



UNIVERSITÀ  
DEGLI STUDI  
DI UDINE

## Università degli studi di Udine

High-school students' mastery of basic flow-control constructs through the lens of reversibility

*Original*

*Availability:*

This version is available <http://hdl.handle.net/11390/1193263> since 2021-09-24T14:29:36Z

*Publisher:*

ACM - Association for Computing Machinery

*Published*

DOI:10.1145/3421590.3421603

*Terms of use:*

The institutional repository of the University of Udine (<http://air.uniud.it>) is provided by ARIC services. The aim is to enable open access to all the world.

*Publisher copyright*

(Article begins on next page)

# High-School Students’ Mastery of Basic Flow-Control Constructs through the Lens of Reversibility

Claudio Mirolo  
University of Udine  
Udine, Italy  
claudio.mirolo@uniud.it

Cruz Izu  
The University of Adelaide  
Adelaide, Australia  
cruz.izu@adelaide.edu.au

Emanuele Scapin  
University of Udine  
Udine, Italy  
scapin.emanuele@spes.uniud.it

## ABSTRACT

High-school students specialising in computing fields need to develop the abstraction skills required to understand and create programs. Novices’ difficulties at high-school level, ranging from mastery of the “notional machine” to appreciation of a program’s purpose, have not yet been investigated as extensively as at undergraduate level.

This work explores high-school students’ code comprehension by asking to reason about reversing conditional and iteration constructs. A sample of 205 K11–13 students from different institutions were asked to engage in a set of “reversibility tasklets”. For each code fragment, they need to identify if its computation is reversible and either provide the code to reverse or an example of a value that cannot be reversed. For 4 such items, after extracting the recurrent patterns in students’ answers, we have carried out an analysis within the framework of the SOLO taxonomy. Overall, 74% of answers correctly identified if the code was reversible but only 42% could provide the full explanation/code. The rate of relational answers varies from 51% down to 21%, the poorest performance arising for a small array-processing loop (and although 65% of the subjects had correctly identified the loop as reversible).

The instruction level did not have a strong impact on performance, indicating such tasks are suitable for K11, when the basic flow-control constructs are usually introduced. In particular, the reversibility concept could be a useful pedagogical instrument both to assess and to help develop students’ program comprehension.

## KEYWORDS

Program comprehension; Novice programmers; High school; Flow-control constructs; Iteration; SOLO taxonomy

### ACM Reference Format:

Claudio Mirolo, Cruz Izu, and Emanuele Scapin. 2020. High-School Students’ Mastery of Basic Flow-Control Constructs through the Lens of Reversibility. In *Proceedings of the 15th Workshop in Primary and Secondary Computing Education (WiPSCE ’20)*, October 28–30, 2020, Online Conference. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3421590.3421603>

## 1 INTRODUCTION

While learning to use basic programming constructs, students manage to write working programs but they appear to lack a more comprehensive grasp of the overall computation carried out by each block of code. In other words, many students do not fully develop the abstraction skills — and perhaps accurate enough mental

models — that would allow them to reason about and interact with code. Several studies report problems and misconceptions even for such basic flow-control constructs as conditionals and loops in the context of tertiary education. Kaczmarczyk et al. [13], for instance, identified “a number of misconceptions all related to an inability to properly understand the process of while loop functioning”, and Cherenkova et al. [6] found that “students have significant trouble with conditionals and loops, with loops being particularly challenging”.

Up to now, however, novices’ difficulties with basic programming concepts have received far less attention in high school contexts. Vahrenhold et al. [37], in their recent broad literature review of K–12 computer science research, reported only eight studies addressing a variety of programming topics specifically for the upper secondary level. Our own literature review of ISSEP and WiPSCE conference proceedings, combined with keywords search in the ACM digital library, found no more than a dozen papers about high school learning of flow-control constructs and recursion.

In this paper we address high-school students’ ability to understand the behaviour of small code fragments by looking at how they deal with the concept of *reversibility*, i.e. the possibility to undo a state transformation in *every* conceivable situation. A relevant feature of this approach is that, to cope with reversibility, the behaviour of program constructs must be understood as a whole, by analysing the interactions of the constituent parts in different potential execution flows.

A sample of 205 students, attending the last three years of secondary instruction in different institutions that offer a specialisation in computing topics, were asked to engage in a set of *reversibility tasklets*. Following a short explanation of reversibility in the test sheet, students were asked to decide if each given (tiny) program can be reversed or not, as well as to justify their answer by writing the reversing code (*Yes* option) or by providing suitable *counterexamples* (*No*). Thus, the test covers both code reading — at an abstract level — and code writing abilities. It is also worth remarking that the involved students had no experience at all of similar tasks in class, so ensuring their higher-order thinking skills, instead of simple recall, were put into play in the endeavour. For four of the six items assigned in the test, namely two conditionals and two iteration constructs, we extracted the recurrent patterns in the answers and then we analysed the justifications within the framework of the SOLO taxonomy [1].

In essence, here we will address the following research questions:

RQ1 – Can students grasp the state transformation enacted by simple conditionals and loops in an accurate and comprehensive way?

- RQ2 – When a statement is reversible, are students able to write correct code to recover the original state?
- RQ3 – Is student’s performance consistent over all tasks, reflecting their current level of comprehension? Or does their comprehension vary for each task example?

The main contribution of this study is twofold. Firstly, it provides an interesting pedagogical perspective to assess and develop students’ mastery of program constructs in the high school. Secondly, it provides a first insight into their ability to reason about basic flow-control structures. Additionally, the tasks proposed here have the potential to disentangle the understanding of program behaviour from the concepts pertaining to some specific application domain. This may be especially helpful to assess progress of novices who already master the “atomic concepts” of syntax and semantics addressed in [22].

The rest of the paper is organized as follows. In section 2 we outline the background of this work. Section 3 presents the tasklets and the methodology of data collection and analysis. Section 4 describes the main results of the analysis and in section 5 we discuss what emerges from the analysis and the implications for instructors and for planning future research lines. Finally, we summarise the conclusions of this study in section 6.

