Towards an Analysis of Program Complexity
From a Cognitive Perspective

Rodrigo Duran  
Aalto University, Finland  
rodrigo.duran@aalto.fi  

Juha Sorva  
Aalto University, Finland  
juha.sorva@aalto.fi  

Sofia Leite  
University of Porto, Portugal  
lpsi05114@fe.up.pt

ABSTRACT

Instructional designers, examiners, and researchers frequently need to assess the complexity of computer programs in their work. However, there is a dearth of established methodologies for assessing the complexity of a program from a learning point of view. In this article, we explore theories and methods for describing programs in terms of the demands they place on human cognition. More specifically, we draw on Cognitive Load Theory and the Model of Hierarchical Complexity in order to extend Soloway’s plan-based analysis of programs and apply it at a fine level of granularity. The resulting framework of Cognitive Complexity of Computer Programs (CCCP) generates metrics for two aspects of a program: plan depth and maximal plan interactivity. Plan depth reflects the overall complexity of the cognitive schemas that are required for reasoning about the program, and maximal plan interactivity reflects the complexity of interactions between schemas that arise from program composition. Using a number of short programs as case studies, we apply the CCCP to illustrate why one program or construct is more complex than another, to identify dependencies between constructs that a novice programmer needs to learn and to contrast the complexity of different strategies for program composition. Finally, we highlight some areas in computing education and computing education research in which the CCCP could be applied and discuss the upcoming work to validate and refine the CCCP and associated methodology beyond this initial exploration.

CCS CONCEPTS

• Social and professional topics → Computer science education; Model curricula; Student assessment;

KEYWORDS

Model of Hierarchical Complexity; Cognitive Load Theory; Program Cognitive Complexity; Complexity; Plan-Composition Strategies

1 INTRODUCTION

For instructional design to be successful, the designer must consider the trajectories that students move along as they learn. Ideally, learners engage in activities that are neither too hard nor too easy for them; with growing expertise, each learner can eventually tackle increasingly complex tasks. This goal was famously captured by Vygotsky in the concept of zone of proximal development, which continues to inspire developments in instructional design [e.g., 68].

Teachers routinely assess the complexity of the tasks they give to their students and seek to sequence those tasks in an effective manner. As a part of this effort, a programming teacher assesses the complexity of the programs that feature in examples and assessments and the programs they expect their students to write.

So, how does one tell how complex a program is, or whether one program is more complex than another? Typically, the teacher draws on their intuition and experience to make an informal assessment. If the teacher’s luck aligns with their ability, it works.

To assist teachers in instructional design, research in educational psychology has produced frameworks such as 4C/ID [84] that sequence classes of activities by increasing complexity. Establishing a theoretically motivated, empirically sustainable, and pedagogically feasible methodology for sequencing topics by complexity is one of the major goals in curriculum development [64]. Within computing education research (CER), scholars have explored the relationships between programming concepts and suggested a number of learning trajectories by complexity [e.g., 35, 55, 62]. However, there exists no well-established methodology for evaluating complexity or tracing such trajectories in programming.

In this article, we set our sights on a more solid theoretical and methodological footing for assessing the complexity of programs. We focus on a facet of complexity that can be extracted, with the help of theory, from concrete programs: the cognitive structures that are required to mentally manipulate a program as one studies or writes it. Recognizing (like [46]) that programming requires the programmer to think about low-level elements of code (the trees) as well as how those elements combine to achieve a higher-level purpose (the forest), we seek a model that attends to both aspects.

Our primary contribution is to suggest a theoretical framework for reasoning about the complexity of computer programs. The framework is meant for analyzing the cognitive schemas present in the design of a given program and the way those schemas are intertwined. In addition to providing a more nuanced analysis, our framework can be used to generate two numerical metrics that summarize program complexity; unlike the more technical metrics from software engineering and algorithm analysis, our metrics reflect a cognitive perspective which is meaningful for instructional design and CER and which can be applied to the sort of short programs that are common in introductory-level programming. To illustrate the
application of our model, and as a preliminary proof-of-concept evaluation of it, we discuss three programs as case studies.

Our framework derives from two main sources. The first is schema theory and the related Cognitive Load Theory (CLT) [2, 10, 80], which are concerned with the limitations of working memory and the growth of expertise as schemas in long-term memory; here we extend the earlier work in CER by Soloway, Rist, and others who have analyzed schemas as reflected in programs [63, 73, 77]. Our other main influence is the Model of Hierarchical Complexity (MHC) [16], a neo-Piagetian theory concerned with the relative complexities of tasks; to the best of our knowledge, the MHC has not been previously applied in computing education.

