Automated Assessment of Learning Objectives in Programming Assignments

Arthur Rump
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
hello@arthurrump.com

ABSTRACT

With online forms of education, it has become harder to 'gauge the room' and get an impression of how well students are following along. We introduce Apollo, a tool that automatically analyses code uploaded by students to get an overview of their progression towards the learning objectives of the course. First, typical learning objectives in Computer Science courses are analysed on their suitability for automated assessment. A set of learning objectives is analysed further to get an understanding of what achievement of these objectives looks like in code. Finally, this is implemented in Apollo, a tool that assesses achievement of learning objectives in Processing projects. Validation of the tool is not conclusive, but early results suggest an agreement in assessment between Apollo and teaching assistants.

Keywords

Programming Education, Automated Assessment, Automated Feedback

1. INTRODUCTION

Education has been shifting to digital environments, recently accelerated due to the coronavirus pandemic. This has made it harder for teachers to keep track of the progress and understanding of their students. With lectures over a live stream (or even pre-recorded), they receive very little feedback from the audience and it has become impossible to get a quick impression of the room during a tutorial session.

For the Creative Technology programme at the University of Twente, the Atelier [7] system was developed to aid with communication between students and teaching assistants during programming tutorials. Atelier lets student upload the code they are working on and share it with teaching assistants. Teaching assistants can leave comments on the code to remind students of the points they discussed. One of the main features of Atelier is an automated code checker that makes comments on the code based on common problems [6].

The use of Atelier enables a smooth experience for individual communication during tutorials, but it does not solve the problem of 'gauging the room.' In this paper, we introduce Apollo, which aims to analyse the student programs in Atelier and give an overview of the overall progress towards the desired learning outcomes for the course. This allows teachers to get a global view of the topics where students are struggling and which students potentially need more help to get up to speed.

The main question this paper aims to answer is:

How can a student's mastery of a programming concept be assessed by automatically analysing their code?

This question is best answered by splitting it into three questions, each building towards an automated tool to assess achievement of learning outcomes. First, there is the question of the curriculum, and deciding which learning outcomes are suited for this automated assessment:

RQ1 What are typical learning outcomes for introductory programming courses and which of these are suitable for automated assessment based on code analysis?

Then, these learning outcomes need to be worked out to specific patterns in code that identify the achievement of such a learning outcome:

RQ2 How can the achievement of learning outcomes be recognized in code?

Lastly, there is the question of how this understanding can be formalized to the extent that a computer could recognize these levels of understanding by analysing the code:

RQ3 How can an automated tool assess the mastery of a concept by analysing code?

With the answers to all these questions, a technical foundation for Apollo is established, which can be used to create a dashboard for teachers to keep track of the achievements of their students.

2. BACKGROUND AND RELATED WORK

In the past, research has been done on automated feedback for programming exercises, which in some cases also includes analysis of different concepts used in the code written by a student. On the side of the computer science curriculum and learning objectives, efforts have been made to provide guidelines for the design of a computer science curriculum. However, we first need to clarify the terminology on learning outcomes, aims, goals and objectives.

2.1 Learning outcomes

Learning outcomes come in a range from vague aspirations for students to achieve at the end of a three-year program to very specific objectives to accomplish in a single lecture. Wilson [24] splits learning outcomes into aims, goals and objectives according to their specificity:

33rd Twente Student Conference on IT July 3rd, 2020, Enschede, The Netherlands.

- **Aims** give a general direction and are not directly measurable. They are useful for guiding the principles of an entire program or subject area. An example from the intended learning outcomes for Creative Technology: "Graduates understand and can use technology in the domain of software, algorithms and physical interaction." [27]
- **Goals** are more specific than aims in terms of scope, but they can still relate to an entire program or subject area. They can be formulated as a concrete action, but don't have to be. "Students can create algorithms for solving simple problems."
- **Objectives** are often written in behavioural terms to describe more specific learning outcomes. They should be observable and measurable. "Students can implement a divide-and-conquer algorithm for solving a problem."

Note that we use the term 'learning outcome' to mean the intended learning outcome of a course, which may or may not be the actual learning outcome for a student. This use of the term seems to be common in educational literature, but there is no general agreement on it. Suskie [23], for example, uses 'learning outcomes' to refer to the actual outcomes of a course and 'learning goals' to refer to the intended learning outcomes. She does make a similar distinction in specificity between aims and objectives.

2.2 Automated feedback on programming exercises

There are multiple ways to approach automated assessment of programming assignments. An approach that has been used since the 1960s is the use of automated testing tools to check student submissions for correctness. Douce, Livingstone and Orwell [5] give an overview of tools that use this approach in their review of the topic. They describe several generations of these tools, starting with simple input/output comparisons in a command-line tool and later moving on to more sophisticated systems based on industry tooling with web-based frontends.

All these tools have one commonality: they require well-defined exercises with supplied test cases to function correctly. This means that this approach cannot provide any value in a setting where the code written by students is more based on their creativity, rather than a well-defined assignment. In later years, other approaches to assessing student code have been taken that do not depend on the program having to generate a predefined output for some input.

