Modelling Expert and Novice Programming Strategies using Python ACT-R

Maria Vorobeva, Robert West, Kasia Muldner
Department of Cognitive Science
Carleton University, Ottawa, Canada

Abstract

Cognitive architectures have been used to model human problem-solving strategies and behaviours in complex domains – here, we focus on programming. However, to date, models of programming have not included various strategies for generating programs. To address this, the present paper describes two cognitive models that simulate a novice and expert strategy for solving a programming problem in Python. The models were based on theoretical frameworks of expert and novice programming. The SGOMS framework was best for modeling experts and competent novices, because it provided functionality to represent goals and plans that mirrored ones used by these individuals.

Keywords: Python ACT-R, cognitive modelling, programming, expertise

Introduction

Programming is a complex skill that requires time and practice to master. To date, however, the components of this skill and corresponding cognitive mechanisms are not clear. As we describe below, one way to fill this gap involves the construction of a cognitive model.

A cognitive model is a formalization of cognitive mechanisms that are hypothesized to impact problem solving and performance within a particular domain. A common cognitive architecture for building cognitive models is ACT-R (Anderson & Lebiere, 1998). ACT-R uses productions (if/then rules) to model problem solving in a given domain, and declarative memory to store facts about the domain. This architecture has inspired other similar architectures, such as Python ACT-R (Stewart & West, 2007), which is used in the present work. Implementing a cognitive model has a number of benefits. It requires the human author (i.e., the model builder) to formally specify the declarative and procedural knowledge needed to solve problems in a given domain. This formalization step is beneficial as it clarifies the cognitive mechanisms (Frischkorn & Schubert, 2018). Moreover, the model provides an environment for testing theories about the hypothesized cognitive mechanisms.

There is substantial work in the ACT-R community and beyond involving cognitive models for a range of tasks. In this review, we focus on problem-solving tasks in science domains. Two common domains used to implement cognitive models include physics and math (Braithwaite et al., 2017; VanLehn et al., 1991). To illustrate, the model Cascade formalized the mechanisms for self-explanation and analogical transfer used during physics problem solving (VanLehn et al., 1991). Another example is FARRA (Braithwaite et al., 2017), which is a model of fraction problem solving. FARRA simulations demonstrated that the distribution of problems in mathematics textbooks may inadvertently strengthen student misconceptions. Relevant to the present work, some researchers have formalized knowledge representations for program generation (Johnson & Soloway, 1985; Pirolli, 1986; Corbett, 2000). The focus of this work was to parse and/or track students' code generation and provide feedback on program statements. To date, however, there does not exist work on implementing cognitive models of programming that simulate different strategies based on a programmer's knowledge (novice vs. expert).

The present paper takes a step towards filling this gap. Specifically, we identify the knowledge representations needed to program and embed them within computational ACT-R models capable of producing solutions to simple programming problems. Programming was chosen as the domain as it represents a complex problem-solving skill, with competing frameworks providing insight into the process programmers engage in while writing programs. Two models are implemented that simulate a novice and expert approach, respectively, to programming.

Novice and Expert Programming Approaches

Since the models we implemented are influenced by theories of expertise, we begin with a brief overview of these, focusing on work in the programming domain. Programmers' mental representations are pivotal to programming performance and ability. For instance, programmers find it easier to read and understand the output of a program when the language uses functions that align with the programmer's underlying problem-solving strategy (Soloway et al., 1983). Of particular interest for the present work are studies investigating programming expertise (Spohrer et al., 1985; Soloway & Ehrlich, 1984).

Early programming frameworks characterizing novice programming focused on identifying the origins of common bugs in novice programmers' code. Spohrer et al. (1985) used a representational framework called GAP trees (Goal and plan networks) to parse the programs of novice programmers, categorize bugs, and identify the problem-dependent knowledge that led to bugs. The GAP framework decomposes a program using a solution space containing a program's goals, and the set of plans that implement those goals (e.g., through decomposition into smaller goals and

plans). Spohrer et al. referred to this solution space as a GAP tree (goal-and-plan tree). There are two types of GAP trees for programs: (1) inferred trees, defined as having goals with multiple executable plans, and (2) solution subtrees, which are branches in the larger inferred GAP tree linking a single execution plan to a goal. Students who were not able to correctly complete programming tasks usually had an error in, or the complete absence of, one or more of the GAP tree components. This suggests that novice errors are caused by missing goal(s), or by incorrect knowledge representation(s).

