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 pre-formed mental models. These models are often incorrect, with students holding entrenched misconceptions. In this paper, we describe a study that investigated whether making explicit connections between our introductory Python programming and computing systems courses could expose mental models and help identify and fix misconceptions. We hypothesised that students would develop a correct mental model by creating a low level systems implementation of a high level program. While we identified misconceptions, these prevented the students from making explicit links and correcting their mental models. We detail these misconceptions, develop a set of hypotheses for why these were held, and suggest future studies.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**;

KEYWORDS

introductory programming; mental models; misconceptions

1 INTRODUCTION

Many students enrolling in university-level introductory programming courses have programming experience, whether from self-study or formal education. These students often hold mental models that are incomplete or even incorrect. Arguably, identifying and correcting entrenched misconceptions is challenging within the constraints of introductory programming courses and the languages that they teach [8].

For example, consider Python's `while` statement, 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, an ambiguity in this definition may give rise to a misconception. 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.

In this paper, we consider whether programming misconceptions could be identified and corrected through explicit connections to an introductory computer systems course. In this course, students are taught how to translate high level (e.g., Python) code 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, demonstrating that the loop condition is evaluated only once at the start of each iteration.

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland

Misconceptions in novice programming are the source of much research, with many being traced back to poor mental models. Programming dynamics is considered a threshold concept [12]. Notional machines [5, 14], in the form of visualiser or tracing methods, have been developed to teach novices how their code will be executed. 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 – 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 concepts common to both programming and computing systems courses, opting instead to see them as distinct topics, 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, which are used to both understand and solve problems. Schemas must be *in equilibrium*, that is, they must be consistent and complete. Beginners start with incomplete knowledge, and then try to establish equilibrium by filling the gaps through experience, sometimes forming misconceptions. As students are taught, they check if 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 through accommodation, and this is when learning occurs and misconceptions corrected.

Bruner [7] builds on Piaget's concept of schemas to propose a *spiral curriculum*, based on these key principles:

(1) a spiral curriculum is *cyclical*, with students returning to the same topic repeatedly;
(2) with each spiral, the topic is revisited at *increased depth*, potentially enabling the disequilibrium required for learning;
(3) new information is combined with students' *prior knowledge* in building and maintaining their schemas.

, This final principle is important: if links between new and prior knowledge are not made, then *new* schemas are created, rather than misconceptions being corrected.

Notional machines [11, 13] fit well with Piaget's concept. Novices lack a concrete or conceptual model of program execution, and notional machines provide an accessible, simplified model. As such, notional machines allow novices to create an initial schema of a programming construct. They can be considered the first spiral of Bruner's curriculum. Much research [1, 5, 11] shows that notional machines are effective in giving students an execution model of their code, helping them form correct mental models.

As notional machines are, by definition, abstractions with details omitted, students may fill the gaps with misconceptions. The notional machine itself may introduce a misconception [3]. Piaget and Bruner recognise that misconceptions likely result from achieving equilibrium. These misconceptions can be corrected in the next spiral, when new information results in assimilation or accommodation.

As we will detail in §3, 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. Anecdotally, we found that some students compartmentalised the notional machines exposed by each course, failing to make the necessary connections between the two. We hypothesise, therefore, that making explicit connections between multiple notional machines, with different abstractions, provides a complementary approach that enables students to resolve misconceptions.

3 STUDY CONTEXT

Our university's first year computing science program includes both introductory programming and computer systems courses. We describe both 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) Python courses, with students self-selecting between them; each full-year course attracts around 200 students.

The CS0+1 course assumes no prior experience, with initial emphasis on code comprehension. Schulte's Block Model [10] is used extensively, teaching students how to explain their code in terms of its grammatical structure (*text surface*), the problem domain and goals (*function*) and how the program executes conceptually (*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. 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 introductory 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 performing, these translations, is to encourage students to make connections between the two execution models. Anecdotally, we find that many students do *not* make these connections. Our study attempts to determine whether these connections can be made 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 carried out one-to-one with each participant. Sessions took place over Zoom and were recorded for later analysis. Participants were provided with an appropriate pre-study briefing, detailing the nature of the study, and indicating that they could opt-out at any time.

```
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

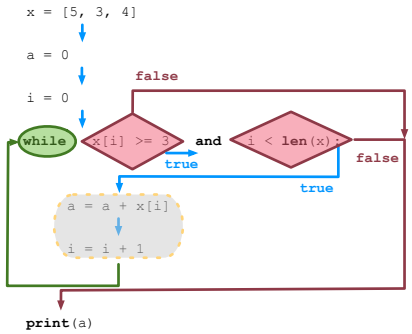

Figure 1: Control flow diagram (from Listing 1)

During the study, think-aloud methods [17] were employed and the screen could be annotated. Participants were encouraged to ask questions and highlight any difficulties. It was emphasised that the study was not a test of their ability; it was important that the participants felt comfortable and received help as needed.

Method — The study focused on Python's `while` loop. There were several reasons for this: (i) it is a construct taught in both introductory programming courses and the computer systems course; (ii) it easily translates into our low level form, unlike Python's complex `for` loop; and (iii) we found that students with prior experience rarely use `while` loops, especially with compound conditions. If unavoidable, they often use infinite loops 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, primed the student for the intervention, and committed the participant to a result that could be compared with any future result.

The computer systems course teaches the students the low level patterns of the high level constructs which they should apply to perform the translation. In this intervention we choose to show the participant an alternative method. The investigator walked through the execution of the code drawing the control flow on top (as in Figure 1).

