications. We propose to use separate actors with the specification concern to monitor the state space and the program's behavior—as also proposed in related work [65]. We extended LeILa with constructs that aid in the specification task: Event handlers for **bootstrap finished** are triggered after all actors have been initialized; these can be used to check the base condition of the specification. All event handlers for **statement finished** are triggered each time the program under analysis conducts a transfer—for an actor that belongs to the program concern. This scheduling corresponds to the activation of observer processes in related work [65]. Handlers for this event ensure that the specification is satisfied, and a *failure* is signaled in case of a specification violation using a particular fail statement. Figure 3 shows a specification skeleton written in LeILa.

Typical Scratch program specifications contain both safety and bounded liveness properties. The liveness properties are typically *time-bound*, that is, expressible in some real-time temporal logic such as MITL [56]. The timed automaton in Fig. 4 illustrates an example property that is also expressible in LeILa.

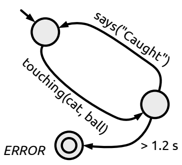


Figure 4: Timed automaton

3.5 Approximations

To make reasoning about Scratch programs feasible, we use several approximations of possible states and behaviors. These are needed to deal with undecidable theories, dependency on the real time (and the execution speeds of machines), the state-space explosion problem that can arise due to non-deterministic behavior (e.g., from the non-deterministic scheduling of processes), and heavily data-dependent properties (e.g., the pixels shown on a screen).

Some of these approximations already happen while translating from Scratch to LeILa programs. We believe that the semantic differences between these languages are not relevant for typical learners tasks such that reasoning results about programs in the intermediate language can be transferred to the original Scratch programs—we evaluate this in Sect. 5.

Time. Time is important for Scratch programs and their specifications. The liveness properties that are relevant in practice can often be time bound. LeILa programs have access to the time that has elapsed since the program has been started. We approximate the time that expires while executing a Scratch program—or a corresponding LeILa program—by mapping a time interval to each operation in a program's control flow.

We use a time profile $\mathcal{T} : Op \rightarrow (\mathbb{N}_\infty \times \mathbb{N}_\infty)$, with $\mathbb{N}_\infty = \mathbb{N} \cup \{0, \infty\}$, to map an interval $[min, max]$ of microseconds to each program operation $op \in Op$, and use this interval to update a global time variable globalTime (an example is given later) that models the time that has expired since the program under analysis was started. This allows analyses to also check real-time properties formulated based on some real-time temporal logic (such as MITL). The *lifting operation* $(\cdot) \Rsh^{\mathcal{T}} : \mathcal{A}^* \rightarrow \mathcal{A}^*$ transforms a given LeILa program $\overline{a}$ to a new LeILa program $\overline{a}_{\mathcal{T}} = \overline{a} \Rsh^{\mathcal{T}}$ by taking a time profile $\mathcal{T}$ into account. That is, program analysis techniques that are not time aware are applicable. Such reductions were done in the past [6, 64] by translating timed automata—or some form of real-time temporal logic in general—to predicate logic.

Technically, we add control transitions after each control transition (l, op, l') of the original program to update the time based on the interval $[min, max] \in \mathcal{T}(op)$. Only actors of the program concern Prog are lifted. The added transitions are labeled with program operations that correspond to following three LeILa statements:

```
declare opTime as integer
assume opTime >= #min and opTime <= #max
define globalTime as globalTime + opTime
```

These statements adjust the time that is assumed to be expired after executing the operation *op*—with #min and #max replaced by corresponding constants. Note that we take advantage of the fact that the variable opTime has a non-deterministic value, which we then restrict using the **assume** statement in its range. We add the control transitions only if an operation's time interval is not empty, that is if $\max - \min \leq 0$.

By convention, we assume that the last concrete state of each *finite* program trace in $[\![\overline{a}]\!] \subseteq C^\infty$ is entered by the *terminating operation* with the statement **halt**. We assign the time interval $[0, \infty]$ to these termination calls—which states that the program is either restarted immediately, stays terminated forever, or is restarted at some time in-between.

Scheduling. Scratch has an inherent notion of concurrency. A scheduling component determines the ordering of the Scratch threads to run, leading to an interleaved concurrency. We do not implement all details of Scratch's scheduler (e.g., particular aspects of how the scheduler deals with loops) and rely purely on LeILa's deterministic round-robin scheduling. The differences in our scheduling strategy can influence the soundness of statements we make about Scratch programs. The scheduler that is implemented in the Scratch VM can lead to non-deterministic schedules in some circumstances—specifically in case one round in the list of processes (threads) takes longer than $(1000/60) * 0.75 = 12.5$ ms, after which the list is processed from its first element.

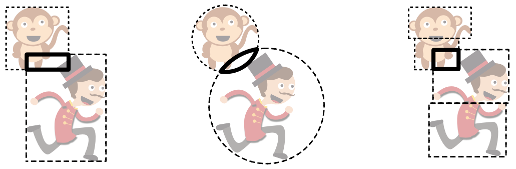


Figure 5: Approximations of the shape of Scratch sprites

```
define atomic mathSin (input: float) begin
  if input >= 0.0 and input <  0.1571 then begin
    assume result > 0.0
    assume result <= 0.1565
  end else if input >= 0.1571 and input < 0.3142 then
    assume result > 0.1565
    assume result <= 0.3091
    ...
```

Figure 6: Excerpt from the sine approximation function

Resources. A central building block of Scratch applications are images and sounds. Images are used as "backdrops" for the stage and "costumes" for the sprites. Attributes of images and their relationship on a canvas are important properties of Scratch programs to reason about. For example, certain behavior might be only visible if the pixels of a sprite are touching the mouse pointer, or if the pixels of two sprites overlap on the canvas.

