Modelling Programming Problem Solving in Python ACT-R

by

Maria Vorobeva

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Cognitive Science

in

Masters of Cognitive Science Program

Carleton University
Ottawa, Ontario

Abstract

Cognitive architectures such a Python ACT-R have been used to model human problem-solving strategies and behaviours in complex domains such as programming. However, to date, models of programming have not investigated various strategies for generating programs. To address this, the present thesis describes the construction of five cognitive models that represented different novice and expert strategies for solving a programming problem in Python. To aid in the design of the models, I conducted a talk-aloud study with expert and novice programmers. The models use a set of goals and steps that were identified in the study transcripts and solutions produced by the programmers in the study. Expert and competent novices were best modelled by the model utilizing an SGOMS framework. The SGOMS framework incorporated the ability to formalize the relationships between goals of the problem and allowed the model to structure the solution in the same way as experts and competent novices.

Acknowledgements

Thanks to my friends and family who supported me through this process. Special thank you to Dr.Kasia Muldner, without whom this thesis would not have been possible. I'd also like to thank Dr.Robert West and the Carleton Cognitive Modelling Lab for all of their help and support in designing the models. Finally, I'd like to thank the Institute of Cognitive Science for providing an incredible environment to conduct this research in.

Table of Contents

List of Tables

List of Illustrations

List of Appendices

Chapter 1: Introduction

The present thesis describes the construction of five cognitive models that represented different novice and expert strategies for solving a programming problem in Python. 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 models is ACT-R (Anderson & Lebiere, 1998). ACT-R uses productions (if/then statements) to model problem solving in a given domain, and declarative memory to store information. This architecture has inspired other similar architectures, such as Python ACT-R (Stewart & West, 2007), that is used in the present thesis. 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 in a given domain. This formalization step is beneficial as it clarifies the cognitive mechanisms in a given domain (Frischkorn & Schubert, 2018). Moreoever, the model provides an environment in which to test these theories about the hypothesized cognitive mechanisms.

There is substantial work in the ACT-R community and beyond on building cognitive models for a range of tasks. Physics and math problem solving are common domains for cognitive models of problem solving (Braithwaite et al., 2017; VanLehn et al., 1991). To illustrate, Cascade formalized mechanisms for self-explanations analogical transfer as the drivers of physics problem solving (VanLehn et al., 1991). Another example is FARRA (Braithwaite et al., 2017) a model of fraction problem solving. The model was used to demonstrate that the distribution of problems in mathematics textbooks may inadvertently strengthen student misconceptions in regards to fraction

division. Relevant to the present work, other models formalized knowledge representations used in programming (Johnson & Soloway, 1985; Pirolli, 1986; Corbett, 2000). Such models have been effective in identifying bugs within student written programs but do not have the ability to write complete programs. APT is close, in that it can write small snippets of code, but it cannot write a complete solution to a programming problem like my models can. In general, to date, there does not exist work on implementing cognitive models of programming that simulate different programming strategies based on a solver's knowledge (novice vs. expert).

My thesis takes a step towards filling this gap. Specifically, I formalize the knowledge representations needed to program within a computational model capable of producing solutions to simple programming problems. Programming was chosen as the target domain as it represents a complex problem-solving skill, with competing frameworks providing insight into the process programmers engage in while writing programs. I implement a series of models to account for individual differences by implementing both novice and expert strategies in these models. To provide insight on model design, I first ran a talk-aloud study in which participants solved programming problems while verbalizing their thoughts and subsequently analyzed the data using a qualitative approach. This data helped shape the model design.

Before presenting the study and the models, I describe a representative sample of the related work.

Chapter 2: Literature review

2.1 Expertise

The models I implement embed different expert and novice problem solving strategies. To provide background on this, I begin with a review of the expertise literature. There are two common ways of characterizing expertise: (a) as a label for exceptionally gifted or talented individuals; or (b) as a state achievable through sufficient education and practice (Chi, 2006). The latter definition is the one I rely on in the present thesis.

In other, non-programming domains, novice and expert performance differences have been attributed to differences in representational knowledge. In their seminal work using the domain of physics, Chi et al. (1981) showed through a series of studies that experts do not simply have more knowledge than novices, but that they hold inherently different representations than novices. In the first study, they asked novices and experts to categorize physics problems by similarity. Novices broadly categorized problems by the superficial features describing the problem situation, such as whether the problem had frictional forces or whether it included inclined planes. In contrast, experts primarily categorized problems by the major physical principles needed to solve the problem, such as conservation of energy and principles of momentum. While novices did not always categorize problems only by superficial features, when they used physics principles to categorize, they were less robust and reliable than the principles used by experts, and relied heavily on the superficial features to guess at the underlying principles. This held true in a follow-up study, that presented twenty new problems that had their superficial features and underlying physics laws crossed. Specifically, the problems included ones

that looked superficially similar but required different underlying physics principles to solve, as well as problems that were superficially dissimilar but required the same underlying physics principles to solve. Experts took longer to categorize the problems, engaging with the problem at a deeper level in order to successfully categorize it. These results showed that not only do novices and experts use different features when classifying physics problems, but that they have different knowledge structures, with experts interpreting the problems according to their deep knowledge of physics principles (what Chi et al. referred to as schemas).

Chi et al. (1981) conducted a third study to determine the nature of the representations. They took the problem categories produced by novices and experts from the first two studies, and asked a new set of novices and experts to describe everything they know about each category. Both experts and novices could describe and elaborate on the superficial aspects of a problem category, such as setting up the equations, and the various objects and forces that may be relevant for that style of problem. However, only the experts could elaborate on underlying physics principles, such as conservation of energy or Newton's force laws. A final study collected talk-alouds of experts and novices describing a basic solution approach to a series of physics problems. Experts would propose a hypothesis around a major principle or physics law early in the problem-solving process that they then used to guide the generation of the solution. Experts also extracted additional features from a problem statement that were used to further refine the problem-solving approach, namely to narrow the search zone in the problem space. Novice problem-solving proposals focused on superficial features. Their approach was guided by surface level schemata containing algebraic formulas with slots related to the

variables of the problem; novices focused on finding numeric values in the problem statement to fit into the slots. These studies demonstrated that experts and novices differ in terms of the contents of their knowledge representations, and that these differences in representation are what determine their differences in problem categorization.

Leonard et al. (1996) investigated the role of conceptual knowledge and its relation to strategy use during physics problem solving. They proposed that the conceptual knowledge required to solve a physics problem was comprised of three components: (1) the major principles and concepts applicable, (2) rationale for the principle's or concept's applicability, (3) an application strategy for principles and concepts. These three components form the basis of the qualitative description for solving a given problem, and are necessary for defining a strategy that includes the *what*, *why* and *how* of a solution approach. Leonard et al. hypothesized that novices taught in a traditional way that did not include instruction on these components tended to view physics problem solving as the memorization and use of equations. Consequently, novice strategies would rely heavily on matching values in the problem description directly to equations, without appealing to any major principals or schema justifying the formula. In contrast, experts would identify and justify their solutions by appealing to their conceptual knowledge, including the major principles and laws that broadly applied to that category of problem, and formulating a strategy on the basis of that conceptual knowledge. Leonard et al. showed that expert strategies can be explicitly taught in a traditional course environment. Specifically, learning from two different physics classes was compared. In the standard classroom a traditional education plan was used, and problem solving was demonstrated by the instructor without discussing the three

components of strategy. In the revised "strategy" classroom, the instructor provided information about the expert approach while problem solving, by first explaining the three components of the relevant strategy and then demonstrating its implementation. Students in the strategy course performed better on a problem categorization task, by categorizing more like experts and recalling more of the major principles when asked about it after 11 months had passed. Thus, this study shows that there is a benefit to teaching the strategies and schemas held by experts.

Differences in mental representation between experts and novices have also been demonstrated in other domains such as chess. Gobet and Simon (1996) found that expert chess players were better than novices at remembering chess board positions presented by the experimenters, but only if the board represented a board state attainable in a game. If the pieces were distributed randomly, experts were no better than novices at recalling the board states. This study showed that experts do not have greater memorization abilities (as they performed as poorly as novices when pieces were distributed randomly), but rather that they rely on mental representations of meaningful board configurations to structure their memory and recall.

In summary, the studies described above demonstrated that there are qualitative and quantitative differences between novices and experts in multiple domains in terms of the mental representations used to guide problem categorization and problem solving.

2.2 Programming Expertise

While much of the research on expertise focused on physics, algebra, or other similar problem-solving domains, there are also studies investigating the expert and novice knowledge structures and representations in the programming domain.