Rist (1989) also studied novices, by analyzing the program-generation process of 10 novices to identify how they used simple programming plans to compose larger, more complex plans. In this study participants were asked to solve programming problems on paper while thinking out loud during their problem-solving process. Similar to Soloway's (1986) conceptual framework, Rist analyzed novice use of goals and plans, codeding the transcripts according to the plans implemented and their order of implementation. The findings showed that novice programmers used the primary goal of a problem to try and identify a set of known, basic programming plans that could be combined to resolve the goal. Novices first identified a *plan focus*, which is the first expression or line of a programming plan that is implemented; the plan focus served as the anchor for a given programming plan. Once the plan focus was implemented, the remainder of the plan was expanded around it (referred to as *program expansion*).

Soloway (1986) used the results of prior studies (Soloway and Ehrlich, 1984; Spohrer et al., 1985) to develop a conceptual framework describing expert programmers' problem-solving approaches. Soloway proposed that expert programmers first obtain an understanding of the goal and plan structure of the problem i.e., develop a rough GAP tree. Experts then use stepwise refinement, which is the breakdown of a problem on the basis of simpler problems the programmer has already solve; the solutions for the simpler problems provide the solutions to create the solution to the current problem. Soloway's framework proposed that novices have difficulty identifying the goals needed to solve the problem, as well as face difficulties recalling appropriate plans needed to implement the goals. In contrast, expert programmers use plan composition to combine the fragments of canned solutions into a final solution plan.

Overall, work described thus far suggests that a key difference between expert and novice programmers relates to the ability to generate plans (i.e., algorithms in the programming domain). Novices are unable to generate a plan either because they lack key information or because they are unable to link programming steps together.

Computational Models of Programming

Prior work has used ACT-R to create cognitive models capturing processes related to programming. For instance, the ACT-R Programming Tutor (APT), developed by Corbett (2000), can write small programs. APT engages in both knowledge tracing and model tracing. Knowledge tracing is used to assess the probability that a student has successfully learned a rule based on application of the rule. For model tracing, the tutor uses an underlying production system, called its ideal student model, which contains the full set of rules to solve all of the practice problems. For each student input, once the student has selected their next goal and next step, the model tracer generates all possible correct next steps and compares these to the student's input. If the student input is correct, problem solving proceeds to the next goal-step combination. If the student's input does not match any of the model's steps, the tutor provides feedback and encourages the student to correct the mistake. The model-tracing component can write the small programs as it has the relevant productions, but it does not taken into account programming strategy.

Soloway's conceptual framework of programming plans discussed above does not formalize plans within a computational model. This was partially addressed by PROUST, a model built by Johnson and Soloway (1985), which could identify strategies in programs students wrote. PROUST took as input finished student programs and parsed these programs by identifying the strategy/goal decomposition used in the program. PROUST used its knowledge base of programming plans, strategies, and bugs to map out the solution path. This allowed PROUST to parse a program and identify deviations from the expected programming plan. Thus, PROUST could identify buggy programs and diagnose the source of the bug(s). While this model could identify strategies used to write a program, it was not designed to write programs.

In sum, to the best of our knowledge, there does not exist a computational model that takes into account programmers' strategies to write programs or that models the differences between expert and novice programmers.

Present Work: Cognitive Models of Programming

We now describe two ACT-R models we implemented, called the *goal expansion* model and the *SGOMS* model. Each model aims to produce a solution to a basic programming problem using the programming language Python – one model simulates a novice approach to solving the problem and the second model an expert approach. Both models solved the rainfall problem, which requires calculating the average of all the positive numbers (including 0) in a list of daily rainfall amounts, and to stop processing the list if a value of -999 is encountered.