Instead of considering every single pixel of every image for reasoning, we *approximate the shape* of images. While there are different ways to do this, we use one rectangle per shape as an approximation and implement this approximation for the Scratch blocks touching (mouse pointer), touching (edge), and touching (Monkey) in our block library written in LeILa.

Note that there are different ways to approximate the shape of sprites. See Fig. 5 for some shape approximations (single rectangles, circles, multiple rectangles): A perfect approximation of the sprites' shapes would indicate that the two given sprites do not overlap, while the presented approximations cannot detect this.

Mathematical Functions. A fundamental problem that limits the applicability of elaborated analysis techniques like model checking is the undecidability of some mathematical logics such as the first-order theories of natural and rational number multiplication.

To reduce the mathematical complexity of some of the analysis tasks in the context of Scratch projects, we overapproximated several of the mathematical methods and provide these approximations in the LeILa library as code. In particular we applied intervalization [51] in some of the methods. By putting these approximations into the libraries the actual decision procedures in our analysis framework can be more generic and do not have to provide support for functions like sine or cosine. Figure 6 shows an excerpt of the sine lookup function which takes a value between 0 and 2π and returns an interval that approximates the sine value.

4 ANALYSIS FRAMEWORK

A central contribution of this work is Bastet, a framework for automatically analyzing and verifying Scratch programs translated to LeILa. The long-term vision of Bastet is to provide the foundations for implementing different program analysis techniques that aid in analyzing programs written in visual, block-oriented programming languages. In particular, we aim at providing the foundations for automatic test generation [10, 30], data-flow analysis [66], unbounded model checking based on predicate abstraction [11], and concolic execution [63]. We focus on describing the framework components for implementing a software model checker to verify safety properties and bounded liveness properties. This section therefore assumes knowledge on software model checking; we refer to the literature for background information [12, 41].

4.1 Overall Architecture

The overall workflow of Bastet for model checking Scratch programs consists of three phases: (1) In the first phase, task models are generated. A verification task consists of a specification and the given program, which is parsed and translated from Scratch to LeILa. Specification and program are both transformed into the set of task actors. (2) In the second phase, the actual program analysis is conducted, which constructs, for example, an abstract reachability graph. More generally, for this phase Bastet builds on concepts from abstract interpretation [24, 25, 34], static analysis [59], software model checking [7, 18, 20], and configurable program analysis [13]. The program analysis consists of a set of *analysis algorithms* (Section 4.3), where one algorithm can *wrap* another one as illustrated in Fig. 7. Analysis algorithms act on an abstract representation of a program's state space and its behaviors provided by a set of state interpreters (Section 4.4). (3) The last phase produces output artifacts, for example, error witnesses, test suites, or correctness proofs if a suitable analysis configuration is used. Since we aim at running program analyses while programming in a Web browser, we have taken a novel route and built Bastet entirely on Web technologies: The framework is written in TypeScript, attached SMT solvers (Z3) are compiled to WebAssembly, and the framework can run in NodeJs or on top of a browser.

4.2 Abstract Domain

To make statements about properties of programs, Bastet relies on abstraction, which is central for constructing finite abstractions of a programs' state space [13, 22, 25], and to cope with the undecidability of most program analysis problems [60]. The *abstract domain* [13] $D = (\ddot{C}, \ddot{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ provides the central operations for abstraction, and for mapping between abstract states and sets of concrete states. The set of *abstract states* E is arranged in a *lattice* $\ddot{E} = (E, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ [15] and is partially ordered by its *inclusion relation* $\sqsubseteq$: $E \times E \rightarrow \mathbb{B}$. The *meet* $\sqcap : E \times E \rightarrow E$, *join* $\sqcup : E \times E \rightarrow E$, *top element* $\top$, and the bottom element $\bot$ are defined as usual. The *concretization* function $[\![\cdot]\!] : E \rightarrow 2^C$ maps an abstract state $\in E$ to a set of concrete states $\subseteq C$, for example, $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$. The *abstraction* function $\langle\!\langle\cdot\rangle\!\rangle : 2^C \rightarrow E$ maps a set of concrete states to an abstract state. Widening is provided by an *abstract domain with widening* $D^\pi = (D, \Pi, \langle\!\langle\cdot\rangle\!\rangle^\pi)$, which defines an *abstraction with widening* $\langle\!\langle\cdot\rangle\!\rangle^\pi : E \rightarrow E$ based on

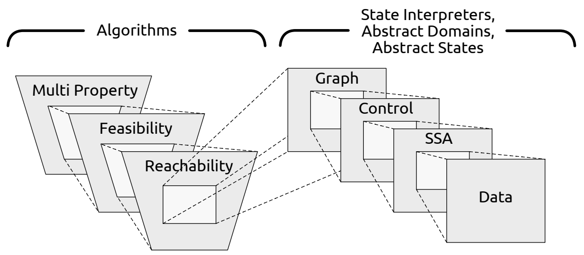


Figure 7: Composed Analysis Procedure

the set of abstraction precisions Π. An *abstraction precision* $\in \Pi$ [21] determines the information to maintain by an abstraction computation. Different realizations of an abstract domain are possible, for example, based on predicate logic [20, 33], or based on combinations of several abstract domains [26]. In this work, we do not evaluate an analysis that conducts any widening, but consider this functionality central for the Bastet program analysis framework.