Programmers' mental representations are pivotal to programming performance and ability. For instance, programming languages are more comprehensible when they 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; Kahney, 1983; Détienne, 1990)

Kahney (1983) showed that novices' representations of recursion had gaps in comparison to experts' representations. Their study investigated whether novices and experts could properly identify all correct solutions to the Kissing Flu problem. Both novice and expert programmers were shown two correct solutions to the problem, which required converting a data structure tracking who kisses who into a data structure of who gave whom influenza via kissing. Participants were asked to identify the solutions that would convert the data structure correctly and to explain the choice(s). Novices could correctly identify solution 1 as a possible solution, but only experts reliably identified solution 2 as also being correct. The researchers concluded that in contrast to experts, novices do not fully understand recursion. Instead, the novices relied on a faulty mental model the researchers referred to as the loop model, where they viewed recursion not as a series of retriggered instances, but as a singular object looping through the database.

Ye & Salvendy (1996) investigated how novices and experts differed in their programming knowledge categorized here by five levels of abstraction; they did this with a 20 question test that had four questions for each level of abstraction. They found the knowledge distinction between expert and novices was evident across multiple levels of abstraction, but differences were greatest at the functional and conceptual levels (which they claimed was analogous to abstract knowledge). These levels corresponded to

language-independent knowledge of programming functions and object forms. Despite this promising start the five levels of abstraction were not formalized into concrete cognitive mechanisms, making it difficult to adapt their framework into a computational model.

Soloway and Ehrlich (1984) identified two representations related to programming expertise, goals and plans, through two studies. Goals here were high-level requirements necessary to resolve the problem. Goals may be resolved by programming plans with code, or as the researchers called it 'canned solutions'. Study 1 used a program completion task where both novice and expert programmers had to complete a missing line of code in a program using only the rest of the program as context (so with no description of the program's problem or stated goal). They found that experts were better able to generate the missing code fragment than novices, and argued this was due to a schema they had for that type of program. The schema allowed experts to infer goals needed to solve the problem. In study 2, a different task was used, namely a program recall task, where participants had to recall as many lines of a program as they could after seeing it briefly. Soloway and Ehrlich found that experts were better able to recall lines of the program. The proposed explanation for these findings was that experts were able to store program lines in meaningful chunks (i.e., programming plans or schemas). This finding mirrors work in other domains. For instance, as already described above, Gobet and Simon's (1996) work showed that showed that chess experts were better able to recall meaningful board configurations than novices, because they use chunks and strategies to structure their memory and recall.

Spohrer et al. (1985) used a representational framework called GAP trees (Goal and plan networks) to parse novice programs, categorize their bugs and identify the problem-dependent knowledge which led to their bugs. The GAP framework decomposes a program into a solution space containing the program's goals, and the plan or set of plans that can implement those goals (often 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 the 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 a missing goal, or by an incorrect knowledge representation used to solve the problem.

Soloway (1986) used the results of the prior studies (Soloway and Ehrlich, 1984; Spohrer et al., 1985) to develop a conceptual programming framework. 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 in their minds. Experts then use stepwise refinement to recall canned solutions that can be applied to the problem goals. Stepwise refinement is the orderly breakdown of a problem on the basis of simpler problems the programmer has already solved, the solutions for the simpler problems are the canned solutions the programmer then uses to create the overall solution to the current problem. Soloway's framework proposed that novices have difficulty identifying the goals needed to solve the problem, as well as difficulties recalling the appropriate

programming plans needed to implement the goals. Finally, expert programmers use plan composition to combine the fragments of canned solutions into a final solution plan; novices struggle with identifying the strategies that best allow for this.

Rist (1989) analyzed the program-generation process of 10 novice programmers 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 talking out loud during their problem solving process. Similarly to Soloway's (1986) conceptual framework, Rist focused his analysis on novice use of goals and plans, and coded 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 can 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*).

Byckling and Sajaniemi (2006) expands on Rist's (1989) work with a framework called role-based analysis. This framework additionally incorporated work showing that variables can be classified by their role in the program and that novices typically used nine variables falling under nine role categories (Sajaniemi, 2002). Role based-analysis uses variable roles as a basis for predicting how novices decomposed problems into goals and plans - Rist's (1989) framework of *plan focus* and *program expansion* is used to predict how these goals and plans get implemented. Byckling and Sajaniemi proposed that the students' focus would be variables, and the expansion would occur to

contextualize variables in the program. Byckling and Sajaniemi analyzed student performance as they progressed through a six-week course in a Pascal like language, using role based-analysis to identify different types of expansion processes. The results showed that students engaged in more strict forward development as they gained experience (i.e., they identified and initialized variables prior to their use in the rest of the program). While their framework proved interesting for analyzing student's use of variables during code generation, unlike Soloway's (1986) GAP tree representation, it did not include information on what expert-level representations novice students lacked.

Bertels (1994) reviewed literature comparing two approaches to modelling novice vs. expert programming differences:

(1) an overlay model, where novice knowledge simply comprises a subset of expert knowledge, so the knowledge novices have is fundamentally correct, but they have less of it;

(2) a buggy model, where expert and novice knowledge representations of programming are fundamentally different and novices have misconceptions as part of their knowledge base rather than simply less knowledge than experts.

These models were used to identify problems in novice knowledge representations that led to bugs in novice code. The review highlighted that novices had incorrect representations (buggy ones), and this best described the cause of their bugs (as opposed to bugs being the result of lacking expert representations). Thus, the buggy model described more of the novice errors overall. From this Bertels proposed that Soloway's (1986) programming plans are strategies of how to implement code. Programming plan components that had bugs were due to incorrect novice conceptions of the plan.

Conversely, Bertels found that bugs at the abstract level resulting from poor goal decomposition and planning were best explained by the overlay model. Some novices simply lacked many of the representations experts had, thus they did not having access to a particular goal or programming strategy that would have been useful.

Based on this analysis, Bertels developed a "dynamic model" that captured overlay-style knowledge representation capturing expert/novice differences at the abstract level, and buggy-style representational expert/novice differences at the functional levels. However, the paper did not describe how the representational structures at those various levels interacted (the abstract and the functional). Thus, more research is needed to assess how novices move from an abstract to functional level.

Castro and Fisler (2020) aimed to identify how and when students move between high level (task-level) programming schema considerations, and low level (code-level) implementational considerations. Here task-level considerations correspond to breaking down and addressing the tasks, analogous to Soloway's (1986) programming goals of the problem. Code-level considerations are ones that must be made while implementing the actual program. Novices in this study (N = 138) learned how to decompose programming problems using the How To Design Programs (HTDP) methodology. HTDP focuses on teaching a high-level approach to students, where prior to programming students first outline a concrete plan. The study involved complicated problems with multiple subtasks and used a talk-aloud method to capture how novices move between task and code levels.

Based on their analysis of novice code and talk-aloud transcripts, which they coded qualitatively. Castro and Fisler identified three types of styles for shifting between task-level and code-level thinking. The first style was *cyclic*, where students alternated

rapidly between task level and code-level. Here, students first mentioned programming goals and then wrote code to fulfill the goals. These students performed best when solving the multi-task programming problems. Castro and Fisler described cyclic programmers as those who followed a HTDP- style solution while making necessary adaptations to the high-level plan. Adaptations were made in response to issues that emerged when chaining the implemented code together into a coherent solution. This was done by cyclic programmers because the high-level plan identified various goals that needed to be addressed, but did not necessarily plan for implementational difficulties in connecting the relevant code together. Thus, the cyclic programmer regularly shifted between high-level thinking, to identify goals, and code-level thinking to resolve compatibility issues for the code written.

The second style of novice programming approach was *code focused*. This approach involved jumping directly into writing of code. These students identified tasks on the fly with none or minimal description of the written code's relation to the task. The third style was a *one-way* style, where students followed the HTDP style and made a high level plan at the start that they then dutifully translated into code. Unlike the cyclic shifting style, they did not actively adapt the programming plan with code-level considerations to make it suitable for the problem. In comparison to the cyclical programming, students who used one way and code-focused strategies both struggled. In general, the results showed it is important to not only plan an appropriate decomposition of goals but to also adjust and connect the implementational code to higher level goals.

Castro and Fisler (2020) demonstrated that students struggle with knowing when to focus on implementational aspect of their programming, and when to focus on higher-

level task considerations. This is similar to the difficulties Bertels (1994) observed. Castro and Fisler's work, as well as Bertel's focused on novice students, but did not identify what knowledge would allow them to behave more like experts.

2.3 Computational Models

Computational models have been used for formalizing representational differences between experts and novices in various domains (de Kleer, 1990; Fleischman & Jones, 2019; Johnson & Soloway, 1985). To illustrate, Cascade is a production rule system capable of both self-explanation of examples and physics (and algebra) problem solving. During these activities, it models various strategies employed by good vs. poor learners, as identified by prior work (Chi et al., 1989; Chi and VanLehn, 1991). This work reported that good learners took the time to explain examples to themselves, by connecting steps in the solution to major laws and principles they had learned, and perform better when solving similar problems (Chi et al., 1989). In contrast, poor learners tended to simply go through the problem without connecting the solution steps to known principles. The content of the self-explanation, was that good learners re-derived the step in the example, using their own knowledge base of known principles that apply to the problem (Chi and Vanlehn, 1991). VanLehn et al. (1991) used these studies' findings to design Cascade, and so to model these behaviors. During example studying, Cascade simulated a good learner by rederiving the solution steps of an example using domain rules that generated the solution steps, i.e., engaged in self-explanation. Cascade could also simulate a poor-learner who simply committed the examples to memory without elaborating on the underlying principles of the solution. While problem solving, Cascade could model a poor student, by copying example steps over to the target problem.