## 2 BACKGROUND

In Piaget’s theory of cognitive development reversibility is a crucial step towards higher-order thinking [27], and according to a neo-Piagetian perspective Piaget’s learning stages apply regardless of age when approaching new knowledge domains [35]. In this respect, reversibility seems to be an appropriate instrument to assess their comprehension in the early stages of learning to code.

The central role of “reverse thinking” in computer science was pointed out in [9]. Lister proposed “asking to reverse a piece of code” [18] as a tool to detect “archetypal manifestation[s] of concrete thinking” in novice programmers, an approach that was then experimented in [36]. In addition, recent work that used the concept of reversibility to investigate the comprehension of conditional constructs in introductory courses at university has shown the potential benefits of a similar perspective [11, 26].

More in general, reversibility can be seen as a tool to assess program comprehension, a broader topic covering, among other aspects: the divide between code tracing, reading (“explain in plain English”) and “chunking” abilities [8, 17, 20, 21]; the characterisation of mental models of program execution [3, 29]; the understanding of loops and nested loops [5]; the issues connected with basic concepts of language notation and operational semantics [22].

Previous work has identified key misconceptions and struggles at high school level. Rahimi et al. [28] addressed high school students’ misconceptions and remarked that programming “puts a high cognitive and knowledge demand on novices including knowledge on a specific programming language and knowledge and understanding of basic programming concepts and constructs such as variables, loops, conditions, abstraction, and procedures.” Kaila et al. [14] compared students’ ability to learn and master a variety of computer programming concepts in two different student groups: university level and junior high school. They reported that “for almost all of the concepts, both groups perform equally well, but students in the

adolescent treatment group perform significantly worse when learning the concepts of loop structures and repetition.” Other works also reported high school student’s difficulties comprehending loop updates [31] and zero-iteration loops [16]. To develop better comprehension, Smetsers-Weeda and Smetsers [34] proposed the use of flowcharts in combination with reflection and evaluation “in order to ensure that students learn from their mistakes through an iterative think-act process”.

At lower secondary school level students are introduced to block-based programming such as Scratch. Research at this level [10, 25] also found loops and variable’s initialization as main sources of misconceptions.

## 3 METHODOLOGY

In this section we describe the four reversibility tasklets and the rationale for using them in our investigation. Then, we outline the data collection process. Finally, we present the criteria underlying our analysis and our application of the SOLO taxonomy.<sup>1</sup>

### 3.1 The task

The exercise analysed in this study comprises four tasklets, which are shown in Figure 1. In the sheet the questions were preceded by a basic explanation of reversibility. An interesting feature of this type of task, unconventional from the students’ standpoint, is that it allows to a large extent to disentangle the understanding of program behaviour from any specific application domain knowledge.

To devise the tasklet format, we chose a mixed method that combines the three features useful in measuring higher order thinking skills: *selection*, *explanation*, and *creation* [15].

Analyse the following code fragments and decide if they are reversible or not:

- If your answer is *Yes*, i.e. the command is reversible, write a piece of code to restore the original state of the variables.
- If your answer is *No*, i.e. it is not always possible to undo the effect, provide examples for which we cannot recover the original state.

```
(a) // int x
    if ( x > 10 ) {
        x = x + 2;
    } else {
        x = x + 1;
    }

(b) // int x
    if ( x < 0 ) {
        x = x + 1;
    } else {
        x = x - 1;
    }

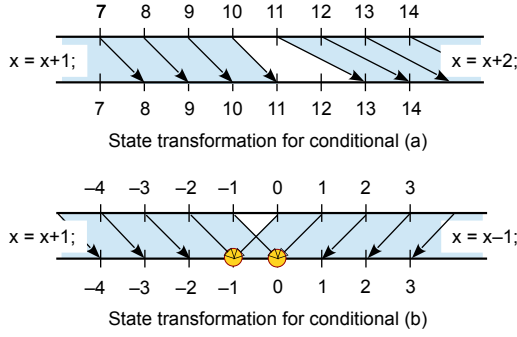
(c) // int x
    while ( x >= 5 ) {
        x = x - 5;
    }

(d) // int[] v
    int x = 0;
    for ( int i=0; i<v.length; i=i+1 ) {
        x = x + v[i];
        v[i] = x;
    }
```

**Figure 1: Reversibility tasklets (a), (b), (c) and (d).**

For each item students had first to identify whether a program is reversible or not (selection); then they were required either to provide a counterexample (explanation), or to write a reversing program (creation).

<sup>1</sup>The complete text of the reversibility task, as well as the recurring patterns and the SOLO classification of students’ justifications will be available from the corresponding author, XX, upon reasonable request.



**Figure 2: Depiction of the state transformations relative to tasklets (a) and (b).**

The statements in the branches of the two conditionals are very simple and straightforward to reverse in isolation. The key insight to decide if the code can be reversed is that there are different computation flows whose outcomes may “overlap”, as shown in Figure 2 for item (b), and in order to find potential overlaps students have to identify border computations. A more explicit overlap occurs in the while loop (c), since all computations starting from  $x \geq 0$  result into a final value in the range  $[0, 4]$ . Hence, such overlap is easier to detect as it does not involve borderline analysis.

Finally, tasklet (d) is more cognitively demanding: students have first to realise that the for loop implements a cumulative sum on a vector; then that the vector’s original state can actually be recovered; finally, they have to write a nontrivial reversing program requiring mastery of the dependencies between the operations carried out at subsequent iterations.

The rationale inspiring the design of the proposed task can be better appreciated in terms of the *Block Model* (BM) [30], a useful framework to map programming concepts and activities into a matrix representing three program dimensions, namely text, execution behaviour and purpose, as well as four abstraction levels: atomic elements, blocks, relationships between blocks and overall macro structure, see Figure 3. Typical tracing tasks, for instance, can be achieved by analysing code behaviour at the atomic level (A), as pointed out in [12], whereas “explain in plain English” (EiPE) tasks pertain to the top-rightmost cell which implies the deepest understanding of a program and its purpose. It is also worth observing that while the first two columns are concerned with *intrinsic* program features, the third one is about *extrinsic* properties in connection with some application domain.

