Re-Use of Programming Patterns or Problem Solving? Representation of Scratch Programs by TGraphs to support Static Code Analysis

Mike Talbot
Technical University of Munich
TUM School of Education
mike.talbot@tum.de

Katharina Geldreich
Technical University of Munich
TUM School of Education
katharina.geldreich@tum.de

Julia Sommer
Technical University of Munich
TUM School of Education
julia.sommer@tum.de

Peter Hubwieser
Technical University of Munich
TUM School of Education
peter.hubwieser@tum.de

ABSTRACT

Novice programmers seem to learn basic programming skills amazingly fast by using visual programming environments like Scratch or Snap. Yet at a second glance, in many cases, the students' programming projects make use of pre-learned solution patterns like collision detection. Aiming to investigate how far such pre-learned patterns are used and adapted, we have to analyze the program structure of a substantially large number of Scratch projects, e.g. from the Scratch repository, in a very detailed way. To automate the static code analysis of these projects, we developed a scheme to transform Scratch projects into a common graph format (TGraph), which was used up to now to analyze programs in Java and Haskell as well as UML diagrams and mathematical solutions. In a second step, this representation enabled us to apply a SQL-like query language for graphs (GReQL) to detect programming patterns in students' Scratch projects. This paper describes the design of our TGraph scheme for Scratch as well as how to query patterns in Scratch code using GReQL, in order to stimulate the use of this methodology by other researchers. As a feasibility study, we report its application on the outcomes of one of our Scratch courses attended by 143 children aged 8-12 years. The study showed that with the presented methodology any code structure can be detected in Scratch projects. To check the validity of the methodology, the programs were additionally checked manually for the occurrence of two patterns - the results were consistent.

CCS CONCEPTS

• **Social and professional topics** → *K-12 education*; • **Software and its engineering** → *Patterns*; *Software verification and validation.*

KEYWORDS

Computer Science Education; Assessment; Programming; Scratch; Patterns, Code Analysis

ACM Reference Format:
Mike Talbot, Katharina Geldreich, Julia Sommer, and Peter Hubwieser. 2020. Re-Use of Programming Patterns or Problem Solving? Representation of Scratch Programs by TGraphs to support Static Code Analysis. In *Workshop in Primary and Secondary Computing Education (WiPSCE '20), October 28–30, 2020, Virtual Event, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3421590.3421604

WiPSCE '20, October 28–30, 2020, Virtual Event, Germany

ACM ISBN 978-1-4503-8759-0/20/10…$15.00
https://doi.org/10.1145/3421590.3421604

1 INTRODUCTION

The discussion about the necessity of computer science (CS) and especially programming in primary education has grown steadily in recent years [4][34]. At the same time, visual programming environments for educational purposes like Scratch, Snap or Blockly have become very popular for teaching introductory programming. One reason for this may be that no knowledge of programming syntax is required, and no compile-time errors can occur [33]. Thus, the students do not receive frustrating error messages. On the other hand, environments such as Scratch are intentionally designed to facilitate engagement among students by supporting many types of projects and making it easy to personalize them [24]. Research even shows that Scratch can improve learning outcomes in disciplines outside CS [21].

During their first steps in programming, students are frequently exposed to existing code fragments that they try to understand or to adapt [5][25][32]. In consequence, when it comes to solving more complex problems, some students may tend to apply or adapt previously learned solution patterns, while others could try to construct their own individual solution from scratch. For example, Smith et al. [27] report that children in *Code Club* had difficulties in making the transition from following guided instructions in Scratch to applying knowledge to unguided open-ended tasks. To investigate how far and under which circumstances students tend to use predefined solutions, we aim to examine the program structure of a substantially large number of Scratch projects.

Besides the investigation of solution patterns on a large scale, we're aiming to analyze Scratch code online in an introductory

MOOC on object-oriented programming [18][19] to provide enriched automated feedback to students. In this MOOC we already use the e-assessment system JACK [13] to analyze Java code.

Our general research question is to what extent and in what form novice programmers reuse solution patterns to solve problems in different contexts using visual programming languages. For this purpose, we have to analyze larger numbers of student projects, e.g. taken from the Scratch repository or provided by MOOCs. To enable such large scale investigations, we need to apply an automated code analysis. The basic idea for this analysis is to detect programming (solution) patterns by searching for certain corresponding combinations of instructions and control structures in the code of the students' programming projects.

To perform this automated code analysis, we have adopted a set of tools that have been developed for the analysis of programs written in textual languages like Java. First, we need to represent the program structure in a way that is easily accessible to automated analysis. For this purpose we have created a method to transform Scratch projects into the TGraph format[9]. Second, we need a query language that operates on this representation. As it was already used to search on TGraphs, it seemed obvious to choose GReQL for this purpose, see [16]. Finally, a management system has to combine these tools and to provide user access. As the automated assessment system JACK was already using TGraphs and GReQL, it seemed natural to use it for our purpose.

In the following, we outline our methodology to find solution patterns in programming projects. In addition, we present a feasibility study that validates the methodology and demonstrates some exemplary application cases. We address the following research questions:

- How can the existing methodology for analyzing textual programming languages be adapted to Scratch?
- To what extent are TGraphs suitable for finding patterns in Scratch projects?

2 THEORETICAL BACKGROUND AND RELATED WORK

2.1 Learning Progress and Visual Programming Languages

Various visual programming languages - especially those developed for the educational context - are widely known for encouraging engagement, creativity and tinkering among novice programmers [37]. At the same time, research is increasingly focusing on the question of which CS concepts students learn in these environments and what their learning progress looks like.

Meerbaum-Salant et al. [20] investigate whether the use of Scratch in extracurricular activities with secondary school students can facilitate the general learning of CS concepts (e.g. initialization, loops, variables). They discovered that students can learn CS concepts by programming with Scratch and that some concepts were learned more successfully than others.

To assess the CS concepts learned during a two-week summer camp, Franklin et al. [11] developed a technique to analyze the students' programming results as well as information from the supporting staff. At the end of the camp, students demonstrated competencies in event-driven programming, initialization of state, message passing and say/sound synchronization.

Related research also focuses on the difference in learning between visual and textual programming and whether it is easier for students to learn textual programming if they first program in a visual programming environment. Armoni et al. [2] found that students who had been introduced to CS concepts in middle-school with Scratch had a better understanding of these concepts when they later programmed with Java or C#. Weintrop and Wilensky [35] developed a commutative assessment to find out how learning block-based programming differs in conceptual understanding compared to learning text-based programming languages. They asked 90 high-school students questions about short programs – half in text form, the other half in block-based form. Their analysis revealed differences in the performance and in misconceptions based on the programming modality.