Keuning, Jeuring and Heeren [12] focus on these types of feedback tools in their review of the space. They explicitly excluded tools that solely depend on the testing approach from their review, to focus on the other possible approaches. They find that the 101 tools they surveyed use a total of 8 approaches to provide feedback to students, of which automated testing is just one. In general, they find three types of approaches: looking at actions the student takes and comparing those with production rules for a correct solution; looking at the final solution the student submits and checking it against some constraints; and the data-driven approach, where feedback is generated based on past submissions by students.

For the specific case of programming, they find five more approaches: automated testing; use of external tools, such as the compiler, to check for correctness; basic static analysis, which can be used to detect patterns and antipatterns; the translation of programs into another language or a simplified form; and intention-based diagnosis, where the tool tries to find the students strategy and matches it with known goals, plans and rules.

Two tools are of particular interest, because of their relevance to this project: PROUST and the ACT Programming Tutor.

PROUST was developed by Johnson and Soloway [10, 11] as a tool that could provide feedback on free-form programming assignments where multiple solutions could be correct. The tool knows some of the high-level goals that would be required to solve the exercise and many potential subgoals. All these goals are related to plans, which can be matched with parts of the code in a student's solution [22].

The notion of goals and plans is useful in the context of assessing learning objectives since these are often written as goals a student should be able to achieve. By working out the plans related to these goals, an automated tool could be able to recognize the achievement of a learning objective.

The ACT Programming Tutor (APT) created by Corbett and Anderson [4] has a Skill Meter that shows the probability that the student has mastered that skill. APT is also goal-oriented, but works with production rules based on the current assignment: it won't even let students take steps that are not on a known path to a correct solution to the problem. Each time the student has the opportunity to apply a production rule, the tool updates the chance that the student has learned the rule, taking into account the previous chance, a general transition probability for this rule and whether or not the rule was correctly applied at this occasion.

While the Skill Meter has a similar goal to this paper, their approach is not applicable in our context, because it depends on keeping track of all the actions the student takes when writing the code. Besides that, all assignments need predefined production rules to arrive at a correct answer, whereas the context of Creative Technology does not have such strict assignments.

2.3 Curriculum and learning objectives

The most recent volume of the Computer Science curriculum guidelines by the Joint Task Force on Computing Curricula, a cooperation between the ACM and IEEE, was published in 2013 [1]. The guidelines define a body of knowledge of all areas and topics that should be part of a computer science curriculum, organized into knowledge areas like "Operating Systems" and "Software Development Fundamentals." These areas are further organized into units (like "Algorithms and Design"), which in turn define a granular list of topics (e.g. "The concept and properties of algorithms") and also provide learning outcomes for courses that teach these units.

The learning outcomes come with a rough categorization in three "levels of mastery":

- Familiarity: the student knows what a concept, or the meaning of a concept, is, but is not able to apply it.
- Usage: the student can concretely use a concept, for example in a program or when doing analysis.
- Assessment: the student can argue for the selection of a concept to use when solving a problem. This also requires the student to understand available alternatives.

These levels are inspired by Bloom's Taxonomy [3], which was revised by Anderson and Krathwohl et al. as Bloom's Revised Taxonomy [13]. They define 6 categories, which

can be seen as a hierarchy of understanding, going from remembering and understanding through application and analysis to evaluation and creation. There have been several attempts to adapt Bloom's Taxonomy for use in a computer science course [9, 14, 25]. The focus in these studies in mainly on the categorization of test questions, which cover a much wider range than just writing code. When looking at code, the main distinction seems to be between the 'apply' and 'create' levels, although a clear line is hard to find [25]. In the Curriculum Guidelines, this is distinguished in the 'usage' and 'assessment' levels.

Finally, the guidelines give an overview of the trade-offs made in the design in introductory courses on computer science. In general, they advise against a solemn focus on programming fundamentals, but otherwise, there are a lot of choices a course designer could make, based on a variety of circumstances. The main takeaway from this chapter in the guidelines is that introductory courses come in a wide variety, especially when diverse audiences are taken into account.

3. LEARNING OUTCOMES IN PROGRAMMING COURSES

To answer **RQ1**, we need to identify common learning outcomes in programming courses and which of these outcomes could assessable by an automated tool. As described in [24], learning outcomes can be split according to their specificity. This is a useful categorization because only learning outcomes that are concrete, observable and measurable, can be assessed by an automated tool. This means that for this research the focus will lie on learning objectives, not on goals or aims.

3.1 Learning outcomes in the Curriculum Guidelines

This analysis starts with a selection of the learning outcomes defined in the Computer Science curriculum guidelines [1]. These are used as a basis for several Computer Science and programming curricula, so they serve as a good starting point. The learning outcomes in the knowledge areas of Software Development Fundamentals (algorithms, fundamental concepts and data structures) and Programming Languages (object-oriented programming) are most relevant for introductory programming courses.

The learning outcomes in the guidelines are separated by three levels of mastery, as described in section 2.3. Of these, only the 'usage' and 'assessment' outcomes are relevant when looking at code. The 'familiarity' outcomes are an important part of understanding the course contents, but the code a student writes cannot display their familiarity with a for-loop if they are not also able to use it.

Next, the learning outcomes can be divided into ones that are directly related to writing code, and those that are about reading, explaining or analysing code or not related to code at all. As a first step, all outcomes at the level of 'familiarity' can be ruled out. On the level of 'usage', the relation to coding is indicated by keywords like 'write', implement' and 'apply', while the keywords 'identify' or 'reason' signal explanation or analysis of code.