Cascade could also model a good student by trying to solve as much of the problem using its own production rule base. In this case, Cascade relied on its domain knowledge until it reached an impasse caused by a knowledge gap. To resolve the impasse, it relied on its commonsense and over-generalized knowledge to infer a new rule needed to overcome the impasse. The commonsense laws specified general mathematical principles.

The evaluation of Cascade showed it can model various student example-studying and problem-solving strategies in the domains of physics (VanLehn et al., 1991), and more recently algebra (Fleischman, 2002). As Cascade formalized the strategies and domain knowledge, Cascade can also be used as an architectural platform upon which to test new theories of learning. To this end, it was used in subsequent work to refine the framework of analogical problem solving and shed light on students' use of examples during problem solving (VanLehn, 1998).

Also outside of the domain of programming, FARRA (Braithwaite et al., 2017) is a model that uses production rules to solve fraction problems. FARRA contains strategy rules and execution rules. Strategy rules have a single precondition - a goal to solve a particular kind of problem; these generate new subgoals that lead to the resolution of the goal. A given problem will only require one strategy production to initialize the problem-solving process. Execution rules implement the problem-solving actions to resolve the subgoals that the strategy rules generate (that are its precondition). Both of these rules can come in correct or incorrect (mal) forms. The correct forms of the strategy and execution rules allow the model to correctly solve a given problem, while the mal rules encode misconceptions (both at the strategy level and execution level) and thus are not able to always accurately or reliably solve the problems. In particular, mal rules do not

always lead to incorrect solutions, as misconceptions are treated by FARRA as overgeneralizations of correct problem-solving procedures. When a rule is overgeneralized, the model may attempt to apply it in settings where it does not apply. The FARRA models are initialized with both mal and correct rules, and problem solving can strengthen the rule salience of either set, by increasing rule weights if they lead to the correct solution. Braithwaite et al. (2017) used FARRA to assess whether the distribution of problems in textbooks presented would lead to optimal reinforcement of the correct rules over the mal ones. They found that the textbook problem distributions lead to the strengthening of certain mal rules in the model. Because division problems were not adequately represented, FARRA strengthened mal rules that led to correct solutions during multiplication, addition and subtraction problem solving but incorrect solutions for division problems.

SimStudent (Li et al., 2015) is a computational model of algebra capable of inductively learning from demonstrations and problem-solving experience. Similar to FARRA and Cascade, it is a production system and its primary function is learning by induction from positive and negative examples. This model formalizes the following assumptions: (a) that production rules can model skills; (b) that during learning multiple learning mechanisms are used and rely on generalizations of: (i) where the rules would apply within the problem; (ii) when to use the rule during the problem-solving process; and (iii) how to use the rule and implement the problem solving steps. The model also includes a perceptual system that guides its attention towards salient features of the environment. The learning of productions is done via the addition of positive and negative examples. Positive examples are ones that help bring the model closer to the

solution of a problem by helping it implement a correct solution step, while negative examples produce incorrect solution steps.

Sim Student is a model of a novice problem solver. Other computational models have aimed to formalize the strategy differences between novices and experts. Koedinger and Anderson (1993) built a model encoding how experts skip steps when generating a solution and embedded it into the ANGLE geometry tutor. The model, called the Diagram Configuration (DC) model, embedded the assumption that experts can engage in step-skipping because of their schemas. The model included visual representations of diagrams included in common types of geometry problems. DC would match the diagram in its memory to the diagram representing the problem. Once the model matched a problem to a diagram schema in its knowledge base, the DC model was able to recall a set of geometry facts and proofs associated with the general schema that applied to the specific problem. In this way the model could skip steps just like experts.

Anderson and Lebiere (1998) developed a well-known cognitive architecture called ACT-R. ACT-R has been used to model students' algebra problem solving (Koedinger & MacLaren, 1997), strategies when solving the Tower of Hanoi (Altmann et al., 2001) as well as numerous other problem-solving and game domains. The above-described studies provided examples of how formalizing cognitive mechanisms within a model can be used to analyze problem solving and the underlying cognitive mechanisms, and thus provide a useful context by which to test hypotheses regarding pedagogy and expertise.

Python ACT-R is a similar cognitive architecture to ACT-R but written in the Python programming language. Python ACT-R is also capable of using an SGOMS

framework, which relies on goals, methods, operators and selector rules to model complex sociological behaviours (West & MacDougall, 2014). *Goals* represent high level problem requirements identified by the model, *methods* represent different plans that can achieve the goal, *operators* are actions within those plans and *selector rules* choose between competing methods. SGOMS has been used to model various complex behaviours such as managing multiple high-level communications demands in an Emergency Operation Center (West et al., 2018).

2.4 Computational Models of Programming

I now describe work on computational models simulating problem solving and related activities in the programming domain. One category of work corresponds to behavioral modelling. An example of this model is Recker and Pirolli's (1995) SURF, a SOAR-style production system. SOAR is a production based cognitive architecture similar to ACT-R (Laird et al., 1986). SURF modelled student behaviour while they interacted with a Lisp recursion tutor. A model was built for each student participant, mapping their mouse-click interactions and verbal self-explanations into productions. The productions were created by hand, by the researchers following data collection. The model also defined the environment as all of the options and text snippets that students encountered in the tutor interface, as well as the interactions possible between the student and the tutor. Recker and Pirolli compared the student models to each other and to a hypothetical ideal student (one that practiced self-explanation at each opportunity), by using hierarchical cluster analysis to identify a set of three classes of strategies employed by students. All of this could have been accomplished without the model, just with the student data, hence the behavioural model seemed redundant. In general, while cluster

analysis did identify classes of student behaviour, the models did not formalize strategies or schemas that the students may have developed.

Some models are architectural, in that they rely on a formalized model of cognitive mechanisms and how they relate to the behaviours produced. An example of one architectural model is GRAPES (Pirolli, 1986), which is an ACT-R inspired model of Lisp learning and programming. GRAPES uses four memory systems: a working memory, a long-term memory, a memory of the programming goals and a set of productions. GRAPES focuses on recursion and uses an ideal model instead of an expert one. The ideal model represents a "good" student's productions, that nonetheless have not been refined by years of practice like an expert's would be. Moreover, the ideal student model has fewer productions than the expert model and the productions are more general. The GRAPES learning mechanisms are similar to ACT-R's and rely on analogical problem-solving during impasses and knowledge compilation mechanisms. Impasses are points during the problem-solving process where the model cannot resolve some step of the problem, often due to not having a production that has the model's present state in the problem as its precondition. GRAPES resolves impasses in its problem-solving process via analogy. This involves generating a mapping between the problem and an example and using the solution steps of the example to guide problem solving, by transferring the steps from the example over to the problem. A rule is inferred as a result of the analogy process. The model saves the rule that solved the problem as a new production, and the state of the problem at the point of impasse as a precondition for that new production.

ACT-R has been used to create models of programming knowledge. The ACT-R Programming Tutor (APT), developed by Corbett (2000), can write small programs. APT

engages in both knowledge tracing and model tracing. Knowledge traing is used to assess the probability that a student has successfully learned a rule based on repeated, correct, application of the rule. For model tracing, the tutor uses an underlying production system, called its ideal student model, that 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 a list of all possible, correct next steps and compares it 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 one of the model's accepted next steps, the tutor provides feedback and encourages the student to correct the mistake. The ACT-R model underlying model-tracing can write the small program snippet solutions as it has the relevant productions, but it does not consider programming strategy.

The models of programming I've discussed up to now have not focused on strategy or schema representation in programming. Earlier I presented Soloway's conceptual framework of programming plans and strategies that investigated the content of programming schemas. The core weakness of that framework was its lack of formalization for all of the programming plans within a computational model. This was partially addressed by PROUST, a model built by Johnson and Soloway (1985), that 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. This process involved PROUST's knowledge base of programming plans, strategies and bugs, used to map out the solution path (or GAP tree) that linked the written program to the problem description. This allowed PROUST to

parse a program and identify deviations from the expected programming plan. Thus PROUST could identify buggy programs and diagnoses the source of the bug(s). While this model could identify strategies, it was not designed to apply them to write programs.