Section 2, below, explains the theoretical background. Section 3 then presents our framework for analyzing programs as well as the case studies that illustrate it. In Section 4, we discuss how the present work relates to, and differs from, earlier efforts in CER, and consider its applications of our framework. Finally, in Section 5, we review the contributions and limitations of this article and consider the future work of empirically validating our theoretical model.

2 THEORETICAL BACKGROUND

2.1 Schemas and Cognitive Load

Cognitive Load Theory (CLT) [2, 10, 68, 80] is a framework for investigating the effects of the human cognitive apparatus on task performance and learning; the primary goal of CLT is to improve instructional design. CLT has its foundation in studies of human cognition. Its basic premise is that cognition and learning are constrained by a bottleneck created by working memory, in which we humans can hold only a handful of elements at a time for active processing. What enables us to carry out complex tasks is our virtually unlimited long-term memory. We learn by chunking related elements into domain-specific schemas that are stored in long-term memory and retrieved for processing in working memory as a single element. As our experience grows, we construct hierarchies of increasingly complex higher-level schemas that encompass numerous lower-level schemas. Even though an expert’s working memory, too, is very limited, the expert can occupy it with high-level schemas they have previously constructed, which enables them to process vast amounts of information that a beginner could not hope to cope with. For example, a novice programmer will be overwhelmed by a “basic” loop that uses expressions, assignment, variables, and selection to process inputs unless they are sufficiently practiced with the lower-level schemas involved; a more experienced programmer will perceive the entire loop as a single instance of a familiar pattern.

Cognitive load is the demand that a situation places on a person’s working memory. It is determined by element interactivity, which is the degree of interconnectedness between the elements of information that one needs to hold in working memory simultaneously in order to perform successfully [37]. Element interactivity, in turn, depends on prior knowledge in the form of existing schemas: someone who can represent the situation with higher-level schemas will require fewer of them as elements in working memory. An estimate of element interactivity can be obtained by identifying interacting elements in learning materials [2, 80]; such estimates necessarily rely on assumptions about learners’ existing schemas [10].

According to CLT, cognitive load can be analytically separated into two components: intrinsic and extraneous [81]. Intrinsic load is caused by interacting elements that are necessary for task performance and learning. Extrinsic load is caused by elements that “don’t need to be there,” but are, whether because of ineffective instructional design, external interference while learning, or some other reason. What counts as intrinsic depends on the learning objectives. For example, in a programming task, syntax can be intrinsic (if the goal is to learn a programming language) or extraneous (if the goal is to learn to solve a problem). Instructional design based on CLT generally seeks to minimize extraneous load, encourages schema formation through practice, and sequences tasks such that intrinsic load is kept in check [e.g., 84].

Cognitive load is an idealistic construct in that it assumes the full attention of a motivated learner. The amount of working memory capacity that a learner actually dedicates to generating processing depends on external factors such as engagement [40, 81].

2.2 Plans: Schemas in Programs

Schema theory has influenced studies of program construction and comprehension. In their seminal work, Soloway and his colleagues [e.g., 73, 74, 77] broke down programs in goal-plan trees: such a tree recorded a hierarchical structure of goals and subgoals and the corresponding plans and subplans that provide solutions to those goals. What Soloway’s group termed “plans” are essentially schemas in the programming domain: a plan represents a stereotypical solution to a programming problem. Building on Soloway’s work, Rist [63] showed how schemas affect programming strategy: both novices and experts program top-down when they can but resort to constructing solutions bottom-up where their existing schemas fail them. The key difference between novices and experts is that experts have a much more extensive “library” of programming schemas in long-term memory.

Within CER, these cognitive theories have inspired pedagogies that explicitly teach plan-like patterns to students [20, 34, 66].

Recently, several studies have examined how students prefer to compose their overall solution from a number of interrelated subplans [25–27]. For instance, one might sequence the subplans (perhaps using separate functions for each) or interleave them (perhaps using a single loop associated with multiple subplans); such decisions may impact on readability and error rates [25, 27, 33, 74]. Plan composition is a potentially significant determinant of cognitive load since it impacts on which elements (i.e., schemas) the writer or reader of a program needs to keep in mind simultaneously; this is something we will explore later in this article.

An established measure of cognitive load for programs does not exist. However, there is an instrument for estimating cognitive load from learners’ ratings of perceived mental effort after a learning task [42], which has been adapted for programming tasks [57].

2.3 Complexity vs. Difficulty

As illustrated in a survey by Liu and Li [47], complexity means different to different people. Following Liu and Li (ibid.), we use the word complexity for the “objective,” learner-independent characteristics of a task, whereas the difficulty of a task additionally depends
on the characteristics of the person who engages in the task, such as prior knowledge and motivation, as well as on contextual factors.