On the level of 'assessment', the scope of the assessment plays a major role: a comparison of multiple approaches cannot be judged in a coding assignment, but whether or not the student chose the right approach can be. In general, outcomes with the keyword 'compare' or 'explain' are not directly related to coding, whereas 'determine', 'identify' or 'choose' indicate a decision that could be identified in code.

Finally, the specificity of these learning outcomes is important: they have to be concrete and observable, otherwise they can not be assessed. For the outcomes that are relevant to writing code, we assigned a label of 'aim', 'goal' or 'objective' guided by the descriptions and examples in [24]. The outcomes in the guidelines were generally found to be specific enough to be observed, so they were mostly classified as objectives.

Using this selection procedure, 15 out of 31 learning outcomes were found suitable for automated assessment. This means that automated assessment of learning objectives in code could be helpful, but can by no means stand on its own.

Note also that this includes none of the 'familiarity' level outcomes. This shows the importance of using other methods of assessment besides automated code analysis: students who struggle with the application of concepts, but do understand them well, have no possibility of demonstrating their knowledge in an automated system.

3.2 Learning outcomes in Creative Technology

Programming in Creative Technology is different from the classical Computer Science way: the focus is on letting students play with the code and create interactive programs, rather than learning algorithms for the sake of it [15]. This is also reflected in the official learning outcomes for programming courses in Creative Technology [26], which have a clear focus on the 'usage' level. (73% of the learning outcomes, against 48% in the Curriculum Guidelines.)

Out of the 11 learning outcomes across two modules, 7 are directly related to coding. Out of these 7, 3 are classified as goals. The 4 objectives related to coding are concrete in the actions the student can take, but the related concepts are still kept vague (e.g. "determine an algorithm for a given problem", "understand and apply guidelines of code quality"). To be assessable, these should first be worked out into observable objectives that concretely specify the content they cover.

3.3 Selection of learning objectives

Drawing inspiration from the curriculum guidelines [1], the official Creative Technology learning outcomes [26] and course materials used in Creative Technology [20, 21], we defined five learning objectives to be further analysed and used in the implementation of Apollo.

- Write a program that uses graphical commands to draw to the screen.
- Usage: Write a program that uses looping constructs for repetition.

  Assessment: Choose the appropriate looping construct for a given task.
- Compose a program using classes, objects and methods to structure the code in an object-oriented way.
- Implement message passing to enable communication between classes in a complex program, instead of using global variables.
- Use elementary vector operations to simulate physical forces on an object.

The objectives are selected to give a fair representation of different types of learning objectives and different degrees of freedom in the resulting code. All are relevant to first year courses of Creative Technology, in which students are introduced to programming using Processing, a language for visual art and new media. These five topics are a

selection of what students are expected to create with Processing throughout the first year of the study.

4. RECOGNIZING LEARNING OBJECTIVES IN CODE

Based on the five learning objectives chosen after answering **RQ1**, **RQ2** can be answered: how can achievement of these learning objectives be recognized in code? For each of these learning objectives, we will describe some of the goals and plans that a tool could use to recognize these objectives and the different aspects of these objectives that indicate that the student has mastered the objective.

4.1 Graphical commands

Write a program that uses graphical commands to draw to the screen.

For the use of graphical commands, the goal is relatively simple: "Draw something on the screen." The plan is similarly simple: use one of the many drawing methods built into Processing.

This is easy to detect, but the usage of one or two drawing methods is not convincing evidence for the mastery of graphical commands. On the other hand, students who use the Transform functions like `pushMatrix` are likely to have a good understanding of the drawing methods. Students who use methods that were not covered in the course could also be considered to have a higher chance of understanding, going beyond what was explicitly taught.

Aspects that indicate mastery:

- Use of a variety of different drawing methods covered in the course material
- Use of advanced drawing methods, like those in the Transform category
- Use of methods that are not explicitly part of the course material

4.2 Looping constructs

Write a program that uses looping constructs for repetition.

In the case of looping constructs, there is a wider variety of goals and applicable plans, for example:

1. Repeat some code as long as a certain condition holds. For this generic case, a while-loop is the only valid plan.
2. Repeat n times, increasing a counter. This can be done with either a for-loop or a while-loop, but using a for-loop is the preferred solution.
3. Iterate over all items in an array. Similar to goal 2, there are multiple valid plans: a foreach-, for-, or while-loop could all be used in this case. The foreach-loop, however, is preferred over the others.
4. Iterate over all items in an array, also using the index independently. While the goal is similar to goal 3, the foreach-loop cannot be used, because access to the index of the item in the array is required. See figure 1 for an example.

```
for (int i = 0; i < ts.length; i++) {
  Thing t = ts[i]; // Get an element
  s += i * t.getValue(); // Use index directly
}
```

Figure 1. Example of an array iteration that requires the use of an index

Aspects that indicate mastery:

- Use of different types of loops
- Use of loops in a variety of situations, e.g. for looping over an array, but also for simple repetitions

4.2.1 *The assessment case*

A common pattern in Computer Science is to have learning outcome on the 'usage' level that specifies multiple ways of doing something and an accompanying objective on the 'assessment' level that requires a student to choose the best option in a situation. In the case of looping constructs: *choose the appropriate looping construct for a given task.*

Mastery of this outcome requires a mastery of the 'usage' level and:

- Use of the most specific appropriate looping construct

4.3 Object-oriented structure

Compose a program using classes, objects and methods to structure the code in an object-oriented way.