Grover at al. [15] designed and tested a seven-week introductory CS course for middle school, called "Foundations for Advancing Computational Thinking" (FACT). The course was conducted in several classroom settings as well as a MOOC. Their assessment of the courses showed significant learning gains in algorithmic thinking and changes in the students' perception of computing. They also found an increase in the students' ability to transfer their learning from Scratch to text-based programming languages. As an extension of their studies, they measured students understanding in introductory CS, identifying misconceptions and challenges in the use of variables as well as the way loops and boolean operator work [14].

In their study, Techapalokul and Eli Tilevich [31] analyzed over 600K projects from the Scratch repository for so-called code smells - recurring patterns and implementations that indicate software quality issues. They found several code smells (e.g. duplicate code, long script, unused variable) that reduce the likelihood of Scratch projects being remixed.

2.2 Solution Patterns

Design patterns are a very important part of software engineering, providing a common vocabulary describing a general solution for a shared problem. For object-oriented programming Gamma et al. [12] presented a catalog of design-patterns that has been widely expanded over the past. Design patterns as a recipe for solving a common programming problem are also considered helpful for programming beginners [7].

To identify patterns that occur in student programs, Amanullah and Bell [1] analyzed 212,250 projects from the Scratch repository. They proposed the use of "elementary patterns" that are intended to impart good programming habits and to teach problem-solving independently from a specific programming language. These patterns can be divided into loop patterns and selection patterns. Loop patterns include solution patterns like *process all items of a collection*, *linear search* to stop a loop when a condition is met or *polling loop* to iterate a sequence until a user-entered value is reached. Selection patterns contain *whether or not* for a single if statement, if-else statements are called *alternative action pattern*, the *independent choice* pattern solves problems related to nested loops. Other research put its focus on mapping solution patterns to aspects of

computational thinking (CT) [36]. Seiter and Foreman [26] introduced the Progression of Early Computational Thinking (PECT) model to measure computational thinking by the level of skills utilized in coding design patterns. As a part of their assessment, they introduce "Design Pattern Variables" as a set of contextual proficiencies based on coding patterns in Scratch such as *animate looks* or *motion*, *conversate*, *collide*, *maintain score* and *user interaction*. The proficiency level of these patterns is then assessed by a qualitative scale: basic, developing and proficient. For example, in the pattern *animate motion*, a simple change of location is considered basic, initializing and changing the position of a sprite is developing while the use of relative movement is considered proficient.

Koh et al. [17] describe a semantic analysis called "Program Behavior Similarity (PBS)" to analyze CT patterns in games created with the visual programming environment AgentSheets. PBS determines the similarity of games by calculating the behavioral similarity on a set of rules by which the game is programmed. For this purpose, they propose the Computational Thinking Pattern Graph (CTPG), which identifies nine canonical patterns that are used in games e.g. *cursor control*, *collision*, *hill climbing*. They designed a quiz to test if students are able to recognize these abstract CT patterns in other contexts such as science simulations. For example, the students were shown a video of two people sledding downhill and then asked about the connection to the game "Frogger", which was implemented with the patterns *transport* and *collision* [3].

2.3 Automated Analysis of Scratch Programs

For investigations of Scratch programs on a large scale such as the Scratch repository or to provide automated feedback to learners, various tools for automated code analysis of Scratch projects have been developed. Boe et al. [6] present Hairball, a basic static code analysis tool, that automatically analyzes Scratch 2 projects. It provides a plugin architecture that allows extending the functionality of Hairball. In a case study, they developed plugins to detect code patterns for *initialization*, *say and sound synchronization*, *broadcast/receive* and *complex animation*. Dr. Scratch [22] builds upon Hairball, providing a web-interface to analyze Scratch projects and offering feedback to improve programming skills. Based on the analyzed Scratch project, Dr. Scratch assigns a score to various concepts such as abstraction, parallelism or logic. Like Dr. Scratch, the Ninja Code Village [23] analyzes Scratch projects online and provides feedback on CT concepts. Quality Hound [30] is another online program analysis tool for the detection of code smells. The ITCH (Individual Testing of Computer Homework for Scratch Assignments) tool provides a run-time analysis of Scratch projects by converting Scratch programs into Python code. Required test cases are written in Python. Stahlbauer et al. [28] from the University of Passau introduce Whisker[1], a very comprehensive testing framework that enables automated property-based testing of Scratch programs. The tests interconnect with the Scratch environment, so test cases can be built up using Scratch blocks. The team also developed Litterbox[2], an automated program analysis tool for static code analysis that provides checks for a wide range of common bug patterns in Scratch [10].

[1]https://github.com/se2p/whisker-main
[2]https://github.com/se2p/LitterBox

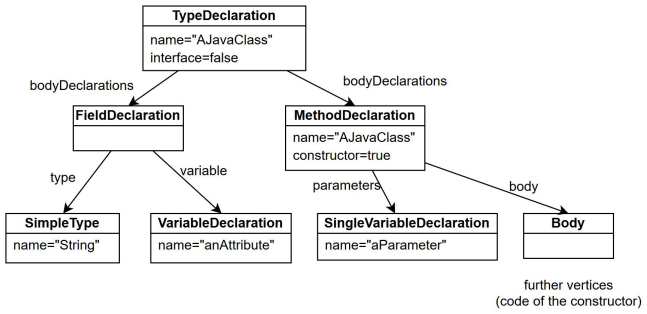

Figure 1: Parts of a TGraph Representation of Java Code

2.4 Code Analysis with TGraphs and GReQL

The assessment of code on a large scale, such as querying for solution patterns in projects from the Scratch repository or from a MOOC, requires an automated analysis of the code. As already explained, we will represent the program structure by *TGraphs*, apply the query language *GReQL* and use the automated assessment system *JACK*. In static code analysis of textual programming languages as an example, there are two main methods to represent program code: an abstract syntax tree or a graph representation [29]. The following describes a graph representation and a query language that is used to assess code from textual programming languages.

2.4.1 TGraph format. TGraph is a graph format specially designed for software re-engineering [9]. TGraphs are directed graphs, vertices and edges are typed, ordered and can carry attribute-value pairs. Sets of TGraphs are represented by an underlying scheme, so a particular scheme describes as an example the set of TGraphs representing Java programs. A TGraph conforms to its underlying scheme with respect to the element types (vertices, edges) and their attribute assignments, as well as the edges incident with a given vertex. The vertex degrees have to conform to the multiplicities defined in the scheme. Multiple inheritance is supported for vertex and edge types. [8] Figure 1 shows a reduced example of a TGraph for a Java class "AJavaClass" defining a string-attribute "anAttribute" and a constructor with a parameter "aParameter". Further information can be found on the website of the Ebert research group of the University Koblenz-Landau[3]. As mentioned above, the TGraph format is currently used to represent code from textual programming languages in a graph format.

