Experience Report: Identifying Unexpected Programming Misconceptions with a Computer Systems Approach

Fionnuala Johnson
University of Glasgow
Glasgow, UK
fionnualajohnson7@gmail.com

Stephen McQuistin
University of Glasgow
Glasgow, UK
sm@smcquistin.uk

John O'Donnell
University of Glasgow
Glasgow, UK
john.t.odonnell9@gmail.com

Quintin Cutts
University of Glasgow
Glasgow, UK
Quintin.Cutts@glasgow.ac.uk

ABSTRACT

An increasing number of students arrive at university with programming experience and mental models already formed. These models are often incorrect, with students holding entrenched misconceptions. We posit that our introductory programming course, using Python, is not sufficient for exposing and correcting these mental models. In this paper, we describe a study that investigated the extent to which making explicit connections between our introductory programming and computing systems courses could expose mental models and help identify and fix student misconceptions. We hypothesised that students would develop a correct mental model by creating a low level systems implementation of a high level program. However, while this approach certainly identified student misconceptions, these were significant enough to prevent the students from being able to see the connection between the low and high level versions, and so their mental models were not corrected. This paper will detail the problematic misconceptions that we identified, develop a set of hypotheses for why these misconceptions were held, and suggest future studies.

1 INTRODUCTION

Many students enrolling in university-level introductory programming courses now have programming experience, whether from self-study or formal pre-university education. While these students develop mental models based on this experience, their models are often incomplete or even incorrect. Arguably, effectively identifying and correcting entrenched misconceptions is challenging within the constraints of introductory programming courses and the languages that they teach [8].

As an example, consider the `while` statement in Python. This is often defined as a construct that allows for the execution of a set of statements for as long as a given condition is true [6, 18]. However, the ambiguity of this definition allows for a common misconception to arise. Consider this snippet of Python code:

```
i = 0
while i < 5:
    i = i + 1
    print(i)
```

On entering the final iteration of this loop, i is 4, and is then immediately incremented to 5. That means that the condition, $i < 5$, if evaluated immediately, would be false. It is not clear, from the definition above, what happens next: the value of i is printed, despite the condition being false. Of course, a more rigorous and detailed description of the `while` statement may correct this particular misconception, but whether such detail is not provided in an introductory programming course, or whether students overlook it, such misconceptions can persist.

In this paper, we consider whether misconceptions that students have in programming could be identified, challenged, and corrected through explicit connections to an introductory computer systems course. In this course, students are taught how to translate high level code (written in Python, for example), into a low level systems language. For example, the snippet of Python code above becomes:

```
        i = 0
loop    if not(i < 5) goto endloop
        i = i + 1
        print(i)
        goto loop
endloop
```

This makes the semantics of the `while` statement explicit, eliminating the misconception that the loop condition is evaluated continuously, instead demonstrating that it takes place only once at the start of each iteration.

Misconceptions in novice programming are the source of much computing science education research. Many misconceptions have been traced back to poor mental models, with students lacking a sound execution model of their code. Programming dynamics is considered a threshold concept [12]. Notional machines [5, 14] have been developed to teach novices in introductory programming courses how their code will execute; these may be in the form of visualisers or tracing methods. While there is evidence to show the success of notional machines in creating correct mental models in novices [11], there is less research on its effect on *existing* mental models.

To that end, and to test our hypothesis – that an explicit link between high level programming constructs and low level systems code would deepen students' understanding of high level constructs

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland
© 2022 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland*, https://doi.org/10.1145/3502718.3524775.

– we conducted a study of 16 students enrolled in introductory programming and computing systems courses at a large university. Our study concluded that students find it difficult to make the necessary links between programming and computing systems concepts, opting instead to see them as discrete subjects, and thus developing separate mental models for the same constructs. However, through conducting the study, we made a number of surprising findings, which we will discuss, and hypothesise about, in this paper. In summary, students with pre-university programming experience may:

(1) lack a sound execution model of their high level code (§5.1), preventing links being made between the programming and computer system courses;
(2) hold more than one context-dependent mental model of a computing concept (§5.2), enabling them to hold potentially conflicting models, and hindering the correction of misconceptions; and
(3) believe that the Python interpreter can correct semantic errors (§5.3), allowing the execution of their Python code to differ from the behaviour of low level code they derived during the computer systems course.