We obtained data on the impact of expertise on programming strategies from a case study we conducted (Vorobeva & Muldner, 2022). In the this study, 12 novice and 7 expert programmers were asked to solve the rainfall problem. While they worked on the problem, participants were asked to think-out loud by verbalizing their thoughts, so that data could be obtained on their reasoning and strategies. The data was analyzed using a qualitative approach to identify participants' goals and problem-solving approaches,

and subsequently informed the design of the two cognitive models we present here that were implemented to solve the same problem.

The two models were implemented using Python ACT-R. Like ACT-R, Python ACT-R distinguishes two types of memory, declarative and procedural. Declarative memory stores information using *chunks*. In the present context, chunks include both steps (here, lines of Python code) and goals representing higher-level strategies. The declarative memory represents information that is known but not immediately actionable. In contrast, the procedural memory encodes *productions*, which are if/then statements that perform actions when their preconditions are met. The preconditions correspond to chunks in the declarative memory. These productions are used to generate the Python program "steps" (program lines), as will be described shortly.

Model Components: Overview

As noted above we implemented two models for simulating programming performance. In our framework, a model corresponds to the set of productions that define the expert and novice problem-solving approaches (the nature of the differences between the models will be discussed in the next section). The productions rely on information chunks stored in the declarative memory and *module buffers* (described below), for their preconditions. While the two models simulate different problem-solving strategies (novice vs. expert), they both rely on the same *modules*. Modules are specialized components in Python ACT-R, specifying distinct functions of the mind (Stewart & West, 2007). A given model within Python ACT-R may rely on a number modules to help carry out its problem-solving process within the environment. Modules exist outside of the model and are called upon by the model using the appropriate buffer that relays commands from the model to the appropriate module (and may also relay information from the module to the productions, as is the case for the DM module). The modules used by both models include (a) the motor module; (b) the environment; and (c) the declarative memory module (see Figure 1 for a visual of the modules and their relations). We now describe the modules and related buffers.

The motor module writes the Python program to a file and produces a log of the program goals and steps. Thus, the log shows a detailed trace of the problem-solving process. The motor module has a corresponding motor buffer (see Figure 1), which is used by the model's production to control the motor module's behaviour.

The environment module contains the description of the rainfall problem for each model. The environment is the same for the two models. Information from the environment is not mediated through a buffer.

The declarative memory module is a general component of the Python ACT-R architecture. The model uses a buffer to communicate with the declarative memory (see *DM Buffer*, Figure 1), and can use the buffer to add chunks to the declarative memory (e.g., reflecting new goals identified) or retrieve chunks from memory. Sometimes the declarative

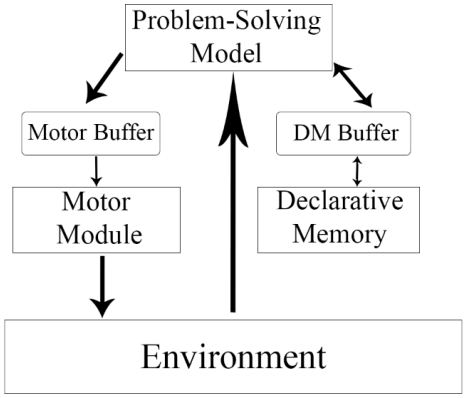

Figure 1: Key system components. The arrows represent the direction of information flow. Problem-solving models receive information directly from the environment, send instructions to the motor module to implement actions in the environment (via the motor buffer) and share information bi-directionally with the declarative memory (through the DM buffer) to manage the problem solving process.

memory may make mistakes and fail to retrieve facts, or retrieve an incorrect fact that matches some of the query terms. This reflects that people will not always correctly recall information. During such events the model will be redirected to repeat the retrieval – it will retrieve the correct fact with sufficient attempts. Additionally, in cases where multiple facts match the query terms, retrieval is determined using a probabilistic calculation, where the association strength of the fact (a measure of how often the fact is retrieved) determines its probability of being retrieved. While both models use the same declarative memory module, they are initiated with different information within.

Both models are initialized with a focus buffer, which tracks where the model is in the program-generation process. Unless otherwise stated, the focus buffer holds chunks corresponding to the primary preconditions that must be met for a production to fire.

