University of Glasgow | School of Computing Science

Exploring Just In Time Code Comprehension in Lab Based Programming Excercises

Angus McIntosh (2182976m)

April 17, 2020

ABSTRACT

This paper tackles the issue of students writing code without considering how it will behave. Research shows that the ability to understand and trace programs is necessary to be able to effectively write programs. It studies a potentially automatable tutoring technique that asks code comprehension questions whilst students are writing code.

1. INTRODUCTION/MOTIVATION

It has been widely accepted in computing science (CS) education research that code comprehension is an important skill for learners to acquire. However, as we show in this paper the primary teaching method used in Computing Science courses, lab-based exercises that revolve around writing code at a machine in an IDE, is ineffective at teaching code comprehension. This results in students with poor code comprehension abilities not improving them or developing good writing skills.

Code comprehension pedagogies exist to address this problem, but they often struggle to engage learners, who view them as *"fake"* since they have no direct link to what is seen as a main activity of CS for many students, of writing code.

If a code comprehension activity could be merged with code writing, the code comprehension activity could become engaging too. A long term goal of this work is to merge code comprehension learning activities into the automatic tools used by learners when they write code. Creating such a system would create a kind of automated tutoring system that can support users with understanding their own code and with identifying and fixing bugs.

Of course code writing cannot take place without code understanding, just I couldn't write this paper with confidence if I only had a poor grasp of English. Hence, this paper explores the potential of merging the learning for both writing and understanding into a single pedagogy. This paper contributes 4 significant deliverables towards this goal.

1. An argument based in the literature that using an IDE to complete writing tasks is ineffective at developing code comprehension skills.

2. A small interview study with tutors known to be effective, to ellicit good questioning methods.

3. An outline of a protocol for generating a sequence of code comprehension questions, given a program that could be followed by a human tutor or an automaton.

4. A study to validate the concept, in which a human tutor applies the protocol to a number of participants as they each work on a sequence of programming problems.

The outcome of the final study finds that the concept is broadly workable and the paper makes suggestions for further work.

2. CODE COMPREHENSION

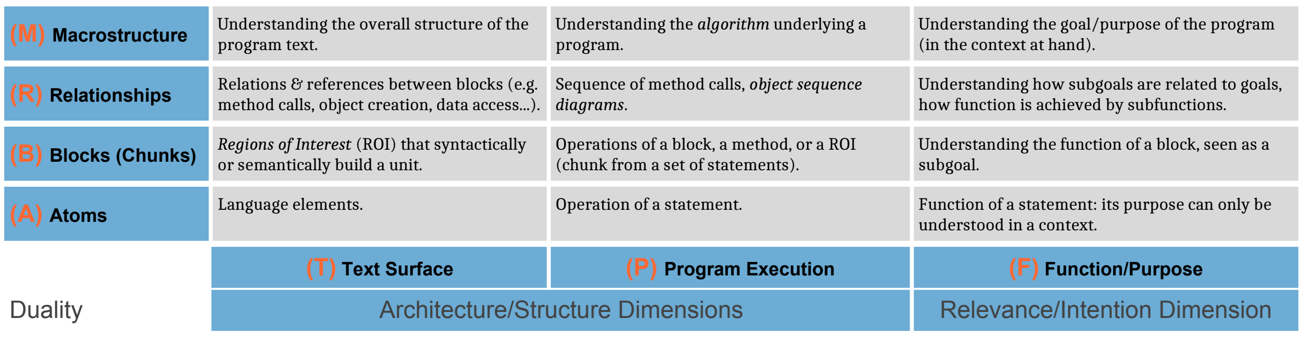
2.1 The Block Model

Schulte describes program comprehension as *"usually conceptualized as a process in which an individual constructs her own mental representation of the program"*[19]. Schulte's Block Model views the process of program comprehension as *"filling in"* a grid of code understanding shown in figure 1. Each cell in the grid represents an aspect of the mental model of the code. The rows of the grid correspond to the size of code structures and the columns of the grid correspond to type of understanding. Schulte separates the types of understanding into two, functional and structural understanding. Functional aspects of code include the intended meaning and purpose of code and are difficult to measure objectively. Structural aspects of code are measurable and are divided into *text surface* that covers what the text of the program actually is, how it is laid out in the file, and *execution* how a machine running the code behaves, what the state of the program will be after running a line of code. [19] Many theories of code comprehension can be understood as different methods of moving around the Block Model[20].

This paper focuses on improving the structural understanding of code at the Atomic and Block levels because as will be argued in section 2.2 structural understanding is an important skill.

2.2 Code Comprehension Is Important

Lopez et al[15] studied the relationships between first year university student's abilities to trace, explain and write code, by analysing the responses to an end of first semester exam. Each question in the exam was placed into a category based on the skill required to answer the question, these included *"Tracing"*, *"Reading"* and *"Writing"* skills. In their study, reading skill refered to the functional aspects of code comprehension and the tracing skill to the structural aspects. By statistically analysing the marks for each question they found that tracing skill was a predictor of reading skills and that both reading and tracing skills were predictors of writ-



(M) **Macrostructure**	Understanding the overall structure of the program text.	Understanding the *algorithm* underlying a program.	Understanding the goal/purpose of the program (in the context at hand).
(R) **Relationships**	Relations & references between blocks (e.g. method calls, object creation, data access...).	Sequence of method calls, *object sequence diagrams*.	Understanding how subgoals are related to goals, how function is achieved by subfunctions.
(B) **Blocks (Chunks)**	*Regions of Interest* (ROI) that syntactically or semantically build a unit.	Operations of a block, a method, or a ROI (chunk from a set of statements).	Understanding the function of a block, seen as a subgoal.
(A) **Atoms**	Language elements.	Operation of a statement.	Function of a statement: its purpose can only be understood in a context.
	(T) **Text Surface**	(P) **Program Execution**	(F) **Function/Purpose**
Duality	Architecture/Structure Dimensions		Relevance/Intention Dimension

Figure 1: Schulte's Block Model[19].

ing skill[15], suggesting that the ability to structurally comprehend code is a key factor in the ability to functionally comprehend code and write good code.

2.3 Existing Code Comprehension Pedagogies

Because code comprehension skills are important multiple pedagogies have been created to help learners *"develop"* their code comprehension ability. These techniques separate the learner from either a machine, creating a level of *"fakeness"* about the task, or from code they write themselves.

2.3.1 *Code Animators*

Code animations or visualisations are animations of how the code behaves whilst it is running. Hundhausen et al[10] found that how the code animation is interacted with is the most important factor in the effectiveness of visualisations and code animations. When students only watched the animations there was no significant learning advantages but when they were used as validation for a task such as predicting how code will run, code animations were educationally effective[10]. They show the structural behaviour of code, but can target different rows of the Block Model. Some animators such as Python Tutor[8] display the step by step behaviour of a notional machine for the programming language targetting the Atomic and Block levels. Other animators like PlanAni focus on the algorithmic steps choosing to ignore language specific details[18] instead targetting the Macro and Relations levels from the Block Model of the code's structural execution.

2.3.2 *Tracing*

When tracing, students are required to imitate the behaviour of the program on paper, tracking the values of variables, the current state of the stack etc[9]. Xie et al suggests using memory tables to track the values of variables while the program executes as a way for students to structure their tracing[29]. This gets the students to excercise their structural understanding of the code at the Atomic level of the Block Model only. UUHistle offers a computer based drag and drop tracing environment for Python[24].

The repetitive and tedious nature of tracing makes it off putting to some students. For other students tracing can be too difficult as they have not mastered the internal execution of a program yet[25].

2.3.3 *Explain In Plain English*

Students are asked to give an explanation of a given program in a single sentence. The aim of this pedagogy is to get the student to develop their ability to get to the top right of the Block Model and create a macro structure level functional understanding of code. Responses that are not at the top right of the block model while not viewed as incorrect are perceived as weaker responses and in exams awarded fewer marks with responses in the bottom lefthand corner of the block model often receiving no marks at all[4].

Despite these questions not involving writing code and being separate from the machine, it is often necessary to explain code to others so *"Explain in Plain English"* questions are realistic.