As described above, there is a wide range of work investigating expert-novice differences based on study data. There is also work on designing computational models capable of problem solving, including some for the programming domain. However, to date there does not exist a computational model that writes programs and models the differences between expert and novice problem solvers, for instance by employing programming schemas to structure problem solving and goal composition. Computational models have either formalized behavioural and other low-level implementational knowledge (thus avoiding formalizing the underlying representations) and/or focused on parsing full solution programs but not writing them (APT could only write small snippets but not full solutions). To address this gap, I built computational models that both formalize the underlying representational knowledge structures and generate programs.

Chapter 3: Study of Expert and Novice Programmers

3.1 Overview

The main goal of the present study was to gather data on the process of program generation. Process data would inform on how participants identified goals and implemented steps by writing Python code, which in turn will inform the implementation of computational models of programming. Earlier research showed that programmers used algorithms (schemas) when problem solving (Soloway, 1986; Spohrer et al., 1985), and that there were individual differences in how effectively novice programmers navigated between high-level algorithmic considerations and low level implementational ones (Castro and Fisler, 2020). However, it is unclear from the earlier research exactly how algorithms are structured to aid programming, or how the algorithm is implemented. This study should provide additional information. Moreover, as both novices and experts were recruited, the study should also shed light on how expertise impacts the programming process.

3.2 Participants

Initially, participants self-identified as novice or expert programmers. The novice participants (N = 12) were undergraduate students recruited from a first-year course that offered a broad introduction to cognitive science. These participants were offered course credit for participating in the study. Novices were required to have completed, or be currently undertaking, one beginner level programming course (or equivalent experience). Exclusion criteria included enrollment or completion of any 2000 or 3000 level undergraduate computer science courses (or their equivalent).

The expert participants (N = 7) were programmers with a programming-related degree, and/or related work experience; this included graduate students, professional software developers and those with degrees in computer science. Experts were compensated with 15$ for completion of the study. Exclusion criteria included insufficient knowledge as assessed by the pretest and by performance on the study problems. All expert participants had sufficient knowledge to successfully complete the pretest; one participant had difficulty completing the problems within the timeframe and thus was excluded from analysis.

3.3 Materials

3.3.1 Instructional Materials: Python Problems

The study involved four programming problems that varied in difficulty. All four problems were solved in Python, a high-level, general-purpose programming language. The four problems covered concepts such as loops, conditions, variables and other concepts typically covered by a first-year programming course. To illustrate, Figure 3.1 shows one of the problems, the rainfall problem. This problem requires the user to calculate the average rainfall over a certain period of time, using a list of rainfall amounts; negative numbers in the list are to be ignored and the program needed to stop processing the list upon encountering the first -999.

```
Design a program that has as a first line in it a list of numbers representing
daily rainfall amounts.
The list may include zero or more -999 values - these indicate an error and as
soon as one is encountered, the program stops processing the list.
The program outputs the AVERAGE of the NON -
NEGATIVE values in the list up to the first -999 (if it shows up).
There may be negative numbers (as although there cannot be negative rainfall
there may be erroneous inputs) other than -999 in the list.
The program must work with a list of any length
==================================
Here is an example. Suppose we have the list [50, 400, 300, -27, 50, -999].
The program outputs:
Problem_1[]: 200
Where -27 is ignored in the list and -999 is the stop signal.
Write your code using the input provided below. Please follow the preset code
so that we can provide feedback as to the correctness of your answer.
```

Figure 3.1 The Rainfall Problem

3.3.2 Questionnaires

Participants completed a five-item pretest that assessed their basic programming knowledge (see Appendix A.1). This pretest was used to confirm participants were categorized correctly as novice or expert (as initially this was done based on self-report). Novices were excluded from analysis if they had more relevant experience than just a single first year programming course, or if they could not complete the pretest (i.e., skipped entire problems as this demonstrated a lack of familiarity with the domain). Experts were excluded from analysis if they could not successfully complete the pretest, and no errors were allowed in their responses.

3.4 Procedure

The study protocol was reviewed and approved by the Carleton ethics board. The study was done via Zoom software. Once participants signed the consent form, they were given 20 minutes to complete the pretest. Upon completion of the pretest, I reviewed the answers to check that the participant was properly categorized as novice or expert. During this time, participants were given remote access to my code editor (VSCodium)

via the Zoom remote user function and asked to read a general instruction document outlining the study expectations and protocol (see Appendix B.1). The instructions specified that the participants must talk out loud as they are problem solving, so that I could get insight into their thought processes while they wrote programs.

Participants then solved 4 programming problems by writing a program for each problem. They were given up to 15 minutes to complete each problem. During this solution phase, there were asked to not test the code (i.e., by running it). After 15 minutes had passed or they indicated they were done, they were asked to run their program and offered an optional two-minute debugging period per problem if the code did not run successfully. Verbal utterances and programming actions on the screen during the programming activities were recorded from within the Zoom software.

Participants could ask for clarification if they had difficulty understanding the problem statement, or had difficulty interacting with the interface through Zoom, or if they had small syntactic questions that did not require elaboration of programming concepts. Beyond this, participants were not provided any help or feedback. The programming segment took no more than one hour and ten minutes, and the full experiment took approximately one hour and forty-five minutes.

3.5 Data Processing

To make the analysis feasible, I focused the analysis on the rainfall problem. Since my aim was to identify strategies used and to compare novice and expert code-generation approaches, I did not analyze the debugging period.

The audio files from the Zoom sessions were transcribed. The first phase of the transcription was done by the software Otter.ai. Otter produced a time-stamped transcript

of the audio file. I subsequently corrected the transcripts by watching each Zoom recording and ensuring that the transcript accurately reflected the verbal utterances. At this point, the transcripts were also supplemented with snippets of the participant's code added at the appropriate location in the transcript, so that they aligned with the verbal protocol. This was done in order to produce a transcript that reflected both verbal data and programming actions in chronological order. To illustrate, a portion of the transcript with supplemental code is provided in Figure 3.2.

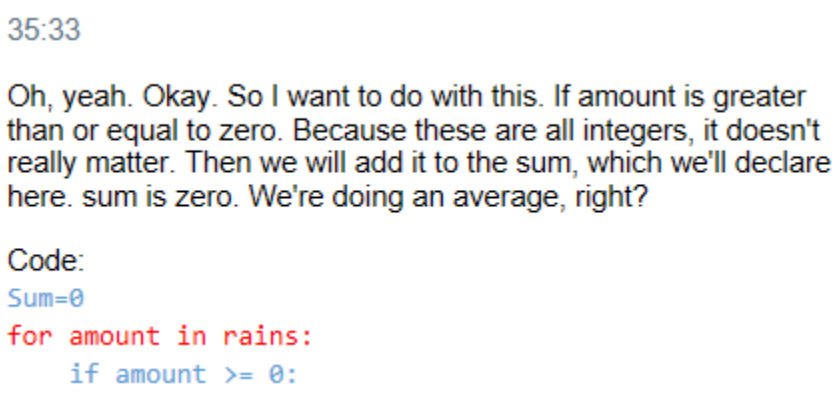

Figure 3.2 Snippet of transcript representing a portion of a participant's problem solving. Text in black represents the participant's talk aloud while text in blue represents what the participant wrote related to the talk aloud. Code written in red was written at a prior time point.

3.6 Qualitative Coding of Transcripts

The transcripts were coded by the thesis author to identify the components of the problem-solving process; this facilitates analysis of problem-solving approaches and novice and expert differences. The coding was based on earlier work analyzing problem solving in the domain of physics (Gertner & VanLehn, 2000). While this domain is different from programming, the approach used was high level and so generally applicable across domains. In this prior work, the knowledge representation was comprised of two key constructs, namely step and goals. This representation was

implemented into a rule-based problem solver that produced solutions to physics problems; the solutions were used by ANDES (Gertner & VanLehn, 2000), a physics tutoring system. ANDES had the complete solution for each target problem (all equations and transformations that led to a correct answer) and thus was able to identify where and how the student deviated from a correct solution path.

I now describe the coding scheme for the present thesis, used to label the transcripts with goals and steps.

3.6.1 Coding Scheme

As noted above, the coding scheme included goals and steps. Goals corresponded to concrete, verbally stated intentions about a high-level programming action that needed to be performed. A sequence of goals is an algorithm, namely a recipe for solving the problem. Depending on the confidence and experience of the programmer, goals could be very clear and explicit as in "I need to initialize variables for the sum and count" or the goal may be stated in a more confused and uncertain manner, as in "I think I need to go through the list somehow". To facilitate comparison between participants, a set of goal labels was developed, shown in Table 3.1. This was done by reading the transcripts to identify common goal patterns. Utterances not related to the programming process were ignored and not coded. Other uncoded utterances included questions to the experimenter, experimenter comments and restatements of the problem statement, or repetition of identified goals by the participant.