2.4.2 GReQL (GUPRO Repository Query Language). GReQL [16] is an expression language to query graph structures like TGraphs. GReQL supports *FWR (from-with-report)* expressions. In the *from* part variables to be used are declared by name and type. In the *with* clause a regular path expression describes the structure to be queried in the TGraph. The edge directions in the path expression are identified by `-->` or `<--`, grouping and subqueries are supported. The report part returns attribute-values of vertices or edges declared in the *from* part and found by the path expression. Aggregate functions are supported. Further information can be

[3]https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/research/intern/TGraphen

found on the website of the Ebert research group at the University Koblenz-Landau[4].

2.4.3 JACK. The University of Duisburg-Essen developed and maintains JACK [13], a system to automatically assess and analyze program code written in languages like Java, Haskell and R as well as UML diagrams and mathematical solutions. To analyze the artifacts with JACK they are transformed into a TGraph. This representation can then be queried for elements and structures using GReQL [16]. The JACK system can be operated both on a scalable server environment and as a stand-alone Java version. We currently use JACK to evaluate Java code in a MOOC to provide automated feedback. As a next step, we aim to evaluate Scratch code in MOOCs online and to analyze code structures in Scratch projects form the Scratch repository on a large scale with the JACK stand-alone version.

3 CONTEXT

To validate a promising approach to investigate patterns in Scratch, we wanted to analyze Scratch projects that were created under consistent conditions. We used the projects created in a series of programming courses with school children between the age of 8 and 12 years that we had conducted recently. They took place in an out-of-school learning facility and were all held by the same instructor. Whole school classes took part in the course as part of a school trip and it was also offered as a holiday course in which children could register voluntarily. A total of 108 boys and 24 girls took part in the courses - in 11 cases, no information on gender was provided (n=143).

The courses consisted of two or three 3-hour units that focused on creating games in Scratch. During the course, the students programmed three different mini-games by following worked examples – a step-by-step demonstration of how a problem can be solved. During these exercises, they had learned several solution patterns, such as the collision-pattern. Afterward, the students were asked to develop individual game ideas and to implement them in Scratch. They were not told what kind of thematic elements the games should contain, nor were they explicitly asked to use the previously introduced patterns. The subsequent analysis of the students' games should focus on how the introduced patterns are re-used and adapted.

4 METHODOLOGY

To enable automated analysis of Scratch code, we aimed to represent the projects by TGraphs. Yet, up to now, TGraphs have been applied only for textual programming languages. Therefore, we had to develop a TGraph scheme for Scratch programs. As a first step, we analyzed the structure of the Scratch file in detail. The scheme of the vertex and edge class hierarchy to represent Scratch projects in a TGraph is based on the structure of Scratch files. Transforming Scratch program code into a TGraph representation is then done by parsing the Scratch code file and building the TGraph according to the scheme for Scratch projects. This enables us to perform GReQL queries for specific structures such as solution patterns within the TGraph representation of Scratch projects on a large scale.

[4]https://www.uni-koblenz-landau.de/en/campus-koblenz/fb4/ist/rgebert/research/Graphtechnology/graph-repository-query-language-greql

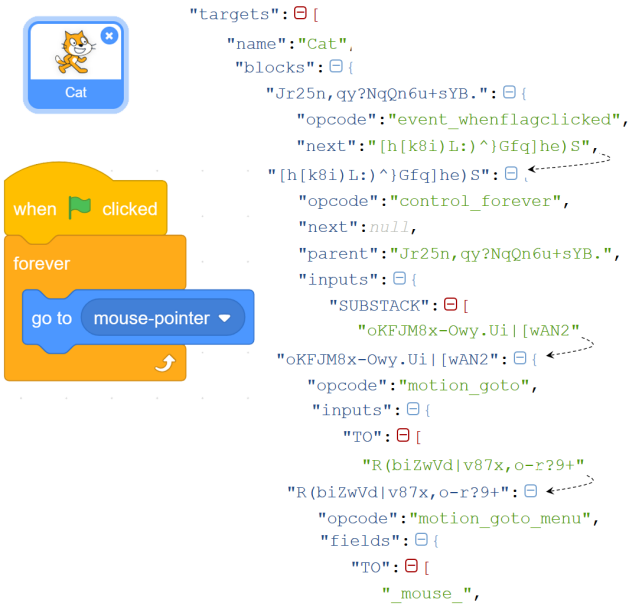

Figure 2: Scratch Example of Code

4.1 Structure of a Scratch Project File

Scratch projects contain a stage and 0-n sprites. Both the sprites and the stage can be programmed by placing sequences of blocks on a sprite or the stage. Images and sound can be used to further animate projects.

A project file in Scratch 3 is stored as a ZIP archive file. It contains all information about the project, including all images- and sound files. The actual program code, the blocks of every sprite as of the stage, is stored in a file named project.json in the Javascript Object Notation (JSON) format[5].

The project.json file is divided into four main parts:

- *targets:* all information about the stage and the sprites of the project
- *monitors:* list of all variables displayed on the stage during the execution of the program
- *extension:* list of the extensions used in the project. Extensions are used to connect external hardware, to use video sensing, music, ...
- *meta:* meta-information on the project, such as the Scratch version

The targets property in the project.json file, as shown in Figure 2, provides the main information for the stage and each sprite: isStage (boolean), name (of the stage or sprite), variables (defined), lists (defined), broadcasts and blocks (used on the stage or the sprite), comments, currentCostume, costumes, sounds, volume, layerOrder.

4.2 Scratch Block Types

The Scratch programming environment offers different block categories such as motion, looks or sound. Besides the category, each

[5]https://en.scratch-wiki.info/wiki/Scratch_File_Format

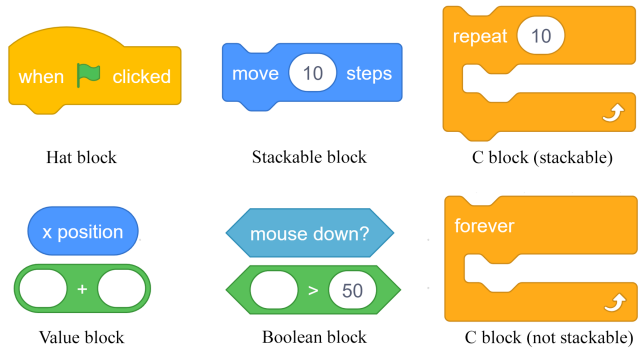

Figure 3: Scratch Block Shape Types

block belongs to a certain shape-type. As shown in Figure 3, the five main types are:

- *Hat:* starting event of a block sequence
- *Stackable:* have a bottom notch to connect with other blocks
- *C blocks:* blocks can be placed in the inside. With the exception of the forever block, all C-blocks are stackable.
- *Value:* blocks to sense, report or manipulate values
- *Boolean:* blocks return boolean values

4.3 Building a TGraph from a Scratch Project

JGraLab is a Java graph library and modeling framework, designed and maintained by the JGraLab team of the University of Koblenz-Landau[6]. It offers support for building and traversing TGraphs that conform to the underlying scheme with respect to the element types (vertices, edges) and their attribute assignments, as well as incidences of the edges and vertex degrees corresponding to the defined multiplicities.