We consider element interactivity to be a key aspect of both complexity and difficulty. The inherent complexity of any task is determined by the interconnectedness of the elements present in the task. It reflects the need to process multiple elements simultaneously in working memory, assuming no prior knowledge in the domain. Existing schemas mediate complexity by helping the learner deal with it in larger chunks, thereby reducing the element interactivity — and, by extension, the difficulty — of a complex task. The more complex a task is, the more schemas the learner must possess so that the task is not too difficult for them.

In this article, we are primarily concerned with complexity — the unmediated element interactivity inherent in a task. More specifically, we are interested in the complexity inherent in programs. That complexity, we argue, accounts for a substantial part of the complexity of any activity in which the learner has to mentally manipulate those programs, such as writing or comprehending them. Of course, complexity alone does not account for real-world learning outcomes; we will say more about difficulty in later sections.

2.4 The Model of Hierarchical Complexity

The Model of Hierarchical Complexity (MHC) [16] is a neo-Piagetian theoretical model for analyzing the complexity of actions within a domain. Moreover, the MHC seeks to characterize the domain-specific stages of development that a learner goes through as they gain expertise in the domain and become capable of successful performance on increasingly complex actions.

According to the MHC, an action is an exhibited behavior with a particular sort of input and a particular sort of output; a person employs cognition to perform an action but the specific cognitive processes that occur are not explained by the MHC. Instead, the MHC is concerned with the structural relationships between actions, in particular, the "recursive" relationships between a more complex action and its less complex sub-actions.

The MHC posits that actions within a domain can be organized in a hierarchy. How high a particular action appears in such a hierarchy reflects its intrinsic complexity and is determined by its recursive relationships with other actions. Not just any dependency between actions is enough for a difference in complexity, however: an action is only more complex than another if it coordinates less complex actions according to the MHC axioms or rules.

The MHC distinguishes between two kinds of actions: 1) primary actions at the lowest level of complexity, and 2) composite actions that organize other actions according to a rule that may or may not imply higher complexity. There are three rules [12]:

- The prerequisite rule applies where succeeding at action A requires successful performance of exactly one other action at the same level of complexity as A. However, this does not mean that A is more complex than its prerequisite, only that successful performance on A is preceded by successful performance on it.
- The chain rule applies where a higher-level action A requires the organization of two or more lower-level actions in an arbitrary way: the lower-level actions are parts of A but can be carried out in any order and the whole is no greater than the sum of its chained parts. For example, the action of calculating \(1 + 2 \times 4\) links the actions of addition and subtraction with the chain rule, as one may carry out those sub-actions in either order.
- Finally, and most importantly, the coordination rule applies where a higher-level action A organizes two or more actions at a lower level of complexity in a non-arbitrary way. This means that the lower-level actions must serve distinct roles within the higher-level action; they cannot be simply swapped for each other or performed in an arbitrary order [13]. The distributive law is an example: computing \(2 \times (5 + 3) = (2 \times 5) + (2 \times 3)\) displays more complex behavior by giving addition and multiplication distinct roles rather than just performing the sub-actions separately.

For an action to be more complex than another, it must organize a minimum of two lower-level actions as per the coordination rule. Every primary (lowest-level) action \(A_0\) within a domain has the complexity level \(h(A_0) = 0\). Every more complex action \(A_k\) coordinates at least two lower-level actions \(A_i, \ldots, A_j\) and has a higher level of complexity than any of them: \(h(A_k) = \max(h(A_i), \ldots, h(A_j)) + 1\).

The MHC characterizes each level of complexity in terms of lower-level actions. In doing so, it postulates that someone who is able to perform at level \(n\) is also able to perform at level \(n - 1\); this implies a learning trajectory from less complex actions to more complex ones. According to the MHC, the developmental level of a learner in a domain equals the level of the highest action that the learner is able to carry out successfully [15].

The MHC has been empirically validated in several educational disciplines. For instance, Commons [12] applied Rasch analysis to show a positive correlation between the predicted complexity of equations in physics (the pendulum test) and measured student performance. In another study, Dawson [19] compared an MHC-based metric of learner development to other developmental scoring systems and found that it measured the same latent variables and was more internally consistent than the other metrics. The MHC has been used for complexity analysis in diverse domains such as physics [78], bias in forensics [14], chemistry [3], and student competence in graduate courses [56].

3 THE COGNITIVE COMPLEXITY OF COMPUTER PROGRAMS

The Cognitive Complexity of Computer Programs (CCCP) framework is a theoretical model for reasoning about the complexity of computer programs and generating metrics that summarize aspects of complexity. The CCCP characterizes the complexity of a program from a cognitive perspective: it describes and quantifies the cognitive constructs that are present in the program design and that are required for mentally manipulating the program.