4.3 Algorithms

The flow of the analysis steps is determined by several *analysis algorithms*, where one analysis algorithm can *wrap* another one (Fig. 7). Each of these algorithms operates on (at least) the set of reached states (reached) and the set of frontier states (frontier, also called worklist or waitlist) of an abstract reachability graph—the nodes of this graph are abstract states. The following analysis algorithms are implemented in Bastet for a resource bounded model checking [20] procedure.

Reachability. The *reachability* algorithm, based on configurable program analysis [13], conducts the reachability analysis as shown in Alg. 1: Starting from an initial set of reached states, and an initial set of frontier states (both are passed as arguments to the algorithm), the algorithm traverses the state space of the analysis task until either (1) a fixed point of the reachable states was attained, (2) a target state was found, or (3) a resource budget was exhausted. A *target state* is an abstract state for that one or more properties to check were signaled to be violated. The algorithm is implicitly parameterized with a state interpreter *SI*, which defines different operators the algorithm uses to compute abstract successor states, merge abstract states, or check coverage. The state-space traversal strategy is determined by the operation choose on the set frontier.

Feasibility. Whenever the analysis reaches a target state the feasibility of this state has to be checked. A target state is *feasible* only if it represents at least one concrete state that is reachable if the program is executed concretely—for example, in the Scratch VM. An abstract state can be *infeasible* (1) in case the wrapped analysis computed an abstraction (widening) that overapproximated the set of reachable states, or (2) a decision procedure—for example, a SMT solver—was not yet invoked to check if the abstract state represents a non-empty set of concrete states. To foster a clear separation of concerns, we use a separate feasibility check algorithm—which can also be extended to implement a CEGAR loop [21] and conduct precision refinements. When a target state has been reached, then the reachability algorithm terminates and returns to the feasibility check algorithm, which might re-invoke the reachability algorithm after an infeasible state was eliminated. Note that we abstracted

Algorithm 1 Reachability($\text{frontier}_0, \text{reached}_0)_{SI}$

```
Input:  reached_0 ∈ $2^E$: Set of initially reached states
        frontier_0 ∈ $2^E$: Initial set of frontier states
        $SI = (D, \text{succ}, \text{widen}, \text{mergeto}, \text{stop}, \text{target}, \text{init}, \text{choose})$
Output: $(\text{frontier}, \text{reached}) \in 2^E \times 2^E$
1:  frontier ← frontier_0
2:  reached ← reached_0
3:  while frontier ≠ ∅ do
4:      $e \leftarrow \text{choose}(\text{frontier})$
5:      $\text{frontier} \leftarrow \text{frontier} \setminus \{e\}$
6:      for each $e' \in \text{succ}(e)$ do
7:          $e'' \leftarrow \text{widen}(e', \text{reached})$
8:          $(\text{frontier}, \text{reached}) \leftarrow \text{mergeto}(e'', \text{frontier}, \text{reached})$
9:          if not $\text{stop}(e'', \text{reached})$ then
10:             $\text{frontier} \leftarrow \text{frontier} \cup \{e''\}$
11:             $\text{reached} \leftarrow \text{reached} \cup \{e''\}$
12:             if $\text{target}(e'') \neq \emptyset$ then
13:                 return (frontier, reached)
14: return (frontier, reached)
```

away some details of how concrete executions are conducted in the Scratch VM, that is, not all feasible target states might be actually feasible due to slightly different semantics—see Sect. 3.5.

Multi Property. The *multi-property* algorithm [4] maintains the status of all properties from the formal specification to check. Among the set of reached states and the frontier states, the algorithm maintains the set unknown of properties for that no verdict was decided, the set violated of properties that have been found to be violated (along with a counterexample), and the set satisfied of properties that were decided to be satisfied (possibly paired with a proof of correctness). The algorithm maintains a budget of resources that is left to check the different properties. To decide a property's verdict, another algorithm is invoked if budget is left.

4.4 State Interpreters

The main functionality for assigning meaning to a program's operations and producing an abstract representation of a program's state space and behaviors is provided by a set of state interpreters. A *state interpreter* implements abstract interpretation [24, 25] and builds on the formalisms and operators that were introduced with the CPA concept [13]. We define a state interpreter by the tuple

$$SI = (D, \text{succ}, \text{widen}, \text{mergeto}, \text{stop}, \text{target}, \text{init}, \text{choose}).$$

A state interpreter generalizes a configurable program analysis by allowing for more control of the process of merging the state space, reflected by the operator mergeto. It further makes the state space traversal process more explicit through the operators init and choose. We define the following components:

1. The *abstract domain* [13, 25] $D = (\ddot{C}, \ddot{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ determines the form of abstraction that is used to represent sets of concrete states as abstract states—see Sect. 4.2 for more details.

2. The *init* operator $\text{init} : () \rightarrow 2^E \times 2^E$ returns the initial sets of frontier and reached states. It defines the state sets that are passed initially to the analysis algorithms as arguments.

3. The *abstract transfer* [13, 25] function $\text{succ} : E \rightarrow 2^E$ provides the set of abstract successor states for a given abstract state. A *labelled abstract transfer* $\text{succ}_{op} : E \rightarrow 2^E$ computes abstract successor states by interpreting a given program operation $op \in Op$.

4. The *widening* [25] operator $\text{widen} : E \times 2^E \rightarrow E$ is used to compute widenings of abstract states, that is, with $e \sqsubseteq \text{widen}(e, \cdot)$. Typically, the widening is parameterized implicitly with an abstraction precision $\pi \in \Pi$ resulting in $\text{widen}_\pi : E \times 2^E \rightarrow E$.