2.3.4 *Thinkathons*

Students answer code comprehension and code writing questions on paper, this compares to Hackathons where the focus is entirely on writing code and producing a working program. The questions are intended to promote mastery learning by having many questions on the same topic and similar difficulty that a student can attempt until they master that topic. Some of the questions are based on tracing and Parson Puzzles described below, some ask for descriptions of code, some to debug code and some to write new code to solve a problem[6]. Since Thinkathons involve a range of comprehension tasks they can excercise all the areas of the Block Model, but because of the focus on paper based work, is seen to have a degree of *"fakeness"* about it.

2.3.5 *Parson Puzzles*

Parsons Puzzles require students to recreate a program where the lines of code have been muddled up. Most Parson puzzles provide a collection of possible lines to choose from but there are variants that allow the student to write the lines themselves or add extra distractor lines to choose from[16]. These attempt to merge aspects of code comprehension and code writing together and results in a mixture of atomic and block level code comprehension with scaffolded writing. Whilst tools to do Parson Puzzles on a machine exist[11], code rarely comes to a programmer mostly written but with lines jumbled up they are seen to have a high degree of *"fakeness"*.

2.3.6 *PRIMM*

PRIMM is a teaching structure for introducing new programming elements that includes code comprehension elements. It has five steps:

1. Predict - Students read an example piece of code, then predict how it will behave when it is ran.
2. Run - The teacher or lecturer runs the code so that students can check their predictions.
3. Investigate - Students investigate whether the code ran as predicted and why.
4. Modify - Students modify the example code to achieve a different purpose.
5. Make - Students make a new piece of code using the concepts introduced in the example to solve a different problem.

The *"predict"* and *"investigate"* steps of PRIMM involve tracing and exercises structural comprehension skills while the *"modify"* and *"make"* steps use code writing skills and are often done at a machine. While both code comprehension and code writing skills are used in PRIMM, the code comprehension exercises and code writing excercises are kept separate by the educator[22].

2.3.7 Intelligent Tutoring Systems

These work by providing automated feedback through techniques such as text, new questions or animations based on student responses to questions or learning tasks. The 2014 working group report of the ITiCSE conference by Brusilovsky et al criticizes the complexity for educators to use smart learning content like intelligent tutoring systems because many require configuration to add new excercises[1]. Keuning et al agree with this assessment[13].

The feedback is typically provided at the end of tasks such as code writing, but there are examples of other times for providing feedback. For example, Silva et al provided feedback to students as they coded, based on model solutions and pre-prepared teacher feedback[23]. The feedback from these systems can target any area of the Block Model, however only the feedback covering the structural comprehension at Atomic and Block level can be applied to arbitrary code as any other feedback tends to be task specific and relies on pre-prepared feedback and model solutions.

3. WHY WRITING CODE DOESN'T NECESSARILY DEVELOP COMPREHENSION SKILLS

It is tempting to assume that simply by writing code, students will gain an ability to comprehend code. Given that as section 2.2 showed, that comprehension is a key factor in code writing, a student writing code must be excercising some form of code comprehension to be able to write a program at all. Thinking about writing in natural language, it seems unlikely that an author of a text does not understand anything that they had written. Whilst this paper accepts this point, this section it argues that there are some common student behaviours that allow them to write code without actively reading the code and developing their code comprehension skills. It is these behaviours that the paper particularly tackles.

3.1 Demonstrations

A study into how engagement with physics demonstrations affected learning by Crouch et al[5] delivered 7 demonstrations to students taking a physics course each in 4 ways:

- No demonstration
- Observe, where the students watched the demonstration.
- Predict, where the students predicted the outcome from a set of choices before watching the demonstration.
- Discuss, where the students predicted the outcome from a set of choices before watching the demonstration, and then discussed the outcome together afterwards in small groups of 3 or 4 students.

The students then did a test at the end of their semester on two things:

- If they could predict the outcome of the exact same demonstrations again, and
- If they could explain the physics behind their prediction.

The students who did not see a demonstration were the worst at predicting the outcome of the same demonstration again, with the observe, predict and discuss students increasingly able to make the correct prediction.

When students had only observed a demonstration, their ability to explain their prediction was little better than a student that had not seen the demonstration at all. Those students that had predicted the outcome beforehand and the students that had both predicted and discussed did much better at explaining their predictions than those that had only observed.[5]

This matches research into the effectiveness of different code animators that got a similar result; when code animators were only watched there was little or no learning gain, but when predictions were made before seeing the code animation the animators were more effective learning tools[10].

These studies show that to make a demonstration educationally effective it is important to consider and predict the outcome of a demonstration before seeing the outcome itself. Applying this to code comprehension, running code can be considered as a demonstration of the structural elements of the code's behaviour, suggesting that to make code writing effective as a code comprehension pedagogy it is necessary to ensure the student understands the code as it is written. However as shown in the next subsection, it is not easy to consider code written yourself.

3.2 Reading What You Have Written Is Hard

It is not necessary to fully understand code to write it. In fact even for experienced developers that fully understand what they intend to write, it is generally accepted that whilst typing out code, mistakes will be made. Either through mistakes or which will lead to differences between what the writer beleives they have written and the text of the program itself. It is then difficult for them to consider the *behaviour* of code as written because they already know the *intended* behaviour of the code. This is demonstrated with a login function in figure 2.

```
def login(username, password):
    if is_valid_username(username) or
       is_matching_password(username,
          password):
        return get_user(username)
```

Figure 2: An example of a possible mistake in a login function that would be easy for the author to miss. They would have intended to write *and* instead of *or* when verifying the username and password. However, since the code writer will have intended to write *and* instead of reading the erroneous *or* they remember intending writing *and* and read the code as saying *and.*

This effect is well known in non code writing and is a form of a top down illusion[2]. Pilotti et al showed that when proofreading text written themselves, people were less effective at correcting errors than others who had not written the text. They propose that this is because they rely on their memory of the text they had written instead of the text itself[17]. They saw what they remembered writing, not what was actually written.

3.3 People Take The Easiest Approach To Reasoning

When reasoning, people will tend towards using the *"easiest"* method to reach their conclusions. Schwartz and Black[21] describe a system of *"fallback"* where people try to use simpler mental models to solve problems before using complex mental models to solve the problem. They performed an experiment that asked people to predict how interlocking gears would behave when one gear was turned. They asked participants to make multiple predictions which direction the rightmost gear in different chains of gears would turn if the leftmost gear was turned in a given direction. They noticed that to begin with the participants would start by moving their hands round in the air tracing circles and the motion of each gear, but after prediciting for some chains of gears, participants would stop making circles with their hands and make the predictions much faster. Schwartz and Black suggested this was because they had realised that the direction that the final gear in a chain of gears will turn in the same direction as the first if their is an odd number of gears in the chain and the opposite direction if there is an even number, so no longer had to go through the effort of modelling how each consecutive gear would turn.[21] The participants switched to using a simpler mental model when they found one but had used a slower more mentally taxing model until the easier one was discovered.

3.4 The ICAP model of educational effectiveness

ICAP is a model for evaluating the likely educational benefit of learning activities reached through an analysis of the large body of published activities[3]. ICAP defines a set of groupings for different types of behaviour students display when engaging with an activity, where each group is better for learning than the previous group. The groups are:

- Passive - No task relevant movement
- Active - Activity that shows picking out or emphasising an existing piece of task relevant information
- Constructive - Activity that generates a new task relevant piece of information
- Interactive - A conversation where both participants are being constructive

ICAP predicts that students displaying interactive actions are learning more than students displaying constructive actions, who again are learning more than students displaying only active actions, who are learning more than students that are passive.

Interactive engagement is envisaged as a human to human conversation, but could be a human to machine conversation if the machine is able to consistently generate meaningful and task relevant new information for the human to build on.

There are cases where it does not hold:

- The activity is off topic, so it is not relevant to the intended learning outcome
- The intended learning outcome is arbitrary, so cannot be derived by inference or deduction from other knowledge
- The learner do not infer or deduce the new knowledge because they not possess other required knowledge from which to make the inferences or deductions.

Writing code is considered a constructive action because it involves creating new knowledge and actively typing it. Running code is not active because it does not involve selecting or emphasising, making it a passive action. Predicting is a constructive action because it generates a new piece of information, the prediction.

Since ICAP does not hold where an activity is off topic, and writing code does not require code comprehension. Code writing is off topic and therefore not constructive with respect to teaching code comprehension skills.