4.3.1 TGraph Scheme for the Representation of Scratch Projects. At first, we designed a scheme for TGraphs representing Scratch projects. The names of the vertex and edge classes and their attributes follow the respective properties in the JSON file of Scratch projects. The attribute values are assigned when creating a corresponding TGraph.

An extract from the proposed scheme to transform Scratch code into TGraphs is shown in a reduced form in Listing 1. Figure 4 shows an extract of the vertex class hierarchy of the scheme. The proposed scheme for Scratch projects largely reflects the hierarchy of the JSON file (Figure 2). In some parts, we added additional vertex classes and made use of multiple inheritance, which is supported by the TGraph format. The use of additional hierarchies and multiple inheritance can greatly simplify the creation of GReQL queries on TGraphs. As an example, Scratch offers three loop blocks: *repeat with a counter*, *repeat until* and *repeat forever*. If the pattern to be investigated shall contain any kind of those loop blocks, it is not necessary to query for all applicable loop blocks, but instead, the parent class vertex definition, ControlRepeatC, can be taken (Figure 4).

[6]https://jgralab.github.io/jgralab/

Listing 1: Extract of the Definition of Vertex Classes from the TGraph Scheme for Scratch Projects

```
GraphClass Scratch3;

abstract VertexClass Node;
VertexClass ScratchProject:Node{name:String};
abstract VertexClass Element:Node;
VertexClass Block:Element;
abstract VertexClass Targets:Element;
VertexClass Stage:Targets{currentCostume:Long, ...};
VertexClass Sprite:Targets{name:String,currentCostume:Long, ...}
abstract VertexClass BlockType:Block;
abstract VertexClass Stackable:BlockType;
VertexClass HatBlock:Stackable;
abstract VertexClass TopStackable:Stackable;
VertexClass StackBlock:TopStackable;
VertexClass CBlock:TopStackable;
abstract VertexClass BlockClass:Block;
VertexClass EventsBlock:BlockClass;

abstract VertexClass MotionStackBlock:MotionBlock,StackBlock;
VertexClass Motion_glideto:MotionStackBlock;

abstract VertexClass EventHat:EventsBlock,HatBlock;
VertexClass Event_whenkeypressed:EventHat;
```

Besides all vertex classes and their hierarchy, the scheme also defines the possible connections between vertex classes through edges and their multiplicities. Listing 2 shows an extract of the scheme definition of edge classes.

Listing 2: Extract of the Definition of Edge Classes from the TGraph Scheme for Scratch Projects

```
abstract EdgeClass ChildTargetCall from Node (0,1) to Node (0,*);
abstract EdgeClass Child:ChildTargetCall from Node (0,1) to Node (0,*);
EdgeClass SpriteDescription:Child from ScratchProject (1,1)
   to Sprite (0,*) role spriteTarget;
EdgeClass CostumeDescription:Child from Targets(1,1)
   to Costumes(1,*) role costume;
EdgeClass BlockBundleDeclaration:Child from Targets (1,1)
   to Block(0,*) role blockStart;
EdgeClass SubStackNext:Child from Substack (1,1)
   to Block(0,1) role next;
EdgeClass StackedNext:Child from Stackable(1,1)
   to TopStackable(0,1) role next2;
EdgeClass InputDescription:Child from Block(1,1)
   to InputType(0,*) role inputType;
EdgeClass InputContentStackDeclaration:Child from InputType(0,1)
   to Block(0,1) role inputBlock;
EdgeClass FieldDescription:Child from Block(1,1)
   to Fields(0,*) role fields;
```

4.3.2 Parsing the Scratch JSON and Generating the TGraph. In the next step, the JSON in the Scratch file is parsed and the corresponding TGraph is created by instances of specific Java classes created by JGraLab based on the Scratch scheme for TGraphs. These classes provide methods for creating new vertices and edges from the generated classes to ensure that only TGraphs that conform to the schema can be built.

A final resulting TGraph is represented by Figure 5. It shows the reduced form of a sprite that follows the mouse-pointer (Figure 2).

4.4 Searching Structures by GReQL Queries

The representation of the Scratch project by a TGraph allows us to search for any block structures within Scratch projects with a GReQL query. The GReQL query for the solution pattern *mouse follower* (Figure 6) is shown in Listing 3. This solution pattern is introduced in our Scratch courses.
In the *from* part the variables used in the *with* part are declared by a name and their vertex or edge type. As a starting point of the script on a sprite, the class "Hatblock" is chosen instead of the class "EventHat" in the path expression. The Hatblock class contains

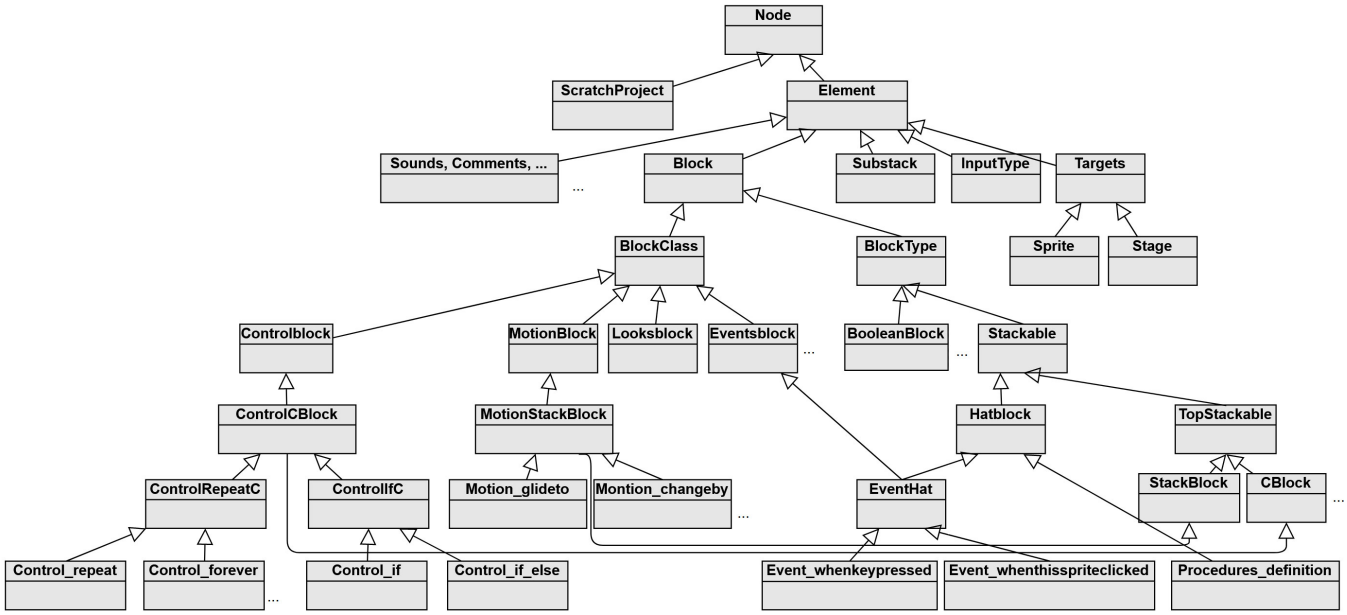

Figure 4: Extract of the Vertex Class Hierarchy of the TGraph Scheme for Scratch Projects