The remainder of this paper is structured as follows. Section 2 lays the theoretical foundations for our work. Next, Sections 3 and 4 describe the context and design of our study. We discuss our results, including the hypotheses that we have developed, in Section 5. Finally, Section 6 describes related work, and Section 7 concludes.

2 THEORETICAL FOUNDATIONS

Piaget [2] states that knowledge is organised into schemas, which are equivalent to mental models. We can use schemas to both understand and solve problems. To be functional, schemas must be *in equilibrium*, that is, they must be consistent and complete. As beginners start with incomplete knowledge of a concept, they try to establish equilibrium by filling gaps in their knowledge through experience, sometimes allowing misconceptions to form. As more material is taught, students check if this new information is consistent with their existing schema, and assimilate it if so. However, if the new information is *not* consistent, it will result in disequilibrium. Equilibrium is restored by adjusting the schema. This process of accommodation is when learning occurs and misconceptions may be corrected.

Bruner [7] builds on Piaget's concept of schemas to propose a spiral curriculum, based on the three key principles. A spiral curriculum is *cyclical*, with students returning to the same topic repeatedly, assimilating more knowledge each time. With each spiral, the topic is revisited at *increased depth*, potentially enabling the disequilibrium required for learning. This new information is combined with students' *prior knowledge* in building and maintaining their schemas. This final principle is important: if the necessary links between new and prior knowledge are not made, then new schemas are created, rather than misconceptions being corrected.

Notional machines [13] fit well with Piaget's concept. Neither the execution of the code in the real machine or run-time system, nor the conceptual model of a programming language shared by experts, is accessible to the novice. Notional machines provide a simplified version of the expert's real or conceptual execution model. As such, a notional machine allows novices to create an initial schema of a programming construct. It can be considered the first spiral of Bruner's curriculum. Much research [1, 5, 11], shows how effective notional machines are at giving students' a concrete execution model of their code, helping them form correct mental models. Many misconceptions are the result of students' lacking this execution model, and how changing students' mental models is particularly difficult.

As notional machines are, by definition, abstractions with details omitted, it is possible that students fill in some of that detail with misconceptions. The notional machine itself may inadvertently introduce a misconception [3]. Both Piaget and Bruner recognise that misconceptions are a likely result of achieving equilibrium. These initial misconceptions can be corrected in the next spiral of the curriculum, where new information is provided and assimilation or accommodation occurs.

However, notional machines are designed with varying levels of detail and abstraction. The introductory programming and computing systems courses at our institution expose students to two different notional machines: the high level Python execution model, and a low level systems execution model. We posit that making connections between multiple notional machines, with different levels of detail and abstraction, provides a complementary approach that enables students to resolve misconceptions in their mental models.

As we will detail in §3, our study was conducted at our institution, a large university that has both introductory programming and computer systems courses. Anecdotally, we found that some students compartmentalised the notional machines exposed by each course, failing to make the necessary connections between the two. The hypothesis of our study, therefore, is that by making the links between the programming and systems courses explicit, we can deepen students' understanding of programming concepts.

3 STUDY CONTEXT

Our university's first year undergraduate computing science programme includes an introductory programming course, taken alongside a computer systems course. We describe both courses in this section.

3.1 Introductory Programming Courses

Introductory programming courses at university need to accommodate a wide range of abilities, from novices through to competent programmers. Our university offers both beginner (CS0+1) and advanced (CS1) courses, with students self-selecting between them; each course attracts around 200 students. Python is taught in both courses, and both are taught across two semesters.

The CS0+1 course assumes no prior experience. Initial emphasis is put on code comprehension. Schulte's Block Model [10] is used extensively, which helps students to learn how to explain their code in terms of its grammatical structure (*text surface*), the problem domain and its goals (*function*) and how the program executes with regard to a conceptual machine and its state (*machine*).