5. The *merge-to* operator $\text{mergeto} : (E \times 2^E \times 2^E) \rightarrow (2^E \times 2^E)$ can be used to merge a given abstract state into existing abstract states to keep the abstract reachability graph compact. It is crucial for the performance of an analysis procedure. Algorithm 2 illustrates one possible implementation of this operator, which mimics the merge functionality of the CPA algorithm [13]. The algorithm uses the function $\text{allowMerge} : E \times E \rightarrow \mathbb{B}$, which defines whether or not a merge is intended, and the operator $\text{merge} : E \times E \rightarrow E$ [13] which actually merges two abstract states—we assume that merge is only defined for states with $\text{allowMerge}(e) = \textit{true}$.

The variant $\text{allowMerge}_{\text{Never}}(e, \text{frontier}, \text{reached}) = (\text{frontier}, \text{reached})$ never merges a given abstract state into the state space, and avoids looping over the states already reached, and thus, reduces the complexity of the reachability analysis.

6. The *stop* operator [13] $\text{stop} : E \times 2^E \rightarrow \mathbb{B}$ defines whether or not the given abstract state should be added to the sets frontier and reached. Typically, this operator conducts a coverage check and is crucial for a fixed-point iteration: The state is only added if not yet covered by the existing set of reached states, that is, $\text{stop}_{\text{Cover}}(e, R) = (\exists r \in R : e \sqsubseteq r)$.

7. The *target* operator $\text{target} : E \rightarrow 2^{\mathcal{S}}$ defines the set of properties $\subseteq \mathcal{S}$ that should be signaled to be reached (or violated) at the given state. These properties can be, for example, trap properties of a test generation procedure, or other safety properties to check by model checking.

8. The *choose* operator $\text{choose} : 2^E \rightarrow E$ determines the state-space traversal strategy of the analysis: It returns the next state to compute successor states for from a given set of abstract states (typically, the set of frontier states). By default, Bastet uses the operator choose_{wam} which implements the *wait-at-meet traversal* strategy. For each control location of the transition systems of the LeILa program to analyze, a wait-at-meet number is computed that ensures that all states that lead to a control location on that the control flow merges are processed before continuing to the merge location.

Partitioning. Three operators in the reachability algorithm take the set of already reached states reached as argument: the widening operator widen, the merge-to operators mergeto, and the stop operator stop. Different subsets of reached might be relevant for the operators to conduct their job in a sound fashion while not sacrificing the degree of completeness. Considering only a subset of reached states can have a considerable impact on the performance (and algorithmic complexity, for example, of Alg. 2) of the analysis. By default, we partition the set of reached states based on the position in the control flow the different processes are in.

Algorithm 2 $\text{mergeto}_{\text{Std}}(e, \text{frontier}, \text{reached})$

```
Input:  $e \in E$: Abstract state to possibly merge
        frontier $\in 2^E$: Current set of frontier states
        reached $\in 2^E$: Current set of reached states
Output: (frontier, reached) $\in 2^E \times 2^E$
1:  remove $\leftarrow \emptyset$
2:  add $\leftarrow \emptyset$
3:  for each $r \in$ reached do
4:      if allowMerge$(e, r)$ then
5:          $e' \leftarrow$ merge$(e, r)$
6:          remove $\leftarrow$ remove $\cup \{ r \}$
7:          add $\leftarrow$ add $\cup \{ e' \}$
8:  frontier $\leftarrow$ (frontier $\cup$ add) $\setminus$ remove
9:  reached $\leftarrow$ (reached $\cup$ add) $\setminus$ remove
10: return (frontier, reached)
```

4.5 Basic State Interpreters

In the following, we describe the state interpreters that we consider to be of wider interest, and which we have evaluated in our empirical study on the applicability of our framework. Figure 7 illustrates the combination of the analysis components.

Graph Interpreter. The *graph interpreter* $SI_{\mathbb{G}}$ keeps track of the predecessor–successor relation of abstract states, and constructs the abstract reachability graph. The graph interpreter uses the abstract domain $D_{\mathbb{G}}$ with the lattice $\ddot{U}$ of graph states U. A *graph state* $u = (w, \widehat{u}) \in U$ consists of a *wrapped abstract state* $w \in E$ and a set of *predecessor* graph states $\widehat{u} \subset U$. In its merge-to operator $\text{mergeto}_{\mathbb{G}}$, the interpreter makes sure that whenever a state is removed from the graph (because is was merged into another state), also all its children are removed from the graph, and also takes care of re-adding states to the set of frontier states frontier in case one of the removed states was in there. Note that our mergeto operator makes explicit that a merge can affect both the set of reached and frontier states, which is not exemplified in the original CPA formalism [13].

Control Interpreter. The *control interpreter* $SI_{\mathbb{C}}$ takes care of modeling the control-flow of LeILa programs, including modeling unrollings of loops, keeping track of the call stack, conducting message passing, and including the scheduling of threads—see Sect. 3.3. The position in the control flow and the computation status of the different threads is modeled explicitly (not symbolically). The control interpreter wraps another interpreter and provides the labeling between on transitions, that is, the labeled abstract transfer of the wrapped interpreter is called.

The control interpreter uses the *control abstract domain* $D_{\mathbb{C}}$ with the lattice $\ddot{Z}$ of abstract control states Z. The interpreter merges two abstract control states only if all its processes are on the same control location, if they have the same call stack, the same loop unrollings, if all processes are in the same computation state, and if also the wrapped abstract states are supposed to be merged.