There can be many high-level goals that lead the student to create a class, but it generally comes down to the goal of structuring the program: grouping related data and actions together. The related plan is that of a class definition.

While having class definitions in your program is an indicator that you have some understanding of object-oriented programming, a student who writes a program with hundreds of empty class definitions is not very likely to master the learning objective. Fehnker and De Man [6] suggest an object-oriented structure that should be used for interactive Processing applications and also provide PMD rules for automated detection of common mistakes in this structure. If a program is structured into multiple classes and none of the common design smells is detected, the student has likely mastered this objective.

The aspects that indicate mastery of this objective:

- Use of classes with various methods
- Relatively few detected design smells in the code structure

4.4 Message passing

Implement message passing to enable communication between classes in a complex program.

This objective is related to the previous objective of object-oriented design, but included as a separate objective since it receives dedicated attention in the course materials and is not covered by the design smells in [6].

The goal related to this objective is sharing information between classes. Take the case of a rocket and its fire tail as an example. When the rocket moves, the tail should be informed of the new position of the rocket to draw itself in the correct position. There are multiple plans to achieve this:

1. Using a global variable that holds the rocket's position. This variable is declared in the global scope and used by both the `Rocket` class and the `FireTail` class to determine the position where they should be drawing. When the rocket moves, the `Rocket` class modifies this global variable.
2. Using method parameter passing. The `Rocket` and `FireTail` classes both store their positions in a local variable. When the rocket moves, the `Rocket` class modifies its local variable and calls a method on the `FireTail` class to communicate its new position.

The second option is preferred, because the information is not shared with other parts of the program that might inadvertently change its value, but only with the parts that

actually need the information.

Indicators for mastery of this learning objective:

- No use of global variables that are changed in one class and read in another
- Use of method parameters to pass information between classes

4.5 Simulating physics

Use elementary vector operations to simulate physical forces on an object.

Forces, acceleration, velocity and position are modelled in Processing using the `PVector` class, so physical formulas are usually translated into operations on these vectors. Shiffman [21] gives an example to apply a force to an object with a certain mass, which is shown in figure 2. This code represents the formula $\vec{F} = m\vec{a}$ when the force is applied to an object.

```
void applyForce(PVector force) {
  PVector f = force.get();
  acceleration.add(f.div(mass));
}
```

Figure 2. Function that applies a force to an object with mass

This is an example plan for a common goal when dealing with physics: "apply a force to an object with mass." Some other common goals in the course materials include: "calculating the drag force for an object in a medium", "model gravity using constant downward force" or "calculate friction".

If one or more plans related to known goals when working with physics are found, then that is a good indicator for mastery of this learning outcome:

- Use of a known plan for working with forces

There are, however, many more physical phenomena than the 6 that are covered in the course materials. Since creative students are encouraged to use their imagination and make up their own forces, a more general indicator for the use of physics is also required. In the very general case:

- Use of operations on `PVector`s

This is only a weak indicator because it doesn't necessarily mean the student is modelling physics; a `PVector` could just as well be used to do abstract linear algebra. If there are no `PVector`s used in the code, however, this does mean that the student is not using vector operations to do physical modelling and is thus not showing any mastery of the learning objective.

5. PROGRAMMATIC ASSESSMENT

In this section **RQ3** is answered, by designing an automated tool called Apollo that can programmatically assess the achievement of learning objectives, and by conducting a preliminary validation of its results. Due to time constraints, this validation takes place on a small scale. The contributions to the Atelier project will however be part of a study over a longer timeframe, conducted as part of that project.

Inspired by Corbett and Anderson [4], Apollo could be used to model the chance that a student has achieved a learning objective. In this context, learning objectives are assumed to follow a two-state model: a student has achieved the learning objective, or they didn't. To enable this, Apollo calculates the probability that a program contains convincing evidence for the mastery of a learning objective. This can then be used to update the probability that the student has mastered the learning objective using a probability updating rule.

Apollo uses static analysis to detect the relevant aspects and indicators, based on the characteristics of the learning objectives uncovered in answering **RQ2**. Since previous research regarding static analysis of Processing projects [2, 6] was successful in adapting PMD[1] to work with Processing, it was chosen to serve as the basis for Apollo.

5.1 Implementation

Since version 6, PMD includes a framework for defining code metrics [16], which can produce a numeric output based on the analysis of a class or method. Apollo implements every countable aspect of the learning objectives as such a metric. Unfortunately, PMD is not able to directly report these metrics to the caller, so for each learning objective, a ReportRule is defined that reports every metric as a rule "violation", such that the values can be read from PMD's output. Apollo listens for all reported violations and uses the metrics to calculate the final probability for each learning objective.

In general, many of the aspects rely on counting, for example, the number of calls to a certain method or the number of class definitions. A generic mapping function is used to compute the probability of having convincing evidence for each of these aspects, defined as:

$$S_{a,b}(n) = 1 - \frac{1}{\frac{1}{a}n^b + 1}$$

This function defines a family of "S"-shaped curves in the range $[0, 1\rangle$, where the slope is determined by the parameters a and b, which can be varied from aspect to aspect, and from course to course.