The CS1 course proves to be a greater challenge. A student's history of learning can vary greatly, and lecturers need to teach and test the basics while still motivating and engaging very competent students. In our experience, students rely heavily on Python's built-in functions and libraries, searching the Internet for methods that will perform the desired action. For example, rather than iterating over the indices of a list if the position rather than the values of a list is needed, a student will iterate over the values and use `.index()` to get its position. When it fails to return the correct result, they are lost or over-engineer the solution. Their pre-university learning often appears to have focused on the textural surface and function of the Block Model, with less knowledge of the machine understanding. Considering that the program dynamic is considered a threshold concept [16] this is hardly surprising. Though the CS1 course does teach a machine understanding of the code, the prior reliance on built-in methods and libraries can prevent students' existing misconceptions from being exposed and corrected.

3.2 Computer Systems

Alongside the second semester of the introductory programming courses, most students take a course in computer systems. One of the main components of this course is a section that aims to develop a concrete execution model of high level programming constructs. Snippets of high level code, written in a simple, Pascal-like language, are translated to a low level equivalent that makes the control flow of the program explicit. Each line of the low level code can then be further translated into assembly code. The purpose of understanding, and being able to perform, these translations, is to allow students to make connections between a high level execution model, and a concrete, low level model. Anecdotally, however, we find that many students do not make these connections. The focus of our study, then, is to determine whether it is possible to make these connections explicitly.

4 STUDY DESIGN

In this section, we describe our study design, in terms of the participants, the logistics, and the method used.

Participants — The study was advertised to students participating in either of the CS0+1 or CS1 streams. We hypothesised that more experienced students would benefit less from our proposed intervention, as they are most likely to have correct mental models of the material used in the study.

Logistics — The study consisted of two separate sessions, each carried out one-to-one with each participant. Each session was conducted over Zoom and recorded for later analysis. Participants were provided with an appropriate pre-study briefing, describing the method of the study, and indicating that they could opt-out at any time.

Participants chose the time of each session. During the study, think-aloud methods were employed and the screen could be annotated. Participants were encouraged to ask questions and point out any difficulties they encountered. It was emphasised that the study was not a test of their ability or knowledge. It was important that the participants felt comfortable and received help as needed.

Method — The study focused on the `while` loop. There were several reasons for this:

(1) It is a basic construct taught in both introductory programming courses and in the computer systems course;

```
x = [5,3,4]
a = 0
i = 0
while  x[i] >= 3  and  i < len(x):
    a = a + x[i]
    i = i + 1
print(a)
```

Listing 1: High level Python snippet

```
            x = [5,3,4]
            a = 0
            i = 0
whileloop   if x[i] < 3 then goto endloop
            if i >= len(x) then goto endloop
            a = a + x[i]
            i = i + 1
            goto whileloop
endloop     print(a)
```

Listing 2: Low level version of Listing 1

```
x = [5,3,4]
a = 0
i = 0
while  i < len(x) and x[i] >= 3:
    a = a + x[i]
    i = i + 1
print(a)
```

Listing 3: High level Python snippet as in Listing 1, but with the order of the conditions reversed

(2) It easily translates into our low level form, unlike Python's complex `for` loop;

(3) We have found that students with prior experience rarely use `while` loops, especially with compound terminating conditions. If unavoidable they will opt for an infinite loop with multiple `break` statements.

In the first session, we asked each participant to hand-execute the code (Listing 1) and predict what would be the result. This exposed their existing mental model of the execution, and primed the student for the intervention.

The intervention involved walking through the execution of the code, using a diagram to illustrate the control flow of the program. From this diagram, the low level version of the code was then derived (Listing 2).

In the second session, participants then practiced the intervention by repeating the method (*i.e.,* hand-executing a high level code snippet, drawing the control flow, and translating to the low level form) with a similar code snippet. Finally, the student was then asked to hand-execute further code snippets. This allowed us to compare any changes in approach that could be attributed to the intervention.

Throughout the sessions, participants were asked about the importance of the order of the conditions in the `while` loop. For example, in Listing 3, the order of the `x[i] >= 3` and `i < len(x)`. That the order of the conditions matters is an important aspect of the machine understanding for expressions, and essential to fully understanding the semantics of the `while` loop.