SSA Interpreter. The *SSA interpreter* $SI_{\mathbb{S}}$ transforms the program operations that are passed to its labeled abstract transfer function into a single-static assignment [5, 62] form, such that wrapped interpreters can build on this. In the context of model checking, an SSA transformation cannot be done in a preprocessing step, but depends on runtime information such as the number of loop unrollings or the call stack. The SSA interpreter uses the abstract domain $D_{\mathbb{S}}$ with the lattice $\ddot{S}$ of SSA states S. An *SSA state* $s = (w, \eta) \in S$ consists of a *wrapped abstract state* $w \in E$ and a *SSA map* $\eta : X \longrightarrow \mathbb{N}$, which maps to each data location a current SSA index. The SSA interpreter delegates the decision whether or not to merge two abstract states to the wrapped interpreter. In case two SSA states are supposed to be merged, the merge functionality of the SSA interpreter ensures that also the SSA maps of the states are merged and synchronized, that is, the SSA ϕ functions applied in the merge operation. The wrapped analysis is instructed to also synchronize the information based the ϕ function. This clear separation of concerns is missing in some [13] established frameworks.

Data Interpreter. The *data interpreter* $SI_{\mathbb{D}}$ keeps track of the data state of LeILa programs by encoding all data (values of data locations on the heap and the stack) into predicate logic. We require that this state interpreter is wrapped by an SSA interpreter, which ensures that all program operations that are given to the labeled transfer function are in the SSA form.

The interpreter uses the abstract domain $D_{\mathbb{D}}$ with the lattice $\ddot{M}$ of abstract data states M. One data state $m = (\phi) \in M$ consists of a (block) formula ϕ in predicate logic only, which encodes the full content of the heap and stack data locations.

In its standard configuration, the interpreter always allows to merge two abstract data states. That is, given two abstract data states $m_1 = (\phi_1) \in M$ and $m_2 = (\phi_2) \in M$, the result is a new abstract data state $m_3 = m_1 \sqcup m_2$, where the join $\sqcup$ of the lattice corresponds to the Boolean disjunction $\phi_1 \vee \phi_2$ of the two formulas. The labelled abstract transfer function succ_{op} encodes the semantics of a given operation into the formula of the successor state.

5 PILOT STUDY

To demonstrate the practical applicability of the Bastet framework for analyzing Scratch projects, we conduct an empirical pilot study. To make our results easy to reproduce, we provide a replication package, and provide the Bastet framework as open source: github.com/se2p/artifact-ase2020/. In this study we aim to answer the following research questions:

RQ1 (Soundness). To which extent does translating Scratch projects to LeILa maintain the semantics such that both useful and sound propositions can be made?

RQ2 (Performance). Are the typical limitations of model checking, such as the state-space explosion problem and undecidable theories, limiting factors when applied to Scratch projects?

5.1 Study Objects

We conduct our empirical study based on four Scratch programming exercises taken from the context of primary school programming education [31]. The children are given an informal specification of what the program is supposed to do, and so we can compare their solutions against the specifications, which we formalized in LeILa. Related work [31] describes details on the setup of the course.

Monkey. This exercise uses two sprites: a circus director and a monkey. The goal is to have the circus director move continuously towards the monkey. The formal specification requires that the circus director must make a step at least once every 100 ms and that the circus director may not move away from the monkey.

Elephant. This exercise is implemented with a single sprite: the elephant, which has different costumes. The goal is to create the impression that the elephant is dancing by continuously switching the costumes. Our formal specification requires that a costume change must take place at least every 1.2 s.

Cat. This exercise comprises two sprites: a cat and a ball. The goal is that the cat indicates, via a speech bubble, that it has caught the ball as soon as it touches the ball sprite. The formal specification therefore states that within 1.2 s after the sprites touch, the cat sprite must say that it caught the ball—see Fig. 4.

Horse. This exercise uses a single sprite that reacts to mouse inputs. As long as the sprite is not touched by the mouse pointer it should continuously change its color; as soon as it touches the mouse pointer it should rotate. The formal specification requires that its color must change at least every 100 ms or its direction must change at least every 100 ms if the mouse pointer is touched.

The dataset consists of 279 non-empty solutions to these four exercises. We automatically translated all solutions to LeILa and also wrote the formal specifications in LeILa for all four projects.

5.2 Experimental Setup

Bastet was used in revision aa1026a with a *bounded model checking* configuration, using a time bound of 300 s. All experiments were conducted on machines equipped with Intel Xeon E5-2650 v2 @ 2.60 GHz CPUs with 256 GiB of RAM. Bastet was executed within Docker containers. All processes were limited to consume at most 10 GiB of RAM, and were limited to at most 4 CPU cores. The measured execution (Wall clock) time excludes the time needed to parse the given Scratch project or corresponding LeILa program, and the time for starting and initializing the NodeJs environment.

5.3 Experiment Procedure

We conducted the following experiments to answer our questions:

RQ1. To learn about the soundness of Bastet, we first inspected all student solutions *manually* and checked if they satisfy the specification. We executed Bastet on all of these 279 solutions, and then compared Bastet results with our manually assigned verdicts.

RQ2. To make statements about the applicability of model checking to Scratch projects and the resulting performance, we use Bastet configured as a *time-bound* model checker and run it on the verification tasks—pairs of student solutions and formal specification—described earlier. Our hypothesis is that Scratch programs tend to be simple, and might not have the complexity of traditional programs, and thus, also expensive techniques like model checking seem in reach to get results efficiently.

5.4 RQ1 Results (Soundness)

Table 1 presents the comparison between the results of the manual inspection with those computed by Bastet automatically.

