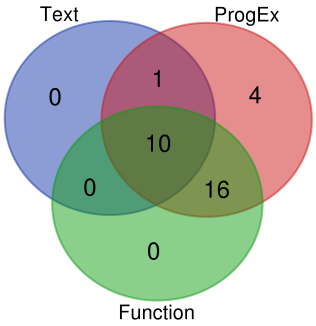

Figure 5: Venn diagram showing coverage of Block Model dimensions by teachers.

MP and RP are more frequently addressed by secondary teachers than they are by tertiary instructors, whereas AP is less.

We suggest that the final goals of achieving *ProgComp* are related to the cells found on the upper right corner of the Block Model. This view is shared by the teachers as the MF category (understanding the goal/purpose of the program) is the most frequently cited.

Interestingly, all interviewees have at least one reference to Program execution (P) as shown in Figure 5, most commonly in combination with Function (F) (51%). Note that one third of them talk about *ProgComp* in terms that matched all 3 dimensions (T/P/F), while none referred only to text surface (T).

As mentioned above, this is probably due to practitioners' experiences on the difficulty and hence importance of different aspects in teaching. In the following section we present different learning activities, obtained — in part — in the interviews, and then in section 6.3 present some qualitative results of the interviews with regard to difficulties and hints for possible learning trajectories.

5 COLLECTION AND CLASSIFICATION OF PROGRAM COMPREHENSION TASKS

Based on literature analysis, discussions within the working group, and examples of activities provided by our interview participants, we collected and categorised several types of activities that are intended to help students develop *ProgComp*.

The tasks listed in this section also include some common types of tasks analyzed in Section 3: tracing tasks (AT, AP, RP), a Parsons problem (BP), "explain in your own words" tasks (BF, RF, MF).

5.1 Methodology

We categorized *ProgComp* tasks using the Block Model framework. To implement this, we analysed each of the available tasks considering both the level of complexity upon which it focuses (atoms, blocks, relational, macro) and what dimension of the program it concerns (Text Surface, i.e. syntax; Program Execution, i.e. notional machine; Functional, i.e. function, purpose or intent of the code).

This approach allowed us to enrich the list of available *ProgComp* tasks. Indeed, during the analysis, we found that some parts of the Block Model matrix were not covered by any task, and this led us to devise new types of task with the potential to fill those gaps.

We traverse the Block Model column by column, presenting first task types that pertain to the Text Surface, then the Program when executed and finally the Function of the program. For each column we group the task types starting from the atom level upwards. These types of task then need to be further specified by applying them to a particular code fragment. An example of this is presented in Figure 6.

5.2 Text Surface Tasks

The text dimension of the Block Model is based on the perceivable representation of a program. In terms of the comprehension process, reading and comprehending starts by perceiving, which implies identifying and discriminating between atomic elements in the text, then recognising their organisation into language structures of growing complexity, culminating in the overall program structure.

The types of tasks in this category are then focused on *statically detectable properties*, i.e. syntax as well as static typing. Even though we restrict our attention to program notational features, the inherent complexities of language constructs and dependencies may be overwhelming to students, as demonstrated by Luxton-Reilly et al. in their in-depth analysis [65]. Luxton-Reilly and colleagues also provide valuable suggestions as to how to decompose a complex task (in a novice's perspective) into more focused components.

Other types of tasks considered in the literature to assess or develop novices' understanding of the static properties of programs include, in particular:

- Tasks requiring fixing compile-time errors introduced within the code in order to test students' ability to identify the actual sources of the problems [55].
- *Fill-in-the-gap* (e.g. choosing the right keyword) and *highlighting* (e.g. identifying the occurrences of a syntactic concept) tasks to test students' basic competency in terms of language syntax [30, 52].
- Parsons-like puzzles involving only the language notation and tasks requiring to translate an accurate formal definition into code [74].

To be more concrete, we list several specific task examples, either drawn from the literature or suggested by working group members and participants during the interviews. The examples are classified according to the rows of the Block Model:

Atom–Text (AT).

- Identify the keywords in a piece of code;
- Box all the assignment statements;
- List all integer variables;
- Box all arithmetic expressions (arithmetic expressions can be recognised from purely syntactic items);
- Box the headers of all methods/procedures/functions;
- Transform between alternative syntactic forms of atomic elements (e.g. from `i++` to `i=i+1`).

Block–Text (BT).

- Draw a box around the code of each conditional construct;
- Draw a box around the code of each loop;
- Box the body of each method/procedure/function;

- Check if the parentheses are placed correctly;
- Draw nested boxes to represent the structure of a complex expression.