Now, to begin with, all tasklets (a–d) essentially pertain to the *program execution* dimension (P: middle column in Figure 3), i.e. with properties intrinsic to the code but requiring a viable mental model of the underlying notional machine. Thus, to know if a program is reversible there is no need to figure out its function. In addition, the analysis of each item can be done at a different (minimal) level of abstraction: tasklet (c) can possibly be achieved by simply tracing the code in a few cases (AP), since the probability of getting “conflicting” outputs is quite high; a reversal of code (a) requires considering the *if* block as a whole (BP), but can be written even without having actually considered borderline computations; to see that code (b) cannot be reversed, on the other hand, such

			EiPE tasks
Macrostructure	MT	MP (d) read+write	MF
Relationships	RT	RP (b) read	RF
Blocks	BT	BP (a) read+write	BF
Atoms	AT	AP (c) read tracing tasks	AF
	Text surface intrinsic program features	Program execution intrinsic program features	Function / purpose extrinsic features

**Figure 3: Block Model analysis of the tasklets (a–d).**

borderline computations must be identified (RP), what implies that the *if* branches cannot be analysed independently of each other; finally, a comprehensive grasp of the behaviour of both the given program (MP) and the reversing code is a prerequisite to provide a solution for tasklet (d).

### 3.2 Data collection

We administered the test on reversibility at the end of the school year to a sample of 205 students of the age range 16–19 from (11 classes of) two technical schools, located in two different areas of the country, that offer a specialisation in computing topics. All the students had completed at that time an introduction module of imperative programming, which is scheduled in the third high school year. 101 students were finishing third year (K11), 52 fourth (K12), and 52 their last year of high school (K13). Each collected sheet was then anonymised and digitised prior to analysis.

### 3.3 Data analysis

The reference framework for the core part of our analysis is the SOLO taxonomy [1], an instrument of widespread use to assess code reading and writing tasks, e.g. [21, 33, 38], which is deemed to provide “a means of evaluating cognitive or mental models, to see if the novices are able to make connections between what they have learnt.” [33]. From this perspective, the learners’ achievements are classified in terms of complexity and quality of the interrelationships between parts they are able to deal with.

After measuring the percentages of correct options, the analysis has been carried out as follows. As a first step, a researcher identified and listed the recurrent patterns of code or justification in students’ answers by an inductive content analysis process [23]. Next, a second researcher revised the patterns, possibly adjusting the list by splitting some patterns into more refined ones or merging pairs of essentially equivalent patterns, and assigned a SOLO level to each pattern according to the guidelines summarised in Table 1. During this process, the same researcher checked more than half of the students’ sheets via deductive content analysis (see again [23]), based on the set of identified patterns, as well as classified into

Table 1: SOLO Classification features

SOLO Level	Answer features (reasoning or code)
<b>Prestructural</b> (1)	Poor answer showing lack of understanding of either the task or some basic programming construct.
<b>Unistructural</b> (2)	Attempt to reverse part(s) of the code that could work for limited values, as it disregards interactions with other parts, for example by doing a cursory “syntactic manipulation”; defective justification that the code cannot be reversed.
<b>Multistructural</b> (3)	Answer indicating that the goal of the task is clearly understood and pursued with a reasonable approach, but somehow incomplete, e.g. the reasoning may be ambiguous or the code may be affected by minor flaws.
<b>Relational</b> (4)	Correct and accurate answer, providing either some appropriate reversing program or a clear counterexample showing that the program cannot be reversed.

SOLO categories all isolated (non-recurrent) answers which were left unprocessed from the previous steps.

Finally, a third researcher reviewed the categorisation of justification patterns and discussed with the second researcher a few cases where different possible perspectives had emerged. In particular, they eventually agreed that tasklet (c) could be fully achieved at the unistructural SOLO level, so that insights to motivate higher SOLO levels could hardly be found in the students’ answers. In addition, further discussion resolved minor issues and led to the final classification presented in section 4.

It is worth noting that all items except tasklet (c) are in fact relational in that students need to consider non-trivial relationships between parts. Tasklet (c), on the other hand, is unistructural since it makes only sense to consider the loop as a whole: isolating either the condition or the inner assignment would be meaningless. To identify overlaps between final states for the straightforward while in tasklet (c), students could opt for describing it in words, providing two or more counterexamples, e.g. via tracing, or attempting and seeing that any basic code manipulation will fail to retrieve the original value. Any other answer that fails to describe such behaviour is classified as prestructural.

Table 1 does not include the *extended-abstract* SOLO category, since it does not match with the features of the tasks at hand. It can however be noticed that the proposed tasklets, although simple, require abilities at the *relational* level that cannot be taken for granted in the high school.

## 4 RESULTS

In this section we outline the results of our investigation, as well as providing sample answers to better describe the range of SOLO levels and mental models identified in the analysis.

Overall, 74% of the chosen options about code reversibility are correct, but only 42% are also supported by sound justifications, either at SOLO relational level for items (a), (b) and (d), or at the unistructural level for item (c). Table 2 summarises the main figures concerning the selected options as well as the quality of students’ justifications. As it is customary [32], SOLO means are determined by averaging over the weights assigned to each level: 4=relational,

Table 2: Rate of correct options, full justifications and average SOLO mean for each tasklet.

	correct options	full justifications	average SOLO mean
<b>tasklet (a)</b>	92.2%	50.7% (Rel)	2.96
<b>tasklet (b)</b>	59.0%	37.1% (Rel)	2.58
<b>tasklet (c)</b>	81.0%	60.5% (Uni)	—
<b>tasklet (d)</b>	65.4%	21.5% (Rel)	1.65
<b>overall</b>	74.4%	42.4%	2.40

3=multistructural, 2=unistructural, 1=prestructural, and 0=null, i.e. either empty or meaningless justification.

### 4.1 Analysis of conditional tasks

The first two tasks are relational in that students have to identify if there is an overlap in values resulting from the two update paths (if and else). Tasklet (a) has no overlap, hence is reversible and tasklet (b) has an overlap for borderline values 0 and  $-1$ .