3.5 Monkeys At Typewriters Do Not Learn

Most teachers, tutors and lecturers will have experienced a student taking a *"Monkey and Typewriter"* approach to a coding problem. The student types out code without understanding the code they have written, then when the code almost inevitably doesn't behave as the student wants it to, they perform changes on the code until it *"works"*. The student does not seem to have reasons for their changes, but makes them anyway and runs their code after each change to check its behaviour. Some students use a variation of this method into something resembling hillclimbing in machine learning where they keep the change if running the altered code seems closer to the desired behaviour than before and discarding the change otherwise.

Whilst evidence for this behaviour in programming generally anecdotal such as Kinnunen and Simon reporting this behaviour whilst interviewing students about their emotional state and classifying it as *"Success not due to own actions"*[14], there is evidence that it occurs when trying to fix compile errors. Jadud used the BlueJ IDE[7] to track how students handled compile errors in their Java programs and found that students often took many attempts to fix their compilation error[12]. Jadud found that students who repeatedly failed compilation for the same reason performed poorly in their programming courses and proposed the Error Quotient as a course success predictor[12]. Watson et al went further and devised the Watwin algorithm to predict course success. The Watwin algorith takes into account the time and attempts on a particular compile error in relation to the students peers[28]. Again, Watson et al found that some students would be taking large numbers of compile attempts to fix the compile error[27]. That the students took a large number of attempts to fix compile errors suggests they were attempting to fix them by guessing, instead of reasoning about the error message and their code.

4. GOAL AND ANALYSIS OF EXISTING APPROACHES

As covered in previous sections code comprehension is an important skill, but many of the existing pedagogies are seen to have a degree of *"fakeness"*. Code writing tasks on the other hand are generally accepted as engaging activities for learners. The objective of the paper is to explore possible ways of merging these together to create a pedagogy that both develops code comprehension ability but is as engaging as code writing tasks. To achieve this any pedagogy or tool must have these traits to succeed:

- Automatable - The pedagogy must be able to work alongside students coding themselves and it needs to be able to provide feedback on the code they actually write. It is impractical to have teaching staff configuring a tool for every task they set to anticipate every mistake or misunderstanding a student might make on a writing task.
- Effective - The pedagogy needs to excercise the student's mental models of the structural behaviour of their own code, this will improve their ability to fully comprehend their own code.

This section analyses existing pedagogies on how closely they meet these criteria to provide a basis for this new pedagogy.

4.1 Debugging

It could be argued that debugging, although not a specific pedagogy already embeds code comprehension in code writing. However, for the same reasons that writing code itself does not necessarily require code comprehension skills neither does debugging. Error messages are becoming more helpful providing line numbers and informative error messages that make it unnecessary to inspect their code at length to find a fix. Even when the bug is more subtle and does not create a compiler or runtime error, students learn behaviours that fix common classes of bugs and try them in the hope they will fix the bug they have. For example, *"off by one"* errors are common, so some students learn that inserting *"+1"* or *"-1"* into their code sometimes fixes their bug.If it doesn't they try something else until something does work.

4.2 Parson Problems

Whilst Parson Problems can be automatically generated from arbritrary code, they would not be effective for a learner that had recently written the code themselves. They can avoid building or using an understanding of the behaviour of the code to recreate the program and complete the puzzle, by instead using the easier method of recollecting what they had just written.

4.3 Tracing

Tracing does force the learner to develop their understanding of the structural behaviour of their code as it requires them to act as the machine. It is also automatable on arbitrary code as has been demonstrated by UUHistle[24].

There are still issues with tracing however, as it can be incredibly time consuming to fully trace even some short pieces of code. Some learners criticise tracing as too repetitive and [24][**?**] It also does not allow the student to build an understanding of larger code structures as it forces them to hand execute the code step by step, restricting the learner to the Atomic and level of the Block Model.

4.4 Summary

The previous section showed that there is an issue with using code writing on its own as a way of teaching code comprehension. This section examined some existing pedagogies but found that none are totally suitable to merging with code writing in their current form. It is not that any of these pedagogies are bad, they are all valuable techniques, but that they are not suitable for developing a learners understanding of the structural behaviour of their own code.

Out of those methods, tracing shows the most potential and adapting tracing to make it less time consuming and tedious is a promising avenue of development that is explored in the rest of the paper.

5. PROPOSAL

Adapt tracing to make it more applicable to the context of their own code.

The eventual aim of the larger body of work for which this paper is a stating point is to create a system that gets student to develop their structural understanding of the code they write themselves. It will be a development environment that forces the student to answer questions on the structural behaviour of the code before getting to see the results of running their program.

5.1 Research Questions

To help create this system, it is necessary to answer these research questions:

1. RQ1 - What questions can be asked about their code to help a student develop their understanding of the code?
2. RQ2 - When asked detailed structural questions do students use a structural or functional understanding to answer those questions and does that change over time?
3. RQ3 - If I ask these questions what affect does this have on their coding strategy?
4. RQ4 - Do those questions improve a student's confidence at writing their own code?

5.2 Scope and Answering Research Questions

The rest of this paper restricts itself to answering these questions. To answer RQ1, the paper briefly describes a series of interviews held with experienced tutors and the types of questions arising from that. To answer RQ2, RQ3, and RQ4 the paper first describes an initial protocol developed for the system, then a study where students answer coding tasks where that protocol is emulated by a human tutor. It does not attempt to implement the system itself.

6. TUTOR INTERVIEWS

[This section is not]

The intention of the interviews with tutors was to gain a better understanding of the types of questions that tutors ask their students. A particular goal was to determine whether tutors used the kinds of structural code comprehension questions described in the paper.

Two tutors took part in the the interviews selected based on their proven track record as effective tutors in introductory programming in the School of Computing Science. The interviews lasted around 50 minutes. During the interview the interviewer played the role of a student and the tutors were given example buggy code supposedly generated by the interviewer/student. The tutor tutored the *"student"* asking the same kinds of question they would ask another student that had written similar code. The *"student"* responded as best they could within the student role, sometimes proposing changes to the code. After the roleplay session tutors were asked about their opinions on an early proposal for the system. These sessions were recorded for later analysis.

The findings from the interviews are described briefly:

- Tutors use questioning methods that do get students to explain their code they have written to a tutor, that will prompt them to reconsider when they deviate from what is actually written. This requires the students to practice their code comprehensions skills.
- Tutors do ask students questions about the structural behaviour of their code often by performing a trace. They both commented that their general strategies were built around tracing or *"stepping through the code"*. This backs up the decision to build the protocol around tracing.
- At some points the tutors deviated from strict tracing, instead opting to select particular iterations of a for loop to trace through rather than all of them or skip parts they deemed to be unimportant. They indicated a preference for selecting the iterations that behave as students expect first and then cases where it does not afterwards. Whilst determining which behaviours are intentional is beyond the abilities of an automaton the concept of being selective about which iterations to use is integrated into the protocol.
- One tutor expressed concern that students would be incapable of attempting to answer structural questions. If it is the case after most of a year of a computing course it is a cause for great concern in both the affected students and the course's teaching staff and one that needs an effective remedy. The other tutor also hinted at a lack of comprehension skills when they discussed students writing code they couldn't manage to understand.
- The tutors also asked functional understanding questions. Although an automated system could not understand responses to these questions, a mixed approach where the automated system covered the structural understanding while human tutors focussed on functional understanding would be valuable.

The tutors were broadly supportive of the concept. The tutor interviews reinforced the choice of tracing as a base pedagogy, but also introduced the ideas of skipping parts of the trace and being selective over which iterations of loops to trace. The structural questions they asked help inform the structural questions used in the protocol.

7. DEVELOPMENT OF PROTOCOL

This section describes a protocol to adaot tracing to make it more applicable to students writing their own code. The general approach is to make answering the tracing questions compulsory in order to run the code for two reasons, it forces them to answer the tracing questions and consider/predict how each part of the code will run before seeing the final result of the code. Under both the prediction literature and ICAP this will improve the effectiveness of code writing as a pedagogical tool to teach code comprehension.

In this paper/study this will be done manually by a tutor but in the long term should be added as a feature of an IDE that when the run button or equivalent is pressed it asks tracing questions.

A protocol will generate the questions that will be asked based on the written code, the rest of this section details this protocol.

