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

ACM Reference Format:

Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an Analysis of Program Complexity From a Cognitive Perspective. In *ICER '18: 2018 International Computing Education Research Conference, August 13–15, 2018, Espoo, Finland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3230977.3230986

ICER '18, August 13–15, 2018, Espoo, Finland

ACM ISBN 978-1-4503-5628-2/18/08…$15.00
https://doi.org/10.1145/3230977.3230986

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 for introductory programming [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 low-lever 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. *Extraneous 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 germane 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 $1 + 2 - 4$ 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 $A_{i...j}$ and has a higher level of complexity than any of them: $h(A_k) = max(h(A_i), ..., h(A_j)) + 1$.

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.)

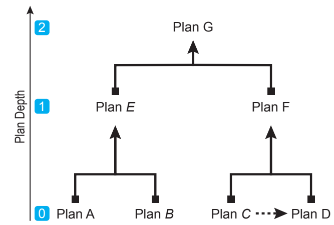

Figure 1: A generic plan tree. Solid arrows indicate that a higher-level plan *coordinates* lower-level plans, as per the MHC axioms. Dotted arrows indicate a *prerequisite*. The plan depths of each level are shown in the blue boxes.

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.

```
int i, input, sum;
sum = 0;
for (i = 1; i <= 10; i++) {
    read(input);
    sum = sum + input; }
```

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 = i` 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

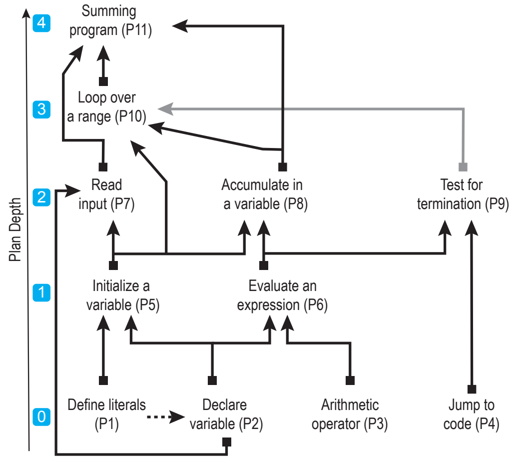

Figure 2: Plan tree for Case Study 1.

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 *summing 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.

```
def average(collection):
    return sum(collection) / len(collection)
l = [1, 2, 3 ,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 $PD(P7) = max(PD(3), PD(8)) + 1 = 2$. 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, $PD(P11) = max(PD(P3), PD(P9)) + 1 = 2$.

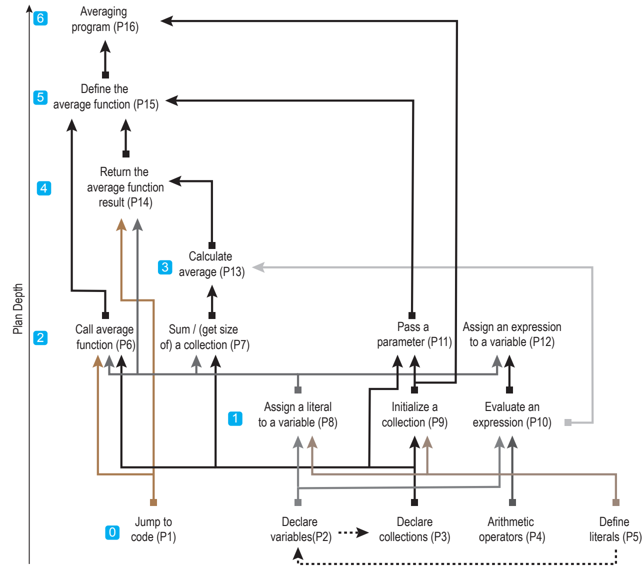

Figure 3: Plan tree for Case Study 2. The arrow colors are for visual clarity only.

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]

3.2.1 *Case Study 1 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 schemas 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.

```
var count = 0
var sum = 0
var average = 0
while (true) {
   val input = readInt()
   if (input >= 0) {
      if (input >= 999999) {
        if (count == 0) { println("No data!"); break }
        println(average);
        break }
    count += 1
    sum += input
    average = sum / count } }
```

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

[1]IA 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.

[2]Since 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.