Building on the analyses of Soloway and Rist cited above, we examine not only abstract, language-agnostic plans but also the lower-level plans that implement the higher-level plans as individual instructions in a concrete program written in a particular language. The CCCP also extends existing plan-based approaches to program analysis by adapting the hierarchy-building rules of the MHC to the study of computer programs. The formal rules of the MHC structure the study of the relationships between plans and provide a foundation for claims about the relative complexity of different plans and the programs the plans appear in.
Taking our cue from Rist [63], we distinguish between plan schemas and plans. A plan schema is a cognitive structure that a programmer mentally manipulates, while a plan is the concrete realization of a plan schema in a program. The CCCP assumes that there is a direct mapping between plans in code and plan schemas in the memory of the programmer who successfully works on the program; therefore, our analysis of plans in a program can be said to provide a cognitive perspective on the plan schemas that the program calls for. Moreover, we posit that applying a plan schema can be viewed in terms of the MHC as an action; therefore, we can adopt the MHC rules for the analysis of plan hierarchies.

We define the cognitive complexity of a program in terms of the hierarchical structure of plans present in the program. The CCCP is concerned with two aspects of this complexity: 1) the complexity level of each plan in the hierarchy, and 2) any interactions between plans that demand simultaneous processing of the plans in working memory, thus contributing to higher intrinsic load. Correspondingly, the CCCP can be used for two types of analysis, which we term hierarchical analysis and interactivity analysis, respectively. A full CCCP analysis of a program starts with the concrete code (or detailed pseudocode) and produces a plan hierarchy as well as the associated metrics of plan depth and maximal plan interactivity.

The subsections below introduce hierarchical analysis and interactivity analysis in turn, demonstrating them with case studies of programs. Our intention here is not to provide an unambiguous algorithm for thoroughly analyzing any given program. We merely seek to illustrate what complexity is in terms of the CCCP, to provide a few proof-of-concept examples of the sort of output that a CCCP-based analysis can produce, and to tentatively explore methods that can generate that output.

All the case studies were analyzed in iterations by the first author, with feedback from the other authors.

3.1 Hierarchical Analysis

Hierarchical analysis of a program produces a description of the program as a tree of plans; Figure 1 shows an abstract example. Each plan appears in the tree at a particular level of complexity, with the primary plans at the lowest level of zero and the more complex plans at increasingly high levels. The plan that corresponds to the entire program is at the top (root) of the hierarchy.

The level of a plan is known as its plan depth (PD); this is our programming-domain equivalent of what is termed an action’s “order” or “level” by the MHC. The plan depth of the top-level plan is the plan depth of the entire program; this is one of the two main numerical metrics of complexity that the CCCP can generate. Programs can be compared in terms of their plan depth. In addition to this summary metric, the plan tree provides a comprehensive look at the plans involved in reading or writing the program.

Following the application of the MHC in other domains, we start hierarchical analysis from the primary plans at the bottom. These elements are chosen so that they represent primitive operations of a notional machine [22, 75] that the programmer instructs. We did not use an explicit specification of a notional machine; instead, we started with system capabilities for which we could identify no coordination of other plans without introducing low-level concepts outside of the target notional machine implied by the program. (E.g., the capability to store a value is a primary notional-machine element, as is the capability to jump to a different part of the code; bit-level operations are not, as they are unnecessary for understanding the programs in our case studies.)

To build the upper levels of the hierarchy, we considered how the plans depend on each other. Specifically, we sought instances of the MHC rules of coordination (i.e., lower-level plans serve distinct roles in a higher-level one) and prerequisites (i.e., a plan depends on one other plan at the same depth). The chain rule (i.e., applying simpler plans in an arbitrary sequence) does not contribute to higher complexity in the MHC/CCCP sense; to avoid unnecessarily complicating our plan trees, we chose to ignore chaining.

3.1.1 Case Study 1: Summing Program

We analyze a program that sums a fixed sequence of numbers using an explicit loop control variable; the code is shown below and the plan tree in Figure 2.

```plaintext
1: int i, input, sum;
2: sum = 0;
3: for (i = 1; i <= 10; i++) {
4:   read(input);
5:   sum = sum + input;
6: }
```

The summing program features four primary plans (P1–P4) at the lowest level. The define literals plan (P1) represents the use of numerical data in the program. P1 is a prerequisite for the declare variable plan (P2), an abstraction of the linking of names to storage in computer memory. The arithmetic operator plan (P3) signifies the use of arithmetic operations such as addition. The jump to code plan (P4) represents the unconditional transition of control to a different part of the code, as at the end of the loop.