Steps corresponded to written code in Python (i.e., in order for something to be labelled as a step, it had to be written in the Python IDE). Steps could immediately follow a goal. For example, once the goal to iterate through the list was expressed, the

subsequent code “for x in rains:” corresponds to a step. Steps could also occur in the absence of a goal - in this case steps were identified on the basis of a written Python line, or short written statement (for example a conditional break, which can be written on 2 lines, but it’s functionally one expression). Finally, steps could occur later on in the solving process after the corresponding goal. Similarly to the goals, to facilitate comparison between transcripts, a set of step labels was created, shown in Table 3.2. Again, the labels were identified by reading the transcripts to identify common step patterns.

Goal Label	**Description**
Initialize variable total	Set up variable total to store sum of positive numbers
Initialize variable count	Set up variable count to store # of positive numbers
Iterate through list	Iterate through the list of values given
Stop loop	Break loop when stop condition (-999) encountered
Track total	Add positive numbers in list to total
Track count	Increment count by 1 for each positive number in list
Calculate average	Calculate average of positive numbers in list using sum and count
Other – find index of -999	find which index value of list corresponds to stop signal -999
Other – length of list	Find length of given list
Other – ignore negative numbers	Ignore number in the list if it is less than 0
Other - idiosyncratic	Goal does not fall into above definitions and is highly idiosyncratic to programmer

Table 3.1 List of goal labels used to code the transcripts.

Step Label	Description
Initialize variable	Initializing a variable that will be used by other steps within the python program
Loop	Iterate through the list of rainfall amounts using a loop
Condition	A Boolean statement that acts as a condition for another step
Increment variable	Add a value to the variables
Calculation	Perform a calculation with values in the variable
Other – idiosyncratic	Performed a step that did not conform to expected solution

Table 3.2 List of step labels used to code the Python solution

3.6.2 Qualitative Coding Process

Applying the coding scheme to the transcripts involved reading each utterance and labelling it with a goal tag (see Table 3.1) if it corresponded to a goal. It also involved labelling the Python code with the step labels (see Table 3.2). Once the transcripts were coded, the goal and the step codings were extracted and recorded in a separate document tracking their chronological order (i.e., the order that goals were stated and steps were implemented; these did not include time-stamps and only snippets of the original utterances). I refer to these latter codings as chronotranscripts (see Figure 3.3 for an example).

Once chronotranscripts were made for all of the participants for the rainfall problem, they were compared to each other and the canonical solution, and general trends were identified. Specifically, I performed a qualitative analysis of the chronotranscripts to identify similarities and differences in the solutions components and problem-solving strategies of experts and novices. To accomplish the latter, I focused on the order of goal identification and step implementation, as well as any variations of the goals and their

implementation. This would help clarify how goal decomposition was done be experts and novices, and identify possible relationships (e.g., between goals and between goals and steps).

2. Goal: **[stop loop]** if – 999 break loop and return average
3. Step: < *loop* > for I in x:
4. Step: < *increment variable* > sum +=i
5. Goal: **[initialize variable total/count]** keep a count of how many nums summed
6. Step: < *initialize variable* > sum = 0
7. Step: < *initialize variable* >num_sum = 0
8. Goal: **[other - ignore negative numbers]** if I is less than 0 do nothing

Figure 3.3 A chronotranscript snippet. The lines labelled as goals are simplified statements of the programmer's verbalized intentions and include both the goal label (bolded and in square brackets), followed by the statement made by the participant. Steps are the Python expressions written to resolve the goals and are coded with step labels (italicized and in angle brackets).

3.6.3 Canonical Solution

An additional expert (not a study participant) provided a canonical solution to the rainfall problem that was used as a baseline solution to which expert and novice solutions were compared. The canonical solution is presented in Figure 3.4. While this solution is not Pythonic in that it could be more compact, it represents a common strategy taught to students in first year programming classes that has the advantage of being language independent. Since the experts had programming experience but were not Python experts, this solution is appropriate for the present study.

Goal: **[initialize variable total]** set up a variable *total* to store positive numbers
    Step <*initialize variable*>: **total** = 0
Goal: **[initialize variable count]** set up a variable *count* to store the # positive numbers
    Step <*initialize variable*>: **count** = 0
Goal: **[iterate through list]** Iterate through the list
    Step <*loop*>: for number in rains
    Goal: **[stop loop]** stop if number is -999
        Step <*condition*>: If number == 999: break
    Goal: **[track total/ track count]** add **positive numbers** to total, keeping track of how many there are
        Step <*condition*>: If number > 0:
        Step <*increment variable* >: total += number
        Step <*increment variable* >: count += 1
Goal: **[calculate average]** calculate average
    Step <*calculation*>: average = total / count

Figure 3.4 A chronotranscript of the canonical solution provided by an expert programmer.

The canonical solution assumes that the program is written in the order shown in the solution. Thus, first the two variables, sum and count are initialized, and are used to track how many positive numbers there are in the list and the sum of the positive list values. The next part of the program iterates through the list of rainfall amounts (using a for loop). Within the loop, the program checks if the current value is the stop signal (-999), which stops the loop; otherwise, the program checks if the current number is positive and if so updates the sum and count. Once all values in the list are processed or the stop value is found, the program calculates the average rainfall.

3.7 Results

As noted above, the results are based on a qualitative analysis of expert and novice programming solutions.

3.7.1 Participant Solutions vs. Canonical Solution

For the majority of cases, both novices and experts' final programs were in line with the canonical solution. To illustrate, Figure 3.5 shows the final solutions of novice P103 and expert P116 (shown is the Python code rather than chronotranscripts to

demonstrate their similarity). Both participants created and initialized variables to track the positive numbers (num and total for P103, and sum and count for P116) and iterated through the list rains (called x in P103's program and rains in P116's) using a loop. Within the loop they both addressed the stop signal (-999) and tracked the sum and count of the positive numbers. Finally, both expert and novices calculated the average using their variables for sum and count outside of the loop. Not all participants followed this solution exactly, like participants P105 and P117, who implemented additional steps not included in the canonical solution (Figure 3.7 lines 17-21 and Figure 3.9 line 6).

a)

```
total = 0
num = 0
for i in x:
    if i == -999:
        break
    if i >= 0:
        total += i
        num += 1
average = total/num
```

b)

```
sum = 0
count = 0
for amount in rains:
    if amount >= 0:
        sum += amount
        count += 1
    elif amount == -999:
        break
average = sum / count
```

Figure 3.5 A correct solution to the rainfall problem produced by a novice participant (P103) (a), and an expert participant (P116) (b).

1. Goal : **[calculate average]** calculate average of the positive numbers
2. Goal : **[iterate through list]** iterate through list rains
3. Step: < *loop* > for x in rains
4. Goal: **[other - idiosyncratic]** identify positive numbers as you iterate through the list
5. Step: < *condition* > if x > 0:
6. Goal: **[track total/ track count]** count numbers in the list
7. Step: < *initialize variable* > total = 0
8. Goal: **[initialize variable count]** the count, so how many numbers there are in the list
9. Step: < *initialize variable* > count = 0
10. Step: < *increment variable* > total = total+1
11. Step: < *increment variable* > Variable 2 Updated: count = count+1
12. Step: < *increment variable* > total = total + x (CORRECTION)
13. Goal: **[other - return average]** return the output total/count
14. Step : < *calculation* > return total/count

Figure 3.6 The chronotranscript for novice participant P100.

3.7.2 Deviations of Expert and Novice Chronotranscripts from the Canonical Solution

While both novices and experts produced final solutions that were similar (if not identical) to the canonical solution, they did not produce identical chronotranscripts. The implication of the latter is that they did not follow the same solution strategy and goal order. For example, neither experts or novices consistently identified initializing the variables as their first goal, and initialization was often not the first step implemented, as was done in the canonical solution. For example, novice participant P100 identified the goal to use a loop to iterate through the list rains and implemented the goal (see Figure 3.6, lines 2 and 3). They did not deal with the variables until the condition for handling the positive numbers was implemented (see Figure 3.6, line 5) and the variables sum and count were required for the goal "count the positive numbers in the list" (see Figure 3.6, lines 6 and 8). At this point the participant initialized the variables (Figure 3.6, lines 7 and 9). Most of the other novices also did not follow the canonical solution in their order of goal implementation, though their exact deviations from the canonical differed from one another.

1. Goal: **[iterate through list]** iterate through every element in the list
2. Step: < *loop* > for I in range(0, end_input)
3. Goal: **[other - find index of -999]** create variable to identify end-point (as it is variable)
4. Step: < *initialize variable* > end_input = ……
5. Goal: **[other - positive numbers id]** if 0 or positive number in list should be added
6. Step: < *condition* > if (x[j] >= 0):
7. Goal: **[initialize variable total]** sum variable to numbers added
8. Step: < *initialize variable* > sum = 0
9. Goal: **[initialize variable count]** count variable identified
10. Step: < *initialize variable* > elements_amount = 0
11. Goal: **[track total]** add element (convert to int for safe)
12. Step: < *increment variable* > sum += int(x[j])
13. Goal: **[track count]** add to count
14. Step: < *increment variable* > elements_amount += 1
15. Goal: **[calculate average]** calculate and print average
16. Step: < *calculation* > average = sum/elements_amount
17. Goal: **[other - find index -999]** find the index value of the end_input -999
18. Goal: **[iterate through list]** iterate through the list
19. Step: < *loop* > for j in range (0, len(x))
20. Goal: **[other - find index of - 999]** if j element in the list is -999 set j as end_point
21. Step: < *condition* > if (x[j] == "-999"): end_input = j

Figure 3.7 The chronotranscript for novice participant P105

Certain goals and steps were typically addressed in the same order as in the canonical solution, such as for example calculating the average, which most participants (expert and novice) performed last, as was done in the canonical solution. However, this was not universal, as participant P105 went on to implement a separate loop construct after they implemented the step to calculate the average (Figure 3.7, lines 15-20).