5 RESULTS & DISCUSSION

16 students participated in the student, comprised of 7 from the CS0+1 course, and 9 from the CS1 stream. The first session typically lasted for one hour, while the second session took around thirty minutes. Of the 16 participants, 2 showed little to no understanding of the semantics of the `while` loop. 10 participants hand-executed the code line-by-line, following each iteration; of these, only 3 correctly noticed the "*out of range*" exception that would result from the order of the conditions (as shown in Listing 3). The remaining 4 students hand-executed each line of the code once, integrating the function of each line together to form a structure, which they used to predict the result; none of these participants noted that the "*out of range*" exception would be thrown.

Our study design assumed that participants would identify misconceptions that they held about the high level code snippet during the process of translating the code to the low level equivalent. In carrying out the study, it became clear that this assumption was incorrect: participants could *incorrectly* hand execute the high level code, indicating a misconception, before proceeding to *correctly* translate it to the low level form. It was only in having participants hand-execute the low level code that they identified their misconceptions about the high level code. This supports our hypothesis: dealing with the same programming constructs within two different notional machines helps to identify and correct misconceptions.

The nature of the study – using a small group of participants, and being led by their understanding – meant that a significant amount of time was spent in identifying where each participant's hand execution had failed. This provides us with rich, qualitative data, that, while unsuitable for thematic classification, allows us to identify three common issues and areas for future work: participants lacking a concrete low level execution model (§5.1), holding multiple conflicting mental models about the same high level construct (§5.2), and participants assuming intelligence within the high level interpreter (§5.3).

In the sections that follow, we use quotes from participants, with thick description [17], to illustrate these three issues, and to propose new hypotheses and future work. Participants are labelled from *P1* through *P16*.

5.1 Lack of Concrete Execution Model

It was clear that many participants, and those from CS0+1 in particular, did not hold a concrete execution model for the low level code. This was surprising: the low level execution model (where each line is executed in turn, with explicit `goto` statements dictating control flow) is much simpler than that for high level programming languages.

For some participants, including *P5*, clarifying the low level execution model was useful (*P5*: " *clears up more of it, especially like when you said, like in low level the computer always like execute next line*"). *P13* was confused by `goto` statements, which jump to another location in the code when the condition is true, in contrast to the high level `while` loop, which is entered into when the condition is true (*P13*: "*Okay, so then it will go straight to the end and print.....Why is it suddenly confusing?*"). *P11* and *P10* were also confused by the low level hand-execution task.

Most of the confusion resulted from participants over-complicating the low level execution model, and layering in their own understanding rather than methodically following the execution model. However, with practice, all participants appeared to understand the execution model. *P11*, who struggled with hand executing both the high and low level forms, remarked that "*I think when you put them like side to side, and you can see, and when you must do it like one by one, it makes so much more sense than just reading it off*". This supports our hypothesis.

While it was common for participants to lack a concrete execution model for the *low level* form, some participants also appeared to lack an understanding for the *high level* execution model. *P5*, for example, struggled with the dynamic nature of the high level, remarking that "*it won't matter for like the high level, but like for the low level, we have to since it goes one by one*". The simple, line-by-line nature of the low level execution model had become clear, while the high level execution model remained difficult to understand.

Similarly, *P10*, a CS1 student, when asked what the purpose of low level form was, remarked "*low level is like more simple, I would say and high level we use like a lot of inbuilt functions in high level, I would say that in low level we just use the simplest version of all the loops and everything*". This participant, when asked to perform the translation from the high to low level forms of the `while` loop, took more than 6 minutes to perform the task. They did not split each condition into individual operations, and also failed to form the correct `goto` statements in the low level code. This indicates a lack of understanding of the dynamic nature of the high level form.

Finally, while *P7* correctly identified the "*out of range*" error (the only CS1 student to do so), they still did not realise or understand that the order of the conditions in the high level form mattered, saying "*So say they are swapped, it would still check x[i] greater than equal to three wouldn't it so it would still throw an error*".