The initialize variable plan (P5) represents the assignment of a literal value to a variable. It organizes two lower-level plans (P1 and P2) using a coordination rule. (The order of the assignment matters; 1 = 1 would be an error.) $PD(P5) = max(PD(P1), PD(P2)) + 1 = max(0, 0) + 1 = 1$. The evaluate an expression plan (P6) organizes P2 and P3, so expressions are evaluated over variables. Thus, $PD(P6) = max(PD(P2), PD(P3)) + 1 = 1$. The accumulate in a variable (P8) plan coordinates the assignment of a literal to a variable (P5) and the evaluation of the expression to be assigned (P6). Therefore, $PD(P8) = max(PD(P5), PD(P6)) + 1 = 2$. The test for termination plan (P9) is a selection plan that coordinates P6 and P4, evaluating an expression and branching to the appropriate part of the code. $PD(P9) = max(PD(P4), PD(P6)) + 1 = 2$. The read input plan (P7) represents a library function call. To apply P7, it is only necessary...
to comprehend the purpose of the function and consider its input [41], a variable (P2); P7 coordinates P2 and the return value that is assigned to a variable with P5. (While the assignment of the return value and an initialization plan have different goals, they coordinate the same plans and have the same plan structure.) Thus, \( PD(P7) = \max(PD(P5), PD(P2)) + 1 = 2 \).

Loop over a range (P10) coordinates the initialization (P5), test for termination (P9) and increment (P8) plans in a non-arbitrary order defined by the desired control flow. Thus, \( PD(P10) = \max(PD(P5), PD(P8), PD(P9)) + 1 = 3 \). Finally, the entire summation program (P11) coordinates the loop (P10), input (P7), and accumulation (P8). \( PD(P11) = \max(PD(P10), PD(P7), PD(P8)) + 1 = 4 \).

3.1.2 Case Study 2: Averaging Program. While presenting the hierarchical analysis of the following program, we only discuss what is new compared to the first case study.

```
1 def average(collection):
2     return sum(collection) / len(collection)
3 l = [1, 2, 3, 4]
4 average(l)
```

As shown in Figure 3, the primary plan declare variable (P2) is a prerequisite for the declare collections plan (P3). Initialize a collection (P9) assigns a sequence of literals to a collection, coordinating P5 and P3, thus having a \( PD(P9) = 1 \). Assign a literal to a variable (P8) similarly coordinates P2 and P5.

The library-function-calling plans sum/size of a collection (P7) coordinate the input of the function, a collection (P3), and the assignment of the return value to memory (P8), where \( PD(P7) = \max(PD(P3), PD(P8)) + 1 = 2 \). The call average function plan (P6) has a similar structure, adding a jump to code plan (P1) and activating the user-defined function. \( PD(P6) = \max(PD(P3), PD(P8), PD(P1)) + 1 = 2 \). The assign an expression to a variable (P12) plan coordinates the evaluation of an expression (P10) and the assignment of its result to a variable (P8). \( PD(P12) = \max(PD(P8), PD(P10)) + 1 = 2 \). The pass a parameter (P11) plan is the use of a collection as a parameter (P9), which is later instantiated within the function activation (P3). Therefore, \( PD(P11) = \max(PD(P3), PD(P9)) + 1 = 2 \).

![Figure 2: Plan tree for Case Study 1.](image)

![Figure 3: Plan tree for Case Study 2. The arrow colors are for visual clarity only.](image)

The calculate average plan (P13) coordinates the results of the sum and size function calls (P7) and the evaluation of an expression (P10). \( PD(P13) = \max(PD(P7), PD(P10)) + 1 = 3 \). The return the average function result plan (P14) takes as input the result of the average calculation of P13 and terminates the function activation, performing a jump to the main program (P1) and assigning its result to memory in the global scope (P8). Therefore \( PD(P14) = \max(PD(P1), PD(P8), PD(P13)) + 1 = 4 \). The define the average function plan (P15) encapsulates the whole function activation, coordinating its return statement (P14), its parameters (P11), and the function invocation (P6). \( PD(P15) = \max(PD(P6), PD(P11), PD(P14)) + 1 = 5 \). The whole averaging program (P16) coordinates the function definition (P14) and the initialization of the collection (P9). \( PD(P16) = \max(PD(P9), PD(P15)) + 1 = 6 \).

3.2 Interactivity Analysis

We have shown how hierarchical analysis (HA) gives an overall sense of the complexity of a program. It also suggests how learning to understand a particular aspect of the program (plan) is predicated on first learning to understand other aspects. HA does not, however, attend to simultaneous processing in working memory, which is central to element interactivity and cognitive load. For this purpose, the CCCP extends HA with interactivity analysis (IA).