Similarly, experts also did not identify the goals or implement the steps in the same order as the canonical chronotranscript, or each other. For example, expert participant P115 did not initialize the variable sum or count (called *num_sum*, see Figure 3.8, lines 6,7) until after identifying the goals "*track the positive numbers in the list*" and "*break the loop at -999*" (see Figure 3.8, lines 1 and 2), implementing the step to implement the loop (Figure 3.8, line 4) and incrementing the sum by the current list value (Figure 3.8, line 5). Both experts and novices tended to initialize the variables when they

needed them to resolve other goals, usually when they needed the variables within the loop (Figure 3.7, lines 5-10; Figure 3.6, lines 5-9; Figure 3.8, lines 4-7).

A common deviation of both experts and novices from the canonical solution was that they both tended to implement the loop step first (Figure 3.6, line 3; Figure 3.7 line 2; and Figure 3.8, line 3), while the canonical solution addressed that step only after initializing the variables.

1. Goal: **[track total]** I'm going to sum up all the values until I hit -999 - check if number is positive before summing it
2. Goal: **[stop loop]** if – 999 break loop and return average
3. Step: < *loop* > for I in x:
4. Step: < *increment variable* > sum +=i
5. Goal: **[initialize variable total/count]** keep a count of how many nums summed
6. Step: < *initialize variable* > sum = 0
7. Step: < *initialize variable* >num_sum = 0
8. Goal: **[other - ignore negative numbers]** if I is less than 0 do nothing
9. Goal: **[track total/ track count]** if I greater than/equal to 0, add I to sum and 1 to num_sums
10. Step: < *condition* >for i> = 0:
    < *increment variable* > num_sum+=1
    < *increment variable* > sum+=i
11. Goal: **[stop loop]** break if num is -999
12. Goal: **[calculate average]** print average
13. Step: < *condition* > if i == -999 :
    < *calculation* > print(sum/num_values)

Figure 3.8 The chronotranscript for expert participant P115.

3.7.3 Novice vs Expert Solution Processes

Most novices (and some experts like participant P115) identified and implemented the goal and step of initializing the variables related to *sum* and *count* in sequence (see Figure 3.8 lines 6 and 7; Figure 3.7 lines 8 and 10; and Figure 3.6 lines 7 and 9). In contrast, the experts were more variable in terms of when they implemented this goal and step. Some experts initialized *sum* before *count*, with at least one other non-variable related step or goal in between. For example, expert participant P117 first initialized the variable *sum* (called *rain_sum* in their program) (see Figure 3.9, lines 1 and

2). They then did not initialize the variable *count* (called *rain_count*) until they had nearly completed their solution (see Figure 3.9, lines 11 and 12).

Experts differed from novices in their ability to identify multiple goals in sequence, with no step implementations in between (Figure 3.8, lines 8, 9; and Figure 3.10, lines 4, 5). This also allowed experts to consolidate steps together. To illustrate, participant P115 calculated the average and broke out of the loop in a single expression (Figure 3.8, line 13). This strategy is in contrast to all novices and most other experts, who calculated the average outside of the loop and not within the break statement.

1. Goal: **[initialize variable count]** if something do sum
2. Step: < *initialize variable* > rain_sum=0
3. Goal: **[iterate through list]** iterate through x
4. Step: < *loop* > for rainfall in x:
5. Goal: **[other - ignore negative numbers]** ignore if number in list is less than 0
6. Step: < *condition* > if rainfall <0:
   < *other* > continue
7. Goal: **[stop loop]** break statement
8. Step: < *condition* > if rainfall <0:
   < *condition* > if rainfall == -999:
   < *stop* > break
   < *other* > continue
9. Goal: **[track total]** sum positive rainfall
10. Step: < *increment variable* > else: rain_sum += rainfall
11. Goal: **[track count]** track count of positive numbers
12. Step: < *initialize variable* > rain_count = 0
13. Step: < *increment variable* > rain_count += 1
14. Goal: **[calculate average]** give answer
15. Step: < *calculation* > answer = rain_sum/rain_count

Figure 3.9 The chronotranscript for expert participant P117.

1. Goal: **[iterate through list]** I need a for statement
2. Step: *< loop >* for rainfall in rains
3. Step: *< condition >* if rainfall >= 0:
4. Goal: **[other - ignore negative numbers]** Exclude negative numbers
5. Goal: **[stop loop]** check for -999 (stop signal) first
6. Step: *< condition >* if rainfall == -999: break
7. Goal: **[initialize variable total]** need a total variable
8. Step: *< initialize variable >* total = 0
9. Goal: **[initialize variable count]** need count variable
10. Step: *< initialize variable >* Count = 0
11. Goal: **[track total/ track count]** track positive numbers (sum and count)
12. Step: *< increment variable >* total += rainfall
13. Step: *< increment variable >* count += 1
14. Goal: **[calculate average]** calculate average
15. Goal: **[other - idiosyncratic]** avoid dividing by 0
16. Step: *< condition - calculation >* if count > 0 : Answer = (total * 1.0)/count
17. Goal: **[other - idiosyncratic]** let user know there is no input for program
18. Step: *< other - idiosyncratic >* Answer = "Can't tell. No input"

Figure 3.10 The chronotranscript for expert participant P120.

Some novices had additional sources of variability in their solutions compared to the experts' solving process, due to inefficiencies and inaccuracies in their solution. For example, participant P100 only summed and counted the positive numbers greater than 0 (see Figure 3.6, line 5), and did not include the value 0 in their average calculation. Additionally, novice participant P105 aimed to identify when to stop processing the list (-999) in a separate loop (see Figure 3.7, lines 18-21). This solution does not account for the fact that the list may contain multiple -999s, and therefore does not stop the loop at the first -999 encountered as it should.

Some novices struggled more with the problem, and would use extra steps and had misconceptions in their approach. To illustrate, participant P109 not only added additional code not required for the problem, which was not used in the canonical solution nor by any other participant (see Figure 3.11, line 2), but also made errors when calculating the average. Specifically, rather than keeping a count of the positive values, P109 relied on the built in len function and used that in the average calculation (see

Figure 3.11, line 13). Thus, their average calculation incorrectly divided the sum of the positive numbers by the total length of the list (that included both the positive and negative numbers).

In contrast, experts broadly implemented the standard canonical solution (even though their chronotranscripts deviated from the canonical order). Some deviations did exist, but unlike for the novice deviations they did not represent inaccuracies in the solution process, instead reflecting idiosyncratic variation. To illustrate, expert P117 dealt explicitly with negative values in the list rains, which they handled at the same time as the stop code (see Figure 3.9, line 8), and participant P115 identified the goal to ignore negative numbers but did not implement any steps targeted towards it (see Figure 3.8, line 8).

1. Goal: **[calculate average]** take in values from rainfall and lookup average
2. Step: < *other - idiosyncratic* > def rainfall(list):
3. Step: < *loop* > for x in list
4. Goal: **[other - idiosyncratic]** Check for only positive numbers
5. Step: < *condition* >if x > 0
6. Step: < *other - idiosyncratic* > def rainfall(x):
   < *loop* > for i in x:
7. Goal: **[track total]** if number is equal to and greater than zero add it to the sum
8. Step: < *initialize variable* > sum = 0
9. Step: < *increment variable* > sum += x
10. Goal: **[stop loop]** if -999 stop program
11. Step: < *condition* > elif I == -999: break
12. Goal: **[calculate average]** calculate average
13. Step: <*calculation*> average = sum / len(x)

Figure 3.11 The chronotranscript for novice participant P109.

3.7.4 Summary

To recap, participants' talk-alouds and solutions to the rainfall problem were coded into explicitly stated goals and the written Python steps, listed in chronological order. Their final solutions tended to include the same steps as the canonical solution, but both experts and novices varied the order that they produced the steps in a way that did not impact the validity of the solution. This demonstrates that participants addressed the problem goals in variable order. For example, both novices and experts varied whether to first stop the loop or increment the sum and count variables within the loop. Novices had other sources of variability that did affect the validity of their solution and reflected gaps and/or inefficiencies in their domain knowledge, such as identifying the index of the stop signal in a separate loop.