While both models produce code and corresponding goals associated with the solution process, neither model is capable of *learning* new chunks or productions, i.e., the models do not infer new algorithms or programming syntax through experience. Instead, the models are initialized with this information by the human model builder (and so for the present work, each of the two models' declarative memory was initialized prior to problem solving). Both models do add goals to the declarative memory, but they do not reflect learning of new goals as the goals are generated by the model's productions and thus already exist within productions of the model (though unspecified for the problem at hand).

Two ACT-R Models for Program Generation

Goal Expansion Model This model is inspired by Rist's (1989) framework characterizing novice programming and uses the goals and steps identified in Vorobeva and Muldners' (2022) study. In this framework, novices first identified a plan focus, and then expanded the plan focus by implementing the program steps (e.g., lines in the Python program). However, unlike the Rist framework, our goal expansion model can identify several goals (more than one) directly from the problem statement, based on *keyword – goal* associations – this translates the problem text into high-level goals. Goals correspond to the intention to perform a high-level programming action, needed to solve the problem; for example, calculating the average rainfall. The model expands these goals into related goals based on *goal – goal* associations. For example, once the model generates the goal to calculate the average rainfall, it generates the related goal to initialize the variables needed for the average calculation.

Like the behavior of novices in prior work, the goal expansion model does not generate a high-level algorithm that orders the goals and steps in advance. Instead, the model identifies and addresses goals in the order it retrieves the relevant goal associations. Once a goal is generated by the model, either from reading the problem statement or after goal expansion from one of the keyword-associated goals, the chunk representing the goal and associated step is retrieved from the declarative memory and stored in the DM buffer. The retrieved chunk stored in the DM buffer satisfies the precondition for the firing of the production that implements the step (i.e., line of Python code) that resolves the goal. The step-implementing production sets the model's focus buffer to contain the precondition used by the production that generates other related goals, i.e., directs the model to engage in goal expansion. If a goal is generated using a *goal - goal* association, it goes through the same implementation process as described above; it will subsequently be used to check for further goal-goal associations.

SGOMS Model SGOMS is a cognitive framework that adds planning units and unit tasks to ACT-R in order to model complex behavior (West & Pronovost, 2009; West & Nagy, 2007; West & MacDougal, 2015). Planning units represent goals, such as *calculating average rainfall* and *initializing variables*, and reflect the goals identified during the coding of the participants' verbal protocol and written program in Vorobeva and Muldner (2022). Planning units are used to initialize the model's declarative memory at the start of problem solving and structure the problem-solving process. Each planning unit is composed of unit tasks that must be completed to resolve the planning unit; collectively. The unit tasks for a given planning unit will be referred to as a sub-algorithm. Unit tasks can either define high-level operations or implementational-level operations.

Implementational unit tasks are what the model uses to control implementation of the step, and reflect actions that must be taken to implement the code, as well as to make the written code fit with the rest of the programmed solution. High-level unit tasks are used to implement a planning unit hierarchy by allowing planning units to call upon other planning units as part of the initial planning unit's sub-algorithm. This reflects that the resolution of some goals requires the resolution of other goals, and that this creates a sort of goal hierarchy. When a high-level unit task calls another planning unit (when the planning unit goal requires another goal to be resolved), it redirects the model to that new planning unit and this unit must be completed first. Once the called upon planning unit is complete, i.e., the unit tasks that defines its sub-algorithm have all been completed, the model redirects to the next unit task of the calling planning unit. For example, the *calculate average rainfall* planning unit has as its first unit task to call upon the *initialize_variables* planning unit. This redirects the model to resolving the *initialize_variables* planning unit (by writing the code to initialize the variables) before continuing to the next unit task, namely the *calculate_average* planning unit.