Plans are again our basic unit of analysis; IA examines how plans are put together in a program. Specifically, we estimate which plans must be kept in mind simultaneously as the programmer mentally manipulates the (higher-level) plans of the program.

Whereas HA was concerned with complexity alone, IA additionally deals with difficulty: we must consider the programmer’s prior knowledge and the chunking of multiple subplans into larger wholes that is processed as a single element. As a starting point for IA, we take the program code and the plan tree produced by HA, and state our assumptions about the prior knowledge of the programmer: which plans do we expect the programmer to be able
to deal with as single elements because they are sufficiently familiar with that plan’s individual subplans? We can then use IA to estimate the plan interactivity for programmers that meet the assumption.

We build on the concept of focus of attention (FoA) [18, 60], which we have adapted to a programming context. At any given time while a programmer works on a program, the FoA is a single plan that has been activated in working memory for immediate processing. It is linked to a subset of other plans in the program that need to be considered simultaneously with the FoA; these other plans form a region of direct access (RDA)[60] that must also be stored in working memory. As a programmer processes a program, their FoA will shift from one plan to another at which point they will also rearrange the RDA in working memory as required.

To conduct IA, we examine the control and data flow of the program. We trace the execution of the program step by step, considering how the program flow and (we posit) the FoA shift from one plan to another. We ignore the lower-level plans that the programmer is expected to have abstracted away from working memory. At each FoA shift, we compute a plan interactivity (PI) metric that equals the number of plans inside the RDA; PI is essentially a programming-domain estimate of element interactivity as defined by CLT [cf. 2]. Maximal plan interactivity (MPI) is the highest PI value at any FoA in the program.1

Which plans fall within the RDA of a particular FoA depends on the way plans are merged, sequenced, and nested. We consider a plan A to be in the RDA of another plan B if A is directly nested within B or vice versa, or if the execution of A interleaves with that of B. If A’s execution is done before B’s starts, the plans can be considered non-simultaneously and are not in each other’s RDA.2

3.2.2 Case Study 2 revisited for IA. For the IA of Case Study 1, we color-coded the Summing Program so that each color maps to a relatively high-level plan: loop (P10) in blue; accumulate (P8) in red; and reading (P7) in green. In this example of IA, we assume that the programmer has sufficient prior knowledge that they can process each of these schemata as a chunk. The control and data flow of these three plans are interleaved. It is difficult to isolate the control flow of the reading plan from the accumulate and loop plans: the shared iterative control structure dictates when the plans start and end. The data flow also suggests an interaction between reading and summing plans (a shared input variable). This means that all three plans need to be active in the RDA in order to process any one of them as the FoA, which yields an MPI of 3.

3.2.2 Case Study 2 (Averaging Program) revisited for IA. In the IA of Case Study 2, we assume the programmer can process the initialization (P9, cyan), average function definition (P15, purple), sum (P7, red) and size (P7, brown) plans as single chunks. The initialization of a collection plan can be processed in isolation in the RDA, with a PI of 1. Calling the average function (line 4) activates the function-definition plan in the RDA. The sequenced plan-composition strategy enables the programmer to evaluate plans in isolation and later compose only the results of the plans. They can shift the FoA to a specific plan, compute its result, and shift to the next FoA with the result of the previous FoA as input. For example, the FoA shifts from the summing plan (nested inside the average plan, PI 2) to the size plan (nested with the average plan, PI 2). Shifting the FoA back to the function definition plan, activated in the RDA, we compute the average by activating the results of the function calls (not all their constituent parts) and evaluate an expression (from the average plan), computing the return value of the function. The MPI for case study 2 is therefore 2.

3.2.3 Case Study 3: Averaging Rainfall. In this case study, we consider a version of the Rainfall Problem [69, 73]. We compare two solutions that differ in plan composition: a merged-plans solution (Figure 4) and a sequenced-plans solution (Figure 5), both adapted from [76]. Each color corresponds to one of the main high-level plans identified in other work on the same problem [26, 69]: iteration (light blue), a sum (red), read (light green), average (purple), count (brown), guard against negative (dark blue), guard against division by zero (orange) and sentinel (light green). We assume the programmer can deal with each highlighted plan as a single chunk. With the exception of the exit-in-the-middle loop with a while (true) statement (which coordinates a literal and a jump-to-code plan), we already analyzed the other plans (or their close variants) in the previous case studies.

```plaintext
1 var count = 0
2 var sum = 0
3 var average = 0
4 while (true) {
5   val input = readInt()
6     if (input >= 0) {
7       if (input >= 999999) {
8         if (count == 0) { println("No data!"); break }
9         println(average);
10       } else {
11         break
12       }
13       count += 1
14       sum += input
15       average = sum / count
16     }
17     else { break }
18   }
```

Figure 4: A color-coded merged-plans solution for Rainfall.

