Generating Timed UI Tests from Counterexamples

Dominik Diner, Gordon Fraser, Sebastian Schweikl, and Andreas Stahlbauer

University of Passau, Germany

Abstract. One of the largest communities on learning programming and sharing code is built around the Scratch programming language, which fosters visual and block-based programming. An essential requirement for building learning environments that support learners and educators is automated program analysis. Although the code written by learners is often simple, analyzing this code to show its correctness or to provide support is challenging, since Scratch programs are graphical, game-like programs that are controlled by the user using mouse and keyboard. While model checking offers an effective means to analyze such programs, the output of a model checker is difficult to interpret for users, in particular for novices. In this work, we introduce the notion of Scratch error witnesses that help to explain the presence of a specification violation. Scratch error witnesses describe sequences of timed inputs to Scratch programs leading to a program state that violates the specification. We present an approach for automatically extracting error witnesses from counterexamples produced by a model checking procedure. The resulting error witnesses can be exchanged with a testing framework, where they can be automatically re-played in order to re-produce the specification violations. Error witnesses can not only aid the user in understanding the misbehavior of a program, but can also enable the interaction between different verification tools, and therefore open up new possibilities for the combination of static and dynamic analysis.

Keywords: Error Witnesses · Model Checking · Reachability · Dynamic Analysis · Test Generation · Block-Based Programming · UI Testing

1 Introduction

Block-based programming languages like Scratch have gained momentum as part of the general trend to integrate programming into general education. Their widespread use will crucially depend on automated program analysis to enable learning environments in which learners and educators receive the necessary help for assessing progress, finding errors, and receiving feedback or hints on how to proceed with a problem at hand. Although learners' programs tend to be small and their code is usually not very complex, Scratch programs nevertheless pose unique challenges for program analysis tools: they are highly concurrent, graphical, driven by user interactions, typically game-like and nondeterministic, and story-components and animations often lead to very long execution times.

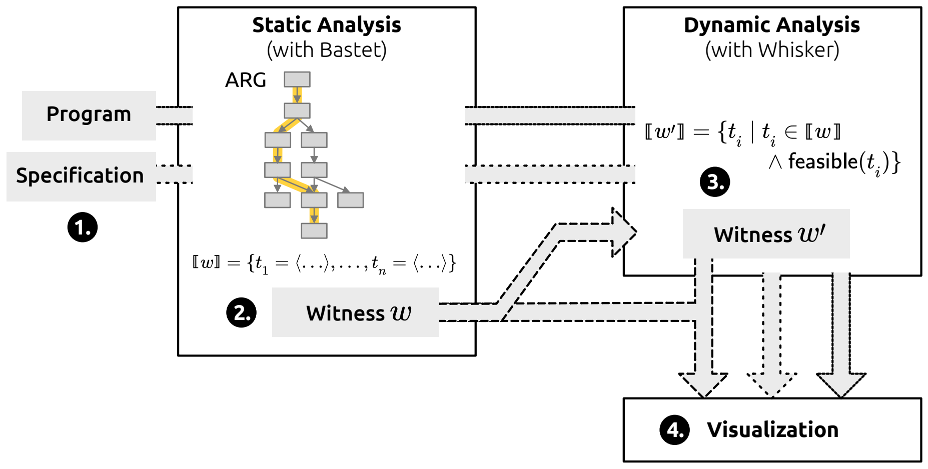

Static Analysis
(with Bastet)
ARG
Program
Specification
1.
$[\![w]\!] = \{t_1 = \langle\ldots\rangle, \ldots, t_n = \langle\ldots\rangle\}$
2.
Witness w
Dynamic Analysis
(with Whisker)
$[\![w']\!] = \{t_i \mid t_i \in [\![w]\!] \wedge \mathsf{feasible}(t_i)\}$
3.
Witness w'
4. Visualization

Fig. 1. Generation, verification, and visualization of Scratch error witnesses

Model checking has been suggested as a solution for tackling these challenges [19], but verification results such as counterexamples are abstract and neither suitable for interpretation by learners, nor for application in dynamic analysis tools that aim to generate explanations or hints.

Observing program executions in terms of the user interactions and their graphical responses is potentially a more intuitive way to communicate counterexamples to learners, as it hides all details of the internal models of the analysis and verification tool and instead shows what a user would see. In this paper we therefore introduce the concept of Scratch *error witnesses* as a means to explain the presence of specification violations, and describe an automatic approach for extracting error witnesses for Scratch programs from counterexamples. Scratch error witnesses describe sequences of timed inputs (e.g., mouse and keyboard inputs) to Scratch programs leading to a program state that violates the specification. Error witnesses are, essentially, UI tests, and thus enable any form of dynamic analysis to help produce more elaborate explanations or feedback, such as fault localization or generation of fix suggestions.

Figure 1 provides an overview of the overall process of generating, verifying, and visualizing error witnesses. A Scratch program and its formal specification is given 1. to the static analysis tool, in this case to Bastet [19]. To analyze the program, Bastet constructs an abstract reachability graph (ARG), which represents an overapproximation of all possible states and behaviors of the program—a node in this graph is an abstract state, representing a set of concrete program states. When Bastet runs into an abstract state in which the specification is violated, an error witness w is produced. An abstract witness may represent multiple concrete test candidates, and depending on the analysis configuration of Bastet (for example, model checking, or data-flow analysis), some of these may be false positives. To increase confidence in the witness, it is handed over 2. to a dynamic analysis (in this case WHISKER [20]), which runs the tests that are described by the witness, and produces a new error witness w' 3., with $\llbracket w' \rrbracket \subseteq \llbracket w \rrbracket$. In case none of the tests in w are feasible, the result is an empty witness, that is, $\llbracket w' \rrbracket = \emptyset$. With this increased confidence in the presence of a specification violation, the refined witness can be visualized 4. for the user, without reducing his or her trust in the analysis results. Trust in analysis results is crucial, for example, for learners who are not familiar with program analysis, and for automated grading or feedback approaches.

Error witnesses do not only have the potential to aid the user in understanding the misbehavior of a program, but they can also be exchanged among different verification tools [3]. This makes it possible to take advantage of the complementary strengths of both dynamic and static analyses. For example, one can use dynamic analysis for verifying the witnesses, for applying fault localization to narrow down the origin the failure, for generating fixes and repair suggestions, or for guiding the state-space exploration to reach a particular state. Error witnesses thus lay the foundations for future research on presenting counterexamples for specification violations in SCRATCH programs to users.

