Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers

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

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

ABSTRACT

Novice programmers should develop program comprehension skills as they learn to code so that they are able both to read and reason about code created by others, and to reflect on their code when writing, debugging or extending it. This work takes a little-explored perspective on the comprehension of small programs by asking students to decide if two code segments are equivalent or not in terms of carrying out the same computation.

A variation of Euclid's algorithm, that extends the greatest common divisor calculation to more than two numbers, was chosen for this work, as it has an adequate level of complexity and its semantics are not obvious. Four program transformations of the original code were developed: two transformations were equivalent and two were not. 73.5% of students were able to identify correctly the four options and 75.5% provided good insights on the equivalent program flow to justify their choices. The overall task has a SOLO mean of 3.19, which indicates code equivalence is a suitable and approachable task to analyse program execution at novice level.

In addition, the data analysis suggests that students' code-reading abilities beyond basic tracing may be generally underestimated and we should investigate how to bridge the potential gap between reasoning about program execution and extracting its purpose.

KEYWORDS

code comprehension; semantic equivalence; novice programmers; program execution

ACM Reference Format:
Cruz Izu and Claudio Mirolo. 2020. Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers. In *Innovation and Technology in Computer Science Education (ITiCSE '20), June 15–17, 2020, Trodheim, Norway*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341525.3387425

1 INTRODUCTION

Most CS1 courses teach programming by introducing one language construct at a time, providing examples, and asking students to implement simple algorithms that required that new construct. Once each construct is well understood, it is simpler to compose longer programs by combining multiple constructs in sequential or nested ways. However, previous work, e.g. [2, 4, 11, 15], shows some students have a superficial understanding of the notional machine that causes misconceptions, which may appear later on the course as program complexity increases.

Submitted to ITiCSE '20, June 15–17, 2020, Trodheim, Norway – Published article: https://doi.org/10.1145/3341525.3387425

A major objective of this study is to investigate on novices' ability to construct mental abstractions of the overall behaviour of small program chunks when executed on a *notional machine*. In terms of the *Block Model* framework [24], this amounts to trying to analyse to which extent — and how — learners are able to move from the atomic level (AP) to the relational or macro levels (RP, MP) within the program dimension (P) represented in the related matrix.

The tracing tasks of widespread use are not satisfactory enough to this aim, as pointed out by Izu et al. [9], since they can be achieved by reasoning mostly at the atomic level. On the other hand, "Explain in plain English" (EiPE) tasks [5, 22] focus only on the program's purpose and not on its execution. In conventional writing tasks, such as those asking to complete or modify code, program behaviour and functional purpose are so strictly entangled with each other, that it may not be clear if failures to achieve the task are to be ascribed to poor abstraction relative to the operation of the notional machine rather than to problem-solving issues. A study by Lister et al. [15], for instance, appear to suggest that weaknesses in the former are often at the very root of the latter.

To quote Luxton-Reilly et al., as a consequence of using assessments that "combine numerous heterogeneous concepts [...] teachers may not be able to diagnose the actual difficulties faced by students and students are not provided with accurate feedback about their achievements" [18], thus the need for new comprehension tasks that facilitate identification of misconceptions at the macro level [24].

Recent work to address the operational dimension of program comprehension independently from other dimensions proposes the use of reversibility tasks [8, 21]. However, such tasks require students to *write* tiny (reversing) programs, whereas we are looking for tasks that test only their reading comprehension.

Asking to decide on equivalence between short programs may fill this gap: in order to understand if two code fragments are equivalent in terms of state transformation, it is necessary not only to read/trace the code but to construct a viable abstraction of the overall computation (without the need of formulating it in words) while reasoning at the program behaviour level.

A new task is proposed in which students are asked to assess the functional equivalence of a short program with four structural transformations of it. Each transformation changes the program flow by editing the order of conditions to be tested and/or the nested constructs (*if* vs. *while*) inside an enclosing loop. Thus, the focus of the analysis should be on the overall program flow.