We posit that the reliance on built-in functions and libraries results in students having a text surface understanding of their code, but fails to allow them to develop a machine understanding. This absence of a previous schema or seeing the new material as a dynamic process prevents students from linking their existing programming knowledge, which may contain misconceptions, with the material taught in the computer system course.

New hypothesis — Students with pre-university programming experience and already-formed mental models may also lack a concrete execution model of their code.

Further study — We will investigate to what extent students with pre-university programming experience understand their code, independently of the hypothesis explored in this paper.

5.2 Multiple Conflicting Mental Models

The remarks quoted so far also lead to a related, but distinct, conclusion: that students think of the high and low level forms as being discrete and unrelated, despite the step of translating between the two. It can be concluded that our participants held separate, and conflicting, mental models for the execution of a `while` loop, dependent on whether it was being expressed in the high or low level code.

P5 noted that the `while` loop, despite the translation from the high level, was "*suddenly like a different structure*", and upon realising that they were the same, remarked that they were surprised.

P10 exhibited this behaviour – of compartmentalising the high and low level forms, and producing separate mental models – more concretely. When asked if the order of the conditions mattered, *P10* replied "*When we do it in high level I guess it doesn't matter, but when we do it in low level, it does.*". Even after discussing the connection between the high and low level forms, the participant did not appear to be confident (*P10*: "*Maybe it will show an error here as well then*").

Similarly, after it was explained to *P9* that the order of operations mattered in the high level, they failed to map this to the low level, remarking that "*I was thinking in high level again*". That the form of the same concept mattered to the participant's understanding indicates that they treat the two separately. Finally, *P3* treated the high and low level forms as being completely separate, arriving at two different outputs (*P3*: "*I mean like both are correct, depending on their purpose*").

It appears that participants are not assimilating new information, discovered in the translation to, or hand-execution of, the low level form, into their existing mental model of the high-level construct. This prevents the necessary disequilibrium in understanding: rather than trying to accommodate their new knowledge into their existing schema, they create another, discrete schema. This does not identify or correct misconceptions in their understanding.

New hypothesis — Students can hold a number of conflicting mental models of the same construct, preventing the identification and correction of misconception.

Further study — We will explore how students in later years of their degree understand the same construct in different languages, investigating whether they have merged their mental models or maintain separate ones.

5.3 Intelligent Interpreter

Pea [9], who coined the term *superbug*, defined it as "*the default strategy that there is a hidden mind somewhere in the programming language that has intelligent interpretive powers*". This leads to students assuming more intelligence of the high level language interpreter than actually exists, resulting in misconceptions. We observe a number of participants in our study that exhibited behaviour consistent with the superbug.

P6, a CS1 student, despite not noticing the "*out of range*" error, was only 1 of 2 participants that was able to explain why the order of the conditions mattered, saying that it had been covered during their programming lectures and tutorials. After displaying a thorough understanding, the participant remarked "*so presumably your compiler would realize that and swap those two statements around automatically*". This suggests that they believe that Python is sufficiently intelligent to understand the programmer's intention, rather than the code as written.

Additionally, *P9*, when describing the high level form noted that "*high level is it more almost a humanized version of low level, given a lot of a lot of tips and tricks basically to to implement low level stuff*". The phraseology here (e.g., "*humanized*" and "*tips and tricks*") suggests that this participant also believes that the Python interpreter is able to understanding their intent.

If students believe that high level language interpreters are intelligent, and that their intention and rationale when writing their code matters more than the code that they write, they will struggle to understand the importance of syntax, and how to debug or reason about it. This will ultimately prevent them from forming correct mental models.

New hypothesis — Students believe that Python's interpreter knows what they *want* their code to do and will execute that, rather than following the defined semantics of the program.

Further study — We will explore to what extent students believe that IDEs, interpreters, and compilers for high level languages can identify and resolve syntax and semantic errors, and instead execute their code correctly, based on their intention.

5.4 Summary

We found that the degree and strength of the misconceptions held by participants in our study made it difficult to test our original hypothesis. This shifted the focus of our study: much more time was spent on understanding where and how misconceptions arose. We found that misconceptions resulted from three main issues: that participants lacked a concrete execution model, held multiple conflicting mental models, and assumed that the high level interpreter was intelligent. These categorisations are overlapping, and it was often difficult to ascertain which applied to each participant.