5.1.1 Graphical commands

For the implementation of recognizing graphical commands, Apollo uses a list of all graphical commands in Processing [17], grouped by category (e.g. "Shape / Curves" or "Color / Setting"). A visitor that runs over the abstract syntax tree of a program and visits all method calls, checks these calls against the set of defined drawing methods and keeps a list of the different methods that were used in the program. This list only keeps track of which drawing methods are used, not how often they are called, so in a program where all drawing is done using the `circle` method, this list will contain one entry for the `circle` method.

The different metrics are then calculated as follows:

1. Use of a variety of different drawing methods covered in the course material

   Filter the list of found drawing methods to those categories of methods covered in the course materials. The length of this list is the input for the function S, mapping the number of different drawing methods used to a probability.

2. Use of advanced drawing methods, like those in the Transform category

   Similar to the covered methods, but counting method calls in the Transform category of drawing methods.

3. Use of methods that are not explicitly part of the course material

[1] PMD is a popular tool for detecting code smells in Java, available at https://pmd.github.io

These are the methods that were not part of the count for the first metric. The length of this list is again mapped to a probability using the function S.

Finally, these probabilities need to be combined into the probability that the analysed program contains convincing evidence that the student can *write a program that uses graphical commands to draw to the screen.* This is done as a weighted average of the results for the three aspects, with the first aspect having weight 3, the second having weight 2 and the last having weight 1. Students that do not use the covered drawing functions sufficiently are unlikely to master the drawing commands, even if they do use advanced or non-covered methods.

5.1.2 Looping constructs

In the analysis of the learning objective for looping constructs, several different goals and related plans were uncovered. Instead of listing and matching on every plan related to using loops, Apollo tries to characterize loops based on the characteristics of the different plans in the analysis. This characterization differs for the three types of loops:

For a while-loop, there are two defining features: (1) the type of condition, either a relation (numeral or not), an equality (numeral or not) or some other condition; and (2) the variables used in the while loop, which have three boolean properties indicating if the variable (a) is used to dereference an array, (b) is being manipulated, or (c) is being manipulated using a constant value (e.g., a uniformly increasing counter).

In a for-loop, there are three features: (1) the variable used as the iterator; (2) the type of condition, either a relation with the length of an array, a different relation, an equality or some other condition; and (3) how the iterator is used in the loop body, with the same properties as the variables in a while-loop.

A foreach-loop always iterates over an array and has no other defining features.

The two metrics are then calculated as follows:

1. Use of different types of loops

   The number of different types of loops used in the program. So for a program that only uses for-loops, the result would be 1; for a program that uses both while- and foreach-loops, the result would be 2. This count is mapped to a probability using the function S.

2. Use of loops in a variety of situations, e.g. for looping over an array, but also for simple repetitions

   Using the characterization of all loops, map them to a triple that includes the type of loop, whether variables were used to dereference arrays and whether variables were manipulated. The number of situations loops are used in is determined to be the number of different triples found in the program. This count is again mapped to a probability using the function S.

The two metrics are averaged two calculate the probability that the program contains convincing evidence that the student is able to *write a program that uses looping constructs for repetition.*

5.1.2.1 The assessment case.

The probability that a program contains convincing evidence that a student can *choose the appropriate looping construct for a given task* is calculated as the ratio of loops that are the most appropriate in that situation over all used loops. In general, a loop is assumed to be the appropriate choice if there is not a more specific type of loop that can be used. The characterization of the 'usage' case is used to determine if a more specific loop would be applicable.

5.1.3 Object-oriented structure

The classes in a program are easy to count in PMD, using a visitor over the AST that increments a counter every time it finds encounters a class definition. To make sure only classes that have some logic in them are counted, the visitor takes an argument to specify the minimum number of methods a class needs to define before being counted. When counting classes in a Processing project, Apollo uses a minimum of two methods, following the structure suggested by Fehnker and De Man [6].

The metrics are calculated as follows:

1. Use of classes with various methods

   This is simply the number of classes counted by the visitor, converted to a probability using the function S.

2. Relatively few detected design smells in the code structure

   For this metric, Apollo runs the PMD rules defined in [6] to detect code smells in the program. Because the chance of a small mistake is bigger in a large program, the amount of smells is divided by the number of classes counted for the first metric.

   The function S is again used to convert this to a probability, but with a slight difference: normally the probability approaches 1 as the argument goes to infinity, but in this case, a higher number represents a lower chance. Using a negative value for the parameter b flips the function S around, starting at 1 and decreasing to approach 0.

The final probability that the program contains convincing evidence that the student is able to *compose a program using classes, objects and methods to structure the code in an object-oriented way*, is calculated as the average of the two metrics.

5.1.4 Message passing

To see if the student can apply message passing, Apollo has to detect the use of global variables and the passing of parameters between classes. To be able to do this, Apollo uses the more advanced scoping features of PMD to check where a variable or method is defined.