```
1 def isNotSentinel (input : Int) = input < 999999
2 def isValid (input : Int) = input >= 0
3 def average (numbers : List [Int]) =
4     if (numbers.nonEmpty)
5         numbers.sum / numbers.size
6     else 0
7 val validInputs = inputs.takeWhile(isNotSentinel).filter(isValid)
8 val averageRainfall = average(validInputs)
```

Figure 5: A sequenced-plans solution for Rainfall.

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, 48, 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.

It has been suggested that students find merged, interleaved plans difficult and that sequential plan composition is easier to deal with [73, 76]. There is also evidence that students mistakenly concatenate plans instead of merging and generally struggle with plans which interact in complex ways [28], that students sometimes read merged plans as if they were composed sequentially [30], and that a student's plan composition strategy interacts with the likelihood they will write defective code [25, 74]; a sequenced-plans approach may lead to better outcomes [25]. The CCCP offers a partial explanation for these findings — merged plan composition demands more plans simultaneously in the region of direct access, which may result in cognitive overload — and puts forward the plan interactivity metric as an indicator of program complexity.

Student performance on code-writing and code-reading questions has been assessed in a number of multi-institutional studies, usually to the conclusion that introductory-level students commonly perform at a level below their teachers' expectations [45, 54, 83]. In order for measurable progress to be made and reasonable expectations set, the complexity of the programs in such assessment instruments should be evaluated. Simon et al. [72] argue: "If we are to have a reliable measure of students' abilities at reading and writing code, we would need to consider a minute analysis of the difficulty levels of code-reading and code-writing questions at the micro level." We expect that a tool like the CCCP could contribute towards such a "micro-level" analysis.

A number of studies over the past few decades have investigated bugs in student code and mapped the bugs to the plans in which they appear [23, 25, 36, 59, 69]. As the CCCP extends Soloway's plan-based analysis by introducing the MHC rule set, it may contribute to the methodology of future studies in this vein.

Another point of reference for our work is the use of program metrics in CER. Luxton-Reilly et al., like ourselves, analyzed individual programs in order to characterize their structure [50, 51]; their analysis and metrics are fundamentally different from ours in that they focused exclusively on syntactic features whereas we have focused on cognitive plans. In a related line of CER, surface-level metrics from Software Engineering such as Cyclomatic Complexity [53], lines of code, and block depth have been applied to code-comprehension tasks [38], code-writing tasks [24], and multiple-choice questions [67]. Such metrics provide one perspective on software complexity but do not indicate how different constructs and plans affect complexity [1], have a low correlation with perceived complexity [39], and disregard the role of prior knowledge.

Finally, a different sort of metric was validated by Morrison et al. [57], who estimated cognitive load by asking students for their subjective perceptions of mental effort after the students had engaged in various types of CS study. We certainly see such post-activity surveys as being valuable, too, but at the same time we hope that the CCCP will mature into a complementary tool for the analytical *a-priori* assessment of individual *programs* during instructional design [cf. 2, 61, 81].

5 CONTRIBUTIONS AND LIMITATIONS

We have outlined a theoretical framework, the CCCP, for assessing the cognitive complexity that is manifested in computer programs. 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].

REFERENCES

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. 2017. Syntax, predicates, idioms: what really affects code complexity?. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 66–76.

[2] Jens F Beckmann. 2010. Taming a beast of burden–On some issues with the conceptualisation and operationalisation of cognitive load. *Learning and instruction* 20, 3 (2010), 250–264.

[3] Sascha Bernholt and Ilka Parchmann. 2011. Assessing the complexity of students' knowledge in chemistry. *Chemistry Education Research and Practice* 12, 2 (2011), 167–173.