*Tasklet (a).* Most students (more than 90%) rightly conjectured that code (a) is reversible, but only a little more than half of them (55%) were also able to write correct reversing code, while one third provided instead code pertaining to the unistructural SOLO level, usually an instance of pattern *Uni1* (28% overall, see Figure 4) that lacks awareness of the interrelationships between statements in the if branches and if condition. The two most “popular” code patterns are shown in Figure 4; other recurrent patterns (one more relational, two multistructural, two more unistructural and two prestructural) are far less frequent.

*Tasklet (b).* 37.1% of students identify the overlap and clearly describe it as in the following excerpt from an instance of pattern *Rel1*:

“If  $x$ ’s initial value is either  $-1$  or  $1$ , then the result will be  $0 \dots$ ”

As the overlap is small, it is possible to write code that reverses all values except 0 or  $-1$ , as shown in Figure 5 for pattern *Mul1*. Such code reflects a good understanding of reversibility and ability to code. However, it assumes the final value 0 was originally “ $-1$ ” which is plausible but ignores the other potential original value of “ $+1$ ”. In comparison, pattern *Uni1* does not adjust the condition of the statement. Although both of them fail for a small number of cases, they show a different level of comprehension.

The third item in Figure 5 (*Uni2*) is another interesting recurrent pattern that seems to indicate a quasi-syntactic manipulation of the code, where both the condition as well as the operations in the if branches are reversed, resulting into a program that fails to undo the state transformation in almost all the cases.

We speculate that students that answer Yes to both (a) and (b) may be failing to check the relationship between the parts and treating them separately. In fact, more than one third of the students choose the Yes option for both items (a) and (b).

(i) Rel1

(ii) Uni1

Figure 4: Most common code patterns for tasklet (a).

## 4.2 Analysis of iterative tasks

As mentioned before, the first iterative task (c) is unistructural while task (d) is relational. Thus, we will discuss them separately.

*Tasklet (c).* Based on the characterisation in section 3.1, tasklet (c) appears to be the least cognitively demanding among the considered ones. And as many as 60% of students were indeed able to identify the (large) overlap relative to the output state.

Examples of the most common acceptable justifications in words are: “We cannot know how many times the loop repeats” (pattern *Uni1*, 31%); or “The final state is the same,  $x = 0$ , for any multiple of 5.” (*Uni2*, 12%). Moreover, 9 students attempted to write the code shown in Figure 6, pattern *Uni5*, which led them to conclude that the reversal was not feasible.

Most of the 25% prestructural answers reported transformations to the while statement that mimic the transformations performed for items (a) and (b), by editing both the variable update and the condition as shown in pattern *Pre1*. It appears none of these superficial edits were tested; in fact, 18 students wrote code giving rise to no iterations or infinite loops in a large number of relevant cases (patterns *Pre2* and *Pre3*, the latter being shown in Figure 6). Finally, 15% of the answers were empty.

*Tasklet (d).* As expected, tasklet (d) turned out to be quite challenging: although 65% of the students had correctly conjectured that the loop is reversible, only one third of them, i.e. 21% overall, managed to write a correct reversing program and about 10% wrote code at the multistructural level.

The challenges faced by a novice programmer in order to approach tasklet (d) are manifold:

- Students have first to understand the computation as a whole, i.e. that the for loop implements a cumulative sum on a vector (usually this aspect is implicit in their answers).
- Then, they have to make the link between final state (cumulative sum) and original state. In short, they should realise that the value of any element but the first one can be restored by subtracting the previous one (its final value). (SUB) Although this appears simple, some students (about 11%) failed to see such connection, as shown in this instance of pattern *Uni3*: “We don’t know  $x$ ’s previous values to be subtracted.”
- Once they identify that SUB link, they have to plan a loop to process all vector’s elements. (PRC) Although seemingly obvious, there are solutions where all vector elements are left unchanged, see for example Figure 7, item *Pre\**.

- More importantly, students have to *be aware* of the dependencies between actions at subsequent iterations, e.g. by choosing a smoother processing order. (DEP)
- Without DEP it is not possible to write correct code, but even when students are aware of the dependencies, they still have to *write* nontrivial reversing code, requiring mastery of the aforementioned dependencies and where the role of  $x$ , if used, must be kept *consistent*. (COD)

As to the last two points, DEP and COD, each original value for index  $i$  is now  $v[i] - v[i-1]$ ; as a consequence, the update needs to consider data dependencies and either (i) update the elements from last to first using a *downward* loop (pattern *Rel1* in Figure 7) or (ii) preserve the partial sum at  $i-1$  while proceeding upwards to the next step (pattern *Rel2* in Figure 7). Based on the experience of Kumar and Dancik relative to the high school, “down-counting loops are more difficult for computer science students than up-counting loops” [16]. However, the main difficulty is not to write a correct downward loop but to realise the need to reverse the loop direction.

A potential reason why students’ code fails to manage the dependencies between subsequent iteration steps is that a variable, usually  $x$ , plays conflicting roles in the loop body. Consider, for instance, the code fragment for the *Mul1* pattern (see Figure 7). While we could infer from the use of a downward for that the student was aware of such dependencies,  $x$  is used in two different roles: in the right-hand side of the first inner assignment it is implicitly meant to hold a partial sum, whereas in left-hand side and in the next statement its value is supposed to be the restored original value of  $v[i]$ . Incidentally, in terms of the roles-of-variables pedagogical model [4], the above roles would be *gatherer* and *temporary*, respectively.

Students’ answers reveal the following combinations of the above features, each related to a corresponding SOLO category:

- SUB and/or PRC (usually both, as for items *Uni1* and *Uni2* in Figure 7): unistructural code;
- SUB, PRC and DEC (DEC = insight suggesting *awareness* of dependencies, e.g. *Mul1* in Figure 7): multistructural code;
- SUB, PRC, DEC and COD (i.e. essentially correct code, e.g. *Rel1* or *Rel2* in Figure 7): relational code;

Notice, in particular, how the ascending vs. descending order of iteration can make a big difference by comparing the correct and compact solution *Rel1* to the correct but more involved *Rel2* or to the superficially similar but incorrect *Uni2* that disregards DEP. The difficulty of approaching the problem by an upward loop emerges clearly from the data in Table 3, that reports the numbers of correct and incorrect solutions relative to the chosen processing order.