PMD can provide a map of declarations and all uses throughout the program when looking at a scope. To find all global variables that are used for message passing, Apollo goes over the global scope to find all globally declared variables and filters out all of the usages that are in the global scope themselves. Only variables that are mutated, either by being reassigned or by internal mutation through a method call, are used for message passing. With all global constants filtered out, Apollo counts the number of uses of these global variables across classes to count how often they are used for message passing.

The detection of parameter passing happens similarly: first, all methods are found by recursively accessing every nested scope. The usages of these methods are filtered to the calls from outside the defining class because only message passing between classes should be considered. To get a number similar to that for global variable use, Apollo looks at the arguments and only counts locally declared values as instances of parameter passing.

Instead of calculating a probability for each aspect and averaging those, the probability that the program contains convincing evidence that the student can *implement message passing to enable communication between classes in a complex program* is calculated as the ratio of parameter passing instances over the total amount of communication, using both parameter passing and global variables.

5.1.5 Simulating physics

Detecting physics simulation is the only learning objective that strongly depends on the detection of plans. Whereas the other objectives could all be simplified using a characterization of the correct solution, physics simulation requires very specific plans to work correctly. To keep things manageable, Apollo only considers operations on `PVector`s when comparing physics plans and simply ignores the calculation of other parts of the solution. Apollo recognizes five plans for simulating physics, which are defined as a list of method calls on different `PVector` instances. See figure 3 for an example.

```
new PhysicsPlan("Apply acceleration",
  new PVectorMethod("velocity", "add",
                    new PVector("acceleration")),
  new PVectorMethod("position", "add",
                    new PVector("velocity")))
```

Figure 3. Example definition of a physics plan

First, all `PVector` operations in a program are identified in a way similar to the detection of global variables, except that this time all scopes are searched recursively. The operations are grouped based on the lowest block they are in, which can be a method body, a loop body etc. A match between a block and a plan is found by looping over both and comparing the statements in them. One limitation, due to the limited type-inference available in PMD, is that all `PVector`s on which operations are done have to be declared as variables. If an operation is done directly on a new object instantiation or a function parameter, it will not be available in the scope.

The two metrics are calculated as follows:

1. Use of a known plan for working with forces

   Apollo counts the number of times one of the five specified physics plans matches with the code, as explained before. The number of matches is converted to a probability using the S function.

2. Use of operations on `PVector`s

   Simply count the number of operations based on the declared `PVector`s across all scopes and use the function S to convert the number to a probability.

The final probability that the program contains convincing evidence that the student can *use elementary vector operations to simulate physical forces on an object*, is calculated as the weighted average of both metrics. The first metric has weight 1, the second 2.5. This means that even when no plans are recognized, a chance of 0.7 can still be reached based on the number of `PVector` operations, which is desired because of the mentioned limitations on plan detection.

5.2 Calibration and validation

For an initial validation and determining the parameters for converting counted metrics into probabilities, Apollo was used on an old dataset of final student projects used in the evaluation of [6]. While the full analysis of each aspect of the five learning outcomes would be too lengthy to repeat here, an example of the approach can be discussed using the example of counting the use of drawing methods covered in the course materials.

All programs used between 4 and 17 different drawing methods, with a median of 9. The first quartile is at 8 drawing methods, the third quartile at 11. See figure 4.

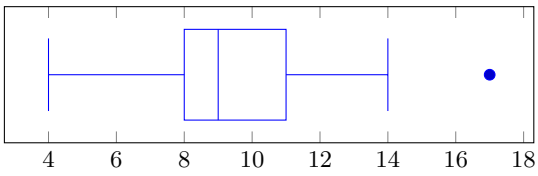

Figure 4. Number of drawing methods used in final student projects

To determine the correct parameters for the function S, these statistics were mapped to the desired chance. The first quartile is mapped to 50%, the median to 70% and the third quartile to 95%. This means that a chance of understanding above 95% is assigned to the 25% best programs, above 70% to the 50% best programs etc.

For the covered drawing methods this means that 8 used drawing methods maps to a 50% chance of being convincing evidence, 9 used maps to 70% and 11 used to 95%. Using a standard curve fitting algorithm the parameters for the function S were determined to be $a = 27748375.73$ and $b = 8.22$.

For most of the learning objectives, the returned statistics looked realistic and could be used to determine the required parameters. Since the dataset originates from the first module of programming in Creative Technology in which the students were not yet introduced to simulating physics, no useful data could be gathered for that objective.

5.2.1 Integration with Atelier

To test Apollo in practice, the tool was integrated with Atelier. Atelier is used during programming tutorials for students to upload their code and discuss it with teaching assistants. Apollo created comments with its assessment on every submission. To reduce the chance that teaching assistants would interpret Apollo's assessments as grading, the probabilities calculated by Apollo were translated into words, following a mapping defined in [18]. A chance of 20% or less is rendered as 'improbable', for example.

Teaching assistants were asked to reply with "+1" if they think the assessment by Apollo is correct, or "-1" if they think it isn't. They were also asked to explain their comment if they had time to do so. See figure 5 for an example.

Integrated with Atelier, Apollo ran on 65 student submissions in the final week of the tutorial period where students could ask for feedback on their projects. On 11 of these, a teacher or teaching assistant indicated whether or not they agreed with Apollo's assessment. Out of these 11, 8 responses included "+1" to indicate agreement with Apollo and 3 responses included neither "+1" nor "-1". Message passing and physics were both mentioned three times in these comments. In the case of message passing, always because Apollo indicated a low score on these programs and the commenter mentioning they agree with that assessment. With physics this was the case twice, and once because there was some physics in the program that did not seem to be picked up by Apollo.