Bastet yields 3 false negatives [36]—falsely reports program correctness. The false negative for Monkey is due to a lost ordering of the actors while translating to LeILa, which leads to an interleaving of the script executions that hides the bug—an imprecision that can be fixed in principle. The other two false negatives were produced for the Cat exercise and have the same cause: They both only check whether the cat touches the ball once at the start of the program, rather than in a forever loop. Since the programs do not initialize the sprites' positions, Bastet assumes that the positions are non-deterministic, and thus the solutions are actually correct. This is a good example for the value of formal specifications, and the challenges of producing them. We can also see 5 true negatives where Bastet was able to reach a fixed-point in the state space and to actually prove the correctness of the solutions.

Table 1: Model Checking Soundness

Exercise	True Positive	True Negative	False Positive	False Negative	Missed Safe	Missed Unsafe
Monkey	5	5	1	1	26	16
Cat	4	0	1	2	49	12
Elephant	22	0	0	0	19	39
Horse	9	0	0	0	12	2

Table 2: Model Checking Performance

Exercise	Min Time	Max Time	Median Time	Min Reached	Max Reached	Median Reached
Monkey	8.5	18	12	728	9027	6147
Cat	9.0	54	34	1038	11885	3969
Elephant	4.8	300	23.5	572	28164	5701
Horse	5.4	20	17	664	4795	4644

The evaluated configuration produced 40 true positives—cases where we can observe undesired behavior—and 2 false positives—cases in which a violation was reported while the solution is correct. The false positives were produced because we assumed the mouse position to be non-deterministic as an approximation, while in true program executions, it can change only after handling messages from the event dispatcher loop.

The configuration we studied is conceptually only able to provide correctness proofs for programs without infinite loops. Many of the solutions contain forever loops, thus, we expected a high number in the column "Missed Safe". For 36 verification tasks, Bastet was not able to terminate with a solution because of the undecidability of arithmetic with multiplication, which is used by the atan2 function, invoked by the pointTowards(..) block.

In total, 95 % of the violations reported were true positives. Consequently, translating Scratch projects into LeILa and interpreting them by Bastet maintains the semantics in the clear majority of the studied verification tasks.

5.5 RQ2 Results (Performance)

This question addresses the performance of model checking, using the Bastet framework, when applied to Scratch projects. Table 2 shows the performance measures for Bastet in terms of time and size of the abstract reachability graphs, restricted to results with the verdict *false*. The median time lies between 12 s for the Monkey exercise and 300 s for the Elephant exercise. While the number of states is substantial considering the size of the programs, we cannot observe dramatical explosions of the reachability graphs. Analysis of the runtime behavior of BASTET suggests that large portions of the time were spent in the SMT solver. Consequently, while SCRATCH programs may seem simple and playful, they bear a high mathematical complexity, which is due to their game-like nature, involving multiplication and division of natural and real numbers. The majority of the analysis time is spent in the SMT solver. Nevertheless, BASTET managed to identify 30 % of all bugs.

5.6 Discussion

This pilot study clearly demonstrates the practical applicability of BASTET for analyzing real-world SCRATCH programs. Instead of demonstrating this based on a basic data-flow analysis, we demonstrated how even a heavy-weight model checking procedure can be implemented on top of BASTET to check learners' programs. While most of the time for running a verification task is spent in the SMT solver, we identified further potential for performance improvements by making use of the potential to use alternative implementations of the analysis operators, for example, by using versions of the operators stop and mergeto that iterate over smaller partitions of the reached states. Since LEILA uses a deterministic interleaving concurrency, we do not face the state-space explosion problem that is typical for concurrent programs with non-deterministic schedulers. Note that the scheduler implemented in SCRATCH can have non-deterministic behavior. Our results indicate that this design decision does only have a minor impact on the soundness of BASTET's model checking configuration. Approximations of mathematical functions such as *sin* and *cos* were crucial to cope with some of the undecidable characteristics that are inherent to SCRATCH program, and also to deal with limitations of the SMT solver theories.

6 RELATED WORK

The increasing popularity of SCRATCH as an introductory programming environment has triggered research on analyzing the resulting programs. In particular, the observation that SCRATCH programmers tend to develop certain negative habits while coding [49] has led to investigations into the general quality problems in SCRATCH programs. It has been shown that various types of code smells are prevalent [3, 39, 61, 70] and have a negative impact on code understanding [38]. Similar habits have been observed outside of SCRATCH in the wider area of scenario based programming[32]. Most existing tools for analysis of SCRATCH programs have their roots in the HAIRBALL [16] Python script which parses SCRATCH 2.0 programs into a kind of abstract syntax tree. For example, the DR. SCRATCH [52] website includes code smell reports produced by HAIRBALL when analyzing learners' programs. Further analysis tools aiming to detect code smells include QUALITY HOUND [69] and SAT[19]. These tools mostly analyze SCRATCH programs only by matching patterns, while BASTET is a full fledged and configurable program analysis framework, and inherits concepts from well-adopted tools like CPACHECKER [14].

A common application of program analysis in the educational domain is automated grading; The ITCH tool [42] translates a small subset of SCRATCH programs to Python programs (textual interactions via say/ask blocks) and then runs tests on these programs. The WHISKER tool [65] executes automated tests directly in the SCRATCH IDE, and supports property-based testing. Autograding has also been applied to the related SNAP! [35] language [73]. Beyond grading, an important application area for automated program analysis is automated hint generation, for example by identifying suggested next blocks based on the evolution of similar programs [57]. Furthermore, automated refactoring [71] has been considered as a route to helping students improve their coding skills. BASTET provides a foundational framework that can better support these activities. Furthermore, BASTET is implemented directly in TypeScript in order to support the direct integration of program analyses into educators' and learners' environments, which are often Web based.