This task was presented to students at the end of a full year CS1 course. We should note these students were not exposed to this type of task before, but they have experience tracing small programs. In other words, the endeavour was new to them, aiming to assess their

higher order skills instead of simple recall. The analysis of their answers provides an interesting peek into their current abstraction abilities. In particular, this study addresses two research questions:

RQ1. Are students able to compare two *similar* small programs and decide if they are functionally equivalent?

RQ2. What level of abstraction is reflected in their justifications for equivalence?

This study provides new insights into novice programmers' ability to analyse code in a comprehensive way. We will use the SOLO taxonomy to classify how well students complete the task and the Block Model framework as a reference to measure their level of abstraction.

In addition, comparing small programs for equivalence should help to strengthen their design skills by requiring them to consider alternative implementations for a given task, as well as by providing their first exposure to equivalence transformations for code refactoring.

The rest of the paper is organised as follows: after outlining some background in Section 2, Section 3 describes how the empirical data was collected and analysed; Section 4 presents the quantitative and qualitative results, which are further discussed in Section 5 and summarised in Section 6

2 BACKGROUND

Program comprehension by novices has been the subject of studies flourished in the 80s, e.g. [7, 27], and since then has been explored from a variety of perspectives, addressing the interconnections between code tracing, reading and "chunking" abilities [6, 14, 16, 17]; the mental models of program behaviour [2, 23]; the understanding of loops and nested loops [3]; the issues connected with basic concepts of language notation and operational semantics [18].

That even simple programming constructs may be challenging emerged, in particular, from the analysis of a huge dataset by Cherenkova et al., who reported that "a significant number [of students ...] exhibit the common errors of failing to check the border condition or reversing the conditional" [4].

In this general context, the focus of our present work is on reading comprehension, as demonstrated through the ability of comparing programs to establish or disprove their equivalence. Some emphasis on program comparison has been previously put by Thompson et al. [28], who have analysed students' ways of classifying code fragments based on perceived similarities and differences; however, to the best of our knowledge the task presented here has not been proposed before. As to the importance of the reading skills, in their recent survey on introductory programming Luxton-Reilly et al. report that "the relationship between code reading and other skills, notably code writing, has also been widely studied, leading to the conclusion that code-reading skills are prerequisite for problem-solving activities including code writing" [19].

The reference frameworks for the core part of our analysis are the SOLO taxonomy [1] and the Block Model [24]. SOLO is an instrument of widespread use to assess code reading and writing tasks, e.g. [17, 26, 29]: from this perspective, the learners' achievements are classified in terms of complexity and quality of the interrelationships between parts they can master. The Block Model, on the other hand, is precisely meant to analyse key aspects of program comprehension, whose merits have been recognised, e.g. [30], in particular for the accuracy of the resulting categorisation.

The input to the program **P** shown below is an array v of *positive* integers. In general, **P** applies a transformation to the values stored in the array and its state at the end of the iteration represents its output.

```
// ∀k ∈ [0, v.length−1] . v[k] > 0
int i = 0, j = v.length - 1;
while ( i < j ) {
  if ( v[i] < v[j] ) {
    v[j] = v[j] - v[i];
  } else if ( v[i] > v[j] )  {
    v[i] = v[i] - v[j];
  } else {
    i = i + 1;
}}
```

Analyse the following programs (**A**–**D**) and decide for each of them if it is *equivalent* to **P**, i.e. if for any input array of positive integers the resulting output is the same as the one produced by **P** (and possibly it does not terminate whenever **P** does not terminate).

If your answer is *Yes*, explain in a few words the reasons why you think that the program is equivalent to **P**. If your answer is *No*, provide an example of input in which the output of the program is different from that produced by **P**.