Relational–Text (RT).

- Link each occurrence of a variable with its declaration;
- Identify the scope of a variable (assuming static binding);
- Identify where a particular function is called;
- Verify if all expressions are correctly typed;
- Verify if every potential flow path of a function's body ends with a return statement;
- Draw a box around the initialization/termination/increment expression of a `for` loop (relational for novices first learning about loop control).

Macro Structure–Text (MT).

- Represent the overall program structure by drawing a "block-nesting" tree;
- Restructure a program's code so that library links are at the top, followed by the definition of global variables and functions/methods, followed by the main program;
- Describe the overall program block structure by drawing nested boxes;
- Draw a diagram showing the overall program structure;
- Represent the overall program structure by drawing a tree of function/procedure dependencies (relative to invocations);

5.3 Program Execution Tasks

In order to deal with the dynamic aspects of execution, the information provided by the program text is not sufficient, but must be supplemented with a conceptualization of machine *state*, establishing the context(s) in which the program is in action. Thus, the program dimension of the Block Model focuses on code execution, or, in technical terms, the *operational semantics* of a program.

At the heart of any characterisation of the program dimension lies the construction of a viable mental model of the *notional machine* [31]. In this respect, Sorva [112] presents a comprehensive review of research threads "that have contributed to our understanding of the challenges that novice programmers face when learning about the runtime dynamics of programs and the role of the computer in program execution". When engaging with the task of tracing the execution of some piece of code, "sketching" is a common practice for students in order to overcome the working-memory load which would be implied by following a long progression of actions and states in their mind [22, 23, 30, 133].

Several types of tasks focus on scaffolding the learning. Here is a list of those most frequently encountered in the literature:

- Tasklets that focus on "atomic" aspects of the operational semantics, by taking a "reductionist" approach to novices' understanding and learning of programming [65].
- Tracing, predicting and "fill-in-the-gaps" (within code) tasks designed to assess novices' program comprehension [60, 75].
- *Proglets*, i.e. little programs aimed at reducing the learners' cognitive load by exploring a single programming concept [37], when used as the basis of tasks requiring to predict the program outcome, to modify the code, or simply to experiment freely with it.
- *Parsons programming puzzles* focusing on the understanding of the notional machine [28, 45, 82].
- Tasks requiring to trace recursive computations, [43, 74, 99, 102].
- Tasks requiring either to verify reversibility or to write reversing code [47, 59, 76, 116].

At a fine-grain level, also considering the suggestions emerging from the working group and the interviews, we can classify a range of examples in terms of the rows of the Block Model:

Atom–Program (AP).

- Trace the program execution for some given input data, where the program does not include procedural units (note that this task can be accomplished at the atom level, as a sequence of several atomic steps, each next step being determined by the previous one);
- Determine the program output (e.g. what is printed) for given input data, again where the program does not include procedural units;
- Determine the value of an expression for given values of the involved variables;
- Trace a particular sequence of statements for given values of the involved variables.

Block–Program (BP).

- Determine the number of iterations of a loop construct for a given initial state (here recognising which repeated step is to be counted implies reasoning at block level — the repeated block; in particular, think of a nested conditional in a loop);
- Identify recurring instrumental blocks such as that for swapping the values of two variables (the assumption is that the identification is based on reasoning about the execution of short sequences of statements);
- Identify the block(s) implementing some specific program pattern, e.g. among those catalogued by [3] or [87];
- Solve a Parsons puzzle for a specific programming pattern.
- Change a *for* loop into a *while* loop.

Relational–Program (RP).

- Identify the variable(s) playing a specific role (in the example of Listing 1: *stepper*, *most recent holder*, *most wanted holder*, *walker*);
- Trace the program execution for a given input, where the program includes calls to procedural units (this task requires to establish connections between states relative to the caller's code and to the called procedural unit);
- Verify whether some branches of a switch/case statement are redundant, i.e. can never be executed;
- Identify states, i.e. values of one or more variables, that could result in an infinite loop;
- Identify the scope of a variable.

Macro Structure–Program (MP).

- Verify if a program statement or block is ever reachable during program execution;
- Identify a comprehensive set of inputs to check *all possible* computation flows of a program;

- Select from given options the program that is computationally equivalent to a reference one, i.e. which gives rise to the same sequence of variable states for every admissible input data;
- Explain why two given programs are not computationally equivalent;
- Estimate the computational costs of the program.

5.4 Function or Purpose Tasks

Relative to the function dimension of the Block Model, a new context, introducing properties *extrinsic* to the program at hand, comes into play.