2 Preliminaries

We stick to the notation that is used in recent work on formalizing SCRATCH programs [19,20]. Uppercase letters $A, \ldots, Z$ denote *sets*, lowercase letters $a, \ldots, z$ denote set *elements*. *Sequences* are enclosed in angled brackets $\langle a_1, a_2, \ldots \rangle$, *tuples* are enclosed in parentheses $(a_1, b_1, \ldots)$, *sets* are enclosed in curly braces $\{a_1, \ldots\}$. Symbols with an overline $\overline{a}$ denote sequences, lists, or vectors. Symbols with a hat $\hat{a}$ denote sets. Symbols with a tilde $\widetilde{a}$ denote relations. The set of all *finite words* over an alphabet A is denoted by A^*, the set of all *infinite words* by A^ω.

Scratch Program A SCRATCH program *App* is defined by a set $\mathcal{A}$ of *actors*. There is at most one actor that fills the role of the *stage* and several other actors that are in the *sprites* role [16]. An actor [19] can be instantiated several times; each actor instance is represented by a list of processes. A *concrete state* $c \in C$ of a program is a list of concrete process states $c = \langle p_1, \ldots, p_n \rangle$. A process state $p_i : X \to V$ is a mapping of typed program variables $x \in X$ to their values $v \in V$.

A *concrete program trace* is a sequence $\overline{c} \in C^\infty$ of concrete program states. The set of all possible *concrete program traces* $C^\infty = C^* \cup C^\omega$ consists of the set of finite traces C^* and the set of infinite traces C^ω [19]. The semantics $\llbracket App \rrbracket$ of a SCRATCH [16, 19] program *App* are defined by the set of concrete program traces it exhibits, that is, $\llbracket App \rrbracket \subseteq C^\infty$.

SCRATCH programs and their actors have a well-defined set of programs with defined meaning, along with user-defined variables. The variables are either actor-local or globally scoped. The set of actor-local variables of sprite actors includes, for example, the variables $\{\mathsf{x}, \mathsf{y}, \mathsf{direction}\}$, which define the position and orientation of a sprite.

Abstract Domain To cope with the restrictions of reasoning about programs, abstraction is needed [8]. Multiple concrete states can be represented by an abstract state. The *abstract domain* $D = (C, \ddot{E}, \langle\!\langle\cdot\rangle\!\rangle, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle^{\pi}, \Pi)$ [19] determines the mapping between abstract states E and concrete states C. An *inclusion relation* between the abstract states E is defined by the partial order $\sqsubseteq \subseteq E \times E$ of the *lattice* $\ddot{E} = (E, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$. The mapping between the abstract and concrete world is realized in the *concretization* function $[\![\cdot]\!] : E \to 2^C$ and the *abstraction* function $\langle\!\langle\cdot\rangle\!\rangle : 2^C \to E$. The *widening* function $\langle\!\langle\cdot\rangle\!\rangle^{\pi} : E \times \Pi \to E$ computes an abstraction of a given abstract state by removing irrelevant details according to the *abstraction precision* $\pi \in \Pi$ by defining an equivalence relation $\pi : C \to 2^C$. We also use formulas $\mathcal{F}$ in predicate logic to describe sets of concrete states: a *formula* $\phi \in \mathcal{F}$ denotes $[\![\phi]\!] \subseteq C$ a set of concrete states.

Abstract Reachability Graph A reachability analysis constructs an abstract reachability graph to determine whether or not a target state is reachable; it proves the absence of such a state if a fixed point is reached, that is, all states have been visited. An *abstract reachability graph* is a directed graph $\mathcal{R} = (E, e_0, \rightsquigarrow)$ of abstract states E rooted in the *initial abstract state* $e_0 \in E$. The structure of the reachability graph $\mathcal{R}$ is determined by its transition relation $\rightsquigarrow \subseteq E \times E$ and we write $e \rightsquigarrow e'$ iff $(e, e') \in \rightsquigarrow$. An *abstract (program) trace* is a finite sequence $\overline{e} = \langle e_0, \ldots, e_{n-1}\rangle$ where each pair $(e, e') \in \overline{e}$ is an element of the transition relation $\rightsquigarrow$. Each abstract (program) trace $\overline{e}$ denotes a possibly infinite set of concrete program traces $[\![\overline{e}]\!] \subseteq C^{\infty}$. An abstract trace is *feasible*, if and only if $[\![\overline{e}]\!] \cap [\![App]\!] \neq \emptyset$, otherwise it is *infeasible*.

Each transition $e \overset{\overline{op}}{\rightsquigarrow} e'$ of the abstract reachability graph can be labeled with a sequence $\overline{op} = \langle op_1, \ldots, op_n\rangle \in Op^*$ of program operations executed to arrive at the abstract successor state e'. The set of program operations Op consists of operations of various types, which can manipulate or check the set of program variables [19]. A Scratch block corresponds to a sequence of operations from Op. To simplify the description, we extend Op with call and return operations, where a call represents the beginning of the execution of such a sequence of operations, and a return marks its end.

Static Reachability Analysis A static analysis (typically) conducts a reachability analysis by creating an overapproximation [8] of all possible states and state sequences of the program under analysis. The resulting abstract reachability graph $\mathcal{R}$ possibly denotes (in case the analysis terminated with a fixed point) a larger set of program traces than the original program has, that is, $[\![App]\!] \subseteq [\![\mathcal{R}]\!]$. An example for a static analysis framework is Bastet [19], which focuses on analyzing Scratch programs. An operator $\mathsf{target} : E \to 2^{\mathcal{S}}$ determines the set of properties that are considered violated by a given abstract state.

Dynamic Reachability Analysis Dynamic program analyses are also a form of reachability analysis: The program under execution is steered by an input generator and its behavior is observed by a monitor process. The tool Whisker [20] guides a Scratch program App in its original execution environment by sending user inputs or providing mocks for functions that interact with the environment, while observing the resulting behavior of App. No abstract semantics are used. In contrast to static reachability analysis the results are always sound.