[4] John B Biggs and Kevin F Collis. 2014. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

[5] Benjamin S Bloom et al. 1956. Taxonomy of educational objectives. Vol. 1: Cognitive domain. *New York: McKay* (1956), 20–24.

[6] Dennis Bouvier, Ellie Lovellette, John Matta, Bedour Alshaigy, Brett A. Becker, Michelle Craig, Jana Jackova, Robert McCartney, Kate Sanders, and Mark Zarb. 2016. Novice Programmers and the Problem Description Effect. In *Proceedings of the 2016 ITiCSE Working Group Reports (ITiCSE '16)*. ACM, New York, NY, USA, 103–118. https://doi.org/10.1145/3024906.3024912

[7] Duane Buck and David J. Stucki. 2000. Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *SIGCSE Bulletin* 32, 1 (2000), 75–79. https://doi.org/10.1145/331795.331817

[8] SN Cant, DR Jeffery, and B Henderson-Sellers. 1995. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology* 37, 7 (1995), 351 – 362. https://doi.org/10.1016/0950-5849(95)91491-H

[9] Francisco Enrique Vicente Castro and Kathi Fisler. 2017. Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*. ACM, 10–19.

[10] Hwan-Hee Choi, Jeroen JG Van Merriënboer, and Fred Paas. 2014. Effects of the physical environment on cognitive load and learning: towards a new model of cognitive load. *Educational Psychology Review* 26, 2 (2014), 225–244.

[11] Tony Clear, Anne Philpott, Phil Robbins, and Simon. 2009. Report on the Eighth BRACElet Workshop: BRACElet Technical Report 01/08. *Bulletin of Applied Computing and Information Technology* 7, 1 (2009).

[12] Michael Lamport Commons. 2008. Introduction to the model of hierarchical complexity and its relationship to postformal action. *World Futures* 64, 5-7 (2008), 305–320.

[13] Michael Lamport Commons, Eric Andrew Goodheart, Alexander Pekker, Theo Linda Dawson, Karen Draney, and Kathryn Marie Adams. 2008. Using Rasch scaled stage scores to validate orders of hierarchical complexity of balance beam task sequences. *Journal of Applied Measurement* 9, 2 (2008), 182.

[14] Michael Lamport Commons, Patrice Marie Miller, Eva Yujia Li, and Thomas Gordon Gutheil. 2012. Forensic experts' perceptions of expert bias. *International journal of law and psychiatry* 35, 5-6 (2012), 362–371.

[15] Michael Lamport Commons, JA Rodriguez, PM Miller, SN Ross, A LoCicero, EA Goodheart, and D Danaher-Gilpin. 2007. Applying the model of hierarchical complexity. *Unpublished manuscript* (2007).

[16] Michael Lamport Commons, Edward James Trudeau, Sharon Anne Stein, Francis Asbury Richards, and Sharon R Krause. 1998. Hierarchical complexity of tasks shows the existence of developmental stages. *Developmental Review* 18, 3 (1998), 237–278.

[17] Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. 2012. Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*. Australian Computer Society, Inc., 77–86.

[18] Nelson Cowan, Emily M Elliott, J Scott Saults, Candice C Morey, Sam Mattox, Anna Hismjatullina, and Andrew RA Conway. 2005. On the capacity of attention: Its estimation and its role in working memory and cognitive aptitudes. *Cognitive psychology* 51, 1 (2005), 42–100.

[19] Theo Linda Dawson. 2002. A comparison of three developmental stage scoring systems. *Journal of applied measurement* 3, 2 (2002), 146–189.

[20] Michael de Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph.D. Dissertation. University of Southern Queensland.

[21] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. ACM, New York, NY, USA, 208–212. https://doi.org/10.1145/1999747.1999807

[22] Benedict du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[23] Alireza Ebrahimi. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies* 41, 4 (1994), 457–480.

[24] Said Elnaffar. 2016. Using Software Metrics to Predict the Difficulty of Code Writing Questions. In *2016 IEEE Global Engineering Education Conference*. 513–518.

[25] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 35–42.

[26] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 12–20.

[27] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 211–216.

[28] David Ginat, Eti Menashe, and Amal Taya. 2013. Novice difficulties with interleaved pattern composition. *Lecture Notes in Computer Science* 7780 LNCS (2013), 57–67. https://doi.org/10.1007/978-3-642-36617-8_5

[29] Richard Gluga, Judy Kay, Raymond Lister, and Sabina Kleitman. 2013. Mastering cognitive development theory in computer science education. *Computer Science Education* 23, 1 (2013), 24–57.

[30] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 267–272.

[31] Michael Hansen, Robert L Goldstone, and Andrew Lumsdaine. 2013. What makes code hard to understand? *arXiv preprint arXiv:1304.5257* (2013).