Drawing a borderline between (abstraction on) code execution features and purpose-driven features is not always straightforward, and it is likely to depend to a large extent on the knowledge assumed at a certain learning stage.

However, well-developed tasks exploring this dimension of program comprehension are more difficult to envisage. As pointed out by Begum and colleagues [5], "[v]ery little research has investigated the behavior of programmers from understanding the problem specification to computer program".

Among the tasks considered in the literature, in which novices are required to understand the program in connection with an extrinsic problem domain we can mention the following:

- Tasks asking to explain in words[3] [61, 75, 129] the purpose of a program.
- "Fill-in-the-gaps" tasks designed to assess novices' understanding of the relationships between a program and the problem being solved [60].
- *Parsons puzzles* focused on the problem to solve [28, 45, 82].
- Tasks requiring to choose more meaningful names for program functional units, or to chunk code segments and define semantically meaningful functions [74].

In more detail, again by integrating suggestions coming from the working group as well as the interview participants:

Atom–Function (AF).

- Identify the purpose of an expression or a simple statement, in connection with the problem domain (e.g. of an expression/assignment for converting Fahrenheit to Celsius)
- Identify the purpose of a condition w.r.t. the problem domain (e.g. divisibility for some positive integer);
- Rename a constant with an appropriate name from the problem.

Block–Function (BF).

- Choose an appropriate name for a simple procedural unit (method, procedure or function, where the unit body consists of a simple block);
- Summarise in a short sentence what the block goal is;
- Identify the program block(s) with a given function, described in problem-domain terms;
- Write comments explaining the purpose of a block and of the statements it is built from.

[3] called in the literature "Explain in plain English" but students may use their native language instead

Relational–Function (RF).

- Choose an appropriate name for a variable (usually the function of a variable can be inferred by establishing relationships between different occurrences of it);
- Summarise in a short sentence the purpose of a simple block invoking one or more methods/procedures/functions;
- Solve a Parsons puzzle for a given code purpose by reordering simple blocks (it requires to identify the sub-purpose of each block and their relationships)
- Identify functionally equivalent blocks, i.e. blocks giving rise to the same overall state transformation (selection from a few predefined options).

Macro Structure–Function (MF).

- Choose an appropriate name for a program;
- Summarise in a short sentence what the program goal is;
- Select the sentence, from a few options, which most accurately summarises the program's purpose;
- Create meaningful test cases for the allowed inputs and expected outputs (test cases are usually based on the program's purpose).

5.5 Towards a repository of Learning Activities

Due to time constraints, we were not able to set up a prototype online repository for the collected tasks. However, it is a long-term goal to either create or join an open-source "live" repository where practitioners/teachers as well as researchers in the field of computer science education can find and contribute *ProgComp* resources. With this goal in mind, we have designed a template to be attached to each submitted *ProgComp* activity, which provides context and supports its use.

The template incorporates the following fields: the coding that maps the activity into a Block Model cell; CS as well as *ProgComp* pre-requisites; materials provided by instructor; instructions for students; the new things that students will learn from this activity; how the activity can be designed as an individual or a team-based activity; and the perceived engagement as in the ICAP model [16].

To validate the template, which is included in appendix C, a subgroup filled a template form for four different types of activities: (a) identifying the role/purpose of variables, (b) commenting selected/key lines of code or code snippets, (c) tracing, and (d) debugging (finding and fixing an error).

It is however to be noticed that discussion within the WG has only addressed two of the six challenges pointed out in [39] in connection with the design of similar online repositories, namely *content* and (partly) *catalogue*. If, on the one hand, the focus of our intended content is per se narrow enough to limit its impact upon other concomitant features and the organisation in terms of Block Model provides helpful keys to access such content, on the other hand several crucial aspects are still to be considered. These include the other four challenges: *curation* (how to maintain the repository), *contribution* (categories of contributors, e.g. those who generate content and those who comment on or rate it, and related incentive and rewards), *community* (how to enhance recognition), and *control* (how to ensure quality and reliability).

In this respect, Fincher et al. [39] illustrate two viable models to cope with the endeavour of running a content repository.

6 MOVING FROM SINGLE TASKS TO LEARNING TRAJECTORIES

Learning trajectories (LT) have garnered the attention of math and science educators [62] because of their ability to model how student's thinking about a specific topic evolves, which supports research-based curriculum development [101]. Such research-based curriculum development has taken place, for example, in the mathematics education community [19].

However, empirical knowledge about LT is largely absent in computer science education. One reason is that there is no established methodology to systematize and define learners' progression in CS disciplines. Some recent studies attempted to extract data from the literature to create learning trajectories for sequence, conditionals, and repetition [91]; abstraction [89] and debugging [90]. These LT provide a path for particular aspects for programming and comprehension, but to the best of our knowledge, there is no learning trajectory for *ProgComp* as an integrative skill.