custom procedures (my blocks), so if the solution pattern is used in a custom procedure, it should also be selected. The path expression in the *with* part in this simple example mainly follows the path of the respective TGraph (Figure 5). The Kleene operator * after the edge "Child" at `hatBlock->{Child}* controlForever ->{Child}* motionGoto` ensures that block structures with additional blocks of any kind before and in the substack of the forever loop are also selected. Figure 6 presents different solution patterns found by the GReQL-query Listing 3.

Listing 3: GReQL Query of the Mouse Follower Pattern

```
from
  project:V{ScratchProject}, sprite:V{Sprite}, hatBlock:V{HatBlock
        }, controlForever:V{Control_forever}, motionGoto:V{
        Motion_goto}, fields:V{Fields}
with
  (project-->{SpriteDescription}* sprite -->{
        BlockBundleDeclaration} hatBlock -->{Child}* controlForever
         -->{Child}*  motionGoto -->{InputDescription} -->{
        InputContentStackDeclaration}-->{FieldDescription} fields
        and (fields.value = "_mouse_" )= true )
report
   project.name as "Project", sprite.name as "Sprite"
end
```

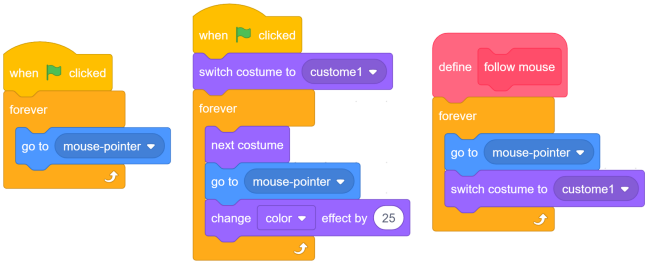

Figure 6: Variations of the Mouse Follower Pattern

4.5 Analysis Process with JACK

Figure 7 shows the main process for analyzing Scratch programs by using GReQL queries on TGraphs. The scratch program code from the JSON file is parsed and a TGraph is created according to the defined scheme for Scratch projects. Any kind of GReQL query can then be executed by the checker process, which as a result returns all structures found in the TGraph. For our research, we used the stand-alone Checker from JACK, which runs on PCs. JACK also provides an online interface so that users can upload their code and receive direct feedback, for example in a MOOC.

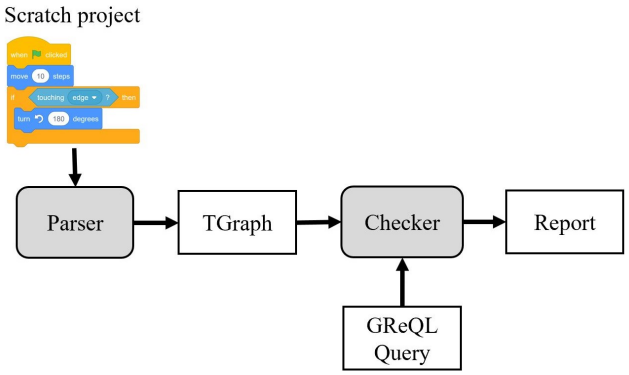

Figure 7: Analysis Process of a Scratch Project

5 FIRST RESULTS

In our feasibility study, 143 students independently developed a total of 295 Scratch game projects. In order to assess how pre-learned solution patterns are reused, adapted, or extended in the individual projects we created a GReQL query in two ways for each

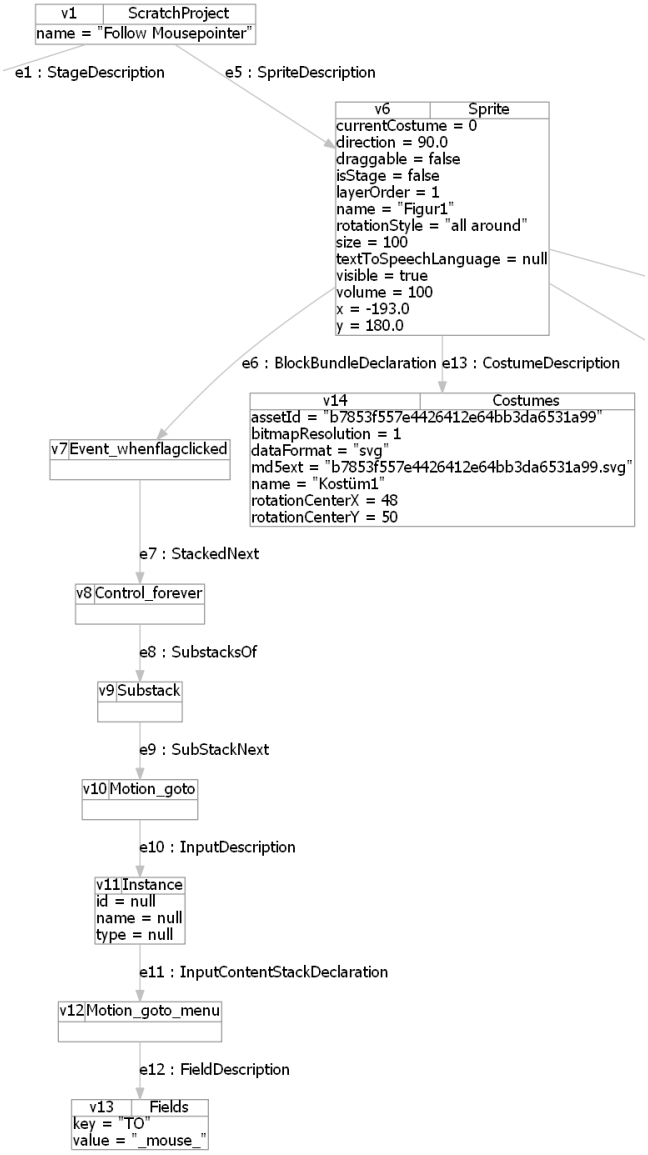

Figure 5: Reduced TGraph of the Mouse Follower Pattern

solution pattern introduced in the guided exercises (mini-games). One query searches for the exact match on the introduced pattern, a second for patterns that have been adapted or extended. An exact match is achieved when the pattern is reused block by block without adaptation or extension. Parameter literals could be included in the GReQL query, in our case we did not check them for equality. An adaption is taken into account when the student exchanges blocks in the pattern. As an example in the collision pattern when a motion block is used instead of the say block. An extension exists when additional blocks are added to the pattern.