A.

```
int i = 0, j = v.length - 1;
while ( i < j ) {
  if ( v[i] < v[j] ) {
    v[j] = v[j] - v[i];
  }
  if ( v[i] > v[j] )  {
    v[i] = v[i] - v[j];
  }
  i = i + 1;
}
```

B.

```
int i = 0, j = v.length - 1;
while ( i < j ) {
  while ( v[i] < v[j] ) {
    v[j] = v[j] - v[i];
  }
  while ( v[i] > v[j] )  {
    v[i] = v[i] - v[j];
  }
  if ( v[i] == v[j] ) {
    i = i + 1;
}}
```

C.

```
int i = 0, j = v.length - 1;
while ( i < j ) {
  while ( v[i] > v[j] ) {
    v[i] = v[i] - v[j];
  }
  if ( v[i] < v[j] )  {
    v[j] = v[j] - v[i];
  } else {
    i = i + 1;
}}
```

D.

```
int i = 0, j = v.length - 1;
while ( i < j ) {
  if ( v[i] < v[j] )  {
    v[j] = v[j] - v[i];
  } else {
    i = i + 1;
  }
  while ( v[i] > v[j] ) {
    v[i] = v[i] - v[j];
}}
```

Figure 1: Equivalence task.

3 METHODOLOGY

In this section we describe the equivalence task used in our investigation. Then, we outline the data collection process. Finally, we present the criteria of our analysis, based on the SOLO taxonomy and the Block Model framework.

The task examined in this paper is shown in Figure 1. This task combines two formats of measuring higher order thinking skills: *selection* and *explanation* [12]. For each of the four programs labeled A–D, students had first to identify whether it is equivalent or not to program P (selection); then they were required to justify their choice (explanation), by either explaining their reasoning or providing a counterexample. The reasons of our interest in exploring students' ability to approach a similar task lie on the features mentioned in the introduction.

Although the proposed task refer to an equivalence concept in terms of input-output relationship, it is worth observing that the equivalent programs (**P**, **B** and **C** in Figure 1) give rise to exactly the same sequence of state transitions, via assignment statements, the only difference being the flow — number and/or order — of conditions tested. This ensures that in order to realise that they are

equivalent it is not necessary to figure out the underlying purpose or to reason about number properties.

This task was presented at the start of the exam paper when students were fresh and active. The task shown in Figure 1 was worth 25% of the paper's marks and appeared also in a second version, where the programs were slightly modified and the order of the corresponding items was different. Note the equivalent transformations, which we will call Y1/Y2 in the analysis, correspond to items C and B, while the non-equivalent ones, which we call N1/N2 correspond to items A and D in Figure 1.

3.1 Data collection

We collected exam answers from a CS1 exam in June 2019 at the University of ZZ. Each exam paper was anonymised and digitised prior to analysis.

Students had received one semester instruction in Scheme, followed by another semester of Java Programming. The sample (151) includes students from two programs: a standard Computer Science program (77 students), and a new computing program focusing on some specific present-day technologies (74 students).

3.2 Data Coding Methodology

After measuring the percentages of students who had chosen correct options, we applied the SOLO taxonomy to conduct a qualitative analysis of the explanations or counterexamples provided to justify each choice.

SOLO coding. To develop the criteria, the two researchers independently performed a deductive content analysis [20] to rate a sample of 20 students' answers. Initially, we tried to rate each individual answer, however some insights were sometimes implicit or said before (and assumed each answer relies on previous explanations to avoid repetition). Thus, after two iterations we decided to rate the four items as a unique, comprehensive task (in terms of SOLO).

Following one more revision of the equivalent (Y1/Y2) and non-equivalent (N1/N2) codes, we discussed and summarised the expected insights into the list shown in Table 1 and applied them to establish the SOLO classification guidelines reported in Table 2. Once the criteria were set, both researchers rated all test papers, marked the insights found, and used Table 2 to classify each answer. Borderline cases between SOLO levels in connection with implicit insights or minor errors were discussed to determine the final classification.

Table 1: Equivalence task's insights.

Insight	Description
1	No *update* statements are added, removed, or run in a different order relative to P. Only the *condition* flow may change
2	If index `i`'s update is preceded by the equality condition, its location is not important. If is preceded by `else`, needs to make sure it only reaches when `v[i] == v[j]`.
3	Vector updates are equally managed by the original "`while-if`" combination from P or the edited "`while-while`" combinations in B/C/D.

Table 2: SOLO Classification guidelines.