```
            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

The investigator then developed the low level (Listing 2), drawing on the control flow diagram. Though the intervention was completed by the investigator, the participant was questioned and encouraged to probe each step of the process. The final low level form was agreed, with the student, to be a correct translation of the high level.

Initially, this intervention completed the first session. However, the study exposed a misconception in the first participant who, despite in the second session being able to correctly translate from high level to low level, continued to hold the same misconception when hand executing the final snippet. Hence our initial hypothesis had failed. It was decided that it was not sufficient to simply uncover misconceptions, but that we should challenge and attempt to correct them. We therefore adjusted the study.

After the investigator had translated the code to low level the participant was asked to hand execute it. When faced with a different result from the low level than the high level, we believed that the necessary disequilibrium in the participant's mental model would result. Through the process of accommodation we hoped to correct any misconception.

This process was not as straightforward as planned, and instead exposed some problematic thinking in the participants that was preventing them from correcting their misconceptions. Section 5 will focus only on this problematic thinking.

In the second session, participants practiced the intervention: given a similar snippet of Python, the participant was asked to translate it to low level form and, given a new snippet of code, asked to hand-execute it. This allowed us examine the participant's ability to perform the translation themselves, and to compare any changes in approach that could be attributed to the intervention. Though this second session provided interesting results, these do not relate to the main findings of the paper, and are omitted.

Throughout the sessions, participants were asked about the importance of the order of the conditions in the `while` loop. 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. Such understanding is essential to enable a student to hand-execute and debug code.

5 RESULTS & DISCUSSION

16 students participated in the study, comprising 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. 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 held about the high level code snippet during the process of translating the code to the low level equivalent. It was only in having participants hand-execute the low level code that they demonstrated 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 comprehensive high 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 [16], 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) appeared to us to be much simpler than that for high level programming languages. Once the participants practised hand-executing the low level form, all but one found it easier than hand-executing high level form.

For some participants, including *P5*, a CS0+1 student, 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*, another CS0+1 student, was confused by conditional `goto` statements, which jump to a 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?*"). Other participants, mainly those from CS0+1, were also confused by the low level hand-execution task.

Most of the confusion resulted from participants overcomplicating 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 the initial lack of a concrete execution model for the *low level* form was resolved with practice, more problematic were the participants who appeared to lack an understanding of the *high level* execution model. *P5*, for example, struggled with the dynamic nature of the high level, and when asked about the order of the conditions remarked 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 hand-executing the Python did not evaluate the condition for the final time but instead exited the `while` loop from the bottom of the loop body. This participant further struggled with the low level execution, needing help with the evaluation of `x[3]`, and had to be told that it would cause an out-of-range error. This indicates a lack of understanding of the dynamic nature of code.

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 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 prior 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 distinct and unrelated, despite the translation step. 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 confident (*P10*: "*Maybe it will show an error here as well then*").

Similarly, after it was explained to *P9* (a CS0+1 student) that the order of the conditions 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* (a CS1 student) 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 concept, 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] coined the term *superbug*, defining it as "*the default strategy that there is a hidden mind somewhere in the programming language that has intelligent interpretive powers*". We observe a number of participants in our study that exhibited behaviour consistent with the superbug, with some assuming that interpreters are intelligent, resulting in misconceptions.

P6, a CS1 student, despite not noticing the out-of-range error, was only 1 of 2 participants that noted that the order of the conditions mattered, saying that it had been covered during their programming course. 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 humanised 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., "*humanised*" 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 the program as written.

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 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. Fincher et al. [5] compare 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: (i) lacking a concrete execution model for both their high and low level code; (ii) having formed multiple, conflicting mental models; and (iii) 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] Josh Tenenberg. 2019. Qualitative methods for computing education. *The Cambridge handbook of computing education research* (2019), 173–207.

[17] Maarten Van Someren, Yvonne F Barnard, and J Sandberg. 1994. The think aloud method: a practical approach to modelling cognitive processes. (1994).

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