*Consistency.* So far we have looked at the student performance for each tasklet. We expect most students to reason at similar or adjacent SOLO levels for every sub-question. We can partition the students by their overall performance in subgroups as described in Table 4.

As we have identified tasklet (d) as challenging for K11–12 students, we consider students are working at relational level even when they skip the last task. Of the 50 students classified at that level, 27 were fully correct, 13 made a partial attempt relative to

```

if (x <= 0)
    x = x - 1;
else
    x = x + 1

```

(i) Mul1

```

if (x < 0)
    x = x - 1;
else
    x = x + 1;

```

(ii) Uni1

```

if (x >= 0)
    x = x - 1;
else
    x = x + 1;

```

(iii) Uni2

Figure 5: Examples of partially correct code patterns, all alleged to be reversals of the conditional construct (b).

```

while (x <= 4)
    x = x + 5;

```

*"It is not reversible because if with the original code  $x = 15$  the outcome will be  $x = 0$ , but with the code I propose at the end of the loop  $x$ 's value will be 5."*

(i) Uni5

```

while (x < 5) {
    x = x + 5;
}

```

(ii) Pre1

```

while (x >= 0)
{
    x = x + 5;
}

```

(iii) Pre3

Figure 6: Examples of recurrent patterns for tasklet (c), used to reason correctly (Uni5) or alleged to reverse it.

```

for (int i = v.length - 1; i > 0; i--) {
    v[i] = v[i] - v[i - 1];
}

```

(i) Rel1

```

for (int i = 0; i < v.length; i = i + 1) {
    v[i] = v[i] - x;
    x = v[i] + x;
}

```

(ii) Rel2

```

for (int i = v.length - 1; i > 0; i = i - 1)
{
    x = x - v[i];
    v[i] = x;
}

```

(iii) Mul1

```

for (int i = 0; i < v.length; i = i + 1)
{
    x = x - v[i];
    v[i] = x;
}

```

(iv) Uni1

```

for (int i = 1; i < v.length; i++)
{
    v[i] = v[i] - v[i - 1];
}

```

(v) Uni2

```

int x = 0;
for (int i = v.length; i = 1; i--) {
    x = v[i];
    x = x - v[i];
}

```

(vi) Pre

Figure 7: Examples of recurrent code patterns meant to reverse loop (d).



item (d) and 10 did not answer it. Most students work at adjacent levels, with less than 14% working below *Uni*.

**Table 3: Use of downward vs. upward for loop to reverse code (d).**

	correct	incorrect	total
upward loop	4	37	41
downward loop	39	28	67
overall	43	65	108

**Table 4: Students performance across all four tasks**

Level	Description	%
Rel	(a) to (c) fully correct.	24.4%
R/M	all correct except one being <i>Mul</i> or empty	10.7%
Mul	all answers are above <i>Uni</i> (excluding c).	12.7%
M/U	some answers above <i>Uni</i> , none below	8.3%
Uni	mostly <i>Uni</i> and one empty	15.6%
Pre	two entries <i>Pre</i> or empty, none above 2	6.8%
Empty	three of four empty entries	6.8%
Mixed	Non adjacent levels in non-empty answers	15%

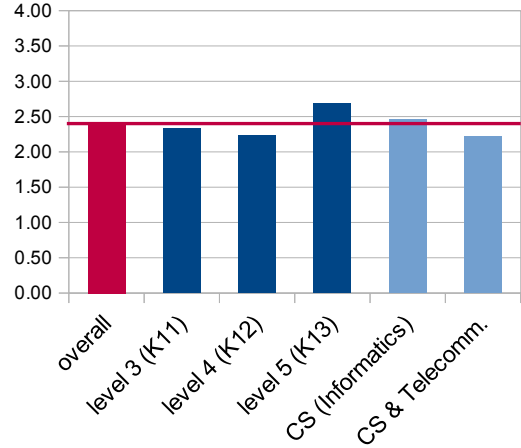
Overall, 15% of students show significant differences in comprehension across the four tasks: for example, they worked at relational level with conditionals, but at prestructural level for one of the iterative tasklets. Only one student failed to see the overlap in tasklet (c), in fact the easiest endeavour, while was relational over other tasklets. Another got tasklet (b) correct but (a) only at unistructural level, while skipping (c) but answering (d) correctly.

### 4.3 Instruction level impact

So far, the analysis has been concerned with the whole cohort. We will now look at the performance for specific subgroups. As we can see from Figure 8, where are reported the average SOLO means covering tasklets (a), (b) and (d), the instruction level had a limited impact on performance. (Notice that tasklet (c) has not been considered because the relational and multistructural levels do not apply to it.)

A more detailed exploration of each item is provided in Figure 9. We can see there are only minor differences between the K11 and K12 cohorts, but the K13 cohort outperforms the rest by about 20% in tasklets (a–c) and of more than 10% in tasklet (d). It is conceivable that this is due to more coding practice and fluency, which means they can produce better code for tasklets (a) and (d), as well as a better ability to describe their reasoning for tasklets (b) and (c).

For the conditionals (a) and (b) at least half of the class are working at multistructural level and above. This indicates the task can be a good learning exercise at any level because it provides a desirable difficulty [2]: the task is challenging enough for students at relational level to revise and confirm their correct mental models, while students at lower levels who partially understand the task can learn from their mistakes. Similar considerations apply to the



**Figure 8: SOLO mean covering the general performance in tasklets (a), (b) and (c) for different student subgroups.**

while loop (c), that appears to be fully understood by a little more than half of the K11 and K12 cohorts.

As expected, tasklet (d) is challenging for all levels, with only the K13 cohort reaching a 40% of answers at multistructural level or above.

## 5 DISCUSSION

In this section we will first revisit the three research questions, then discuss other interesting findings and then draw some conclusions and implications for educators.

### 5.1 Research questions

Based on the results reported in section 4, we will now answer each research question.