SOLO Level	Options correct	insights given
Empty (0)	—	No justifications made
Prestructural (1)	≤ 2	No meaningful insight provided
Unistructural (2)	2-3	At least one correct insight
Multistructural (3)	3-4	Two correct insights (without errors or contradictions).
Relational (4)	4	All three insights provided.

Block Model coding. Although our SOLO classification indirectly captures the abstraction level student used to express each insight, it focuses mostly on the correctness and completeness of their answer. Thus, we used the Block Model's abstraction levels (atomic, block, relation or macro [24]) to better capture the reasoning depth of their justifications.

As expected, most students' answers pertain to the Program Execution dimension, which is concerned with program flow and data flow. However, we did also find a few answers falling into the function dimension. To make the paper self-contained we will briefly describe the elements of the Block Model matrix found in our coding:

- **AP** : show step-by-step execution (tracing) or output for a specific input.
- **BP** : analyse execution of a block, for example discuss the alternative paths of a nested if statement.
- **RP** : analyse the interactions between blocks that are linked sequentially or nested.
- **RF** : identify the function of a (partial) computation resulting from interactions between code segments.
- **MF** : figure out the main purpose of the whole program.

For full details of the Block Model matrix, please refer to its source [24]. As the negative answers asked for an example, while the positive answers asked for an explanation, it makes sense to code Y1/Y2 and N1/N2 separately. The examples in Subsection 4.3 will clarify further the coding.

4 RESULTS

In this section we present first a quantitative analysis of the rates of correct options, followed by a qualitative analysis of students' explanations, using the SOLO taxonomy to measure the depth of their explanations and the Block Model to identify at what abstraction level they are reasoning about programs.

4.1 Correct options

As shown in the summary of Table 3 (see last column), around 3/4 of students selected all 4 correct options, and less than 10% made two or more mistakes.

If we look at individual program transformations values range from 89% for N1 and Y1 to 93% for Y2, as shown by the "Overall" line in Figure 2. We also split the percentage of success for students that made 4, 3, or 2 correct choices in order to detect any patterns or assumptions that weaker students had made.

Of the 24 students who made **one** incorrect choice, 9 were wrong on N2, 8 on Y1, 4 on Y2 and 3 on N1. It then appears that programs

Table 3: Insights (SOLO level) vs. number of correct options.

correct options	relational justifications 4	3	2	1	0	total
4	52.3%	17.9%	2.6%	0.7%	—	73.5%
3		5.3%	9.9%	1.9%	—	17.2%
2			3.9%	3.3%	0.7%	7.9%
1				1.3%	—	1.3%
total	52.3%	23.2%	16.5%	7.3	0.7%	100%

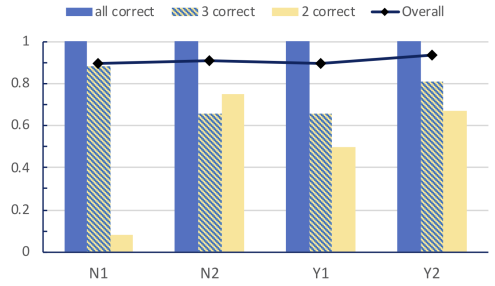

Figure 2: Percentages of correct choices for each program transformation.

N2 (C) and Y1 (D) have been harder to them. Besides, of the 38 students who selected at least one wrong option, 26 made the same choice, both Yes or both No, for N2 and Y1 (and in only one case both were wrong). A possible interpretation of these observations is connected with the structural similarity of N2 and Y1, whose only difference is a swap of the two nested constructs: a number of students probably failed to realise that the implicit condition corresponding to the `else` branch depends crucially on the order of the constructs.

4.2 SOLO analysis

We next move to the SOLO analysis. Table 3 reports the percentage of answers classified at each SOLO level. A related aggregate indicator is the SOLO mean, customarily determined by averaging over the weights assigned to each level, see e.g. [25]: 4=relational, 3=multistructural, 2=unistructural, 1=prestructural, and 0=no justification. Performance on this task resulted in a SOLO mean of 3.19, with 75.5% of the subjects working at the multistructural or relational level.

Besides presenting a few additional figures, in this subsection we will briefly characterise each SOLO level, whereas the excerpts in Table 4 illustrate the insights on which the analysis is based.