7.1 General Principals

The protocol generating the tracing questions to be asked is based on three principals, that questions should be asked in execution order, that only structural tracing questions should be asked and to do only a partial trace.

7.1.1 Ask Questions In Execution Order

Asking questions in execution order follows the normal tracing procedure. Executing the code in order makes it easier for students to follow the questions and track the flow of the program.

7.1.2 Only Ask Structural Tracing Questions

Restricting the questions to ones that excercise their structural tracing ability, ensures that students develop their structural code comprehension ability.

7.1.3 Only Do Partial Traces

This means reducing the amount of tracing that a student needs to complete to run their code. This will reduce the the tedium involved in tracing and makes completing the questions take less time.

The protocol will still ensure questions are asked in certain circumstances including:

- The student has recently answered a question on the same topic incorrectly.
- In loops the protocol will always ask at least three questions.

7.1.4 Target Python

The protocol here is intended to target the Python language. This is because the Python is a common language for introductory programming courses making it a good candidate for a prototype system.

7.2 Introduction To Implementation

This subsection outlines the concepts necessary to understand the subsequent implementation subsection.

7.2.1 Example Code

```
a = 0
for b in [1,2,3]:
    if b % 2 == 1:
        a = b + 2
print(a)
```

Figure 3: Sample code used to demonstrate the developed protocol.

Throughout the explanantion the piece of example code shown in figure 3 is used to explain how the protocol operates. It shows some simple operations a for loop and an if statement, attempting to cover many topics covered early in introductory programming courses.

7.2.2 Syntax Trees

```
a = b + 2
    |
  Assign
    |----------
    |          |
  target     value
    |          |
   Name      BinOp
    |          |-----------------
  id="a"       |        |       |
              left      op     right
               |        |       |
              Name    Add()  Constant
               |                |
              id="b"           value=2
```

Figure 4: A diagram showing how a line of Python code is broken down into its constituent syntax tree by the Python compiler. Some details such as typing comments and where to load values from have been omitted for readability. The nodes types in the tree are named with capital letters and are *"Assign"*, *"Name"*, *"BinOp"* and *"Constant"*.

A syntax tree is created by breaking down code into its constituent parts and are often created during compilation processes. They are built from nodes that represent the separate parts of the code in a program. An example of how a line of the example program breaks down into a syntax tree is shown in figure 4.

7.2.3 Execution Traces

These are similar to the normal output of a trace when done by a student but it is generated by the machine running the program. It is a capture of the values of all declared variables at every step in a programs execution. Figure **??**

```
step=0, line=0, context={}
step=1, line=1, context={a:0}
step=2, line=1, context={a:0, __iterator
    =[1,2,3]}
step=3, line=2, context={a:0, b:1
    __iterator=[1,2,3]}
step=4, line=3, context={a:0, b:1
    __iterator=[1,2,3], __bmod2:1}
step=5, line=3, context={a:0, b:1
    __iterator=[1,2,3], __bmod_2_is_one:
    true}
```

Figure 5: The first five steps of an execution trace of the code in figure 3.

shows the first five steps of an execution trace for the example code. The values of the execution trace at each step is the context at that step.

7.2.4 Contexts

For a stuent to answer a tracing question about the code mid execution, they will often need to know information such as the line or collection of lines currently being executed and the values of variables before that line is executed. The context contains this information so that it is possible for a student to answer questions. This information can often be implied by a human rather than explicitly stated.

7.3 Implementation

To implement the protocol each node visited during an execution trace is mapped to a question type. These question types are used to generate questions based on the context. These questions are then optionally skipped or merged to become questions covering more topics and larger amounts of execution following the skipping rules.

7.3.1 Question Types

There are seven question types for different types of node in the syntax tree.

- Type 0:No Question - This is the null type for nodes that have not been assigned a question.
- Type 1:Value Of Variable - This question seeks to determine the value of a variable in scope. An example question is *"In context X what is the value of variable Y?"*. Nodes of this type will have a method to decide on the variable for *Y*. For an assignment statement *Y* is the variable that is being assigned into.
- Type 2:Value Of Variable In Iteration - This question seeks to determine the value of a variable within a loop. An example question is *"For the Xth iteration of Z loop, what value does Y have?"* This is mostly intended for iterators in a loop.
- Type 3:Value Of Expression - This question asks for the value of an expression or subexpression. An example question is *"In context X what is the value of expression Y?"*, where *Y* is the code represented by the node. This group is intended for expressions.
- Type 4:Boolean Value Of Expression - This question asks specifically for a boolean value and is intended for boolean expressions. An example question is *"In*

context X is the expression Y true or false?", where Y is the code represented by the node.

- Type 5:Next Code To Execute - This question asks what piece of code will be executed next. This is intended for control structures like if statements and loops. An example question is *"In context X what line will execute next?/In context X will Y code execute next?"*

- Type 6:Output - These questions ask for what output there will be. Since there are many possible forms of output, this really many groups each for a different output method and a different question. An example for a call to *print* is *"In context X what is printed?"*

7.3.2 Skipping

Skipping will attempt to avoid asking the students about the same topics too many times if the student shows they understand those topics or areas of code if the student show they understand them by getting questions related to them correct. This can be tracked using a table of question topics, areas of code and the question response history that records whether the responses to questions on a particular topic where correct or not.

When choosing which iterations to do, the protocol chooses the smallest set of iterations that:

- Has at least 3 iterations or every iteration if there are 3 or less iterations.

- Covers every branch of code execution that it is possible to cover. This means an iteration where the if statement is executed is chosen and a an iteration where it does not will also be chosen. If this takes more than 3 iterations, then more than 3 iteration will be chosen.

- Has the earliest possible iterations. This means if all else is equal, choosing to include iteration 1 instead of including iteration 2.

8. STUDY

A study was carried out with a human tutor, the researcher acting as a mechanical tutor would when following the protocol outlined in the previous section.

8.1 Method

The study took the form of one hour tutoring sessions where students were given a series of five coding tasks to complete whilst the tutor emulated the protocol. When completing the coding tasks, the participants were asked to stop before running their code and the tutor asked them tracing questions following the protocol. The participant were asked semi structured interview questions at various points in the sessions.

The study was held at the end of the second semester of first year on a computing science course on machines participants were familiar with from lab sessions in those courses. The programming tasks were done in Python IDLE.

8.1.1 Coding Tasks

There were five coding tasks that required participants to manipulate data in an external module. All the problems and the data to be manipulated are given in Appendix A. Each task was designed to be harder than the previous one and could be solved using code constructs covered in the first year courses the participants were taking. While completing the tasks, whenever a code file was saved as a necessary step to run the code, the contents of the file was copied and stored with a timestamp to be used to match code versions to the questions and discussion that led to them.

8.1.2 Semi Structured Interviews

Before starting the tasks the participants were asked about their confidence in their coding ability. This was an open ended question.

Between each task the participants were asked what they had felt they had learned and how they found the questions asked:

- Do you think you learned anything from this coding task?
- In what ways were the prompting qestions I asked helpful or unhelpful?

After the tasks, the participants were asked about their confidence and how they felt doing the tasks in an semistructured interview, using these questions a prompt:

- How was this style of tutoring in comparison to the tutoring you receive in labs?
- How did the questions I asked compare to the questions other tutors ask?
- What ways might this style change the way you write code? Do you think the code is better?
- How confident do you feel about your coding right now? - Why?
- Does predicting how the code will run change the feel of coding?

8.2 About Glasgow Uni CS1 courses and its tutoring system

Glasgow University's School of Computing has two streams in its first year computing science course each one has a course per semester. One stream sometimes referred to as *"1CT"* is intended for students that have not programmed before and another referred to by the students as *"1P"* is intended for students that had taken programming courses prior to taking the course.

Students work on programming problems in weekly labs with tutors available to help when they get stuck. Tutors receive general training but limited programming specific training.

8.3 Participants

Four students took part in the study, two from the *"1CT"* stream and two from the *"1P"* stream. There were three male participants and one female participant. Based on the exam results the participants achieved in their first semester, the participants represented the spectrum of abilities on the courses.

9. RESULTS AND DISCUSSION

This section covers the outcome of the tutoring sessions held with students. It begins with an annotated transcript of part of the tutoring segment of a session to show the various elements of the protocol in action. It then continues with a discussion based on the responses to the semi structured interviews held as part of those sessions.