The model begins program generation by calling on the highest-level planning unit relevant to the problem. For the rainfall problem, this is the *calculate average* planning unit. This planning unit is considered the highest level as it defines a sub-algorithm for the primary goal stated in the problem statement (to calculate the average rainfall). The sub-algorithm includes unit tasks for both high-level (productions requests to other planning units) and implementational-level productions (requesting variables / conditions and implementing steps). The *calculate_average* planning unit will first require the completion of two other planning units (*initialize_variables* and *iterate_loop).* However, as described above, the called upon planning units may themselves call additional planning units, such as the *iterate_loop* planning unit calling upon the *stop_loop* and *track_variables* planning units. When a planning unit is complete, it directs the model to the next unit task in the planning unit that called it. The program is complete when the highest-level planning unit implements its final unit task; for the rainfall problem this corresponds to the expression that calculates the average in the Python program.

By using planning units to organize information, the program-generation process is guided, but without the need to generate the entire algorithm in advance. In this way the SGOMS model more closely mimics the behavior shown by experts and competent novices in our study (Vorobeva and Muldner, 2022). By relying on planning units instead of a pre-canned algorithm, the model has the ability to recombine the planning units to generate different solutions. This reflects the ability of the SGOMS model to be flexible with its treatment of goals. For example, the SGOMS model is currently capable of generating a simple loop function that sums and counts all of the numbers in a list but that does not give an average for the positive numbers.

SGOMS Model vs. Goal Expansion Model As is the case with the SGOMS model, the goal expansion model is not given a complete algorithm up front. The goal expansion model relies on associations in its declarative memory to generate the goals, but can not specify the exact relationship

between associated goals. Consequently, the goal expansion model has trouble implementing steps in a coherent order. In contrast, the SGOMS model has a concrete structure and hierarchy to the goals that is well defined before problem solving. However, it does not have a *pre-existing* complete algorithm that defines the implementation of the total solution. For example, the planning unit to *calculate average rainfall* initializes the planning units for *iterating the loop* and *initializing the variables*. However, the *calculate average rainfall* planning unit does not define which planning units need to be initialized by the other planning units. Therefore, each planning unit functions semi-independently, and can be called upon by any number of other planning units, as long they are defined by the modeler or by learning mechanisms in advance. In this way planning units may be recombined to generate solutions to new problems, something the goal-expansion model would struggle with.

Simulation of Program Generation via each Model

As described above, both models were implemented using Python ACT-R and initialized with the specification of the rainfall problem. When we ran each model to simulate the problem-solving process by an expert (SGOMS model) and a novice (goal expansion model), the SGOMS model was able to produce a correct solution but the goal expansion model was not. We now describe each model's problem-solving process and output.

Goal Expansion Model Figure 2 shows the output for the goal expansion model. The model was able to form varied solutions to the rainfall problem, because there was no set order of how to address the goals. However, it did not generate a correct program (possibly with sufficient runs it would accomplish it by chance). Specifically, the model had difficulty correctly ordering the program steps (recall that a step corresponds to a single line of Python code). For example, it identified the goal to calculate the average (Figure 2 line 1) and then wrote the line to the top of the Python file to accomplish the goal (Figure 2 line 2). However, the variables that were needed to calculate the average had not yet been initialized or incremented within the loop function (done in Figure 2 lines 4 and 6 respectively). Therefore, the written program would be unable to go through the program at all as it would not have anything assigned to the variables when asked to calculate the average. In general, the goal expansion model currently generates solutions based on the order of keywords it extracts from the problem statement. Thus, adding more refined NLP functionality is needed to appropriately assess its validity as a model of novice programming.

The model produced *some* of the behavior Rist (1989) attributed to novices. It identified goals from the problem statement, and engaged in program expansion to add additional goals and steps. This allowed the model to connect the steps of iterating through the list (a and stopping the loop (Figure 2 lines 7-10). By expanding from a keyword – goal identified plan focus (in this example the keyword – goal plan focus was list - iterating the list), it correctly connected the steps together, but was unable to connect both expressions to the broader problem statement of calculating the average. However, the model was also more sporadic in terms of the ordering of its solution goals / steps. We discuss potential reasons for this in the discussion.

```
1.Goal: [calculate average] I need to: calculate_average
2.Step: < calculation > average = total/count
3.Goal: [initialize variables count/total] I need to: initialize_variables
4.Step: < initialize variables > total= 0, count= 0
5.Goal: [track count/total] I need to:track_variables
6.Step: < condition > if x >= 0:
        < increment variables > count+=1, sum+= x
7.Goal: [iterate through list] I need to: loop_iterate
8.Step: < iterate loop > for x in rains:
9.Goal: [stop loop] I need to: stop_loop_iterate
10.Step: < stop loop > if x == -999: break
```