Relational answers. The justifications provided for the four correct answers focused clearly on equivalent program flows. The fact the task is focused in comparing two values, `v[i]` and `v[j]`, may help students to easily identify the *implicit* equality condition of the `else` branch within the main loop of program P.

Insight 2 is the most often explicitly described one (see Figure 3), while insight 1 is mainly referred to implicitly in the explanations as well as in the examples. Overall, insights 2 and 3 are commonly

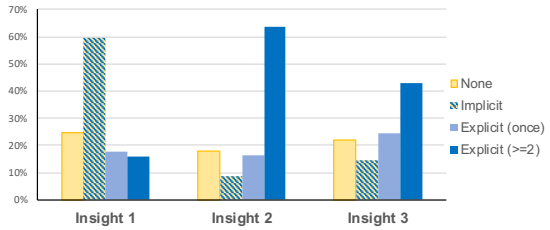

Figure 3: Frequency of task insights (implicitly or explicitly) in student's answers.

repeated or elaborated in subsequent examples, as indicated by their higher frequencies for two or more explicit sightings. Note that in Figure 3 the sum of the percentages for an insight is over 100%, as it may be referred to both implicitly and explicitly, hence counting twice.

Multistructural answers. Justifications at this level are characterised by the following patterns:

- Students selected the correct options but didn't provide any reference to one of the insights in their justifications (6 missed insight 1, 5 insight 2, and 7 insight 3). It is unclear if this can be ascribed to carelessness in articulating the explanation or if they missed some more substantial features. The fact all their choices are correct may support the former.
- Students selected the correct options and essentially addressed all insights, but committed tracing errors when providing the examples for N1/N2 (9 subjects).
- Although referring to all insights, students clearly missed a factor or incorrectly traced the example they provided, and the oversight caused them to incorrectly select one option. There were 8 students in this group; half of them failed to see that D was different to C.

Unistructural/Prestructural answers. At the Unistructural level most students often chose 2 or 3 correct options but made multiple errors when tracing cases that justified their wrong choices. Some managed to articulate one or two insights but the tracing errors indicated small contradictions between what they said and what they traced. We should note also 4 students that have all correct options were classified as unistructural because their explanations were either very short or vague, relating mostly to one of the insights.

We found 11 prestructural answers, 3 of them having at least one insight but also repeated errors and contradictions. Poor answers appear to be due to a superficial analysis of the programs, by stating that they do the same operations while ignoring any flow variations. Others focused on irrelevant facts to make the decision, e.g. if any vector value is set to 0, program P and its equivalent ones won't terminate. There was only one student that made no attempt to analyse the code and was classified as *empty*.

4.3 Block model analysis

Table 4 shows examples of students answers in the program execution dimension at three levels: AP. BP and RP, and Figure 4 shows the percentages of answers classified at each level.

Table 4: Students' justifications examples relative to the program execution dimension (tagged with insights found)

	non-equivalent (N1, N2)	equivalent (Y1, Y2)
AP	N1: *"With input {1, 2, 3, 4}, P gives {1, 1, 1, 1}, whereas A gives {1, 1, 2, 1}"* (inisght 1)	Y1: *"With input {1, 2, 3, 4}, both P and C return as output {1, 1, 1, 1}"* (inisght 1)
BP	N2: *"D is not equivalent to P because* `i = i+1` *is executed when* `v[i]` *is not less than* `v[j]`*, whereas it should be executed only if* `v[i] == v[j]`*."* (insight 2) N2: [In D] *"he index is incremented before testing if* `v[i] > v[j]`*. Thus, the check that index* `i` *=* `[j]` *is lacking."* (insights 2 and 3, the latter being inplicit here)	Y2: [B and P are equivalent] *"since the index is not incremented in every case, but only if* `v[i]` *and* `v[j]` *have the same value, as with P."* (insight 2) Y2: [B] *"gives the same result since program P, instead of using nested* `while`*s to decrement some value in the array, uses a single* `while`*, but the result is the same."* (insight 3)
RP	N2: *"D is not equivalent to P. Indeed,* [the transformation is] *the same at every iteration, except the first one. Indeed, we always arrive at the point* `[if v[i] < v[j]]` *with* `v[i] <= v[j]`*, hence* `i` *is incremented only if* `v[i] == v[j]` *at every iteration but the first one."* (insights 1 and 2)	Y1: [With C] *"*`i` *is incremented only when* `v[i] = v[j]` *since the inner* `while` *loop ensures that* `v[i]` $\leq$ `v[j]` *so that the* `else` *is executed only if* `v[i] = v[j]`*."* (insight 2) Y2: *"P at each cycle decreases the higher value between* `v[i]` *and* `v[j]` *and only if they are equal increases the index* `i`*. B at each cycle executes more cycles where* `v[i]` *decreases if it is greater than* `v[j]` *and conversely; only if* `v[i]` *is equal to* `v[j]` *then the index* `i` *increases."* (all insights, insight 1 being implicit)