3 Scratch Error Witnesses

In general, an error witness is an *abstract entity* that describes inputs from the user and the environment to reproduce (to witness) the presence of a specification violation [3]. By not defining all inputs explicitly and keeping them nondeterministic, the degree of abstractness can be varied: An error witness can be *refined* by making more inputs deterministic, and it can be abstracted by increasing nondeterminism. In this work, we aim at error witnesses for Scratch programs that can be produced and consumed by both static and dynamic analyses, and that are easy to visualize and follow by users, for example, by novice programmers. Scratch error witnesses perform actions that could potentially also be conducted by a user controlling the Scratch program manually and provide means to mock parts of the Scratch environment to control input sources that would behave nondeterministically otherwise.

Note that while a Scratch program can exhibit infinite program traces, the counterexamples and error witnesses we discuss in this work are finite, that is, describing the violation of safety properties—witnessing that something bad (undesired) can happen after finitely many execution steps. We do not consider this to be a practically relevant limitation of our approach since bounded liveness properties—requiring that something good happens within a finite time span—are also safety properties.

A Scratch *error witness* is a tuple that defines inputs from the user and the environment to reproduce a specification violation in a particular program. Formally, it is a tuple $(\widetilde{m}, \overline{u}, s) \in W$ consisting of a *mock mapping* $\widetilde{m} : Op \rightarrow M$, a finite sequence of timed user interface *inputs* $\overline{u} = \langle u_1, \ldots, u_n \rangle \in U^*$, and the *property* $s \in \mathcal{S}$ that is supposed to be violated. The mock mapping is a partial function from the set of operations Op to the mocks M by which to substitute the functionality. A *timed user input* $u = (d, a) \in \mathbb{R} \times A$ is a tuple consisting of an *input delay* $d \geq 0$ in milliseconds, and an *action* $a \in A$ to perform after the delay d elapsed. Note that we abstract from the fact that one mock instance can replace operations of Op of several actor instances. The set of all error witnesses is denoted by W.

For debugging purposes, a timed user input can be enriched by an *expected state condition* $p \in \mathcal{F}$, which is a formula in predicate logic on the state of a Scratch program that characterizes the states that are expected to be reached after conducting the action, that is, $[\![p]\!] \subseteq C$. The expected state condition can be used to (1) check if the witness replay steers the program execution to the expected state space region, and to (2) provide details to the user on the sequence of concrete program states leading to the specification violation.

3.1 User Inputs

Scratch programs are controlled by the user mainly using mouse and keyboard input actions. To specify possible input actions, we adopt an existing grammar [20] to formulate such actions—with the natural numbers $\mathbb{N}$ and the set of Unicode characters $\mathbb{L}$. An input action $a \in A$ is built based on the following grammar:

$$\begin{aligned}
\mathit{input} &= \textsf{Epsilon} \mid \textsf{KeyDown}\ \mathit{key} \mid \textsf{KeyUp}\ \mathit{key} \mid \textsf{MouseDown}\ \mathit{pos} \mid \\
&\quad\ \textsf{MouseUp}\ \mathit{pos} \mid \textsf{MouseMoveTo}\ \mathit{pos} \mid \textsf{TextInput}\ \mathit{text} \\
\mathit{key} &= \textsf{keycode}\ \mathit{code} \\
\mathit{pos} &= \textsf{xpos}\ x\ \textsf{ypos}\ y \\
\mathit{text} &= \textsf{txt}\ \mathit{string} \\
\mathit{code} &\in \mathbb{N},\ \mathit{string} \in\ \mathbb{L}^*,\ x \in\ [-240..240],\ y \in [-180..180]
\end{aligned}$$

3.2 Mocks

Mocks replace specified operations in specified actor instances to control the program execution and steer it towards a target state. Compared to a stub, a *mock* is stateful, that is, the value returned by the mock and side effects can be different from call to call, depending on its *internal state*.

A Scratch block (represented by a sequence of operations from Op) that is supposed to return a new random number with each call (a random number generator) is a typical example that has to be mocked to reproduce a particular behavior. That is, any block that leads to some form of nondeterministic program execution is a good candidate to be mocked. Scratch allows to add various (custom) extensions—to use Scratch for programming hardware components, such as Lego Mindstorms—that add additional variables (or inputs) that require mocking. For example, to sense the motor position, distance, brightness, or acceleration. Even mocking date or time functions might be necessary to reproduce a specific behavior within a dynamic analysis.

We distinguish between different types of mocks. The set of all possible mocks is denoted by the symbol M.

Conditional Effects A *mock with conditional effects* $(\overline{op}_0, \overline{p}, \overline{\overline{op}}, \overline{r})$ is initialized by a sequence of program operations $\overline{op}_0 \in Op^*$ before its first invocation, describes a sequence of state-space conditions $\overline{p} = \langle p_1, \ldots, p_n \rangle \in \mathcal{F}^*$, and has a sequence of assignment sequences $\overline{\overline{op}} = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle \in (Op^*)^*$ and a sequence of mock return values $\overline{r} = \langle r_1, \ldots, r_n \rangle \in V^*$. An initialization operation $op \in \overline{op}_0$ can, for example, declare and initialize mock-local variables to keep track of the mock's state between different invocations. We require that $|\overline{\overline{op}}| \in \{|\overline{p}|, 0\}$ and $|\overline{r}| \in \{|\overline{p}|, 0\}$ (a mock might not produce a return value, or might not conduct any operations but return a value). In case the current program state c is in one of the regions described by a state-space condition p_i when the mock is invoked, that is, if $c \in [\![p_i]\!]$, then also the operation sequence $\overline{op}_i$ is performed and the value r_i returned. A condition p is a formula in predicate logic over the program's variables—including those local to the current actor or mock, and global variables. A nondeterministic (random) value is returned in case none of the conditions $p_i \in \overline{p}$ was applicable for an invocation.