[32] Michael E Hansen, Andrew Lumsdaine, and Robert L Goldstone. 2012. Cognitive architectures: A way forward for the psychology of programming. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 27–38.

[33] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[34] Minjie Hu, Michael Winikoff, and Stephen Cranefield. 2012. Teaching novice programming using goals and plans in a visual notation. In *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*. Australian Computer Society, Inc., 43–52.

[35] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A study of code design skills in novice programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 251–259.

[36] W. Lewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. 1983. *Bug Catalogue: I.* Technical Report. Yale University, YaleU/CSD/RR \#286.

[37] Slava Kalyuga. 2011. Cognitive load theory: How many types of load does it really need? *Educational Psychology Review* 23, 1 (2011), 1–19.

[38] Nadia Kasto and Jacqueline Whalley. 2013. Measuring the difficulty of code comprehension tasks using software metrics. In *The 15th Australasian Computer Education Conference*, Vol. 136. 59–65.

[39] Bernhard Katzmarski and Rainer Koschke. 2012. Program complexity metrics and programmer opinions. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 17–26.

[40] Andreas Korbach, Roland Brünken, and Babette Park. 2017. Differentiating Different Types of Cognitive Load: a Comparison of Different Measures. *Educational Psychology Review* (2017), 1–27.

[41] Sofia Leite, Jose Carlos Principe, Antonio Carlos Silva, Joao Marques-Teixeira, Michael L Commons, and Pedro Rodrigues. In Review. Connectionist models capture hierarchical complexity transitions in development: discrimination between memory-based and operationally-based transitions. (In Review).

[42] Jimmie Leppink, Fred Paas, Cees PM Van der Vleuten, Tamara Van Gog, and Jeroen JG Van Merriënboer. 2013. Development of an instrument for measuring different types of cognitive load. *Behavior research methods* 45, 4 (2013), 1058–1072.

[43] Stanley Letovsky and Elliot Soloway. 1986. Delocalized plans and program comprehension. *IEEE Software* 3, 3 (1986), 41.

[44] Raymond Lister. 2011. Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Proceedings of the Thirteenth Australasian Computing Education Conference* Ace (2011), 9–18.

[45] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, Vol. 36. ACM, 119–150.

[46] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. In *Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*. 118–122. https://doi.org/10.1145/1140123.1140157

[47] Peng Liu and Zhizhong Li. 2012. Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics* 42, 6 (2012), 553–568.

[48] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. *Proceedings of the Fourth International Workshop on Computing Education Research* (2008), 101–112. https://doi.org/10.1145/1404520.1404531

[49] Aleksi Lukkarinen and Juha Sorva. 2016. Classifying the Tools of Contextualized Programming Education and Forms of Media Computation. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2999541.2999551

[50] Andrew Luxton-Reilly, Brett A Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Jacqueline Whalley, et al. 2018. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. ACM, 47–69.

[51] Andrew Luxton-Reilly and Andrew Petersen. 2017. The Compound Nature of Novice Programming Assessments. In *Proceedings of the Nineteenth Australasian Computing Education Conference*. ACM, 26–35.

[52] Susana Masapanta-Carrión and J. Ángel Velázquez-Iturbide. 2018. A systematic review of the use of Bloom's Taxonomy in computer science education. In *Proceedings of the 49h ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, 441–446.

[53] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[54] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.* 33, 4 (2001), 125 – 180. https://doi.org/10.1145/572139.572181

[55] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St. Clair, and Lynda Thomas. 2006. A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. *SIGCSE Bull.* 38, 4 (June 2006), 182–194. https://doi.org/10.1145/1189136.1189185

[56] Patrice Marie Miller and Darlene Crone-Todd. 2016. Comparing different ways of using the model of hierarchical complexity to evaluate graduate students. *Behavioral Development Bulletin* 21, 2 (2016), 223.

[57] Briana B Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 131–138.