In contrast to novices, experts were more likely to identify multiple goals at a time, and implement multiple steps at a time. On the other hand, novices tended to implement the steps as they identified the associated goal. There may be many possible reasons for this difference. One is that experts had a complete algorithm for this style of problem while novices did not. Therefore, experts would verbalize multiple goals that they retrieved from this algorithm from memory and then implement multiple steps at a time. This is unlikely as the goals retrieved were often not retrieved and implemented in the order that would allow for the most straight-forward writing (like the canonical solution). Alternatively, experts may have had smaller sub-algorithms that the goals helped implement, like calculating the average and tracking variables within a loop, while novices lacked the sub-algorithms or had incorrect or incomplete versions of them. Thus, experts may identify multiple goals, and then while implementing the sub-algorithms of

those goals identify additional goals that are necessary to resolve the already identified ones. This seems more likely, especially given that experts generally identified the goal to initialize the variables only once they were already needed for a different goal, which implies the use of sub-algorithms instead of a full algorithm. However, this model may also apply to more competent novices. Future analysis of more complex problems should help further tease apart these two possible programming strategies in experts, as it would give more opportunities to observe new goals being generated during the solution process.

Chapter 4: Cognitive Models of Programming

4.1 Introduction

In this chapter, I present high-level descriptions of five ACT-R models I implemented (code and explanation can be found in Appendices C-D). Each model is capable of producing a solution to a simple programming problem using a different problem-solving approach (the solution produced may not necessarily be correct). The approaches correspond to various novice and expert strategies that are based on the case study from Chapter 3 and/or related work.

I used Python ACT-R for implementing the models. As described in the introductory chapter, ACT-R and Python ACT-R are cognitive architectures developed for understanding and simulating human cognition. These architectures rely on two types of memory: (1) a procedural memory that encodes *productions*, which are if/then statements that perform actions when their preconditions are met, and (2) a declarative memory that stores information in *chunks*. The production preconditions are encoded in the if-part of the production and include internal stimuli (representing mental states) and external stimuli (representing environmental influences). The declarative memory represents information that is known but not immediately actionable. The present models rely on the same knowledge representation as was used for analyzing the transcripts, namely one that includes both goals and steps. With the exception of model 1, goals are stored in declarative memory as chunks. In contrast, steps are generated by productions that use the motor module to write the programming expression corresponding to the step.

Python ACT-R models include a set of productions used for problem solving within the target domain, as well as a system of modules, which are subsystems of productions specialized for particular functions. The model can use the modules to realize various functionalities. Examples of modules include: (1) a motor module that includes productions specialized to the environmental tasks the model must perform, and (2) a declarative memory module that includes productions controlling the model's declarative memory.

Modules, including the declarative memory module, and productions within the model communicate using buffers, which are analogous to the human working memory system. Accordingly, buffers store chunks of information. The information serves as the preconditions for the model's productions; it is also used by the various modules when they are called. The buffers that are used by the present models include the following: (a) DMbuffer, which contains information requested from declarative memory by a production, (b) focusBuffer, which tracks where the model is in the problem solving process, and unless otherwise stated, its contents are one of the primary preconditions that must be met for a production to fire; (c) plan_unit, which represent problem solving schemas that encode an algorithm that the model follows to resolve the goal of the planning unit (this buffer is only used by the SGOMS model, described later); (d) motor buffer, which relays commands from the model to the motor module to implement problem solving actions in the environment.

4.2 General Model Functionalities

While the models presented here will simulate different problem-solving strategies, they will all rely on the same modules for handling the memory system and

interacting with the environment. Modules are initialized prior to problem solving, by defining the module buffers before the model's problem-solving productions. The modules common to all models include the following: (a) the motor module; (b) the environment; and (c) the declarative memory module (see Figure 4.1 for a visual of the modules and their relations).

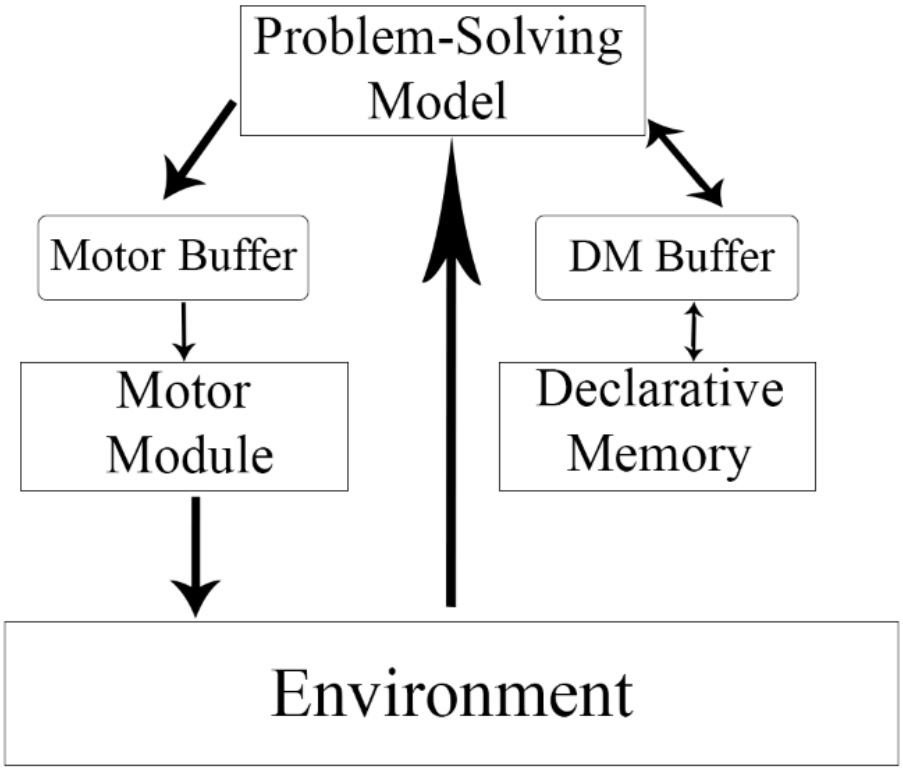

Figure 4.1 Diagram representing modules used by the problem-solving models. The arrows represent the direction of information flow. Problem-solving models receive information directly from the environment, send instructions to the motor module to implement actions in the environment (via the motor buffer) and share information bidirectionally with the declarative memory (through the declarative memory buffer) to manage problem solving.

All of the models use the same motor module to write their Python program and produce a log of their identified goals and implemented steps. Motor modules exist

outside of the model and are called upon by the model using a buffer that relays commands from the model to the motor module.

The model is embedded within an *environment*. The environment contains the same description of the rainfall problem for each model. Environments do not use buffers to communicate with the model – instead, the model is initialized within the environment and has access to its information. The environment is the same for all models.

In contrast to the motor module, which is specialized to a given model, the declarative memory is a general component of the Python ACT-R architecture. The model uses a buffer to communicate with the declarative memory and can use the buffer to add chunks to the declarative memory (reflecting new goals identified) or retrieve known chunks from memory. Sometimes the declarative memory may make mistakes and fail to retrieve known 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 remake the request, and will retrieve the correct fact with sufficient attempts.

Goals identified by the model, as well as steps implemented by the model were based on ones present in the novice and expert solutions and transcripts (as well as those present in the canonical solution). None of the present models are capable of learning new chunks or productions, i.e., algorithms and step implementations of them are not learned through experience, but rather the model is initialized with them by the human model builder. In particular, for the present work, the model's declarative memory was initialized prior to problem solving. Some of the present 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). The addition of these goals to the declarative memory are thus a process of decision-making and not learning.

4.3 Algorithm-Driven Models

The first series of models I implemented are so-called algorithm-driven models. An algorithm is a plan of ordered goals that solve the problem statement. These models simulate the situation where the complete algorithm is accessed or developed *before* any steps are implemented (recall that steps correspond to Python code). I implemented three algorithm-driven models: (a) a *proceduralized* model, where the declarative memory is not used because the entire solution is a retrievable and implementable procedure; (b) a *algorithm-retrieval* model, where a complete algorithm is stored in the declarative memory and retrieved when the problem is encountered in the model's environment; and (c) an *algorithm-generation* model, where the full algorithm is first generated by the model, and then retrieved and implemented. The goals and steps used by the models reflect those identified in the study results Chapter 3, however the strategies modelled do not. Rather, the strategies of these models reflect common models of expertise used by others working with the ACT-R and Python ACT-R architectures (Altmann et al., 2001; Lebiere et al., 2001; MacDougall et al., 2014), as well as work done by Castro and Fisler (2020). As such, these models present a comparison benchmark to the models of strategies that are inspired by the study results (i.e., models 4 and 5).