Complex tasks such as program comprehension favor the holistic integration of knowledge, skills, and attitudes to avoid fragmentation, promote meaningful schema creation and facilitate near transfer [124]. To achieve such goals, frameworks such as the Four Components Instructional Design (4C/ID [123]) provide a blueprint to sequence learning tasks in a trajectory from less to more complex tasks, promoting schema creation while not exceeding learner's cognitive load capacity. It also presents a guide of how to provide scaffolding to learners, decreasing support so learners can gradually accommodate increasingly complex schemas and at the end of instruction be able to perform in an authentic complex environment.

Our work with LT is inspired by such instructional design frameworks and in the following sections, we outline how to sequence tasks based on their difficulty and with decreasing levels of support for *ProgComp*.

This methodology could assist instructors in two ways. First, it provides practical examples for instructors on how to decompose a task that fosters *ProgComp* into sub-tasks that reduce the complexity with respect to the overall task, making it suitable for beginners, and later move to more advanced levels of complexity aimed to advanced learners, working on different aspects of *ProgComp*, as presented by the cells of the Block Model. Second, it provides a guideline that could help instructors identify where a specific task fits into the Block Model and what particular aspects of program comprehension are being fostered.

6.1 Methodology

Using the work of Lister and colleagues as a starting point, the "Leeds" ITiCSE working group Lister et al. [60] concluded that students lacked basic skills pre-requisite for problem-solving, such as comprehending program code. More recently, assessment tasks were found to be more complex than academics expected [65]. For example, tasks typically require both algorithmic thinking (such as initializing a variable before updating it), as well as a more advanced understanding of data representation (assigning a value to a property) [105]. In their research, Luxton-Reilly et al. [65] state that most assessments used in formal examinations combine numerous heterogeneous concepts, resulting in complex and difficult tasks. To develop tasks to determine a student's mastery of particular concepts, the Luxton-Reilly et al. 2017 ITiCSE working group decomposed complex assessments into atomic conceptual elements which can be assessed independently. Their work, which extends the McCracken et al. [70] research, shows that a single code-writing task often involves a plethora of conceptual knowledge.

Duran et al. [34] define these atomic elements as plans and subplans that can be extracted from concrete programs by analyzing the relationship of syntactic and semantic elements in the code and the respective cognitive actions leaners need to perform to comprehend the program. Our work uses Duran et al. model to provide cues on how to decompose a *ProgComp* task into sub-tasks and fit them in the Block Model. What becomes evident is that comprehending code too can be decomposed into multiple facets, each of which can be practiced independently.

The LT for *ProgComp* defines a spectrum of activities that will foster programming comprehension using as many cells in the Block Model as desired by the instructor. Different cells usually will use different activities (see section 5 for examples) that are better suited to achieve the desired learning outcome. Creating a LT is an iterative process where the instructor evaluate leaner's prior-knowledge in a particular context (e.g. using tests [81] or self-evaluation instruments [32, 33]) to identify a sub learning-outcome appropriate to learners' needs (extracted from the main task learning outcome), match this learning outcome to a given Block Model cell and use an appropriate activity to foster *ProgComp* at that cell. The process repeats until the instructor is satisfied with the granularity of the LT (the number of activities in different cells) or if the LT reaches the lowest level of complexity in the Block Model (the sub-task is already simple enough, e.g. uses an Atomic-Text-Surface activity) and no further refinements are required.

LT could be used by instructors in two different ways, depending on their goal. In the first one, the instructor iterates through task's learning outcomes and evaluate if students' needs, prior knowledge, and granularity will be addressed with a proposed spectrum of activities. If tasks are too easy or too difficult the instructor can further decompose the tasks until a saturation point is reached. In a second way, the instructor follows an already defined learning trajectory, using the planned tasks to identify gaps in the existing set of activities and integrate new activities where needed. The LT could also work in tandem with diagnosing tools, where learners' difficulties in a particular cell of the Block Model could be mitigated by using the appropriate activities.

In the next section, we provide an example of a walkthrough of the development of an LT where a task may seem too challenging for some learners or have some implicit assumptions that may not match the instructor's cohort.

6.2 Using the Block Model to Develop a Trajectory

As an example of LT, we take a typical comprehension task, to summarise the goal of a program in a short sentence (i.e. an "explain in plain English" task, as described in section 5), and show how it can be decomposed into subtasks, each fitting into one of the cells in the Block Model matrix.