9.1 Participant 2 Session Excerpt

This section covers part of the session with participant

```
from people import people

no_in_years = {}
year = 0
for i in range(len(people)):
    year = people[i][”birthday”][”year”]
    if year in no_in_years:
        no_in_years[year] += 1
    else:
        no_in_years[year] = 1

greatest_year = ”1995”
no_in_greatest_year = no_in_years[greatest_year]
additional = []
no_in_year = 0
for k,v in no_in_years.items():
    no_in_year = v
    if no_in_year > no_in_greatest_year:
        additional = []
        greatest_year = k
        no_in_greatest_year = no_in_year
    elif no_in_year == no_in_greatest_year:
        additional.append(k)
    else:
        pass

if len(additional) > 0:
    years_unsorted = [greatest_year]
    for i in range(len(additional)):
        years_unsorted.append(additional[i])
    years_sorted = sorted(years_unsorted)
    for i in range(len(years_sorted)):
        print(years_sorted[i])
else:
    print(greatest_year)
```

Figure 6: The code at the beginning of this set of questions.

2. This is their attempt at the third task to calculate the modal birthyear for a collection of people. Further detail on the task is in Appendix A. They had already answered a number of questions about lists, assignment and execution order during the previous task. This allows parts of the first loop to be skipped. Figure 6 shows the code at the beginning of the excerpt. The first topic the participant had not yet been questioned about was the creation of a dictionary so the questions start there on the third line. The question is based on question type 1.

> ***Tutor:*** At the beginning number of years is?
> ***Participant 2:*** An empty dictionary

The answer here is correct. The next question is also based on question type 1 and tests the understanding of dictionary lookup.

> ***Tutor:*** The first time round your loop what year will you get out?
> ***Participant 2:*** 1995

This starts a sequence of 3 questions about the value stored in the variable *year* all based on questionb type 1. If the tutor was fully following the protocol they would have picked 3 different iterations to ask about the first second and tenth, so that both the code inside the if statement and the code inside the else statement are executed to get more path coverage. However they chose the first 3. All of which are answered correctly, though not always in a way the a machine could easily interpret. (See section **??** for more discussion on this point.)

> ***Tutor:*** What will be the value of number_in_years after the first time round the loop?
> ***Participant 2:*** It will be 1995 mapping to 1
> ***Tutor:*** Ok and after the second time round the loop?
> ***Participant 2:*** 1995 to 1 and also 1997 to 1
> ***Tutor:*** And the third time round the loop?
> ***Participant 2:*** Then you add 2000 to 1
> ***Tutor:*** Yep Ok
>
> ***Participant 2:*** Until I get one that's repeated. At which case it will become 2

Here the tutor and the participant deviate from the question and answer protocol. If the tutor had followed the earlier script more closely and asked about the tenth iteration the first iteration where the else block executes the participant would have already been asked about this case.

> ***Tutor:*** Ok so then we have, just before we go into the second loop, the value of *greatest_year* is?
> ***Participant 2:*** 1995

Returning to the protocol, this question was chosen because the next piece of code after the first loop is assigning a value to *greatest_value*, in the first case the string value *"1995"*. This is the second time they are doing an assignment question . The participant answers this question correctly but does not make clear whether they expect 1995 to be a string or a number.

> ***Tutor:*** What about the *no_in_greatest_year*?
> ***Participant 2:*** *no_in_greatest_year*, *no_in_greatest_year* will be whatever the amount of people in 1995 so I don't know how much that is.

This question was of question type 1. The answer provided by participant 2 is incorrect, actually an error will be thrown. From the participant's response it is clear that the tutor did not provide enough context for them to give an exact answer that a machine could understand. The tutor attempts to provide that context by giving an example value for *no_in_years.*

> ***Tutor:*** Ok assuming that *no_in_years* is a dictionary that contains 1950, 1, 1970, 2, 1995, 3. What would you expect this bit to evaluate to?

The value for *no_in_year* provided by the tutor were made up ad hoc by the tutor because it was infeasible to be able to work out the exact values here as it would require emulating the machine through 26 iterations in their head. If the protocol were used here the value would be the actual value at the point of execution as the context, in some contexts the machine can use the rules for finding values of interest to remove some values, but not in this case, but it can still provide all the values.

> ***Participant 2:*** 3
> ***Tutor:*** Ok, that's not correct
> ***Participant 2:*** Oh no ermm huh, what have I done then?
> ***Tutor:*** If *no_in_year* is the number 1950 to 1, the number 1970 to 2

The tutor has still not given the student all the information from the context to answer the question correctly because they did not give the type of the keys, again the tutor attempts to address this by repeating the values of the dictionary with added type information. However this made the error more obvious than a machine that was always relaying type information would make the error.

This part demonstrates a weakness of asking the question verbally. It is not clear whether the spoken response 1995 is intended as a string or an integer or even a float. The only way to convey the types is to actively say the types. In text *"1995"* and *1995* are clearly different. For this part the types were important but conveying them here when they were not usually communicated gave the tutee a hint more obviously than a computer would have.

> ***Participant 2:*** Oh ok, ahh I'm putting in a string. So I need, I'm putting in aye because these aren't strings these are integers. So I'm just gonna go and maintain my streak.

At this point the code in figure 6 is altered so that 1995 is an integer. The bug fixed was an error matching types which would have created a KeyError at runtime. This could be a translation error between the plan of their code and the code they had written. After this fix the code would return the correct answer but still contained planning errors. If participant 2 was following their usual coding technique of running their code then changing it until it gives the correct answer, they would likely have stopped at this point. However they continued to spot another bug.

> ***Tutor:*** Ok you've just changed it
> ***Participant 2:*** Changed it to an integer rather than a string because it will give me a, it will say that it is not in the dictionary because the string 1995 isn't in the dictionary, but the number 1995 is and now it will give me 2.

This explanation of what the participant learned could be captured by a machine if it prompted for a reason for code changes. Whilst the machine itself could not understand the reasons, it could display them back to learners so they can see what they learned, provide them to educators so they can use them to inform teaching and create a rich dataset for computing education research.

After explaining their misunderstanding, participant 2 attempts to anticipate the next question and the tutor accepts this response. This is a deviation from the protocol as the next question should go back in the execution to where the altered code is, so the question should have been about the value in *greatest_year*, followed by a question about what *no_in_years[greatest_year]* evaluates to that the participant anticipated.

> ***Tutor:*** Thats correct
> ***Participant 2:*** Whew

This skips questions asking for the values in *additional* and *no_in_year*. The question about the value in *additional* would have been asked under the protocol, since up to this point the participant had not answered questions about creating lists. The question about the value in *no_in_year* would have been skipped in the protocol as the participant had already correctly answered questions about integer literals and assignment.

The next question is about the values of the tuple k and v during the loop. This question is type 2. This required extra context to be able to answer correctly since in some Python versions the order of iteration over a dictionary is unpredictable. In Python 2 the order of iteration is deterministic but difficult to predict, in Python 3.5 the order is non-deterministic making it impossible to reliably predict correctly, but in Python 3.6 and later the order of iteration is the order the keys were added which is easy to predict. The tutor attempts to pre-empt this issue by telling the participant the order the key value pairs will be chosen but in doing so provides the answer to the question, and the participant answers correctly. This raises an issue about predicting non deterministic behaviour. For example it is impossible to reliably predict what *random.randint()* will return. In these cases the protocol could provide the outcome instead of asking questions on it. The difficulty with that approach is how to identify non deterministic methods and functions.

> ***Tutor:*** Ok. What do you expect the values of k and v to be in the first iteration round this the second loop? Assuming that the first thing is 1995 and 3 and
> ***Participant 2:*** k will be 1995 and v will be 3
> ***Tutor:*** Yeah

This begins a short interaction that does not follow the protocol. It would be difficult for a machine to understand the description of the behaviour the participant gives but a human tutor would be able to pick up on the statement to help them gauge how well the participant understands iterating over a dictionary.

> ***Participant 2:*** Then it will just loop through everything in the dictionary.