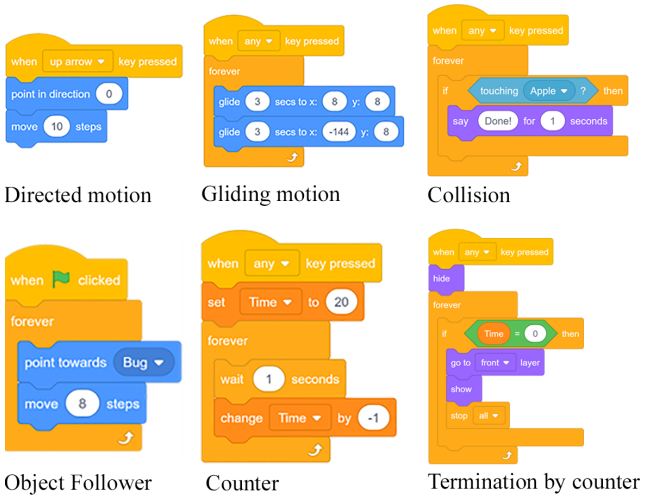

Figure 8: Solution Patterns

5.1 Use of Solution Patterns

We searched the student projects for the solution patterns shown in figure 8 using GReQL queries on the TGraph representation of the Scratch projects. The GReQL query in Listing 4 looks for any representative of the collision pattern (figure 8) in the students' Scratch projects that have not been modified in any way, except for possible changes of the parameter literals. Figure 9 shows an example of a Scratch project, represented in the TGraph format, that contains the collision pattern.

Listing 4: GReQL Query of the Collision Pattern with no Adaption

```
from
project:V{ScratchProject}, sprite:V{Sprite},eventKey:V{
      Event_whenkeypressed}, forever:V{Control_forever},
      substackForever:V{Substack}, if:V{Control_if}, substackIf:V{
      Substack}, sayForSecs:V{Looks_sayforsecs}, booleanOp:V{
      BooleanOp}, sensingTouching:V{Sensing_touchingobject}, fields
      :V{Fields}
with
(project-->{SpriteDescription} sprite -->{BlockBundleDeclaration}
      eventKey-->{StackedNext} forever --> {SubstacksOf}
      substackForever -->{SubStackNext}  if --> {SubstacksOf}
      substackIf --> {SubStackNext} sayForSecs and  isEmpty(
      sayForSecs -->{StackedNext})  and if  -->{InputDescription}
      booleanOp -->{InputContentStackDeclaration} sensingTouching
      --> {Child}* fields and (fields.value = "_mouse_" )= false
      and (fields.value = "_edge_" )= false)
 report project.name as "Project", sprite.name as "Sprite"
end
```

Listing 5 represents the GReQL query that searches for the collision pattern in consideration of adaptions and extensions.

Listing 5: GReQL Query of the Collision Pattern with Possible Adaptions

```
from
project:V{ScratchProject}, sprite:V{Sprite},eventKey:V{
      Event_whenkeypressed}, forever:V{Control_forever},
      substackForever:V{Substack}, if:V{Control_if}, substackIf:V{
      Substack},  booleanOp:V{BooleanOp}, sensingTouching:V{
      Sensing_touchingobject}, fields:V{Fields}, anyNode:V{Node}
with
```

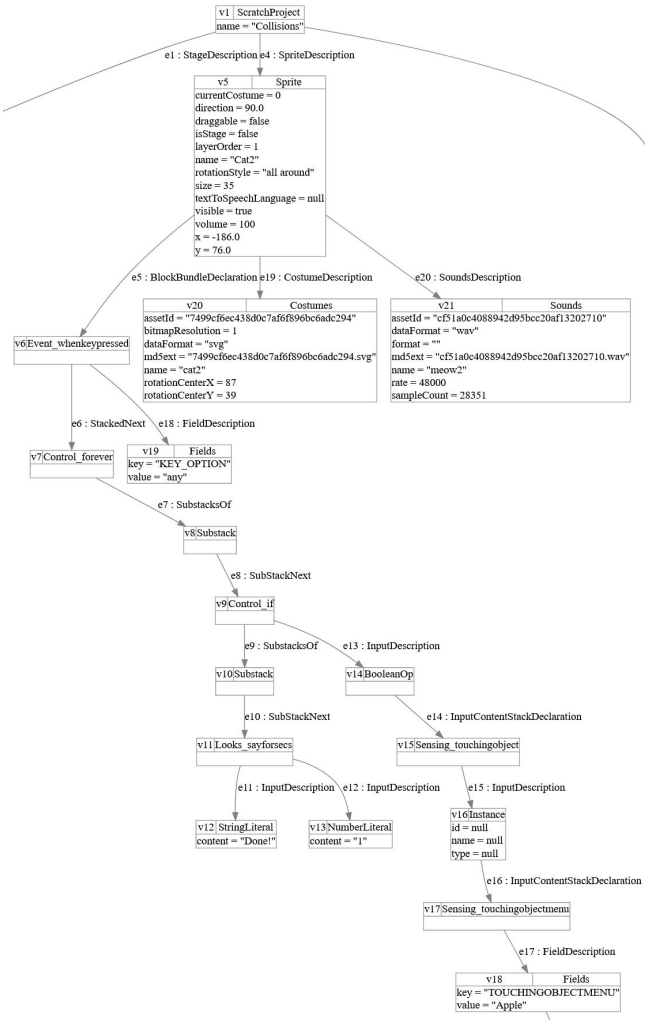

Figure 9: Reduced TGraph of a Collision Pattern without Adaption

```
(project-->{SpriteDescription} sprite -->{BlockBundleDeclaration}
     eventKey-->{Child}* forever --> {SubstacksOf} substackForever
      -->{Child}*  if --> {SubstacksOf} substackIf  --> {Child}+
     anyNode and if  -->{InputDescription} booleanOp -->{
     InputContentStackDeclaration} sensingTouching --> {Child}*
     fields and (fields.value = "_mouse_" )= false and (fields.
     value = "_edge_" )= false)
report  project.name as "Project", sprite.name as "Sprite"
end
```

Table 1 shows the number of each implementation of the pattern with and without adaptation in the student projects. Each use of a pattern was counted only once per project. One GReQL query searches for the solution patterns one to one without adaptation, a second query searches for arbitrary code structures containing the main parts required in the solution pattern. This is achieved by using more general block classes in the GReQL queries, e.g. by querying the block class *MotionStackBlock* instead of a specific motion block (figure 4). The asterisk (*) in the query path expression e.g. `block1 -->{Child}* block2`, ensures that extended solution patterns are also recognized. Only active block sequences that are connected to a *Hat* block are counted. Of all game projects, 99 contained sprites with block sequences but did not make use of any of the presented solution patterns.