Mocks with sequential effects and those with timed effects are specializations of mocks with conditional effects:

Algorithm 1 testGen(App)

```
Input: A Scratch program $App$ to verify
Output: A set of error witnesses $W$ (empty if the program is safe)
1: $(\mathsf{frontier}, \mathsf{reached}) \leftarrow \mathsf{init}_{App}()$
2: $(\cdot, \mathsf{reached}) \leftarrow \mathsf{wrapped}(\mathsf{frontier}, \mathsf{reached})$
3: $\mathsf{targets} \leftarrow \{e \mid \mathsf{target}(e) \neq \emptyset \wedge e \in \mathsf{reached}\}$
4: return $\bigcup_{t \in \mathsf{targets}} \mathsf{toWitness}(\mathsf{testify}_1^{\natural}(\mathsf{reached}, t))$
```

Sequential Effects A *mock with sequential effects* $(\overline{\overline{op}}, \overline{r}) \in (Op^*)^* \times V^*$ describes a sequence of assignment sequences $\overline{\overline{op}} = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$ and a sequence of mock return values $\overline{r} = \langle r_1, \ldots, r_n \rangle$, both with the same length, that is, $|\overline{\overline{op}}| = |\overline{r}|$. The mock has an internal state variable x that tracks the number of the mock's invocations and corresponds to the position in the sequences. That is, at invocation x, the sequence of assignments $\overline{op}_x$ is performed and the value r_x is returned.A nondeterministic (random) value is returned in case the position x is out of the sequences bounds.

Timed Effects A *mock with timed effects* $(\overline{y}, \overline{\overline{op}}, \overline{r}) \in (\mathbb{R} \times \mathbb{R})^* \times (Op^*)^* \times V^*$ describes a sequence of disjoint time (in milliseconds) intervals $\overline{y} = \langle y_1, \ldots, y_n \rangle$, a sequence of assignment sequences $\overline{\overline{op}} = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$, and a sequence of mock return values $\overline{r} = \langle r_1, \ldots, r_n \rangle$. In case the milliseconds $up \in \mathbb{R}$ since the program under test was started is in one of the time intervals y_i, then also the operation sequence $\overline{op}_i$ is performed and the value r_i returned when the mock is invoked.

4 Witness Generation

After we have introduced the notion of a user interface error witness for Scratch programs, we now describe how such a witness can be derived from a concrete program trace that violates the specification—which can be recorded by a dynamic analysis tool such as Whisker and from the abstract reachability graph produced by a static analysis framework such as Bastet.

4.1 Concrete Program Trace from an Abstract Reachability Graph

We first describe how a finite concrete program trace $\overline{c} = \langle c_1, \ldots, c_n \rangle \in C^\infty$ that leads to a (violating) target state $e_t \in E$, with $\mathsf{target}(e_t) \neq \emptyset \wedge c_n \in [\![e_t]\!]$, can be extracted from an abstract reachability graph. This process is typically implemented in a procedure for model checking or model-based test generation.

The outermost algorithm of a model checker with test generation is outlined in Alg. 1. All abstract states that have been reached by the analysis can be found in the set $\mathsf{reached} \subseteq E$, the set $\mathsf{frontier} \subseteq \mathsf{reached}$ contains all abstract states from which successor states remain to be explored. These sets are initialized by the operator init with the *initial abstract states* to analyze the program App. The actual reachability analysis is performed by the wrapped algorithm, represented by the method $\mathsf{wrapped}$, which can, for example, conduct an analysis based on predicate abstraction [13] and counterexample-guided abstraction refinement [7]. This wrapped (pseudo) algorithm terminates when it has reached a fixed point without reaching a violating state or after one or more violations have been identified. The set $\mathsf{targets} \subseteq E$ contains all states that violate the specification.

An abstract reachability graph $\mathcal{R} = (E, e_0, \rightsquigarrow)$ describes the predecessor-successor-relation of the states in this set—represented by the transfer relation $\rightsquigarrow \subseteq E \times E$. An abstract state represents a set of concrete states, that is, $[\![e]\!] \subseteq C$. A sequence $\overline{e} = \langle e_0, \ldots, e_{n-1} \rangle \in E^*$ of abstract states that starts in an initial abstract state e_0 and that is well-founded in the transfer relation $\rightsquigarrow$ is called an *abstract program trace*. An abstract program trace $\overline{e}$ represents a set of concrete program traces, i.e., $[\![\overline{e}]\!] \subseteq C^*$. That is, to get to a concrete program trace $\overline{c}$ that reaches a target state $e \in E$, we first have to select a feasible abstract program trace from graph $\mathcal{R}$, and can then concretize this trace. An abstract program trace is called *feasible* if it denotes at least one concrete program trace. Note that an abstract reachability graph can also contain abstract states that do not have a counterpart in the real world, that is, which are infeasible.

Generic Analysis Operators We define a list of new analysis operators in line with the configurable program analysis framework [5, 19] to extract abstract program traces and concrete program traces from a given set of reached states, reaching a target state:

1. The *abstract testification operator* $\mathsf{testify} : 2^E \times E \to 2^{E^*}$ returns a collection of abstract program traces. Given a set of abstract states $R \subseteq \mathsf{reached}$ and a target state $e_t \in E$, this analysis operator returns only *feasible* program traces—describing only *feasible sequences* of abstract states, all starting in an initial abstract state, and all leading to the given target state e_t. That is, all infeasible traces that would lead to the target are eliminated by this operator. An empty collection is returned in case the given target state is infeasible.

The *abstract single testification operator* $\mathsf{testify}_1 : 2^E \times E \to 2^{E^*}$ strengthens the operator $\mathsf{testify}$ and describes *at most one feasible* abstract program trace.

2. The *concrete testification operator* $\mathsf{testify}^\natural : 2^E \times E \to 2^{C^*}$ returns all *concrete* program traces reaching a given target state. Note that, assuming unbounded value domains, this collection can have infinitely many elements.

The *concrete single testification operator* is supposed to return at most one concrete program trace that reaches the given target state and has the signature $\mathsf{testify}_1^\natural : 2^E \times E \to 2^{C \times C}$.

Note that these operators do not guarantee any particular strategy for choosing abstract or concrete program traces. Nevertheless, different implementations or parameterizations of these operators can be provided that realize different strategies—contributing to the idea of configurable program analysis.

Operator Implementations The implementations of the testification operators vary depending on the composed analysis procedure and its abstract domain—see the literature [5, 19] for details on composing analyses. We provide a first implementation of these operators in the BASTET program analysis framework, in which program traces are chosen arbitrarily.