> ***Tutor:*** Thats fine then. Yeah that will make sense. So your *no_in_year* would be?
> ***Participant 2:*** Whatever the val, so 3 the value of 1995

This returns to the protocol, asking a question of type 1 about the first line of the second for loop. The participant answers this question correctly. The next question generated by the protocol was type 4 and evaluated the expression used in the if statement. The tutor words the question differently but the essence of the question asked follows the protocol.

```
for k,v in no_in_years.items():
    no_in_year = v
    if no_in_year > no_in_greatest_year:
        additional = []
        greatest_year = k
        no_in_greatest_year = no_in_year
    elif (no_in_year == no_in_greatest_year) and (k !=
         greatest_year):
        additional.append(k)
    else:
        pass
```

Figure 7: The code after the second correction correction. Notice the added *"and (k != greatest_year)"* to deal with the case where 1995 is a modal year.

> ***Tutor:*** Yeah. Will *no_in_year* be greater than *no_in_greatest_year*?
> ***Participant 2:*** Oh, no it will be equal. It will be equal to it in which case I'm going to get a duplicate. Argh. You know what I'm just going to do a quick fix. Let me just go. *k* is not so the one that is not equal to *greatest_year*.

By going through the thought process of answering the question the participant notices a planning error they have made. After giving the correct answer, participant 2 alters their code into figure 7 to remove the error. The fix participant 2 uses is arguably not the best strategy, but value judgements about coding strategy is beyond the abilities and aims of the protocol.

> ***Tutor:*** Ok? Ok?
> ***Participant 2:*** Oh yeah, yeah, yeah.

Since the change was after the current point of execution in the code the questioning can continue from where it was and ask about the next part to be evaluated, the boolean expression in the *elif* statement, with type 4 questions. Because the participant had not answered questions on either == or *!=*, the larger *and* expression is broken down into its two subexpressions first before asking about the value of the larger *and* expression.

> ***Tutor:*** We're doing 1995 and 3. Is *no_in_year* equal to *no_in_greatest_year*?
> ***Participant 2:*** Yes

This answer is correct and simple enough for a machine to interpret.

9.1.1 Another Machine Spottable Error

In their code participant 2 places a lone *pass* statement as the code to execute in an *else* case. This code will behave the same if the *else* and *pass* statement were not there and hints at a misunderstanding of how an *if else* statement works and is often seen as bad coding. Linters such as Pylint[26] are well established software development tools that can spot this type of bad coding by pattern matching the code against a predefined set of patterns. For educational purposes, a tool could point out some of the patterns spotted by a linter. They could then either ask why they wrote their code that way and/or provide a prewritten text that explains why that way of coding is seen as bad coding.

9.1.2 Understanding Natural Language Responses

The participants often gave natural language responses such as *"so 3 the value of 1995"* to questions. The tutor had to make on the spot decisions about whether the response clearly showed what they would enter into a text box and to ask again if the response was not specific enough. It is clear from the response *"so 3 the the value of 1995"* that the participant would have entered *3* into a text box. It is not as clear what they would enter by responses like *"K is equal"* so a clarifying question *"Ok so is that true or false?"* was required. The tutor generally accepted answers that stated the participant's answer even if it was surrounded by the reasoning leading to their answer but sought clarification if only reasoning was given.

Responses regularly omitted the types of the value in the responses. The tutor assumed the participant had the correct type instead of frequently seeking clarification of the type.

9.1.3 Summary Of Interaction

The interaction with participant 2 was broadly successful, it helped students not only see their errors but their source, allowing them to find a workable bugfix on their first attempt. It was close enough to the protocol to indicate that an automaton would have prompted similar realisations and code changes.

9.2 Not Directly Possible To Show Structural Understanding Used

It proved difficult to identify definitively whether the participants were using a structural understanding of their code to answer the questions. The answers given by the participants could have been reached by using a functional understanding of the code and by reading the data.

However, a structural understanding was demonstrated when, like participant 2, the participants found an error in their code whilst answering the questions. This would not be possible if the participants had not considered the code they had written at a structural level.

9.3 It made them think

A common theme among the participants was to view the run button as helpful tool to fix errors.

> I'm very much just run it, run it, run it.
> ***Participant 2***

It highlighted problems with their code, and helps them achieve their goal to write programs that *"work"*.

Student 1 reported using print statements to help them when they were stuck.

> ***Participant 1:*** Quite frankly when I am coding its more of try this try that keep trying until it works, if I really am struggling start asking tutor.
> ***Tutor:*** What do you mean by keep trying, what methods would you use when you are trying?
> ***Participant 1:*** Definitely use print statements everywhere
> ***Tutor:*** Print statements everywhere?
> ***Participant 1:*** Like if this, I want to check, put this in, newline print that, let me see what

I've got so far and I wouldn't normally do that first of all but say that I come in a few lines of code later on and it doesn't give me what I'm expecting.
Tutor: How do you decide what you are expecting?
Participant 1: Well based on the question
Tutor: Yeah when you are doing print statements, would you consider what you would expect them to be first before you run the code?
Participant 1: Oh yeah
Tutor: You do that already?
Participant 1: Yeah, otherwise its just like typing monkeys

Some students had a trial and error approach to coding, though they seem to have doubts about it as a strategy.

> Is it ok to just trial and error it?
> ***Participant 1***

> Because usually I would just write it, see if it works, if it works great move on if it doesn't work oh god ok why is not working. Whereas, you know before my style of coding is code and you know run it and hope for the best. Run it and you know maybe it will run, maybe it won't run.
> ***Participant 2***

3 of the 4 students in the study reported looking at their code more closely and felt this was a potential long term effect of the tutoring.

> I feel like this is good for making me pay more attention to details.
> ***Participant 4***

For most of the participants this was seen as a potential long term change in behaviour towards considering their code more deeply.

> I don't think so but I think after doing this for I don't know how long for, but I will be staring at my code for longer before I use the run button I guess maybe inspecting it more.
> ***Participant 3***

> Doing it this way and you know asking questions before you ran it I was a lot slower but I was scrutenising my code in a lot more. I was looking at it a lot more in depth than what I would usually do. Because usually I would just write it, see if it works, if it works great move on if it doesn't work oh god ok why is not working. Whereas, you know before my style of coding is code and you know run it and hope for the best. Run it and you know maybe it will run, maybe it won't run. Whereas the approach that we've had this afternoon, I'm hitting that run button going this is going to run because I've looked at it and its like yeah nothing is getting past me because I've scrutinised it so much.
> ***Participant 2***

Some have become very skeptical, not just doubting what they think they have written behaves as expected but doubting all of their existing knowledge.

> It made me suspicious of everything. Its like does this do what it does?
> ***Participant 3***

For participant 3 that doubt might be justified, during the tutoring session with them it became clear to the tutor that they were struggling, particularly when it came to the operation of dictionaries.

9.4 Affect on Confidence

Three of the four participants reported that the session had resulted in a drop in their confidence as programmers.

> My confidence has gone down.
> ***Participant 3***

> Well after this I'm beginning to question a lot. I'm beginning to say to myself in my head I'd better be revising.
> ***Participant 1***

Participant 4 says that their confidence in their abiliity to catch errors on their own went down, but their confidence in other.areas was unaffected.

> I'm still equally confident about my abilities as far as my language goes and in general problem approach or ability to solve problems, just the attention to detail in like how my able to without testing the code right away actually going through it and thinking about it., like how many errors could I catch if I pay more attention to detail. In this sense my confidence went definitely down.
> ***Participant 4***

When asked why they felt their confidence had fallen, they pointed to things it had shown them they couldn't do. For participant 1 this was a surprise as they believed they were capable of more.

> Probably the number of mistakes I made, probably that and how long it took to solve these kinds of questions because I don't think it should have taken this long cause I'm quite aware that the labs I've done so far are more complex. Like based on the first three questions these are things that I have done and I would have expected more of myself
> ***Participant 1***

Participant 3 brought up areas they had recognised they were weak in in the past and the session had shown that they still had difficulty in those areas.

> The part involving dictionaries and counting cause thats all something I couldn't grasp last year and I've had no contact with it once again.
> ***Participant 3***

While performing these tasks it was pointed out to participant 2 that they had managed to complete all the tasks only actually running each piece of code once each. This had an affect on how they viewed their performance and their confidence. When asked if their confidence had improved or got worse during the session.

> Definitely not worse. Better.Not worse at all. My confidence is better but not in a more co not in a cocky way. After this you know I could hold my head a little bit higher and say brightly yeah I'm a good programmer whereas before I would be like quietly yeah I'm a good programmer.
> ***Participant 2***

They said that it was because they had succeeded at three of the tasks in a single run each time.