*RQ1 - Can student grasp the state transformations enacted by simple conditionals and loops in an accurate and comprehensive way?* If we focus on the first three items, we can see 70% of K13 students have a good grasp of conditional statements and simple while loops compared to 40-50% of their K11–12 counterparts. Similarly, students that attempted the last item exhibit a good comprehension of the code transformation in terms of describing the final state and its link to the original state. Hence, we can conclude that most students have managed a reasonable grasp of conditional and iterative statements.

*RQ2 - When a statement is reversible, are students able to write correct code to recover the original state?* The answer to this question needs to be qualified, due to the difficulty of item (d). The fact 20-30% of high school students completed this challenging task indicates their coding skills are above what is expected.

Most of those students used a downward for loop to undo the state transformation of tasklet (d), in spite of the fact that most examples, both from textbooks and teachers, are stereotypically upwards, hence several of them seem to have developed higher-order thinking skills.



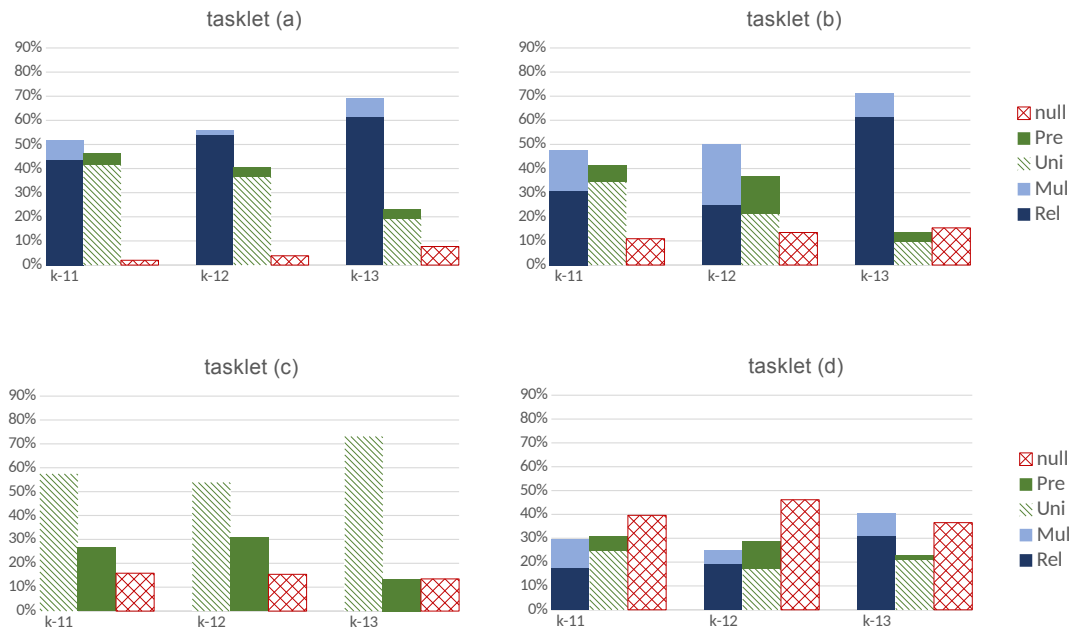


Figure 9: distribution of SOLO ratings for each task and for each student's age subgroup.

On the other hand, there is some concern for the 30% of K11–12 students that made poor attempts to reverse the while loop, and seem to manipulate conditions and update statements without understanding their overall effect. This is best understood looking at patterns *Uni5* and *Pre1* shown in figure 6. The latter shows students do not reflect after editing the code on the limited scope of their transformation which will only reverse numbers in the range {5, 9}. Thus we can see ability to write code is evenly spread over three ranges:

**high** (above average) their code is correct and well-thought as explained above for item (d).

**medium** (average) for the range of students that can reverse conditionals, and their code is mostly correct.

**low** (below average) for those that lack comprehension of the code they produce, even when is correct, such as in task (a).

There are also a small number of students (7%) that did not attempt to code or reason about at least three tasklets.

*RQ3 - Is student's performance consistent over all tasks, reflecting their current level of comprehension? Or does their comprehension vary for each task example?*

Most student answers are mapped into the same or adjacent SOLO levels, as described in table 4 reflecting their current code comprehension level. However 15% show marked differences in performance across tasks. In particular, four students that completed task (d) failed to see the overlap in either items (b) or (c), while ten students that show the overlap in item (b), failed to see it in item (c). Thus, their comprehension is patchy. These students are probably representative of the group that would benefit most from code comprehension activities, as they would become aware of the inconsistencies in their solutions.

## 5.2 Other findings

The results of the analysis seem to corroborate, in the high school context, the findings of previous work addressing students' difficulties with conditionals and loops, see in particular [6, 13].

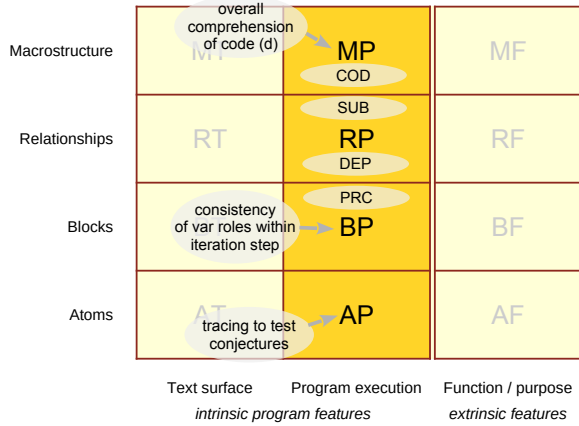
It is also interesting to note that the percentages of SOLO relational justifications (as well as relational+multistructural) decrease precisely in accordance with the abstraction level of the tasklets.

Students working at low SOLO levels appear to lack any skills to analyse the code they read or produce. In fact, from the 70 students who chose the *Yes* option for both items (a) and (b), only 4 provided a really accurate justification for item (c). Thus, 66 students (32%) are answering without showing any analytical skills. The lack of strategies to test their conjectures about program properties may be due to laziness, overconfidence or lack of tracing skills. From a pedagogical perspective, tracing exercises alone that select suitable input cases may not be enough. We should explicitly teach and assess strategies to test programming tasks.