Table 1: Re-Use of the introduced Solution Patterns in the Student Projects (n=295)

Solution Pattern	without adaptation	adapted
Directed motion	110 (37%)	4 (1%)
Collision	84 (28%)	46 (16%)
Gliding motion	58 (20%)	14 (5%)
Random gliding motion	19 (6%)	1 (0.3%)
Object follower	18 (6%)	8 (3%)
Mouse follower	11 (4%)	0 (0%)
Counter	15 (5%)	6 (2%)
Termination by counter	17 (6%)	0 (0%)
Termination by collision	10 (3%)	13 (4%)

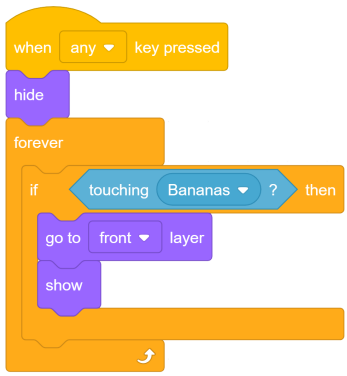

Figure 10: Adapted Collision Pattern

As a proof of concept, we manually checked the occurrence of two patterns without any adaptions in all 295 projects by one person. The manually identified projects were exactly the same as found by the GReQL queries on the TGraph representations of the projects.

5.1.1 Example of an Adaption of the Collision Pattern. Figure 10 represents an adaptation of the pattern *collision*. The student has extended this pattern with elements of the pattern *termination by counter* (figure 8). The main goal was to show a sprite displaying a text in the event of a collision.

5.2 Advanced Code Structures and Concepts

Grover at al. [15] identified misconceptions and challenges in the use of variables, loops and boolean operators. In the guided exercises we introduced some more complex code structures, such as nested conditions in loops and the use of variables and random number blocks. Table 2 shows the number of each use of the code structures. Each structure was counted at each occurrence. Nested conditions were not introduced to the students. As a result, we could only find one project with nested conditions.

Table 2: Advanced Code Structures and Concepts used in the Student Projects (n=295), counted at each occurrence

Code	Count
Nested condition in loop	155
Nested loops	7
Nested conditions	1
Variables	42
Boolean operator	21
Random block	25

6 CONCLUSION AND FUTURE WORK

In this paper, we presented a feasibility study for a methodology to investigate the use of certain solution patterns in a large number of Scratch projects.

The first results demonstrate that we were able to detect patterns like *collision*, *directed motion*, *termination by collision* or *gliding motion*, see Table 1. In addition, we could find out how far advanced programming concepts like nested conditions or nested repetition were used, see Table 2. As part of our feasibility study, we also compared the results of our methodology with those of a manual analysis of the Scratch projects. Since the results were completely consistent, they show the validity of the method.

Our method – conducting GReQL queries on TGraph representations of Scratch programs – offers versatile search options for code structures. Not only one-to-one representations but also adapted or extended patterns can be found by a wildcard search. It must be noted though that GReQL is a very powerful SQL-like query language that requires training and can be quite tricky in detail. Compared to other tools that offer static code analysis of Scratch code, the advantage is that any kind of code structure can be queried without having to modify an existing program, as for example in the plugin system of Hairball [6]. Even though our number of projects (n=295) could have been analyzed manually with great effort, the method opens up the possibility to examine larger quantities of projects, such as the Scratch repository or results of MOOCs.

In summary, the presented method for querying Scratch block structures on TGraphs offers broad support in the search for any solution patterns. Therefore, it can help to answer the question of how students implement and adapt pre-learned solution patterns. However, further analysis of the existing data is needed, using a mixed-method approach where parts of the projects are qualitatively analyzed manually.

We are currently developing a GreQL query editor that generates GReQL queries based on Scratch code. Furthermore, we are working on an importer that brings TGraphs into the graph database Neo4j[7]. This will even make it possible to run queries on millions of projects from the Scratch repository. In combination with JACK [13], the presented TGraph parser could also be used to generate online feedback, e.g. on code smells in Scratch code. To further investigate the use of programming patterns in Scratch, we plan to conduct a study with novice programmers, which will explicitly focus on how students reuse, adapt and extend introduced patterns to solve different types of problems.

[7] https://neo4j.com/

REFERENCES

[1] Kashif Amanullah and Tim Bell. 2018. Analysing Students' Scratch Programs and Addressing Issues using Elementary Patterns. In *Fostering innovation through diversity: Frontiers in Education Conference 2018 : 2018 Conference proceedings 48 (2018, San Jose, Calif.)*. IEEE, Piscataway, NJ, 1–5. https://doi.org/10.1109/FIE.2018.8658821

[2] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2015. From Scratch to "Real" Programming. *ACM Transactions on Computing Education* 14, 4 (2015), 1–15. https://doi.org/10.1145/2677087

[3] Ashok Basawapatna, Kyu Han Koh, Alexander Repenning, David C. Webb, and Krista Sekeres Marshall. 2011. Recognizing computational thinking patterns. In *SIGCSE '11: Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, New York, N.Y., 245–250. https://doi.org/10.1145/1953163.1953241

[4] Tim Bell and Caitlin Duncan. 2018. Teaching Computing in Primary Schools. In *Computer science education*, Sue Sentance, Erik Barendsen, and Carsten Schulte (Eds.). Bloomsbury Academic, London and New York and Oxford and New Delhi and Sydney, 131–150.

[5] Anshul Bhagi. [n.d.]. App Inventor Concept Cards. https://appinventor.mit.edu/explore/resources/app-inventor-flash-cards

[6] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Conrad, Phillip, and Diana Franklin. 2013. Hairball: Lint-inspired Static Analysis of Scratch Projects. In *SIGCSE'13: Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. ACM, New York, N.Y., 215–220.

[7] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and pedagogy. *ACM SIGCSE Bulletin* 31, 1 (1999), 37–42. https://doi.org/10.1145/384266.299673

[8] Jürgen Ebert and Angelika Franzke. 1995. A declarative approach to graph based modeling. In *Graph-Theoretic Concepts in Computer Science: 20th International Workshop. WG '94, Herrsching, Germany, June 16-18, 1994: proceedings (Lecture Notes in Computer Science)*, Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer (Eds.). Springer, Berlin, 38–50.