Table 5: Students' justification examples relative to the program function dimension.

	equivalent (Y1, Y2)
RF	Y1: [B is equivalent to P] *"also because the program purpose is to transform all the elements of the array* [...] *and to sort them in decreasing order."*
MF	Y2: [B is equivalent to P ...] *"In particular, both P and B compute the GCD of a list of numbers (*`v`*), which at the end of the computation will be found at* `v[v.length-1]`*. The other components of* `v` *contain partial results of the computation."*

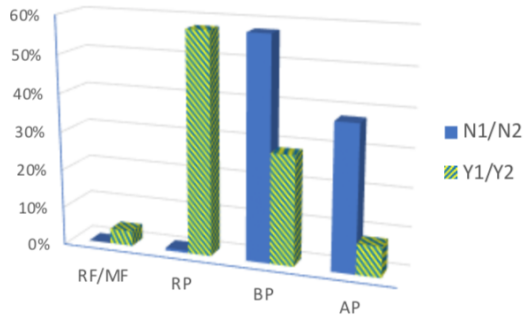

Figure 4: Distribution of YES/NO answers for relevant Block model cell categories.

Functional dimension. Note the reference program P computed the greatest common divisor of the integers stored in an array, but its purpose was not mentioned in the task description. Thus, an interesting question is whether students *spontaneously* look for functional clues in their analysis. In fact, six students, 4 at the relational level and 2 at the macro level, refer to the code purpose; two sample answers are shown in Table 5.

RP – Relational level. Typical explanations of equivalence, as the three listed in Table 4, looked at the block edited in the transformation and described how that changes impact on the block interaction with the other program components.

BP – Block level. In this case the justifications are mostly based on block behaviour. Note this can yield correct or incorrect outcomes, depending on the block location. For instance, the first explanation given for N2 in Table 4 would be in accordance with an incorrect choice for Y1, where the `if-else` block is in a different location.

AP – Atomic level. Examples that provide an input and its output for both programs are classified at this level in accordance with the rationale discussed in [9]. Some students reported the full trace, but many more showed the final result only.

Both RP- and BP-level justifications show that most of our novice cohort decisions, either positive or negative, are based on some abstract pattern of reasoning rather than relying on tracing in order to get insights on program behaviour. As shown in Figure 4, even to disprove equivalence a majority of students' explanations were built upon general block-level arguments, without the need of comparing the outcomes for specific counterexamples. In the case of Y1/Y2, on the other hand, 58% of the answers were given at RP level, 28% at BP, and only 8% at AP. The atomic level usually reflected a superficial approach to the task, as showing an example in which two programs produce the same output, what is no guarantee that this should be the case also with other input instances.

5 DISCUSSION

We proceed now by revisiting the results presented in the previous section in light of the research questions.

5.1 Research questions

RQ1 – Are students able to compare two similar small programs and decide if they are functionally equivalent? Our SOLO analysis answers RQ1 in the positive: both the option choices and correct explanations exceed our expectations, as most complex CS1 assessment tasks are usually around the 50% success range. The SOLO

mean of 3.19 is high compared to other CS1 reading and writing tasks [25]. Furthermore, some of the errors seems to be oversights or failing to list their assumptions.