*Tasklet (d) depth.* The analysis of the answers relative to tasklet (d) was especially instructive because of the richness of its implications. When designing the task, we exhibited a *expert blind spot*: although we new the task to be challenging, we were not fully aware of all the aspects that could potentially emerge by examining the code provided by the students. This can be better appreciated by looking at the features identified in section 4.2 in terms of the block model framework.

As shown diagrammatically in Figure 10, students are required to work at different levels of abstraction in the program execution dimension. At the topmost macrostructure level they have to understand the overall behaviour of code (d) (reading comprehension) and write a reversing program (COD). The connections between

the original value of a vector element and the final value of two subsequent elements (SUB) and the treatment of dependencies (DEP) pertain to the relationships level.



**Figure 10: Mapping of the features implied in tasklet (d) into cells of the Block Model.**

A tentative outline of a loop processing all the vector elements (PRC) can be drafted by considering the looping block as a whole and neglecting details about dependencies, so working at the block level of abstraction. Similarly, although the roles of variables have their effect on the dependencies, possible inconsistencies can be detected by just looking at the block representing an iteration step. Finally, the possible use of tracing to test conjectures about program execution can be done at the atomic level.

### 5.3 Implications for instructors

The instruction level appears to have a limited impact on students' performance, what suggests that reversibility tasks are appropriate for students attending the middle years of high school.

From a pedagogical perspective, designing "low-ceiling" reversibility tasks could be a useful instrument aimed at assessing and fostering students' mastery of basic program constructs from K11 onwards. In particular, similar tasks could help to focus on the need to test program behaviour carefully, and may provide opportunities to examine code at different levels of abstraction and practice with varied success higher-order thinking skills.

Tasklet (d) depth presented a step challenge to students; on the other hand, such complex task allows to expose the links between reading, writing and testing and provides many lessons to be learned. Thus, to use this task as a learning event we should scaffold it, for example by decomposing it into two subtasks: (d1) explain the computation carried out by this small program and reason if is reversible or not; and (d2) write the code that reverse this computation (relative to a different code fragment of similar structure).

More teaching efforts should explicitly introduce methods to approach and analyse a programming task, in particular how to identify suitable test cases in order to check the conjectures on code behaviour.

In addition, instructing students to analyse the code in terms of variable roles could be helpful to let them detect, if not avoid, frequent mistakes, such as that illustrated by item *Uni1* in Figure 7 and commented above.

### 5.4 Future work and perspectives

This study has presented 4 tasklets to explore code comprehension in a comprehensive way. Other approachable tasklets should be developed exploring additional features of core programming constructs.

Future work will also expand on our implications for educators by (i) exploring ways to scaffold tasklet (d) by decomposing it into more focused subtasks, in particular to test separately code reading comprehension and code writing skills; (ii) testing these and related tasks with different cohorts and (iii) developing and testing guidelines for novice programmers.

## 6 CONCLUSIONS

In this paper we explored a promising perspective on code comprehension by asking high-school students to reason about reversing conditional and iteration constructs. We analysed students' performance on four programming tasklets by extracting recurrent patterns in their answers and categorising their code and explanations within the framework of the SOLO taxonomy. We also analysed the structure of the considered tasks in terms of Schulte's Block Model.

Vahrenhold et al. review of studies in computer science education at K–12 levels [37] concluded by inviting researchers to undertake multi-institutional and multinational projects "in the school sector" similar to those carried out within the academic milieu (e.g. [7, 19, 24]). According to Vahrenhold et al., in spite of "the significantly greater complications involved in the school sector," we should nonetheless "aspire to similar studies in schools. This is a grand challenge for Computer Science Education Research."

This paper is a first step in that direction as it covers students from different institutions. The reversibility tasks could represent a useful instrument for multinational projects in this respect, as this study could be replicated in different contexts (and we would be happy to collaborate with educators interested in trying similar investigations).

In the light of our exploration, we think that the reversibility concept could also be a useful pedagogical instrument to assess and develop students' program comprehension in the classroom.

### Acknowledgments

We are very thankful to the teachers who collaborated with us to administer the test to their students.

## REFERENCES

- [1] J. B. Biggs and K. F. Collis. 1982. *Evaluating the quality of learning: The SOLO taxonomy*. Academic Press, New York, USA.
- [2] Elizabeth Bjork and Robert Bjork. 2011. Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society* 2 (01 2011), 56–64.
- [3] R. Bornat, S. Dehnadi, and D. Barton. 2012. Observing Mental Models in Novice Programmers. In *Proc. 24th Annual Workshop of the Psychology of Programming Interest Group* (London Metropolitan University). Article 6, 7 pages.