> Getting it to run first time. Being able to you know complete three, even although its not five, three's better than one and one's better than none. I'm happy with getting three done.
> ***Participant 2***

Participant 2 may have had a more positive experience because answering the questions in the session is similar to the tracing they reported doing normally during planning.

> I like to write the solution down on paper first. I do like to do that a lot, rather than just write and go so like I want to draw my lists and after each step what's happened to them and them I'm able to think am I better using a list am I better using a dictionary? Can I do this in one loop or two?
> ***Participant 2***

The session it was revealed a weakness in a skill that they had not practiced before, while that lead to a decrease in confidence in students, it helps that they are now more aware of what they do not understand. It enables students to target their learning towards areas that that they need to improve. However it highlights how the current instructional design fails to get them to create working programs first time and to teach them good code comprehension skills that are necessary to effectively fix the programs they write that initially do not work.

9.5 Threats To Validity

The study's validity as a study of how an automated system would work is threatened by the humaness of the researcher. The sessions were run by a human and the participants may have reacted differently were a machine asking the same questions and at times the protocol was not exactly followed, but it was followed closely at least 60

10. CONCLUSIONS AND FUTURE WORK

This research aims to automate the process of asking students code comprehension questions as they solve programming problems at an IDE. Four research questions were set in section 5.1. Answers to these questions based on the research the study presented are given here.

A set of questions that could be asked was outlined in section 7.3.1 to answer RQ1 and outlined how they could be automated. For RQ2, it was not possible to directly assertain whether the participants used a structural or functional understanding to answer questions, but they reported studying their code more closely suggesting that they were performing some form of analysis that they wouldn't otherwise do.

The protocol had a clear effect on the student's coding style helping them to analyse their code in more detail and fix planning errors by understanding the source of the error. From the responses given by the students this could become a long term change in behaviour if the protocol were used on a long term basis. This answers RQ3.

The answer to RQ4 on student's confidence is more complex. The questions can have a negative effect on confidnce when they reveal deficiencies in understanding that students were not aware of. The current teaching methods have allowed these deficiencies to go unnoticed for most of an academic year, lulling the students into a false sense of security. Being suddenly confronted with unexpected deficincies in their ability comes as shock. Since the confidence of the student whose deficiencies were not found improved, it suggests that using the technique from thebeginning will prevent the learning deficit ever forming.

11. FUTURE WORK

11.1 Create And Test The System

This will be difficult. It needs to be shown it can be built and still works if a machine does the questioning instead of a person.

11.2 Different Types Of Learners

This study focussed on programmmers in their first year of a university course, some had never coded before, other a little but in different languages. Would similar techniques work for more experienced programmers attempting to learn a new language? Is it suitable for younger coders given the number of participants whose confidence was knocked?

11.3 Other Programming Languages

This work focussed on Python since it is the main language in the two introductory coding courses in the institution the work was carried out in, but there are many different languages in existance. How would the techniques covered in the paper be adapted to an object oriented style of coding or to the graphical programming languages like Scratch often used to teach younger children?

In functional programming languages like Haskell the order of execution is often unpredictable, what effect would that have on how the technique would be implemented. These languages are rarely a programmer's first language but they do diverge significantly from the procedural and object oriented programming paradigms most programmers are familiar with. Do similar techniques work for learning the new coding paradigm?

11.4 Other Code Comprehension Pedagogies

This study focussed on tracing but putting other pedagogies into the code writing process may be effective in getting students to think more deeply about their code. For example Parson problems

12. REFERENCES

[1] P. Brusilovsky, S. Edwards, A. Kumar, L. Malmi, L. Benotti, D. Buck, P. Ihantola, R. Prince, T. Sirkiä, S. Sosnovsky, J. Urquiza, A. Vihavainen, and M. Wollowski. Increasing adoption of smart learning content for computer science education. pages 31–57. ACM, 2014.

[2] C.-C. Carbon. Understanding human perception by human-made illusions. *Frontiers in human neuroscience*, 8:566, 2014.

[3] M. T. H. Chi and R. Wylie. The icap framework: Linking cognitive engagement to active learning outcomes. *Educational Psychologist*, 49(4):219–243, 2014.

[4] M. Corney, S. Fitzgerald, B. Hanks, R. Lister, R. McCauley, and L. Murphy. "explain in plain english" questions revisited: Data structures problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, page 591–596, New York, NY, USA, 2014. Association for Computing Machinery.

[5] C. Crouch, A. P. Fagen, J. P. Callan, and E. Mazur. Classroom demonstrations: Learning tools or entertainment? *American Journal of Physics*, 72(6):835–838, 2004.

[6] Q. Cutts, M. Barr, M. Ada, P. Donaldson, S. Draper, J. Parkinson, J. Singer, and L. Sundin. Experience report: thinkathon - countering an 'i got it working' mentality with pencil-and-paper exercises, 2019.

[7] J. Goslin. Bluej. https://www.bluej.org/.

[8] P. J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 579–584, New York, NY, USA, 2013. Association for Computing Machinery.

[9] M. Hertz and M. Jump. Trace-based teaching in early programming courses. pages 561–566. ACM, 2013.

[10] C. D. HUNDHAUSEN, S. A. DOUGLAS, and J. T. STASKO. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[11] P. Ihantola and V. Karavirta. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education*, 10:119–132, 2011.

[12] M. C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[13] H. Keuning, J. Jeuring, and B. Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2019;2018;.

[14] P. Kinnunen and B. Simon. My program is ok - am i? computing freshmen's experiences of doing programming assignments. *Computer Science Education*, 22(1):1–28, 2012.

[15] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. pages 101–112. ACM, 2008.

[16] D. Parsons and P. Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163. Australian Computer Society, Inc., 2006.

[17] M. Pilotti, M. Chodorow, and K. C. Thornton. Effects of familiarity and type of encoding on proofreading of text. *Reading and Writing*, 18(4):325–341, 2005.

[18] J. Sajaniemi and M. Kuittinen. Program animation based on the roles of variables. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, page 7–ff, New York, NY, USA, 2003. Association for Computing Machinery.

[19] C. Schulte. Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching. pages 149–160. ACM, 2008.

[20] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. Paterson. An introduction to program comprehension for computer science educators. pages 65–86. ACM, 2010.

[21] D. L. Schwartz and J. B. Black. Shuttling between depictive models and abstract rules: Induction and fallback. *Cognitive Science*, 20(4):457–497, 1996.

[22] S. Sentance and J. Waite. Primm: Exploring pedagogical approaches for teaching text-based programming in school. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, WiPSCE '17, pages 113–114, New York, NY, USA, 2017. ACM.

[23] P. Silva, E. Costa, and J. R. de Araújo. An adaptive approach to provide feedback for students in programming problem solving. In A. Coy, Y. Hayashi, and M. Chang, editors, *Intelligent Tutoring Systems*, pages 14–23, Cham, 2019. Springer International Publishing.

[24] J. Sorva and T. Sirkiä. Uuhistle: A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, page 49–54, New York, NY, USA, 2010. Association for Computing Machinery.

[25] V. Vainio and J. Sajaniemi. Factors in novice programmers' poor tracing skills. *SIGCSE Bull.*, 39(3):236–240, June 2007.

[26] Various. Pylint. https://www.pylint.org/.

[27] C. Watson, F. Li, and J. Godwin. No tests required: comparing traditional and dynamic predictors of programming success. pages 469–474. ACM, 2014.

[28] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 319–323. IEEE, 2013.

[29] B. Xie, G. L. Nelson, and A. J. Ko. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 344–349, New York, NY, USA, 2018. Association for Computing Machinery.

APPENDIX

A. STUDENT STUDY

There is a list of people represented as dictionaries. Each person has a name, a date of birth and a list of their siblings' names. The date of birth is broken down into a year and a day of year ranging between 1 (1st January) and 366 (31st December in a leap year). Each person's name is unique. To get the list of people start your Python program with:

```
from people import people
```

It has the format:

```
people = [
    {
        "name":"Alice",
        "birthday":
        {
            "year":1995,
            "day":143
        },
        "siblings":
        [
            "Bob",
            "Clive"
        ]
    },
...
]
```

These questions get more difficult as you progress through them, do not worry if you cannot complete them corectly. You are not being assessed here but try to answer as well as possible. There are expected outputs on the next page. Remember to ask before running your code so you can be asked questions about your understanding.