For a bounded model-checking configuration that does not compute any (block) abstractions, concrete program traces can be produced simply by asking an SMT

Algorithm 2 $\mathsf{toWitness} : C^* \to 2^W$

```
Input: A concrete program trace $\overline{c} \in C^*$
Output: A set of error witnesses $\in 2^W$
1: $\widetilde{m} \leftarrow \mathsf{constructMocks}(\overline{c})$
2: $\overline{u} \leftarrow \mathsf{constructInputSeq}(\overline{c})$
3: return $\{(\widetilde{m}, \overline{u}, s) \mid s \in \mathsf{target}(t)\}$
```

Algorithm 3 $\mathsf{constructInputSeq}_{\mathrm{Ax}} : C^* \to U^*$

```
Input: A concrete program trace $\overline{c} \in C^*$
Output: A timed input sequence $\in U^*$
1: $\overline{\overline{a}} \leftarrow \langle\rangle$
2: for $(op, c)$ in $\Gamma(\overline{c})$ do
3:     $\overline{\overline{a}} \leftarrow \overline{\overline{a}} \circ \langle \mathsf{choose}(\{\ \mathrm{ax}(op, c) \mid \mathrm{ax} \in \mathrm{Ax}\ \}) \rangle$
4: return $\mathsf{foldEpsilonDelays}(\overline{\overline{a}}, \overline{c})$
```

solver for a satisfying assignment (a model) for a formula with which a violating state is supposed to be reached.

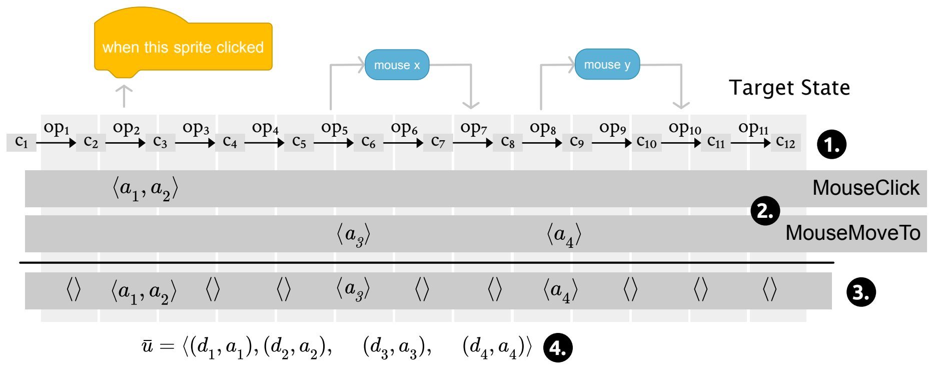
4.2 Error Witness from a Concrete Program Trace

We now describe how Scratch error witnesses $(\widetilde{m}, \overline{u}, s) \in W$ can be produced from a given finite concrete program trace $\overline{c} = \langle c_1, \ldots, c_n \rangle \in C^*$. Such a trace can be created from a model checking run using one of the proposed testification operators, or can be created from the states observed while running the program on a machine, e.g., along with a dynamic analysis. Algorithm 2 outlines the process of generating a Scratch error witness from a concrete program trace.

We assume that there is a transition labelling function $\Gamma : C \times C \to Op^*$ for labelling state transitions. Given a pair $c_1, c_2 \in C$ of concrete states, the function returns a (possibly empty) sequence $\langle op_1, \ldots, op_n \rangle$ of program operations conducted to reach from state c_1 to state c_2. We extend the labelling function to sequences, resulting in an overloaded version $\Gamma : C^* \to (Op \times C)^*$ that produces sequences of pairs of program operations and concrete (successor) states; the first concrete state in the given concrete program trace is skipped.

Timed Inputs A witness contains the sequence of timed user inputs $\overline{u} = \langle u_1, \ldots, u_n \rangle \in U^*$, where each element $u_i = (d, a) \in \overline{u}$ consists of a delay $d \in \mathbb{R}$ (in milliseconds) to wait before conducting an input action a. Algorithm 3 outlines the process of creating this sequence and Fig. 2 provides a visual perspective on the process and the example to discuss. The algorithm is implicitly parameterized with a collection of action extractors Ax. Generally, there is one action extractor ax $\in$ Ax for each class of input action—see the grammar of input actions in Sect. 3.1. In our example, we use 2. an extractor for the action MouseMoveTo and a *composite* action extractor MouseClick that produces two different actions (MouseDown and MouseUp, resulting in a "mouse click").

The algorithm starts from a given 1. concrete program trace $\overline{c} \in C^*$ leading to a target state that violates one or more properties $\subseteq \mathcal{S}$. The trace is



when this sprite clicked
mouse x
mouse y
Target State
$c_1 \xrightarrow{op_1} c_2 \xrightarrow{op_2} c_3 \xrightarrow{op_3} c_4 \xrightarrow{op_4} c_5 \xrightarrow{op_5} c_6 \xrightarrow{op_6} c_7 \xrightarrow{op_7} c_8 \xrightarrow{op_8} c_9 \xrightarrow{op_9} c_{10} \xrightarrow{op_{10}} c_{11} \xrightarrow{op_{11}} c_{12}$
1.
$\langle a_1, a_2 \rangle$
MouseClick
2.
$\langle a_3 \rangle$
$\langle a_4 \rangle$
MouseMoveTo
$\langle\rangle$ $\langle a_1, a_2 \rangle$ $\langle\rangle$ $\langle\rangle$ $\langle a_3 \rangle$ $\langle\rangle$ $\langle\rangle$ $\langle a_4 \rangle$ $\langle\rangle$ $\langle\rangle$ $\langle\rangle$
3.
$\bar{u} = \langle (d_1, a_1), (d_2, a_2), (d_3, a_3), (d_4, a_4) \rangle$
4.

Fig. 2. Generation of the sequence of timed inputs in Bastet

```
Algorithm 4 constructMocks_Mx : $C^* \to 2^{Op \times M}$
Input: A concrete program trace $\bar{c} \in C^*$
Output: A mock mapping $\subseteq Op \times M$
1: return $\{\mathrm{mx}(\bar{c}, \Gamma(\bar{c})) \mid \mathrm{mx} \in \mathrm{Mx}\}$
```

traversed from its start to the end (with the target) state, and the action extractors are invoked along this trace. A call to the action extractor for a given concrete state c that is reached by a program operation op returns a sequence of input actions $\bar{a}$ to execute at this point in the resulting witness. For example, the actions $\langle a_1, a_2 \rangle$ are produced by the MouseClick action extractor for the operation op_2 reaching state c_3, with a_1 = MouseDown xpos 23 ypos 8 and a_2 = MouseUp xpos 23 ypos 8. This action sequence is emitted because operation op_2 signaled a click to the sprite, the mouse position is extracted from the concrete state c_3. MouseMove actions are produced whenever the mouse is expected to be on a particular position, for example, queried by a $\boxed{\text{mouse x}}$ or $\boxed{\text{mouse y}}$ Scratch block. The result 3. of applying the action extractors along the trace is a sequence $\bar{\bar{a}} \in A^{**}$ of sequences of input actions. In case multiple action extractors provide a non-empty sequence for a particular position along the trace, the operator choose chooses an action sequence based on priorities. In the last step 4., empty elements (containing an empty sequence) are eliminated from $\bar{\bar{a}}$ and a delay is added that determines how long to wait before executing a particular action. This functionality is provided by the function foldEpsilonDelays.