Figure 2: Log of Goal Expansion Model's Problem-Solving Process

SGOMS Model Figure 3 shows the output from the SGOMS model. The SGOMS model was able to successfully construct the canonical solution as well as a complete algorithm specific to the problem (note the complete algorithm was not provided to it *a priori*). Additionally, it was able to replicate findings from Vorobeva & Muldner (2022) of experts identifying multiple goals before implementing them (Figure 3 lines 1 and 2), though this ability was restricted to the main goal of calculating the average rainfall. This goal is identified at the start but not implemented until the very end (Figure 3 lines 1 and 10).

Discussion

The aim of the present work was to leverage earlier work on expert and novice programmers' problem solving to develop models capable of program generation. Specifically, the common goals and steps we identified in both expert and novice solutions (Vorobeva and Muldner, 2022) were used to create the declarative knowledge chunks (goals - step) and productions that implemented the step of the solution. Earlier models such as PROUST and APT were capable of processing programs but were unable to write whole solutions for a programming problem. APT had the knowledge base to write small snippets of code, that is expressions that would address one goal of an overall problem, but was unable to chain them together into a complete final solution. While our models are limited in scope in terms of their capacity to solve a range of programming problems, they are capable of identifying and implementing multiple goals and linking them together to provide an overall solution pathway (albeit not a correct one in the case of the goal expansion model).

We expected novices to be best modelled by the goal expansion model, which reflected Rist's (1989) framework of a novice approach to problem solving. Rist argued that

1.Goal: **[calculate_average]** I should calculate the average of the positive numbers
2.Goal: **[initialize variable total/count]** I should initialize the variables sum and count to track the positive numbers
3.Step: < *initialize variables* > count= 0, sum= 0
4.Goal: **[iterate through list]** I should iterate through the list
5.Step: < *iterate loop* > for x in rains:
6.Goal: **[stop loop]** I need to stop iterating the loop when I hit the first -999 in the list
7.Step: < *stop loop* > if x == -999:break
8.Goal: **[track total/count]** I should track the positive numbers in the list using the sum and count variables
9.Step: < *condition* > if x >=0:
    < *increment variables* > count+=1, sum+= x
10.Step: < *calculation* > average = count/sum

Figure 3: Log of SGOMS Model's Problem-Solving Process

novices use a *plan focus*, that is a core step that is written first in the program, and *program expansion* to expand the *plan focus* by implementing additional steps which supported the *plan focus* step. The goal expansion model was able to identify multiple plan focuses from the problem statement using the *keyword – goal* associations in its declarative memory, and expand around those plan focuses with relevant goals. However, many novice participants in our study (Vorobeva & Muldner, 2022) did not rely on or use a plan focus as predicted by the Rist framework. Hence neither the Rist framework or the goal expansion model accurately modelled all of the novices.

Differences between the performance of some novices and the model of novice behaviors (i.e., the goal expansion model) may be due to the simplicity of the rainfall problem not requiring a more generative problem-solving process by not requiring many goals and steps. Additionally, the model's lack of a revision and reflection mechanism made it difficult to engage in program expansion, as the model could not write steps that would precede other already written steps. For example, the model was not able to initialize variables at the top of the file if it had already implemented the loop that incremented them. Thus, the goal expansion model was limited to only depicting strict forward expansion, where *program expansion* would follow the same order as the final working solution. In earlier work, Byckling and Sajaniemi (2006) found that strict forward expansion occurred only in more competent novices, and thus the goal expansion model is limited as a model of all novices.

The SGOMS model best represented the performance of the experts and competent novices, as it produced output that showed the greatest degree of similarity to the outputs produced by experts (and some novices) as determined by a qualitative analysis. It replicated some of the expert's behaviours, such as identifying multiple goals in a row (without step implementation), thus demonstrating some pre-planning capabilities.