1. Write a program that prints out the names of the people with no siblings in the order they appear in the list people.
2. Write a program that calculates the total length of all the names of the people, then prints it.
3. Write a program that finds the modal birth years. This means finding the years that most people were born in (there may be more than one). Then print out each modal year on a separate line in ascending order.
4. Write a program that finds all the sets of twins, triplets ect in the dataset. You can assume twins and triplets share the same birthday and have each other as siblings. It should then print out each set of twins or triplets on a separate line printing the name of each twin or triplet on the same line. The twins names should be in alphabetical order. The sets of twins or triplets should be in age order with the oldest set of twins or triplets first.
5. Write a program that finds all people that have more older siblings than younger siblings. People born on the same day are assumed to be the same age. It should then print out the names of these people each on a single line and in alphabetical order.

```
people = [

    {
        "name":"Alice",
        "birthday":
        {
            "year":1995,
            "day":143
        },
        "siblings":
        [
            "Bob",
            "Clive"
        ]
    },
    {
        "name":"Bob",
        "birthday":
        {
            "year":1997,
            "day":267
        },
        "siblings":
        [
             "Alice",
             "Clive"
        ]
    },
    {
        "name":"Clive",
        "birthday":
        {
            "year":2000,
            "day":8
        },
        "siblings":
        [
             "Alice",
             "Bob"
        ]
    },
    {
        "name":"Doris",
        "birthday":
        {
            "year":1950,
            "day":342
        },
        "siblings":
        [
            "Isabel",
            "Leonard",
            "Nancy",
            "Toby",
            "Wallace",
            "Vivian"
        ]
    },
    {
        "name":"Eve",
        "birthday":
        {
            "year":2009,
            "day":136
        },
        "siblings":
        [
            "Zoe",
            "Xander"
        ]
    },
    {
        "name":"Fred",
        "birthday":
        {
            "year":1970,
            "day":267
        },
        "siblings":
        [
            "George"
        ]
    },
    {
        "name":"George",
        "birthday":
        {
            "year":1970,
            "day":267
        },
        "siblings":
        [
            "Fred"
        ]
    },
    {
        "name":"Harris",
        "birthday":
        {
            "year":2018,
            "day":302
        },
        "siblings":
        [
            "Rebecca",
        ]
    },
    {
        "name":"Isabel",
        "birthday":
        {
            "year":1950,
            "day":342
        },
        "siblings":
        [
            "Leonard",
            "Nancy",
            "Toby",
            "Wallace",
            "Doris",
            "Vivian"
        ]
    },
    {
        "name":"James",
        "birthday":
        {
            "year":1930,
            "day":278
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Kirsty",
        "birthday":
        {
            "year":1990,
            "day":198
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Leonard",
        "birthday":
        {
            "year":1950,
            "day":342
        },
        "siblings":
        [
            "Isabel",
            "Nancy",
            "Toby",
            "Wallace",
            "Doris",
            "Vivian"
        ]
    },
    {
        "name":"Maibhe",
        "birthday":
        {
            "year":1995,
            "day":143
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Nancy",
        "birthday":
        {
            "year":1946,
            "day":342
        },
        "siblings":
        [
            "Isabel",
            "Leonard",
            "Toby",
            "Wallace",
            "Doris",
            "Vivian"
        ]
    },
    {
        "name":"Olivia",
        "birthday":
        {
            "year":2010,
            "day":320
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Poppy",
        "birthday":
        {
            "year":1985,
            "day":87
        },
        "siblings":
        [
            "Quinn"
        ]
    },
    {
        "name":"Quinn",
        "birthday":
        {
            "year":1982,
            "day":36
        },
        "siblings":
        [
            "Poppy"
        ]
    },
    {
        "name":"Rebecca",
        "birthday":
        {
            "year":2020,
            "day":60
        },
        "siblings":
        [
            "Harris"
        ]
    },
    {
        "name":"Siva",
        "birthday":
        {
            "year":1994,
            "day":207
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Toby",
        "birthday":
        {
            "year":1951,
            "day":320
        },
        "siblings":
        [
            "Isabel",
            "Leonard",
            "Nancy",
            "Wallace",
            "Doris",
            "Vivian"
        ]
    },
    {
        "name":"Ursula",
        "birthday":
        {
            "year":2001,
            "day":156
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Vivian",
        "birthday":
        {
            "year":1951,
            "day":320
        },
        "siblings":
        [
            "Isabel",
            "Leonard",
            "Nancy",
            "Toby",
            "Wallace",
            "Doris"
        ]
    },
    {
        "name":"Wallace",
        "birthday":
        {
            "year":1944,
            "day":195
        },
        "siblings":
        [
            "Isabel",
            "Leonard",
            "Nancy",
            "Toby",
            "Doris",
            "Vivian"
        ]
    },
    {
        "name":"Xander",
        "birthday":
        {
            "year":2007,
            "day":345
        },
        "siblings":
        [
            "Zoe", "Eve"
        ]
    },
    {
        "name":"Yana",
        "birthday":
        {
            "year":1965,
            "day":167
        },
        "siblings":
        [
        ]
    },
    {
        "name":"Zoe",
        "birthday":
        {
            "year":2006,
            "day":230
        },
        "siblings":
        [
            "Xander", "Eve"
        ]
    },
]
```

Information Sheet: Run Button Study

What Is The Study About?

This study focuses on how people learning to program can be helped to better understand the code they write. It is important for programmers to be able to fully understand their own code so that they can alter their work, fix bugs and explain it to others. The study looks at whether being asked questions about how the code will behave during execution before seeing the code execute forces programmers to be clearer about how their propgrams work.

What Data Do You Collect?

This study collects the code written by participants and their responses to questions. If you consent a sound recording of the study will be made, if consent for an audio recording is witheld, written notes will be made instead. These are stored anonymously and identifying comments or code such as names or student ids will be removed.

Who Can I Contact?

If you have any questions regarding this study or wish to withdraw consent please contact either of:

- Angus McIntosh 2182976m@student.gla.ac.uk
- Quintin Cutts quintin.cutts@glasgow.ac.uk

Participant Consent Form: Run Button Study

The purpose of this study is to assess the effectiveness of different ways of interacting with the run button as a tutoring technique.

The study will take about an hour.

During the study you will be presented with a series of coding problems for Python. While answering these problems you will have access to a run button to test your work. However, you will be required to answer questions about your code before using the run button. After 50 minutes you will be asked to take part in a short interview about your experience answering the problems

If you do not consent to an audio recording of the study, written notes of the study will be made. Your responses to questions and the code you write will be recorded. All responses will be held in strict confidence, ensuring the privacy of all participants. No personal participant information will be stored with the data. Online data will be stored in a password protected computer account; paper data will be kept in a single-occupant locked office.

It is the tutoring technique, not you, that is being evaluated. You may withdraw from the study at anytime without prejudice, and any data already recorded will be discarded. Participation in this study will not affect any grades you receive.

If you have any further questions regarding this study, please contact either of:

- Angus McIntosh 2182976m@student.gla.ac.uk
- Quintin Cutts quintin.cutts@glasgow.ac.uk

I have read this information sheet and agree to voluntarily take part:
Name: ____
Signature: ____
Email: ____
Date: ____
I agree to an audio recording of the interview ☐

B. CODE QUESTION MAPPING

This is a mapping between Python code constructs and the question type numbers.

```
import , 0
assignment , 1
empty dict creation , 1
for X in list , 2
dict loopup , 3
list creation using [X], 1
list creation using [], 1
!= dict , 4
and , 4
== dict , 4
string in list(membership), 4
list.append(), 1
list.sort(), 1
if, 5
cast int to int , 3
int + int, 3
int * int, 3
int - int, 3
dict assignment , 1
dict.keys(), 3
sorted(list, key), 3
sorted(list, key, reverse), 3
lambda expression , 0
string.join(), 3
print(), 6
dict[v] X= v (dict assignment and lookup
    using +=/-= ect), 1
while, 5
int += int, 1
if ... else, 5
[X for x in y] (list comprehension), 1
for k, v in dict, 2
elif, 5
range(int), 3
len(list), 3
>, 4
list.items(), 3
pass, 0
"string literal", 3
list[i], 3
dict.iteritems(), 3
```