BASTET uses LEILA as its intermediate language, which was designed to reflect the semantics of SCRATCH as well as possible, to provide constructs for program analysis and verification, and to be comprehensible by teachers and learners. The use of intermediate languages is a common approach to enable analysis, and other example languages include an intermediate language for timed asynchronous systems [17], a translation of Timed CSP into LLVM IR [8], the C Intermediate Language [53], or a textual representation of timed state charts [44]. Nevertheless, none of them was immediately applicable to the programs we aim to analyze.

Building on concepts found in established frameworks such as CPACHECKER, BASTET contributes an evolved and holistic perspective on the components of a program analysis framework. For example, we refined the CPA algorithm by providing a more general merge operator that can manipulate both the reached and the frontier states. Among the conceptual differences, BASTET is the first program analysis framework entirely built on Web technologies.

7 CONCLUSIONS

Block-based programming languages are tremendously popular, and their popularity leads to a need for automated program analysis. However, the game-like and concurrent nature of typical programs, their web-based execution environments, and their ad-hoc semantics make this challenging. To address this problem, in this paper we introduced BASTET, a framework for program analysis and verification for SCRATCH programs based on abstract interpretation and software model checking. BASTET is based on a clean definition of the semantics of SCRATCH programs, and a translation to the LEILA intermediate language. A pilot study on 279 real children's programs demonstrated the soundness and potential of the approach. The pilot study also revealed that, despite their playful nature and small size, the programs represent fundamental challenges for verification, offering potential for future work on improving the performance and capabilities of the program analysis approach. Besides principle enhancements and optimizations of BASTET, future work will also explore different ways to apply BASTET for supporting learners and educators. To foster research on these aspects, BASTET is available as open source at https://github.com/se2p/bastet.

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". We thank the reviewers whose comments and suggestions helped to improve this manuscript.

REFERENCES

[1] Martín Abadi and Gordon D. Plotkin. 2010. A Model of Cooperative Threads. *Log. Methods Comput. Sci.* 6, 4 (2010).

[2] Gul Agha and Carl Hewitt. 1985. Concurrent Programming Using Actors: Exploiting large-Scale Parallelism. In *FSTTCS (Lecture Notes in Computer Science, Vol. 206)*. Springer, 19–41.

[3] Efthimia Aivaloglou and Felienne Hermans. 2016. How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 53–61.

[4] Sven Apel, Dirk Beyer, Vitaly O. Mordan, Vadim S. Mutilin, and Andreas Stahlbauer. 2016. On-the-fly decomposition of specifications in software model checking. In *SIGSOFT FSE*. ACM, 349–361.

[5] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (1998), 17–20.

[6] Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. 2002. Bounded Model Checking for Timed Systems. In *FORTE (Lecture Notes in Computer Science, Vol. 2529)*. Springer, 243–259.

[7] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *CAV (Lecture Notes in Computer Science, Vol. 2102)*. Springer, 260–264.

[8] Björn Bartels and Sabine Glesner. 2011. Verification of Distributed Embedded Real-Time Systems and their Low-Level Implementations Using Timed CSP. In *APSEC*. IEEE Computer Society, 195–202.

[9] David Bau, D Anthony Bau, Mathew 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*. 445–448.

[10] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Generating Tests from Counterexamples. In *ICSE*. IEEE Computer Society, 326–335.

[11] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. 2009. Software Model Checking via Large-Block Encoding. *CoRR* abs/0904.4709 (2009).

[12] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. 2018. Combining Model Checking and Data-Flow Analysis. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 493–540. https://doi.org/10.1007/978-3-319-10575-8_16

[13] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *CAV (Lecture Notes in Computer Science, Vol. 4590)*. Springer, 504–518.

[14] Dirk Beyer and M. Erkan Keremoglu. 2009. CPAchecker: A Tool for Configurable Software Verification. *CoRR* abs/0902.0019 (2009).

[15] Garrett Birkhoff. 1935. On the structure of abstract algebras. In *Mathematical proceedings of the Cambridge philosophical society*, Vol. 31. Cambridge University Press, 433–454.

[16] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 215–220.

[17] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. 1999. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *World Congress on Formal Methods (Lecture Notes in Computer Science, Vol. 1708)*. Springer, 307–327.

[18] Guillaume P. Brat and Willem Visser. 2001. Combining Static Analysis and Model Checking for Software Analysis. In *ASE*. IEEE Computer Society, 262.

[19] Zhong Chang, Yan Sun, Tin-Yu Wu, and Mohsen Guizani. 2018. Scratch analysis Tool (SAT): a modern scratch project analysis tool based on ANTLR to assess computational thinking skills. In *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE, 950–955.

[20] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34.

[21] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (Lecture Notes in Computer Science, Vol. 1855)*. Springer, 154–169.

[22] Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1512–1542.

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

[24] P. Cousot. 2003. Verification by abstract interpretation. In *Verification: Theory and Practice*. Springer, 243–268.

[25] P. Cousot and R. Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547.

[26] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2011. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 6604)*. Springer, 456–472.

[27] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457.

[28] Caitlin Duncan, Tim Bell, and Steve Tanimoto. 2014. Should your 8-year-old learn coding?. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. 60–69.

[29] Wallace Feurzeig and Seymour Papert. 2011. Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments* 19, 5 (2011), 487–501.

[30] Angelo Gargantini and Constance L. Heitmeyer. 1999. Using Model Checking to Generate Tests from Requirements Specifications. In *ESEC / SIGSOFT FSE (Lecture Notes in Computer Science, Vol. 1687)*. Springer, 146–162.