Mock Mappings A Scratch error witness contains a mock mapping $\widetilde{m} : Op \to M$, which specifies mocks used for substituting particular operations of the program or the runtime environment to steer a program execution (or a state space traversal) towards a target state that violates the specification.

The creation of the mocks from a given concrete program trace is implemented in the function constructMocks, which is outlined in Alg. 4. The algorithm is implicitly parameterized by a list of mock extractors Mx. A *mock extractor* is a

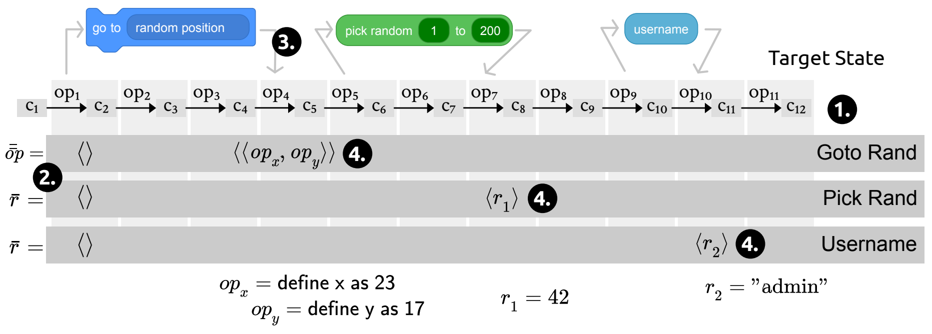

go to random position
3.
pick random 1 to 200
username
Target State
c1 op1 c2 op2 c3 op3 c4 op4 c5 op5 c6 op6 c7 op7 c8 op8 c9 op9 c10 op10 c11 op11 c12
1.
$\bar{op} =$ ⟨⟩ ⟨⟨op_x, op_y⟩⟩ 4. Goto Rand
2.
$\bar{r} =$ ⟨⟩ ⟨r_1⟩ 4. Pick Rand
$\bar{r} =$ ⟨⟩ ⟨r_2⟩ 4. Username
op_x = define x as 23
op_y = define y as 17
$r_1 = 42$
r_2 = "admin"

Fig. 3. Generation of mocks in Bastet

function that creates a mock for a Scratch block or a function of the runtime environment based on a given trace. Section 3.2 already motivated why we need mocks for Scratch programs, and discussed mocks with different degrees of expressiveness. Typically, we have one mock extractor for each block that interacts with the environment (the operating system, the runtime environment, connected hardware components).

Figure 3 illustrates the process of generating mocks based on a given concrete program trace 1. leading to a target state, which violates one or more properties $\subseteq \mathcal{S}$. Three mock extractors are in place: Goto Rand produces a mock for the Scratch block go to random position, Pick Rand produces a mock for pick random (..) to (..), and Username produces a mock for the block username. All mock extractors operate by consuming the input trace from left-to-right, starting with an empty mock 2., and then enriching it from step-to-step. In contrast to action extractors, mock extractors determine their behavior *after* the observed block returns 3. to the calling block, then, the mock is updated based on the concrete state found at that point in the trace 4..

Note that each mock extractor can produce another type of mock—see Sect. 3.2 for mock types. The mock extractor Goto Rand returns a mock with sequential effects and operation sequences to perform: It assigns new values to the sprite's variables x and y in each invocation, and does not have a return value. The extractor Username returns a mock with conditional effects: This mock returns the value "admin" in case the condition *true* applies, that is, always.

5 Evaluation

We illustrate the practicality of generating and replaying (validating) UI error witnesses for Scratch programs. In particular, we are interested if our concepts are *effective* and if they contribute to a more *efficient* tool chain to show the presence of bugs in UI centered programs.

5.1 Experiment Setup

Implementation We implemented the concepts presented in this paper in the static program analysis framework Bastet [19] and in the dynamic analysis

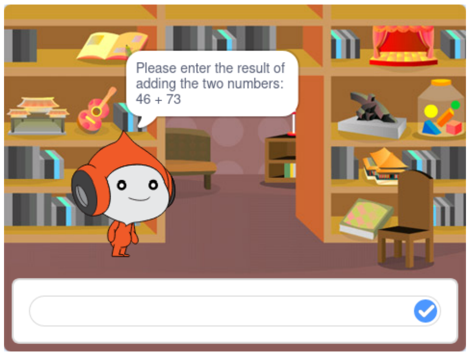


Fig. 4. The Brain Game example program

tool WHISKER [20]. We added support to generate error witnesses from an abstract reachability graph into BASTET, and enriched WHISKER with support for replaying these witnesses. We also defined a witness exchange format based on JSON to exchange error witnesses between analysis tools.

Benchmarking Environment Students and teachers in educational contexts such as schools typically do not have access to large computing clusters. For this reason, we tried to aim for a more practical setting and conducted our experiments on a single desktop workstation featuring an Intel(R) Core(TM) i7-2600 processor with 3.40GHz and 32GiB of RAM (although as little as 2 to 4 GiB would have been sufficient for our case study). The machine runs Debian GNU/Linux 10 and the current LTS version of Node.js (v14.16.0 at the time of writing). Our additions to support SCRATCH error witness generation are implemented in BASTET (version `af0a20db`) and its replay in WHISKER (version `392712bf`). We used the Node.js API provided by Puppeteer[1] to control a browser and automatically stimulate our case study SCRATCH programs with user input.

Case Study SCRATCH is backed by a large online ecosystem and community. For example, Code Club[2] is a global network of free coding clubs for 9 to 13 year-olds with the aim of helping children develop programming skills in SCRATCH, among other languages. We took inspiration from one of their SCRATCH projects called "Brain Game"[3] and use it as a case study.