Interestingly, CS1 students seem to be successful when analysing complex program flows, whereas previous work with a CS1 cohort reported failure to analyse the impact of a simple `if` statement [10]. We attribute the higher success rate, apart from the differences in cohorts, to the fact that all possible path flows are explicit and this guides students' analysis.

RQ2 – What level of abstraction is reflected in their justifications for equivalence? Both the SOLO and the Block Model analysis have indicated that more that half of the CS1 cohort were confident to analyse the code at the relational level and to clearly describe the insights, as shown in Table 4.

Contrary to our expectations, tracing the code with an example may not be students' first choice to figure out program properties, as also confirmed by the significant rate of students (58%) who tried to approach the justifications relative to items N1/N2 in more general terms, in spite of being asked to provide specific counterexamples.

In this respect, we should note that N1 (A) is tricky to trace because at first glance the two "`if`" statements can be mistaken to be mutually exclusive. Instead, once the first `if` statement is executed the condition of the next one may become true, and this subtle pattern seems to trip many students. Similarly, for N2 (D) ignoring the `while` after `i` is incremented may indicate again the preconception that only one condition can be true at each iteration. This may explain why 52% of the 81 papers providing answers for a specific input (AP level) at least one such outcome is incorrect. Tracing errors may be due to carelessness or attempts to reason at multiple levels while tracing, moving from BP down to AP and back in an inconsistent manner.

In short, the equivalence task has engaged students to reason above the atomic level and it is clear from this analysis that many of them have the potential to grasp the macro level of small programs.

5.2 Other findings

Asking students to compare program behaviour seems to provide better outcomes that asking them to "explain-in-words". Although poor performance in EiPE tasks has been ascribed to "not seeing the forest for the trees" [16], it is conceivable that, for a fair amount of students, weaknesses to explain program behaviour may be due to not being able to articulate their reasoning in appropriate and clear ways. Or that some scaffolding is needed to move from the BP/RP levels of abstraction to the BF/RF. Given the limited scope of our exploration, from our analysis we cannot of course draw compelling conclusions regarding the students' code reading abilities. Nevertheless, there is a clear indication that it would be important to test the alternative explanation that, in a significant number of cases, major issues lie in their poor "verbalisation" skills, rather than in their program understanding.

Implications for educators. This work has proposed a new type of code comprehension task that engages students to work at higher levels of abstraction compared with tracing tasks. Educators should consider including similar tasks for multiple reasons. Firstly, analysing a program's execution flow is very useful when debugging and testing code. As explained in [13], any improvement in program comprehension would help novice programmers to develop educated hypotheses about the bug they are trying to locate and fix. In particular, the equivalence task chosen for this study can support weaker students to evaluate the need and impact of chained conditions. Choosing different program transformations to the ones explored here could help novices to reason about other aspects of program execution; for example, another task may direct them to analyse the need to set/reset variables at the start of a subsequent iteration cycle.

Secondly, looking at equivalent programs towards the end of CS1 could also provide opportunities to discuss code quality metrics such as readability and efficiency. For example, once they have identified three equivalent programs (in our sample task, programs P, B and C) you could ask students: Which version is more readable? Is any of the version more efficient than the others?

Limitations. The main limitation of this study is the fact we tested the task with a single cohort, so it is unclear to which extent it can be generalised to other contexts. As with many comprehension tasks, a careful choice of examples that are neither trivial not overly complex is critical to trigger meaningful reading comprehension. Hence, testing the task with a medium size CS1 class is sufficient to demonstrate its potential merits.

6 CONCLUSIONS

While most CS1 students are usually exposed to tracing, which helps them build a correct mental model of the notional machine, such tasks do not force them to consider program execution beyond the atomic level of each command. However, chunking from statement into blocks and their relationships is important when writing, testing and debugging. In this study we have proposed a new code comprehension task with the aim of developing such level of code comprehension.