[31] Katharina Geldreich, Alexandra Funke, and Peter Hubwieser. 2016. A programming circus for primary schools. In *ISSEP 2016*. 49–50.

[32] Michal Gordon, Assaf Marron, and Orni Meerbaum-Salant. 2012. Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. 198–203.

[33] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (Lecture Notes in Computer Science, Vol. 1254)*. Springer, 72–83.

[34] S. Gulwani and A. Tiwari. 2006. Combining abstract interpreters. In *Proc. PLDI*. ACM, 376–386.

[35] Brian Harvey, Daniel D Garcia, Tiffany Barnes, Nathaniel Titterton, Daniel Armendariz, Luke Segars, Eugene Lemon, Sean Morris, and Josh Paley. 2013. Snap!(build your own blocks). In *Proceeding of the 44th ACM technical symposium on Computer science education*. 759–759.

[36] Sarah Smith Heckman and Laurie Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM*. ACM, 41–50.

[37] Matthew Hennessy and Gordon D. Plotkin. 1979. Full Abstraction for a Simple Parallel Programming Language. In *MFCS (Lecture Notes in Computer Science, Vol. 74)*. Springer, 108–120.

[38] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[39] Felienne Hermans, Kathryn T Stolee, and David Hoepelman. 2016. Smells in block-based programming languages. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 68–72.

[40] Gerard J. Holzmann. 2000. Logic Verification of ANSI-C Code with SPIN. In *SPIN (Lecture Notes in Computer Science, Vol. 1885)*. Springer, 131–147.

[41] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 1–54.

[42] David E Johnson. 2016. ITCH: Individual Testing of Computer Homework for Scratch Assignments. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 223–227.

[43] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.* 37, 2 (2005), 83–137.

[44] Yonit Kesten and Amir Pnueli. 1992. Timed and Hybrid Statecharts and Their Textual Representation. In *FTRTFT (Lecture Notes in Computer Science, Vol. 571)*. Springer, 591–620.

[45] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *AGERE!@SPLASH*. ACM, 31–40.

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

[47] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4 (2010), 16:1–16:15.

[48] Linda McIver and Damian M Conway. 1999. GRAIL: A Zeroth Programming Language. In *Advanced Research in Computers and Communications in Education New Human Abilities for the Networked Society*. IOS Press, Netherlands, 43 – 50.

[49] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 168–172.

[50] Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer.

[51] Antoine Miné. 2007. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. *CoRR* abs/cs/0703076 (2007).

[52] Jesús Moreno-León and Gregorio Robles. 2015. Dr. Scratch: A web tool to automatically evaluate Scratch projects. In *Proceedings of the workshop in primary and secondary computing education*. 132–133.

[53] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC (Lecture Notes in Computer Science, Vol. 2304)*. Springer, 213–228.

[54] Terence John Parr and Russell W. Quong. 1995. ANTLR: A Predicated- *LL(k)* Parser Generator. *Softw. Pract. Exp.* 25, 7 (1995), 789–810.

[55] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

[56] Amir Pnueli and Aleksandr Zaks. 2008. On the Merits of Temporal Testers. In *25 Years of Model Checking (Lecture Notes in Computer Science, Vol. 5000)*. Springer, 172–195.

[57] 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*. 483–488.

[58] Partha Pratim Ray. 2017. A survey on visual programming languages in internet of things. *Scientific Programming* 2017 (2017).

[59] Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726.

[60] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[61] 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.

[62] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *POPL*. ACM Press, 12–27.

[63] Koushik Sen. 2007. Concolic testing. In *ASE*. ACM, 571–572.

[64] Maria Sorea. 2002. Bounded Model Checking for Timed Automata. *Electron. Notes Theor. Comput. Sci.* 68, 5 (2002), 116–134.

[65] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *ESEC/SIGSOFT FSE*. ACM, 165–175.

[66] Bernhard Steffen. 1991. Data Flow Analysis as Model Checking. In *TACS (Lecture Notes in Computer Science, Vol. 526)*. Springer, 346–365.

[67] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4 (2013), 19:1–19:40.

[68] Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, and Heonshik Shin. 2002. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread. *Inf. Process. Lett.* 84, 4 (2002), 221–225.

[69] Peeratham Techapalokul and Eli Tilevich. 2017. Quality Hound—an online code smell analyzer for Scratch programs. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 337–338.

[70] Peeratham Techapalokul and Eli Tilevich. 2017. Understanding recurring quality problems and their impact on code sharing in block-based software. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 43–51.

[71] Peeratham Techapalokul and Eli Tilevich. 2019. Code quality improvement for all: Automated refactoring for Scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 117–125.

[72] Jake Trower and Jeff Gray. 2015. Blockly language creation and applications: Visual programming for media computation and bluetooth robotics control. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 5–5.

[73] Christiane Gresse Von Wangenheim, Jean CR Hauck, Matheus Faustino Demetrio, Rafael Pelle, Nathalia da Cruz Alves, Heliziane Barbosa, and Luiz Felipe Azevedo. 2018. CodeMaster–Automatic Assessment and Grading of App Inventor and Snap! Programs. *Informatics in Education* 17, 1 (2018), 117–150.

[74] David Weintrop, David C Shepherd, Patrick Francis, and Diana Franklin. 2017. Blockly goes to work: Block-based programming for industrial robots. In *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 29–36.

[75] David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children*. 199–208.

[76] Niklaus Wirth. 1971. The Programming Language Pascal. *Acta Inf.* 1 (1971), 35–63.

[77] Niklaus Wirth. 2002. Pascal and Its Successors. In *Software Pioneers*. Springer Berlin Heidelberg, 108–119.