Here, the task is to implement a game with a quiz master asking the player for the result of five randomly chosen arithmetic computations (see Figure 4). Only a correct answer increases the player's score. The game ends when all questions were answered correctly (in which case a green check mark sprite is displayed) or when a wrong answer was given (in which case a black cross appears).

We chose Brain Game because its size and complexity are typical of the programs developed by learners. Moreover, it exhibits randomness and requires user interaction, which is challenging for program analysis tools.

[1] https://github.com/puppeteer/puppeteer [2] https://codeclub.org/en
[3] https://projects.raspberrypi.org/en/projects/brain-game-cc

Table 1. Effectiveness, execution times in seconds rounded to two significant digits

Variant	Replay Successful?	Replay Time
V1	×	3.0
V2	✓	4.0
V3	✓	8.0
V4	✓	4.1
V5	✓	5.0

Afterwards, we devised four properties that constitute our notion of a correct Brain Game implementation:

P1 The score must have been initialized with 0 before the first question is asked.
P2 The green check mark must be shown within 200ms when all questions were answered correctly.
P3 The black cross must be shown within 200ms when a question was answered incorrectly.
P4 The score must not decrease.

We formalized these properties as both LeILa [19] programs and Whisker tests. The former can be fed to Bastet with the aim of checking a given program against this specification and generating an error witness, and the latter is handed to Whisker to verify the error witness. We implemented five erroneous variants V1–V5, each violating one of the above properties:

V1 Violates P1: the score is not initialized at all.
V2 Violates P2: the sprite for the wrong answer is not shown when a question was answered incorrectly.
V3 Violates P3: the sprite for the correct answer is not shown when all five questions were answered correctly.
V4 Violates P4: the score decreases by one when an incorrect answer is given.
V5 Violates P4: the score decreases by one when an incorrect answer is given *except* when it would turn negative.

5.2 Witness Replay and Validation (Effectiveness)

Effectiveness describes the ability to replay and validate the statically generated witnesses by a dynamic analysis. To this end, we ran each of the five erroneous Brain Game variants along with the specification in Bastet and extracted the error witnesses. Then, we ran Whisker together with the specification and the Scratch error witness on each program under test to investigate if the witness generated by Bastet can be verified in Whisker.

The results are summarized in Table 1 and show that four out of five errors were reproducible. In detail, the violations of properties P2–P4 by programs V2–V5 were revealed via static analysis by Bastet and confirmed by dynamic replay in Whisker. V2 and V4 both require a wrong answer for the fault to be exposed. P3 requires five correct answers. Finally, V5 requires at least one correct answer followed by a wrong answer.

Table 2. Efficiency for different verification tasks, execution times in seconds rounded to two significant digits

Variant	Error Witness Generation and Replay				Random Input Generation (estimated)
	Analysis	Concretization	Replay	Combined	
V1	25	0.52	3.0	28	400
V2	50	0.83	4.0	54	400
V3	1500	10	8.0	1500	7.6×10^{10}
V4	47	1.0	4.1	51	400
V5	210	2.1	5.0	220	500

While V2–V5 were validated successfully, the tools disagree when it comes to the violation of property P1 by variant V1: Bastet detected a violation but this could not be confirmed by Whisker. When first reading the score variable, Bastet detects that it has not been initialized yet, thus deeming its usage unsafe and reporting a violation. This requires no user interaction and the generated replay contains no user input. When replaying the generated witness in Whisker, however, no violation is detected. This is because uninitialized variables in Scratch have a default value of 0 before the first program execution, which just so happens to be the value demanded by the specification. However, the violation *could* be detected by Whisker when at least one correct answer is given (thus increasing the score to at least 1) and the game is played for a second time, where the score would still be 1 as it is not reset from the previous game.

The failure to detect V1 highlights a limitation in our work: the current definition of a Scratch error witness only allows for mock mappings but not for setting the initial state of a Scratch program. While the formalism in Section 3 can be easily extended, more implementation work in Whisker is necessary to support this. We plan to address both issues in future work.

We conclude that Scratch error witness reuse among different tools is possible, but may reveal differences in implicit assumptions or approximations.

5.3 Sequential Tool Combination (Efficiency)

The second question we investigate is whether guiding a dynamic analysis by tests generated from a static analysis can increase the testing efficiency. For this purpose, we measured the combined execution times of Bastet and Whisker to generate and replay an error witness, and compare it against the expected average runtime of Whisker when purely unguided random input generation were to be used. Table 2 contains the results of this experiment.

Looking at the combined times in Table 2, we see that the fault in program variant V1 is easiest to reveal for Bastet since it requires no user interaction. V2 and V4 entail similar effort, both require one wrong answer. V5 requires a wrong and a correct answer and poses more challenges to Bastet, increasing verification time by one order of magnitude. V3 requires 5 correct answers; as this requires covering more program states, the additional analysis effort increases the time by another order of magnitude.

Table 3. Scaling experiment conducted on differently sized variants of program V3, execution times in seconds rounded to two significant digits

Variant	Analysis	Concretization	Replay	Combined
$V3_1$	57	0.90	5.9	63
$V3_2$	170	2.4	5.2	170
$V3_3$	410	4.7	6.0	420
$V3_4$	810	7.0	7.0	820
$V3_5$	1500	10	8.0	1500

To contrast this with the time it would take to reveal the faults using only random input generation in Whisker, we consider the average expected execution time of this inherently randomized approach: The space of possible answers to each question asked in Brain Game consists of $200 - 2 = 198$ numbers. (The two summands range between 1 and 100). In the best case scenario, Whisker manages to generate the correct answer on the first try. In the worst case scenario, there is no upper limit to how many tries are necessary. However, assuming that the random number generator produces evenly distributed numbers the average number of tries can be computed as $198/2 = 99$. Moreover, from Table 2 we can infer that a Scratch error witness replay for one question takes roughly 4 seconds. With this, the average execution time can be estimated as $4 \times 99 = 396$ seconds for V2 and V4. Similarly, for V5 (which requires one correct and one wrong answer), a wrong answer is given in one try on average but replay lasts longer (5 seconds). Thus, the estimated time is $5 \times (99 \times 1) = 495$ seconds.