- [4] Pauli Byckling and Jorma Sajaniemi. 2006. A role-based analysis model for the evaluation of novices' programming knowledge development. In *ICER '06: Proceedings of the second international workshop on Computing education research* (Canterbury, United Kingdom). ACM, 85–96.
- [5] Ibrahim Cetin. 2015. Student's Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective. *Canadian Journal of Science, Mathematics and Technology Education* 15, 2 (Feb. 2015), 155–170. <https://doi.org/10.1080/14926156.2015.1014075>
- [6] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) (SIGCSE '14). ACM, New York, NY, USA, 695–700. <https://doi.org/10.1145/2538862.2538966>
- [7] Tony Clear, J.L. Whalley, Phil Robbins, Anne Philpott, Anna Eckerdal, and M. Laakso. 2011. Report on the final BRACElet workshop: Auckland University of Technology, September 2010. (2011).
- [8] Kevin Cox and David Clark. 1998. The Use of Formative Quizzes for Deep Learning. *Computers & Education* 30, 3-4 (April 1998), 157–167. [https://doi.org/10.1016/S0360-1315\(97\)00054-7](https://doi.org/10.1016/S0360-1315(97)00054-7)
- [9] David Ginat and Michal Armoni. 2006. Reversing: An Essential Heuristic in Program and Proof Design. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) (SIGCSE '06). ACM, New York, NY, USA, 469–473. <https://doi.org/10.1145/1121341.1121488>
- [10] Shuchi Grover and Satabdi Basu. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) (SIGCSE '17). Association for Computing Machinery, New York, NY, USA, 267–272. <https://doi.org/10.1145/3017680.3017723>
- [11] Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. 2018. Novice Programmers' Reasoning About Reversing Conditional Statements. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) (SIGCSE '18). ACM, New York, USA, 646–651. <https://doi.org/10.1145/3159450.3159499>
- [12] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland UK) (ITICSE-WGR '19). Association for Computing Machinery, New York, NY, USA, 27–52. <https://doi.org/10.1145/3344429.3372501>
- [13] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) (SIGCSE '10). ACM, New York, NY, USA, 107–111. <https://doi.org/10.1145/1734263.1734299>
- [14] Erkki Kaila, Rolf Lindén, Erno Lökkila, and Mikko-Jussi Laakso. 2017. About Programming Maturity in Finnish High Schools: A Comparison Between High School and University Students' Programming Skills. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) (ITICSE '17). Association for Computing Machinery, New York, NY, USA, 122–127. <https://doi.org/10.1145/3059009.3059021>
- [15] F.J. King, L. Goodson, and F. Rohani. 1998. Higher order thinking skills: Definitions, strategies, assessment. Florida State University – Center for Advancement of Learning and Assessment.
- [16] A. Kumar and G. Dancik. 2003. A tutor for counter-controlled loop concepts and its evaluation. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, Vol. 1. T3C–7. <https://doi.org/10.1109/FIE.2003.1263331>
- [17] Raymond Lister. 2007. The Neglected Middle Novice Programmer: Reading and Writing without Abstracting. In *Proceedings of the 20th Conference of the National Advisory Committee on Computing Qualifications (NACCCQ '07)* (Port Nelson, New Zealand), S. Mann and N. Bridgeman (Eds.), 133–140.
- [18] Raymond Lister. 2011. Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conf. Res. Pract. Inf. Technol. Ser.* 114 (2011), 9–18.
- [19] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITICSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) (ITICSE-WGR '04). ACM, New York, NY, USA, 119–150. <https://doi.org/10.1145/1044550.1041673>
- [20] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) (ITICSE '06). ACM, New York, NY, USA, 118–122. <https://doi.org/10.1145/1140124.1140157>
- [21] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proc. 4th Int. Workshop on Computing Education Research* (Sydney, Australia) (ICER '08). ACM, New York, USA, 101–112. <https://doi.org/10.1145/1404520.1404531>
- [22] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. 2017. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITICSE Conference on Working Group Reports* (Bologna, Italy) (ITICSE-WGR '17). ACM, New York, USA, 47–69. <https://doi.org/10.1145/3174781.3174784>
- [23] Philipp Mayring. 2014. *Qualitative content analysis: theoretical foundation, basic procedures and software solution*. Klagenfurt, Austria.
- [24] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.* 33, 4 (Dec. 2001), 125–180. <https://doi.org/10.1145/572139.572181>
- [25] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. 2010. Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research* (Aarhus, Denmark) (ICER '10). ACM, New York, NY, USA, 69–76. <https://doi.org/10.1145/1839594.1839607>
- [26] Claudio Mirolo and Cruz Izu. 2019. An Exploration of Novice Programmers' Comprehension of Conditionals in Imperative and Functional Programming. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland UK) (ITICSE '19). ACM, New York, NY, USA, 436–442. <https://doi.org/10.1145/3304221.3319746>
- [27] Jean Piaget. 1999. *The Construction of Reality in the Child*. Routledge. Original edition: La représentation du monde chez l'enfant, F. Alcan, Paris, 1926.
- [28] Ebrahim Rahimi, Erik Barendsen, and Ineke Henze. 2017. Identifying Students' Misconceptions on Basic Algorithmic Concepts Through Flowchart Analysis. In *Informatics in Schools: Focus on Learning Programming*, Valentina Dagièné and Arto Hellas (Eds.). Springer International Publishing, Cham, 155–168.
- [29] Ian Sanders, Vashiti Galpin, and Tina Götschi. 2006. Mental models of recursion revisited. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) (ITICSE '06). ACM, New York, USA, 138–142.
- [30] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) (ICER '08). ACM, New York, NY, USA, 149–160. <https://doi.org/10.1145/1404520.1404535>
- [31] Philipp Shah, Dino Capovilla, and Peter Hubwieser. 2015. Searching for Barriers to Learning Iteration and Runtime in Computer Science. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) (WiPSCE '15). Association for Computing Machinery, New York, NY, USA, 73–75. <https://doi.org/10.1145/2818314.2818326>
- [32] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to Assess Novice Programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (Madrid, Spain) (ITICSE '08). ACM, New York, USA, 209–213. <https://doi.org/10.1145/1384271.1384328>
- [33] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2009. A Taxonomic Study of Novice Programming Summative Assessment. In *Proc. 11th Australasian Conf. on Computing Education - Volume 95* (Wellington, New Zealand) (ACE '09). Australian Computer Society, Inc., Darlinghurst, Australia, 147–156.
- [34] Renske Smetsers-Weeda and Sjaak Smetsers. 2017. Problem Solving and Algorithmic Development with Flowcharts. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (Nijmegen, Netherlands) (WiPSCE '17). Association for Computing Machinery, New York, NY, USA, 25–34. <https://doi.org/10.1145/3137065.3137080>
- [35] Peter Sutherland. 1999. The application of Piagetian and Neo-Piagetian ideas to further and higher education. *International Journal of Lifelong Education* 18, 4 (1999), 286–294. <https://doi.org/10.1080/026013799293702>
- [36] Donna Teague and Raymond Lister. 2014. Programming: Reading, Writing and Reversing. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education* (Uppsala, Sweden) (ITICSE '14). ACM, New York, USA, 285–290. <https://doi.org/10.1145/2591708.2591712>
- [37] Jan Vahrenhold, Quintin Cutts, and Katrina Falkner. 2019. Schools (K–12). In *The Cambridge Handbook of Computing Education Research*, S. A. Fincher and A. V. Robins (Eds.). Cambridge University Press, Cambridge, Chapter 18, 547–583. <https://doi.org/10.1017/9781108654555>
- [38] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. *Conferences in Research and Practice in Information Technology Series* 114 (2011), 37–45.