[9] Jürgen Ebert, Volker Riediger, and Andreas Winter. 2008. Graph Technology in Reverse Engineering: The TGraph Approach. In *10th Workshop Software Reengineering (GI-Edition - Lecture notes in informatics, Vol. 126)*, Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter (Eds.). Gesellschaft für Informatik, Bonn, 67–81. http://www.uni-koblenz.de/\protect\unhbox\voidb@x\protect\penalty\@M\ist/documents/Ebert+2008GTI.pdf

[10] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common Bugs in Scratch Programs. In *ITICSE '20*, Michail Giannakos, Guttorm Sindre, Andrew Luxton-Reilly, and Monica Divitini (Eds.). ACM, New York N.Y., 89–95. https://doi.org/10.1145/3341525.3387389

[11] Diana Franklin, Conrad, Phillip, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, Gerardo Aldana, Paulo Almeida-Tanaka, Brynn Kiefer, Chelsea Laird, Felicia Lopez, Christine Pham, Jessica Suarez, and Robert Waite. 2013. Assessment of computer science learning in a scratch-based outreach program. In *SIGCSE'13: Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. ACM, New York, N.Y., 371. https://doi.org/10.1145/2445196.2445304

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design Patterns Abstraction and Reuse of Object-Oriented Design. In *ECOOP '93: object-oriented programming: 7th European conference, Kaiserslautern, Germany, July 26 - 30, 1993 ; proceedings (Lecture Notes in Computer Science)*, Oscar M. Nierstrasz (Ed.). Springer, Berlin, 406–431.

[13] Michael Goedicke and Michael Striewe. 2017. 10 Jahre automatische Bewertung von Programmieraufgaben mit JACK - Rückblick und Ausblick. In *Informatik 2017: 25.-29. September 2017, Chemnitz: proceedings (GI-Edition - Lecture notes in informatics)*, Maximilian Eibl and Martin Gaedke (Eds.). Gesellschaft für Informatik, Bonn, 279–283. https://doi.org/10.18420/in2017{_}21

[14] Shuchi Grover and Satabdi Basu. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In *SIGCSE '17: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, N.Y., 267–272. https://doi.org/10.1145/3017680.3017723

[15] Shuchi Grover, Roy Pea, and Stephen Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education* 25, 2 (2015), 199–237. https://doi.org/10.1080/08993408.2015.1033142

[16] Manfred Kamp. 1996. *GReQL - eine Anfragesprache für das GUPRO-Repository 1.1*. Koblenz. https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/research/graph-technology/GReQL

[17] Kyu Han Koh, Ashok Basawapatna, Vicki Bennett, and Alexander Repenning. 2010. Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2010: 21 - 25 September 2010, Leganes, Madrid, Spain ; proceedings*. IEEE, Piscataway, NJ, 59–66. https://doi.org/10.1109/VLHCC.2010.17

[18] Johannes Krugel and Peter Hubwieser. 2017. Computational thinking as springboard for learning object-oriented programming in an interactive MOOC. In *2017 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 1709–1712.

[19] Johannes Krugel and Peter Hubwieser. 2018. Strictly Objects First: A Multipurpose Course on Computational Thinking. In *Computational Thinking in the STEM Disciplines*, Myint Swe Khine (Ed.). Springer International Publishing, Cham, 73–98. https://doi.org/10.1007/978-3-319-93566-9{_}5

[20] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2010. Learning computer science concepts with scratch. In *ICER '10: Proceedings of the Sixth International Workshop on Computing Education Research*. ACM, New York N.Y., 69–76.

[21] Jesús Moreno-León and Gregorio Robles. 2016. Code to learn with Scratch? A systematic literature review. In *Proceedings of 2016 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, Piscataway, NJ, 150–156. https://doi.org/10.1109/EDUCON.2016.7474546

[22] Jesús Moreno-León, Gregorio Robles, and Román-González, Marcos. 2015. Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. *RED : revista de educación a distancia* 14, 46, Sept. 15 (2015). http://www.um.es/ead/red/46

[23] Go Ota, Yosuke Morimoto, and Hiroshi Kato. 2016. Ninja code village for scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, Piscataway, NJ, 238–239. https://doi.org/10.1109/VLHCC.2016.7739695

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

[25] Natalie Rusk. [n.d.]. Scratch Cards. https://resources.scratch.mit.edu/www/cards/en/scratch-cards-all.pdf

[26] Linda Seiter and Brendan Foreman. 2013. Modeling the learning progressions of computational thinking of primary grade students. In *ICER '13: Proceedings of the ninth annual international ACM conference on International computing education research*. ACM Press, New York, 59–66. https://doi.org/10.1145/2493394.2493403

[27] Neil Smith, Clare Sutcliffe, and Linda Sandvik. 2014. Code Club: Bringing Programming to UK Primary Schools through Scratch. In *SIGCSE '14: Proceedings of the 2014 ACM SIGCSE Technical Symposium on Computer Science Education*, J. D. Dougherty (Ed.). ACM, New York, NY, 517–522. https://doi.org/10.1145/2538862.2538919

[28] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. ACM Press, New York, N.Y., 165–175. https://doi.org/10.1145/3338906.3338910

[29] Michael Striewe and Michael Goedicke. 2014. A Review of Static Analysis Approaches for Programming Exercises. In *Computer Assisted Assessment: Research into E-Assessment: International Conference, CAA 2014 Zeist, The Netherlands, June 30 – July 1, 2014 Proceedings (Communications in Computer and Information Science)*, Marco Kalz and Eric Ras (Eds.). Springer, Cham, 100–113.

[30] Peeratham Techapalokul and Eli Tilevich. 2017. Quality Hound — an online code smell analyzer for scratch programs. In *VL/HCC 2017: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing : October 11-14, 2017, Raleigh, North Carolina, USA : proceedings /*. IEEE, Piscataway, NJ, 337–338.

[31] 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*. IEEE, Piscataway, NJ, 43–51. https://doi.org/10.1109/VLHCC.2017.8103449

[32] David S. Touretzky. 2016. Kodu Idiom Flash Cards. https://www.cs.cmu.edu/~dst/Kodu/FlashCards

[33] Rebecca Vivian, Katrina Falkner, and Claudia Szabo. 2014. Can everybody learn to code?. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Päivi Kinnunen and Simon (Eds.). ACM, New York, N.Y., 41–50. https://doi.org/10.1145/2674683.2674695

[34] Mary Webb, Niki Davis, Tim Bell, Yaacov J. Katz, Nicholas Reynolds, Dianne P. Chambers, and Maciej M. Sysło. 2017. Computer science in K-12 school curricula of the 2lst century: Why, what and when? *Education and Information Technologies* 22, 2 (2017), 445–468. https://doi.org/10.1007/s10639-016-9493-x

[35] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-Based and Text-Based Programs. In *ICER '15: Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM, New York, N.Y., 101–110. https://doi.org/10.1145/2787622.2787721

[36] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (2006), 33–35. https://doi.org/10.1145/1118178.1118215

[37] Hatice Yildiz Durak. 2020. The Effects of Using Different Tools in Programming Teaching of Secondary School Students on Engagement, Computational Thinking and Reflective Thinking Skills for Problem Solving. *Technology, Knowledge and Learning* 25, 1 (2020), 179–195. https://doi.org/10.1007/s10758-018-9391-y