To reveal the fault in V1, however, we would require at least one correct answer, followed by a restart of the game. The replay time for this cannot be extracted from the table (since we did not have user interaction) but using a conservative estimation of 4 seconds, similar to V2 and V4, the estimated execution time is also 396 seconds. Exposing the fault in V3 requires 5 correct answers in a row. An average number of 99^5 tries with a replay time of 8 seconds results in a total runtime of 7.6×10^{10} seconds, which is more than 2400 years.

While these results suggest the combined approach is more efficient, this depends on how Bastet's performance scales with increasing size of the programs to generate error witnesses for. We therefore analyze the impact of the size of the state space in Bastet on the verification time using four alternate versions $V3_1$, $V3_2$, $V3_3$ and $V3_4$ of V3 requiring one, two, three and four correct answers instead of five, respectively. We use $V3_5$ synonymously for V3. Afterwards, we generated error witnesses for each of the four new variants using Bastet. The run times are presented in Table 3. For each additional question asked, the results indicate that the verification time increases linearly by a factor of 2. Since error witness generation dominates the costs, the same increase can also be seen for the combined execution time.

Overall, the results indicate that guiding a dynamic analysis by tests generated from a static analysis can increase the testing efficiency, and scales well with increasing test program size.

6 Related Work

As it can be beneficial to hide the internal models of analysis and verification tools to support adoption by users or developers [21], the idea of producing executable tests from counterexamples has been revisited in different contexts over time. An early approach to produce executable tests from counterexamples [2] was implemented for the BLAST model checker [14], and many alternative approaches followed. For example, Rocha et al. [18] generate executable programs for counterexamples produced for C programs by ESBMC [9], Muller and Ruskiewicz [17] produce .NET executables from Spec# programs and symbolic counterexamples, Csallner and Smaragdakis [10] produce Java tests for counterexamples generated by ESC/Java [11], and Beyer et al. [4] presented an approach that converts verification results produced by CPAChecker [6] to executable C code. Our approach applies similar principles, but considers interactive, graphical programs, where verification tasks consider possible sequences of user interactions. Executable error witnesses for interactive programs with user interactions need to mock not only user inputs, but also other environmental dependencies. Gennari et al. [12] described an approach that also builds mock environments, but again targets C programs. Besides the interactive nature of UI error witnesses, a further property that distinguishes our problem from prior work is that we are considering timed traces. Timed counterexamples are produced, for example, by Kronos [22, 23] or Uppaal-Tron [15]; however, we are not aware of any approaches to produce executable tests from such counterexamples. Testification of error witnesses has not only been proposed for producing executable tests, but also as an interchange format for different verification tools [3]; again a main difference of our approach is that our interchange format considers UI error witnesses rather than C function invocations. Aljazzar and Leue [1] produced interactive visualizations of counterexamples to support debugging. By producing UI tests from UI error witnesses we achieve a similar goal: Users can observe program executions and the interactions with the program along described by the error witness.

7 Conclusions

This paper introduced the notion of error witnesses for programs with graphical user interfaces—controlled by mouse and keyboard inputs, sent at particular points in time. We illustrated our concepts and implementation in the context of the analysis of game-like programs that were developed using visual- and block-based programming, in Scratch. We (1) formalized the notion of UI error witnesses, (2) described how these witnesses can be generated from the abstract reachability graph that was constructed with an SMT-based (Satisfiability Modulo Theories) software model checker, and (3) demonstrated their practicality for confirming the presence of errors in a dynamic analysis.

The exchange of error witnesses between different verification tools opens up possibilities to develop hybrid approaches that increase efficiency. Our findings also indicate that error witnesses can be useful in order to cross-check and test tools. Besides the technical aspects, however, there also remains the larger problem of making UI error witnesses accessible and useful for learning programmers.

References

1. Aljazzar, H., Leue, S.: Debugging of dependability models using interactive visualization of counterexamples. In: QEST. pp. 189–198. IEEE Computer Society (2008)
2. Beyer, D., Chlipala, A., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: ICSE. pp. 326–335. IEEE Computer Society (2004)
3. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: ESEC/SIGSOFT FSE. pp. 721–733. ACM (2015)
4. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses - execution-based validation of verification results. In: TAP@STAF. Lecture Notes in Computer Science, vol. 10889, pp. 3–23. Springer (2018)
5. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: CAV. Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer (2007)
6. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. CoRR **abs/0902.0019** (2009)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000)
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. IEEE Transactions on Software Engineering **38**(4), 957–974 (2011)
10. Csallner, C., Smaragdakis, Y.: Check'n'crash: Combining static checking and testing. In: Proceedings of the 27th international conference on Software engineering. pp. 422–431 (2005)
11. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. pp. 234–245 (2002)
12. Gennari, J., Gurfinkel, A., Kahsai, T., Navas, J.A., Schwartz, E.J.: Executable counterexamples in software model checking. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 17–37. Springer (2018)
13. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. Lecture Notes in Computer Science, vol. 1254, pp. 72–83. Springer (1997)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 58–70 (2002)
15. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: EMSOFT. pp. 299–306. ACM (2005)
16. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The scratch programming language and environment. ACM Trans. Comput. Educ. **10**(4), 16:1–16:15 (2010)
17. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: International Symposium on Formal Methods. pp. 73–87. Springer (2011)

18. Rocha, H., Barreto, R., Cordeiro, L., Neto, A.D.: Understanding programming bugs in ansi-c software using bounded model checking counter-examples. In: International Conference on Integrated Formal Methods. pp. 128–142. Springer (2012)
19. Stahlbauer, A., Frädrich, C., Fraser, G.: Verified from scratch: Program analysis for learners' programs. In: ASE. IEEE (2020)
20. Stahlbauer, A., Kreis, M., Fraser, G.: Testing scratch programs automatically. In: ESEC/SIGSOFT FSE. pp. 165–175. ACM (2019)
21. Visser, W., Dwyer, M.B., Whalen, M.W.: The hidden models of model checking. Software and Systems Modeling **11**(4), 541–555 (2012)
22. Yovine, S.: Model checking timed automata. In: European Educational Forum: School on Embedded Systems. Lecture Notes in Computer Science, vol. 1494, pp. 114–152. Springer (1996)
23. Yovine, S.: KRONOS: A verification tool for real-time systems. Int. J. Softw. Tools Technol. Transf. **1**(1-2), 123–133 (1997)