We devised a new type of task engaging students to reason about equivalence between small programs. The equivalence task was used in a CS1 exam, in which 74% of students succeeded to select the correct equivalent pairs. A thorough analysis, using both SOLO and the Block Model to classify the answers has shown 62% of student are consistently reasoning at the relational level. Hence, this equivalence task is approachable for non-experts, in this case CS1 students which are still novice programmers. It has also proven to be appropriate for our goals of forcing students to think about program execution at the block or relational level. Their reasoning skills turned out to be better than expected: 3 out of 4 students were able to provide at least two insights into the program execution and 53% provided in their own words the three key insights that explain why or why not the program flows are equivalent.

Open lines from this study include (1) further testing of this task with different cohorts and (2) the development of equivalence tasks that explore additional aspects of code transformations, such as when to initialise a variable, or equivalence of iteration constructs.

REFERENCES

[1] J. B. Biggs and K. F Collis. 1982. *Evaluating the quality of learning: The SOLO taxonomy.* Academic Press, New York, USA.

[2] 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*. Article 6, 7 pages.

[3] 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

[4] 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 (SIGCSE '14)*. ACM, New York, USA, 695–700. https://doi.org/10.1145/2538862.2538966

[5] Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. 2012. Some Empirical Results for neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions. In *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123 (ACE '12)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 77–86.

[6] 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

[7] Benedict du Boulay. 1986. Some Difficulties of Learning to Program. *J. of Educational Comput. Research* 2, 1 (1986), 57–73.

[8] Cruz Izu, Cheryl Pope, and Amali Weerasinghe. 2017. On the Ability to Reason About Program Behaviour: A Think-Aloud Study. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, USA, 305–310. https://doi.org/10.1145/3059009.3059036

[9] 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 (ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 27–52. https://doi.org/10.1145/3344429.3372501

[10] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 251–259. https://doi.org/10.1145/2960310.2960324

[11] 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 (SIGCSE '10)*. ACM, New York, NY, USA, 107–111. https://doi.org/10.1145/1734263.1734299

[12] 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.

[13] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a Framework for Teaching Debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference (ACE '2019)*. Association for Computing Machinery, New York, NY, USA, 79–86. https://doi.org/10.1145/3286960.3286970

[14] 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 (NACCQ'07)*, S. Mann and N. Bridgeman (Eds.). 133–140.

[15] 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 (ITiCSE-WGR '04)*. ACM, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[16] 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 (ITICSE '06)*. ACM, New York, USA, 118–122. https://doi.org/10.1145/1140124.1140157

[17] 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 (ICER '08)*. ACM, New York, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[18] 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 (ITiCSE-WGR '17)*. ACM, New York, USA, 47–69. https://doi.org/10.1145/3174781.3174784

[19] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion)*. ACM, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[20] Philipp Mayring. 2014. *Qualitative content analysis: theoretical foundation, basic procedures and software solution*. Klagenfurt, Austria.

[21] 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 (ITiCSE '19)*. ACM, New York, NY, USA, 436–442. https://doi.org/10.1145/3304221.3319746

[22] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in Plain English' Questions: Implications for Teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, USA, 385–390. https://doi.org/10.1145/2157136.2157249

[23] Ian Sanders, Vashti 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 (ITICSE '06)*. ACM, New York, USA, 138–142.

[24] 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 (ICER '08)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[25] 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 (ITiCSE '08)*. ACM, New York, USA, 209–213. https://doi.org/10.1145/1384271.1384328

[26] 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 (ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, 147–156.

[27] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: an empirical study. *Commun. ACM* 26, 11 (Nov. 1983), 853–860. https://doi.org/10.1145/182.358436

[28] Errol Thompson, Jacqueline Whalley, Raymond Lister, and Beth Simon. 2006. Code Classification as Learning and Assessment Exercise for Novice Programmers. In *Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications*, S. Mann and N. Bridgeman (Eds.). NACCQ in cooperation with ACM SIGCSE, 291–298.

[29] 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.

[30] Jacqueline Whalley and Nadia Kasto. 2013. Revisiting Models of Human Conceptualisation in the Context of a Programming Examination. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136 (ACE '2013)*. Australian Computer Society, Inc., AUS, 67–76.