The merged-plans program and sequence-plans program employ different plans and therefore yield different PD scores. We will leave that aside, however, since our present purpose is to explore the interactivity between plans in each program.

Analyzing the merged-plans program in Figure 4, we observe that all plans share the same control flow through the loop plan (using the while (true) loop) and some plans share the same data flow. For instance, the variable `input` is shared among the input, negative, sentinel and sum plans, while count is shared by the count and guard plans. This interleaving of control and data flow forces the activation of all these subplans in working memory for every FoA shift. In order to be able to process the program and extract its meaning (or write it), the programmer needs to evaluate the impact of each plan on the data and control flow. Therefore, for the merged-plans program in Figure 4 all plans must interact in the RDA, yielding an MPI of 8.

The sequenced-plans approach in Figure 5 uses, where possible, a compartmentalization strategy by making extensive use of

1IA is loosely related to the work of Letovsky and Soloway [43], who studied delocalized plans scattered in the program text. However, IA is based on program flow rather than the textual organization of program code.

2Since programs are generally written down, the program text can in practice serve as a tool for external cognition [65], reducing working memory load. IA, as presented, does not account for this and may overestimate load in some cases.
functions. By switching the FoA at each function call (or function evaluation), sequenced plans composition induces a switch-process-store-output (SPSO) processing pattern, reducing the number of simultaneously activated plans in the RDA.

To process the reading plan (line 7), the FoA switches to the sentinel plan, composed with the loop plan, with both simultaneously in the RDA (PI 2). At the end of the input plan (sentinel found), the plans collapse to a single result (a collection), stored in the RDA. The FoA switches to the negative plan (also line 7), which is processed using (only) the result of the previous step as input and which outputs a collection to the reading plan (PI 1, processing just the negative plan). The reading plan is then processed with its inputs and stored for later composition.

At line 8, a function call switches the FoA to the averaging plan of line 3. The averaging plan activates the guard-against-zero plan (lines 4 and 6) and the computation of the average itself. To compute the average (with guard and average active in the RDA), the FoA switches to the sum plan (using an SPSO pattern), which makes the sum plan the only active plan in the RDA for now. After another switch to the count plan (SPSO again) we are back to the first RDA in order to compute the averaging expression. Having the average and guard plans activated yields a PI of 2. Overall, the sequenced-plans Rainfall program of Figure 5 has an MPI of 2.

4 RELATED WORK AND DISCUSSION

Of the theoretical tools that have been used in CER for categorizing activities by complexity, Bloom’s Taxonomy [5] is probably the most common [7, 29, 52]. Meanwhile, the SOLO taxonomy [4] has been used for analyzing students’ responses to code-reading tasks [71], the structure of students’ code [11], and aspects of program design such as testing and abstraction [9]. In addition to classifying skills such as tracing and writing code in taxonomies, researchers have investigated the dependencies between the skills and the way the skills evolve with growing expertise [17, 44, 82].

The studies cited above emphasize the general types of activities that programmers engage in (Bloom and/or the general degree of structuredness in learning outcomes (SOLO)). Our work on the CCCP differs from them in that we seek to characterize the content of the programming activities—the programs themselves—and to do so at a relatively fine level of detail. We believe that this is a useful complement to the existing work that examines the complexity of different programming activities and the relationships between them. For example, to establish a progression of skills, some studies have sought to compare student performance on reading code vs. writing code of “similar complexity” [45, 72] and would benefit from a better definition of program complexity than is currently available. Simon et al. [72] ask: “Is an assignment statement easier or harder to read and understand than a print statement? Is a nested loop easier or harder than an if-then-else? Is the difficulty of a piece of code simply the linear sum of the difficulties of its constituent parts?” By adopting the axioms of complexity from the MHC, the CCCP provides a theoretical grounding for claims about the relative complexity of different programming constructs; by further considering the shifting focus of attention and cognitive load, the CCCP suggests that difficulty is not simply a linear sum of the entire code.

We are not the first to propose a set of cognitive complexity metrics for programming. Cant et al. [8, 32] conceptualized CCM, a tentative framework for measuring complexity, which resembles ours in its goals and in that it, too, draws on schema theory and related findings from cognitive psychology. Our present work overlaps that of Cant et al.; we focus on a narrower set of metrics, operationalize them, and bring them to bear on actual programs.