While the models were informative, there are various improvements we are working on. For instance, the models could benefit in terms of validity if they had productions capable of reflecting on and revising the programs written (this would allow the model to insert written code in between or in front of existing lines of already written code, as needed). This problem with the lack of reflection and revision is most apparent in the inability of the models to capture the novice and expert ability to engage in backwards program expansion (the models are only capable of strict forward expansion). This could be implemented through the expanded use of planning units in the future. Moreover, given that the present models have only been tested on a single problem, more work is needed to test and generalize them with a range of problems.

In the future it might also be beneficial to extend the model with the ability to construct a GAP tree of the type described in prior work (Spohrer et al., 1985; Soloway 1986). A GAP tree would allow the model to have a high-level representation of the overall problem and the current state of the problem and would supplement the existing hierarchical structure of the planning units. For problems more complex than the rainfall problem, the current version of the SGOMS model may have difficulties managing more complex arrangements of goals/planning units and determining the best implementation order. One potential solution to this issue could be to add an additional buffer that constructs and tracks a hierarchy tree of goals. Another possibility is the addition of declarative knowledge of how to best arrange multiple subgoals during implementation (such as ensuring that the step for variable tracking is always written within the loop iterating the relevant list) to the declarative memory.

In spite of these limitations, the models presented in this paper matched some of the novice and expert performance from our study (Vorobeva and Muldner, 2022). Additionally, they were capable of capturing various problem-solving strategies from the study conducted and prior research.

Acknowledgments

This work was supported by an NSERC Discovery Grant (#1507).

References

Anderson, J. R., & Lebiere, C. (1998). The atomic components of thought. Lawrence Erlbaum Associates.

Braithwaite, D. W., Pyke, A. A., & Siegler, R. S. (2017). A computational model of fraction arithmetic. *Psychological Review*, 124(5), 603–625.

Corbett, A. (2000). Cognitive mastery learning in the act programming tutor. In *Adaptive User Interfaces. AAAI SS-00-01*.

Frischkorn, G. T., & Schubert, A. L. (2018). Cognitive models in intelligence research: Advantages and recommendations for their application. *Journal of Intelligence*, *6*(3), 34-56.

Johnson, W. L., & Soloway, E. (1985). PROUST: Knowledge-based program understanding. IEEE Transactions on Software Engineering, 3, 267–275.

Pirolli, P. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction*, *2*(4), 319-355.

Rist, R. S. (1989). Schema Creation in Programming. Cognitive Science, 13(3), 389–414.

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 37–39. https://doi.org/10.1109/HCC.2002.1046340

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11), 853–860. https://doi.org/10.1145/182.358436

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, (5), 595-609.

Spohrer, J. C., Soloway, Elliot, & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human–Computer Interaction*, 1(2), 163–207.

Stewart, T. C., & West, R. L. (2007). Deconstructing and reconstructing ACT-R: Exploring the architectural space. Cognitive Systems Research, 8(3), 227–236. https://doi.org/10.1016/j.cogsys.2007.06.006

VanLehn, K., Randolph, J. M., & Chi, M. T. H. (1991). Modeling the Self-explanation Effect with Cascade 3. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, 132–137.

Vorobeva, M., & Muldner, K. (2022). Investigating Expert and Novice Programming Problem Solving. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, 44.

West, R. L., & Pronovost, S. (2009). Modeling SGOMS in ACT-R: Linking Macro- and Microcognition. *Journal of Cognitive Engineering and Decision Making*, 3(2), 194–207. https://doi.org/10.1518/155534309X441853

West, R. L., & Nagy, G. (2007). Using GOMS for modeling routine tasks within complex sociotechnical systems: Connecting macrocognitive models to microcognition. *Journal of Cognitive Engineering and Decision Making*, *1*(2), 186-211.

West, R. L., & MacDougall, K. (2014). The macro-architecture hypothesis: Modifying Newell's system levels to include macro-cognition. *Biologically Inspired Cognitive Architectures*, *8*, 140-149.