The low response rate can at least partially be attributed to the fact that Apollo was only deployed in the last week of tutorials when many students were asking questions

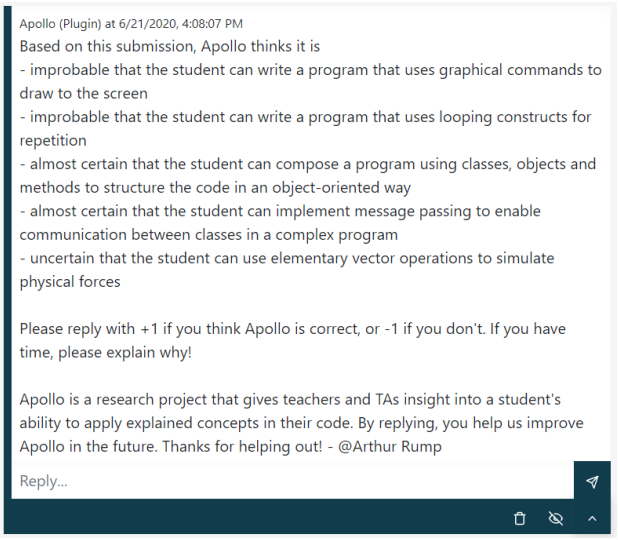

Figure 5. An example comment by Apollo in Atelier

about their final projects. Teaching assistants rightly prioritise helping as many students as possible over answering questions in the limited time they have during a tutorial.

In separate interviews with two teaching assistants, they both indicated to agree with the comments made by Apollo most of the time. One of them indicated that they would like to get more information on why Apollo made the assessment, the other mentioned that the overall information presentation in Atelier could be better and that it sometimes feels "rather spammy." Both did find the provided information useful when looking at a project uploaded by a student.

6. CONCLUSIONS

The main question in this paper is concerned with **how the level of a student's mastery of a programming concept can be assessed by automatically analysing their code?** To conclude with a final answer to this question, the three research questions leading up to it need to be answered first.

RQ1 What are typical learning outcomes for introductory programming courses and which of these are suitable for automated assessment based on code analysis?

In section 3, we looked at learning outcomes in the Computer Science Curriculum Guidelines and compared them with learning outcomes for programming courses in Creative Technology. In the Curriculum Guidelines, 15 out of 31 learning outcomes were found to be directly related to writing code and specific enough to be assessable with automated means. The learning outcomes for Creative Technology were generally too vague or dependent on knowing the content of the course to concretely implement in an automated tool. Based on these findings, a set of five representative learning objectives was created to use for the development of Apollo.

RQ2 How can the achievement of learning outcomes be recognized in code?

In section 4 the five learning objectives were dissected to find how achievement of these objectives could be recognized in code. Soloway's concept of goals and plans [22] served as a useful basis for working this out. For each learning objective, different aspects of the code were identified that would indicate that a student has achieved this objective.

RQ3 How can an automated tool assess the level of mastery of a concept by analysing code?

In section 5 the implementation of Apollo was described, showing the different approaches taken for each learning objective. Interesting to note here is the difference between characterizing the plans matching a learning objective, or trying to match a list of known plans. While the first is easier to implement and gives more consistent results, the second is easier to generalize to any learning objective for which goals and plans can be worked out.

While validation of the tool was limited due to time constraints, feedback from teaching assistants does indicate that Apollo is a useful addition in Atelier, even though the presentation of all information is not optimal. Early signs also tend to indicate that Apollo's assessment is mostly correct, but further validation is required to make a clear statement on this. This will happen as part of a longer term study in the Atelier project.

Returning to the main question of this paper, we can conclude that it is possible to assess a student's mastery of a programming concept by automatically analysing their code, but also that it requires a thorough specification of what the student should be able to do at the end of a course. It is also worth pointing out that not all learning outcomes can be assessed this way, so automated code analysis will not be able to replace all or even most of the other methods of assessment used during a course.

6.1 Further research

The main goal of this paper is to lay the foundations of a system that could 'gauge the room' in a digital tutorial setting and provide teaching staff with a global overview of students' progress towards the desired learning outcomes. In its current state, Apollo is able to assess individual programs, but it cannot yet translate this to the chance that a student has achieved a learning objective, nor combine the results to provide this overview for a group of students. Further research should look into ways to use Apollo's output towards this goal. A potential way to do this could be a probabilistic update rule like probability kinematics [8], which can be used to update the chance that a student has mastered the learning objective given the probability of convincing evidence reported by Apollo.

Related to this is the presentation of Apollo's results. The current textual representation was mainly chosen due to limitations of the Atelier platform, but it would be interesting to look into different ways to present the results to users. This becomes especially important when the results are combined into student- and group-level numbers, possibly tracked over time. Further research is needed to determine the most intuitive way to present this data.