The MHC has been applied in other domains to define learning trajectories of increasingly complex tasks [e.g., 3]. The CCCP similarly suggests a progression from the concepts required at the leaves of the plan trees towards the higher-level roots. In this respect, our work shares some goals with earlier work that has proposed learning trajectories for introductory programming. The proposal of Mead et al. [55] links concepts in intuitively-constructed “anchor graphs,” in which learning an earlier concept ought to carry some of the cognitive load of later concepts; the most obvious difference between their work and ours is that they focused on generic programming concepts whereas we analyze plans in individual programs. Rich et al. [62] created a set of K-8 learning trajectories for three concepts: sequencing, repetition, and conditionals; they identified challenges in basing the trajectories on a heterogeneous and sparse set of prior reports of concept difficulty. Izu et al. [35] suggested an intuitive, SOLO-inspired learning trajectory in which the learner abstracts increasingly complex “building blocks” (language constructs and plan-like “templates”) in order to tackle the next concepts; the CCCP differs from this work, and the other research just cited, in its use of MHC to structure the plan hierarchy.

The CCCP may also help interpret some earlier results in CER. For instance, Mühling et al. [58] gave students a psychometric test whose items featured different programming constructs. Contrasting student performance on different items, Mühling et al. found that simple sequences were easy for the participants and that loops with a fixed number of iterations were easier than all items involving conditionals. Their most surprising result, the authors suggest, was that nested control structures were easier than a loop with an exit condition. The CCCP predicts these findings (albeit with the proviso that the CCCP deals with complexity, whereas measured difficulty is affected by prior exposure to different constructs). For example, nesting fixed-iteration loops does not increase plan depth beyond that of a conditional loop exit. As another example, Ajami et al. [1] measured the performance of professional developers on program-comprehension tasks that were otherwise similar but featured different programming constructs. Their findings suggest that conditionals were less complex than for loops, that the size of the expressions used as inputs for constructs had an impact on performance, and that flat structures are slightly easier than nested ones; these results match what is predicted by our plan depth metric.
We have outlined a theoretical framework, the CCCP, for assessing writing code, we would need to consider a minute analysis of the which may result in cognitive overload — and puts forward the cognitive complexity that is manifested in computer programs.

5 CONTRIBUTIONS AND LIMITATIONS

We have explored how to analyze plan hierarchies and the interactions between plans in terms of the CCCP and demonstrated the complexity metrics of plan depth and plan interactivity. Moreover, we have discussed how such analyses can contribute to debates about the relative merits of curricular decisions as well as the design and interpretation of research on student programming.

Our methodological exploration so far has been tentative. We have a goal, a framework, and examples of plausible analyses of programs in terms of the framework: we do not yet have a well-defined analysis process. Before the CCCP can be applied more easily and transparently, we must further refine the steps that an analyst must take in order to delimit plans and apply the MHC-derived rules of the CCCP to them. Even so, our work lends preliminary support to the idea that the MHC, which has not been previously applied to CER, can provide structure to analyses of program complexity.

The CCCP is built on general theoretical models for which there is empirical support. Nevertheless, if the CCCP is to be more than an idealistic construct that fails in practice, it must be directly evaluated and refined based on empirical findings. Each of our case studies reflects one possible breakdown of an example program in terms of the CCCP, and while we have provided a rationale for this analysis in theoretical terms, it remains to be seen whether it aligns with student performance, for instance. Since student performance reflects the difficulty of a task, any empirical evaluations will need to account for prior knowledge [cf. 10, 64].

We have limited our analysis of complexity to a particular facet: the cognitive complexity present in program designs. We believe this to be a very significant aspect of complexity in programming tasks, but it is not the only one. Another significant facet of task complexity is what the learner is expected to do with the program. We envision that the CCCP could be used in combination with other frameworks that emphasize the activity aspect: for instance, in the 4C/ID model of instructional design [84], students engage in different activities within a task class (e.g., worked examples followed by completion tasks followed by problem solving) before proceeding to another task class with more complex content. The CCCP could help in identifying and ordering task classes.

The complexity of a programming task is additionally influenced by factors such as task presentation [70], contextualization [6, 49] and syntax [21, 79], which are not covered by the CCCP as presented. In the future, we may expand the framework by adapting a generic task model from the literature [e.g., 47] to programming education.

Violating programmers’ expectations of code structure leads to poorer comprehension, as existing schemas fail to apply [e.g., 31]. In the present work, we have only considered programs that are “planlike” and unsurprising.

Our example programs cover only a handful of basic plans. Additional work is required in order to extend the present work to other content (e.g. recursion, objects) and more complex plans.

In the future, we intend to adopt a mixed-methods approach to evaluating the CCCP and developing the analysis process. The evaluation may incorporate elements such as expert validation, empirical measurements of prior knowledge (cf. earlier work in estimating cognitive load [2]), correlation of predicted complexity with task performance through Rasch analysis (cf. how the MHC has been validated in other domains [13]), and triangulation against mental effort ratings [57].