[58] Andreas Mühling, Alexander Ruf, and Peter Hubwieser. 2015. Design and first results of a psychometric test for measuring basic programming abilities. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM, 2–10.

[59] Laurie Murphy, Sue Fitzgerald, and Scott Grissom. 2015. Bug infestation! A goal-plan analysis of CS2 students' recursive binary tree solutions. In *Sigcse '15*. 482–487. https://doi.org/10.1145/2676723.2677232

[60] Klaus Oberauer. 2002. Access to information in working memory: exploring the focus of attention. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 28, 3 (2002), 411.

[61] Fred Paas, Juhani E Tuovinen, Huib Tabbers, and Pascal WM Van Gerven. 2003. Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist* 38, 1 (2003), 63–71.

[62] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. 2017. K-8 Learning Trajectories Derived from Research Literature: Sequence, Repetition, Conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 182–190. https://doi.org/10.1145/3105726.3106166

[63] Robert S Rist. 1989. Schema creation in programming. *Cognitive Science* 13, 3 (1989), 389–414.

[64] Peter Robinson. 2001. Task complexity, task difficulty, and task production: Exploring interactions in a componential framework. *Applied linguistics* 22, 1 (2001), 27–57.

[65] Yvonne Rogers. 2004. New theoretical approaches for HCI. *Annual review of information science and technology* 38, 1 (2004), 87–143.

[66] Jorma Sajaniemi and Marja Kuittinen. 2005. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education* 15, 1 (2005), 59–82.

[67] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H. Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. 2013. The Canterbury QuestionBank: Building a repository of multiple-choice CS1 and CS2 questions. In *ITiCSE'13 Working Group Reports (ITiCSE WGR '13)*. 33–51. https://doi.org/10.1145/2543882.2543885

[68] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educational psychology review* 19, 4 (2007), 469–508.

[69] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the Rainfall Problem is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. ACM, 87–96. https://doi.org/10.1145/2828959.2828963

[70] Judy Sheard, Angela Carbone, Donald Chinn, Tony Clear, Malcolm Corney, Daryl D'Souza, Joel Fenwick, James Harland, Mikko-Jussi Laakso, Donna Teague, et al. 2013. How difficult are exams?: a framework for assessing the complexity of introductory programming exams. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*. Australian Computer Society, Inc., 145–154.

[71] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L Whalley. 2008. Going SOLO to assess novice programmers. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 209–213.

[72] Simon, Mike Lopez, Ken Sutton, and Tony Clear. 2009. Surely we must learn to read before we learn to write!. In *Conferences in Research and Practice in Information Technology Series (ACE '09)*, Margaret Hamilton and Tony Clear (Eds.), Vol. 95. Australian Computer Society, 165–170.

[73] Elliot Soloway. 1986. Learning to program = Learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.

[74] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860.

[75] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2 (2013), 1–31. https://doi.org/10.1145/2483710.2483713

[76] Juha Sorva and Arto Vihavainen. 2016. Break Statement Considered. *ACM Inroads* 7, 3 (Aug. 2016), 36–41. https://doi.org/10.1145/2950065

[77] James C Spohrer, Elliot Soloway, and Edgar Pope. 1985. A goal/plan analysis of buggy Pascal programs. *Human–Computer Interaction* 1, 2 (1985), 163–207.

[78] Kristian Stålne, Michael Lamport Commons, and Eva Yujia Li. 2014. Hierarchical complexity in physics. *Behavioral Development Bulletin* 19, 3 (2014), 62.

[79] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 19.

[80] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[81] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review* 22, 2 (2010), 123–138.

[82] Donna Teague and Raymond Lister. 2014. Programming: reading, writing and reversing. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 285–290.

[83] Ian Utting, Dennis J. Bouvier, Michael E. Caspersen, Allison Elliott Tew, Roger Frye, Yifat Ben-David Kolikant, Mike McCracken, James Paterson, Juha Sorva, Lynda Thomas, and Ta Wilusz. 2013. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the 2013 ITiCSE working group reports (ITiCSE -WGR '13)*. ACM, 15–32. https://doi.org/10.1145/2543882.2543884

[84] Jeroen JG Van Merriënboer and Paul A Kirschner. 2017. *Ten steps to complex learning: A systematic approach to four-component instructional design*. Routledge.