Finally, further research could be done into showing Apollo's results directly to students. While Apollo is currently intended to be used by teaching staff, it could be useful for students to know how they are progressing through the course and in which areas they might need more practice. Sadler [19] describes three requirements for providing formative feedback to students: the student has to know (1) the standard they should aim for, (2) how their actual performance compares to the standard, and (3) what action they can take to close the gap between the desired and actual performance. Apollo can fulfil the second of these requirements, but further research is needed on if and how an automated tool could satisfy the other two criteria.

7. REFERENCES

[1] ACM Computing Curricula Task Force ed. 2013. *Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science*. ACM, Inc.

[2] Blok, T. and Fehnker, A. 2017. Automated program analysis for novice programmers. *Proceedings of the 3rd international conference on higher education advances* (Jun. 2017).

[3] Bloom, B.S., Englehart, M.D., Furst, E.J., Hill, W.H. and Krathwohl, D.R. 1956. *Taxonomy of educational objectives. The classification of educational goals: Handbook 1: Cognitive domain*. David McKay Co., Inc.

[4] Corbett, A.T. and Anderson, J.R. 1995. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modelling and User-Adapted Interaction*. 4, 4 (1995), 253–278. DOI:https://doi.org/10.1007/bf01099821.

[5] Douce, C., Livingstone, D. and Orwell, J. 2005. Automatic test-based assessment of programming. *Journal on Educational Resources in Computing*. 5, 3 (Sep. 2005), 4–es. DOI:https://doi.org/10.1145/1163405.1163409.

[6] Fehnker, A. and de Man, R. 2019. Detecting and addressing design smells in novice processing programs. *Communications in computer and information science*. Springer International Publishing. 507–531.

[7] Fehnker, A. and Mader, A. Atelier for creative programming: Stimuleringsregeling open en online onderwijs.

[8] Jeffrey, R. 1987. Alias Smith and Jones: The testimony of the senses. *Erkenntnis*. 26, (1987), 391–399. DOI:https://doi.org/10.1007/BF00167725.

[9] Johnson, C.G. and Fuller, U. 2006. Is Bloom's taxonomy appropriate for computer science? *Proceedings of the 6th baltic sea conference on computing education research koli calling 2006 - baltic sea '06* (2006).

[10] Johnson, W.L. and Soloway, E. 1984. Intention-based diagnosis of programming errors. *Proceedings of the 5th national conference on artificial intelligence, austin, tx* (1984), 162–168.

[11] Johnson, W.L. and Soloway, E. 1985. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*. SE-11, 3 (Mar. 1985), 267–275. DOI:https://doi.org/10.1109/tse.1985.232210.

[12] Keuning, H., Jeuring, J. and Heeren, B. 2019. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*. 19, 1 (Jan. 2019), 1–43. DOI:https://doi.org/10.1145/3231711.

[13] Krathwohl, D.R. 2002. A revision of Bloom's taxonomy: An overview. *Theory Into Practice*. 41, 4 (Nov. 2002), 212–218. DOI:https://doi.org/10.1207/s15430421tip4104_2.

[14] Lister, R. and Leaney, J. 2003. Introductory programming, criterion-referencing, and Bloom. *Proceedings of the 34th sigcse technical symposium on computer science education - sigcse '03* (2003).

[15] Mader, A., Fehnker, A. and Dertien, E. 2020. Tinkering in informatics as teaching method. *Proceedings of the 12th international conference on computer supported education* (2020).

[16] PMD - using and defining code metrics for custom rules: 2017. *https://pmd.github.io/pmd-6.24.0/pmd_userdocs_extending_metrics_howto.html*. Accessed: 2020-05-28.

[17] Processing language reference (API): *https://processing.org/reference/*. Accessed: 2020-05-27.

[18] Renooij, S. and Witteman, C. 1999. Talking probabilities: Communicating probabilistic information with words and numbers. *International Journal of Approximate Reasoning*. 22, 3 (Dec. 1999), 169–194. DOI:https://doi.org/10.1016/s0888-613x(99)00027-4.

[19] Sadler, D.R. 1989. Formative assessment and the design of instructional systems. *Instructional Science*. 18, 2 (Jun. 1989), 119–144. DOI:https://doi.org/10.1007/bf00117714.

[20] Shiffman, D. 2015. *Learning processing: A beginner's guide to programming images, animation, and interaction*. Elsevier Science.

[21] Shiffman, D. 2012. *The nature of code*.

[22] Soloway, E. 1986. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*. 29, 9 (Sep. 1986), 850–858. DOI:https://doi.org/10.1145/6592.6594.

[23] Suskie, L. 2018. *Assessing student learning: A common sense guide*. Wiley.

[24] The aims, goals and objectives of curriculum - what are the differences? 2014. *https://thesecondprinciple.com/instructional-design/writing-curriculum/*. Accessed: 2020-05-07.

[25] Thompson, E., Luxton-Reilly, A., Whalley, J.L., Hu, M. and Robbins, P. 2008. Bloom's taxonomy for CS assessment. *Conferences in Research and Practice in Information Technology Series*. 78, (2008), 155–161.

[26] University of Twente: OSIRIS - course offerings: *https://osiris.utwente.nl/student/OnderwijsCatalogusZoekCursus.do*. Accessed: 2020-04-30.

[27] 2015. *Report on the bachelor's programme Creative Technology of the University of Twente*. Quality Assurance Netherlands Universities (QANU).