6 RELATED WORK

We found that Pea's [9] statement, that "*much more research is needed on how best to help students see that computers read programs through a strictly mechanistic and interpretive process, whose rules are fairly simple once understood*" holds. du Boulay [4], who first coined the term notional machine, a pedagogical tool to aid novices create correct mental models of the mechanics of the program, furthered this field. Sorva [13] showed how using a notional machine, such as a visualiser, was effective in aiding beginners. Finch et al. [5] a large number of notional machines available to teachers, their different forms, and their various strengths and weaknesses. For example, hand execution [3] exposes the student's mental model, but does not provide a means for showing the correct method. Conversely, visualisers [15] accurately show a correct execution model, but they do not reveal the user's own mental model.

The computer systems approach that we describe in this paper adds to the set of notional machines that are available to teachers. Not only does it provide a concrete execution model, similar to other notional machines, it does so at a markedly different level of abstraction. It is in translating between these different levels of abstraction that students can both expose and correct their mental models.

7 CONCLUSION

We set out to test our hypothesis that misconceptions and incorrect mental models could be corrected by making explicit links between high level concepts, as taught in an introductory programming course, with the same in a low level form, as taught in a computer systems course. Though our initial results are promising, we were surprised by the scale and depth of the misconceptions held by our participants. As discussed in §5, we found three main issues, with students:

- lacking a concrete execution model for both their high and low level code;
- having formed multiple, conflicting mental models; and
- believing that the high level language interpreter has the ability to understand the intent, rather than only the syntax and semantics, of code.

These issues combine to prevent students from developing and correcting their existing mental models, instead preferring to explain newly learned behaviour by creating a new mental model. Further research is required to fully investigate the new hypotheses that we've formed, and to design effective interventions.

REFERENCES

[1] Michael Berry and Michael Kölling. 2014. The state of play: a notional machine for learning programming. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 21–26.

[2] Hamidreza Babaee Bormanaki and Yasin Khoshhal. 2017. The Role of Equilibration in Piaget's Theory of Cognitive Development and Its Implication for Receptive Skills: A Theoretical Study. *Journal of Language Teaching & Research* 8, 5 (2017).

[3] Paul E Dickson, Neil CC Brown, and Brett A Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 159–165.

[4] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of man-machine studies* 14, 3 (1981), 237–249.

[5] Sally Fincher, Johan Jeuring, Craig S Miller, Peter Donaldson, Benedict Du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, et al. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 21–50.

[6] The Python Software Foundation. 2022. Python 3 Documentation: The `while` statement. https://docs.python.org/3/reference/compound_stmts.html#while Date accessed: 21/1/22.

[7] Ronald M Harden. 1999. What is a spiral curriculum? *Medical teacher* 21, 2 (1999), 141–143.

[8] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. 2020. Analysis of Student Misconceptions using Python as an Introductory Programming Language. In *Proceedings of the 4th Conference on Computing Education Practice 2020*. 1–4.

[9] Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (1986), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

[10] 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*. 149–160.

[11] Juha Sorva. 2007. Notional machines and introductory programming education. *Trans. Comput. Educ* 13, 2 (2007), 1–31.

[12] Juha Sorva. 2010. Reflections on threshold concepts in computer programming and beyond. In *Proceedings of the 10th Koli calling international conference on computing education research*. 21–30.

[13] Juha Sorva et al. 2012. *Visual program simulation in introductory programming education*. Aalto University.

[14] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–64.

[15] Juha Sorva, Jan Lönnberg, and Lauri Malmi. 2013. Students' ways of experiencing visual program simulation. *Computer Science Education* 23, 3 (2013), 207–238.

[16] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 49–54.

[17] Josh Tenenberg. 2019. Qualitative methods for computing education. *The Cambridge handbook of computing education research* (2019), 173–207.

[18] W3Schools. 2022. Python While Loops. https://www.w3schools.com/python/python_while_loops.asp Date accessed: 21/1/22.