4.3.1 Model 1: Proceduralized

In the *proceduralized* model, the programmer does not need to recall goal information from the declarative memory to produce the solution because the goals have

been embedded within the productions. Thus, the model by-passes declarative memory and directly implements the necessary steps by relying on the productions. In this way the algorithm in this model has been proceduralized and does not rely on having goals in its declarative memory. Here, a chain of productions implements the steps needed to solve the problem (i.e., to write a program to calculate the average of the positive numbers in a list), simulating no conscious reflection on the part of the programmer.

The first production fires when the problem statement is loaded. Each production calls upon the motor module to write a single Python expression (or line) to the Python file and then sets the focus buffer to contain the precondition for the next production in the algorithm. For example, the production that implements initializing the variables sets the focus buffer to contain the chunk acting as the precondition for the production that implements iterating through the loop. Thus, there is no need to decompose the problem goal, "Calculate the average of the positive numbers in the list", into a series of smaller goals as there is a chain of productions that implements all of the steps needed to resolve the problem goal. This is a common model of expertise in ACT-R (Altmann et al., 2001; Kennedy & Trafton, 2007). One of the hallmarks of these models is that the solution is generated quickly and efficiently, due to the fact that all solutions steps are generated by pre-existing productions.

This approach is not a good simulation of the behavior I observed in the study, as neither novices nor experts showed evidence of a proceduralized solution to the problem. Specifically, I anecdotally observed both groups (novices, experts) often stopped to think between steps, while a proceduralized solution would implement each step in rapid succession.

4.3.2 Model 2: Algorithm Retrieval

In contrast to the *proceduralized* model, the *algorithm-retrieval* model's declarative memory contains the complete algorithm required to generate the solution. The algorithm corresponds to the goals needed to solve the problem. The first goal is retrieved when the model encounters the problem statement. Retrieval is done by requesting the goal information from the declarative memory and storing that information in the DMbuffer. When the goal is retrieved and added to the DMbuffer, it satisfies the precondition to fire the corresponding step-implementing production. Once the step has been implemented (i.e., the Python expression resolving the goal has been written to the Python file), the model retrieves the next goal from its declarative memory. This cycle of goal retrieval and step implementation proceeds until all the goals have been retrieved and all of the steps implemented; at this point the program is considered complete. The declarative memory is not updated during problem solving, as the algorithm is already encoded in the memory.

The difference between this model and model 1 is that the present *algorithm-retrieval* model does not have the complete solution proceduralized and instead has the algorithm stored in its declarative memory. This means that the model interweaves retrieving the relevant goals from memory and implementing the steps to resolve them. This is in contrast to model 1, that relies on a set of productions to directly implement the problem-solving steps immediately upon encountering the primary problem goal.

This model represents an expert programmer who has internalized the canonical solution. This is a common way of modelling algorithm-based problem solving within the ACT-R architecture (Lebiere et al., 1998.), although in domains other than programming.

4.3.3 Model 3: Algorithm Generation

The final algorithm-driven model does not have the algorithm proceduralized (model 1) or stored in its declarative memory (model 2). Instead, model 3 first generates a *complete* algorithm based on the problem statement and then retrieves it to implement the algorithm components by writing the Python code. This model is inspired by the one-way programming style that some novices use as identified by Castro and Fisler (2020), where some novices first generated a complete solution plan and then implemented it. The difference between model 3 and model 2 is that model 3 first *generates* the algorithm in the declarative memory, while model 2 does not include this component. Once the algorithm is generated, the two models behave identically.

The generation of the algorithm by the model begins by identifying goals from the problem statement and from goal expansion. Goals are not stored in the declarative memory (as in model 2). Instead, they are generated by the model using the keyword – goal, goal – goal and goal – variable/condition associations, added to the model's declarative memory at the start of the problem-solving process. As natural language processing is beyond the scope of this thesis, the model relies on keyword – goal associations to generate goals based on keywords in the problem statement. Additionally, as in the case study, participants identified goals that required the resolution of other goals, and not all goals were directly readable in the problem statement, therefore the model is also capable of searching its declarative memory for goal – goal associations, and then initializing the associated goal. For example, calculating the average requires initialization of variables to track the sum and the count of the numbers being averaged, so even though such a goal is not directly identifiable from keywords of the problem

statement the model can check its declarative memory for a *calculate_average – initialize variable* association. To check for these related goals, the productions that generate goals also set the precondition (using the focus buffer) for the production that searches for any goals related to the current goal being added to the algorithm. Once a goal – keyword association or goal – goal association is identified, the model retrieves the variables or condition relevant to the identified goal, and then adds the goal to the algorithm it builds it in its declarative memory. By the end of the generation process, each goal in the algorithm will have a line in the declarative memory to represent it.

Once all goals have been added to the declarative memory (from both keyword search and related goals search) the algorithm is considered complete. At this point, the model 's focus is set to retrieve the first goal in the algorithm and begin implementation of the algorithm. Implementation proceeds identically to the process for model 2, as the algorithm generated by model 3 is identical to the one in model 2.

4.4 Non-Algorithm Driven Models

4.4.1 Model 4: Goal Expansion

Recall that for model 3, the algorithm was generated in its entirety before any code was written. The *goal-expansion* model does not generate an algorithm up front. Instead, it cycles between productions that engage in goal generation, goal retrieval, and step implementation during the program generation process (generate a goal, retrieve it, implement it; generate next goal, retrieve it, implement it, and so on)[1]. The model also expands goals into related goals after implementing them (similar to the functionality to

[1] Note that model 2 *retrieves* goals from a stored algorithm and implements them, while model 4 *generates* a goal, retrieves it and implements it by generating the Python step

search for related goals in model 3). This model is inspired by Rist's (1989) framework of novice programming, where novices identified a *plan focus*, a core step that is written first in the program, and then engaged *program expansion* around the *plan focus* by implementing additional steps that supported and expanded the *plan focus* step. However, unlike the Rist framework, the model identifies a few goals (more than one) directly from the problem statement, based on its known keyword – goal associations, and then expands these goals into related goals based on its known goal – goal associations.

The *goal-expansion* model uses a set of keyword - goal, goal – goal and goal – variable/condition associations to translate the problem text into goals, similar to what was done in the *algorithm-generation* model (model 3). Unlike model 3, model 4 does not generate a complete algorithm before engaging in implementation of steps. Therefore, 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, it is immediately retrieved from the declarative memory. Once retrieved it acts as the precondition for the firing of the production that implements the step that resolves the goal. The step-implementing production sets the model's focus buffer to contain the precondition for the production that seeks out any other related goals i.e., directs the model to engage in goal expansion.

4.4.2 Model 5: SGOMS

Model 5 relies on SGOMS. SGOMS is a cognitive framework that adds planning units and unit tasks to Python ACT-R in order to model complex behavior.

The planning units represent goals, such as *calculating the average* and *initializing the variables*, and reflect the goals identified during the coding of the participants' chronotranscripts. However, planning units treat goals in a more

complicated manner than the treatment of earlier models (models 1-4), as I describe shortly. Planning units are initialized into the model's declarative memory at the start of problem solving. Each planning unit defines a set of operations, called unit tasks, that must be completed to resolve the planning unit; the set of unit tasks for a given planning unit will be referred to as a sub-algorithm. For example, the planning unit *calculate_average* has a sub-algorithm composed of four unit tasks that must be completed to resolve the goal of calculating the average (see Figure 4.2).

As noted above, a unit task is an operation that must be performed as part of a sub-algorithm that resolves a goal. Unit tasks can either define high-level operations or implementational-level operations. High-level unit tasks are analogous to the task-level considerations of competent programmers identified by Castro and Fisler (2020). Task level considerations involve identifying and organizing the goals of the problem. High-level unit tasks within a planning unit calls upon productions that initialize other planning units; other planning units reflect goals that must be resolved before the present one can be. Implementational unit tasks are what the model uses to control implementation of the step, and are analogous to the code-level considerations identified by Castro and Fisler. Code-level considerations reflect actions that must be taken to implement the lines of code, and to make the written code fit with the rest of the programmed solution. In the SGOMS model implementational unit tasks correspond to productions that implement a step (by writing code to the Python solution file – there is exactly one of these in a given planning unit), and productions that retrieve the variables or condition necessary to implement the step.

Planning Unit	Unit Tasks
calculate_average	1) Fire initialize_variables planning unit 2) Fire iterate_loop planning unit 3) Fire request_variables unit task 4) Fire type_calculate_average unit task
initialize_variables	1) Fire request_variables unit task 2) Fire type_initialize_variables unit task
iterate_loop	1) Fire request_variables unit task 2) Fire type_initialize_variables unit task
...	...
...	...

Figure 4.2 Subset of planning units and unit tasks used in the SGOMS model

The high-level unit tasks implement a planning unit hierarchy by allowing planning units to call upon other planning units as part of their sub-algorithm. When a high-level unit task of a planning unit calls another planning unit, it redirects the model to that new planning unit and this unit must be completed first. Once the called upon planning unit is resolved, i.e., the unit tasks its sub-algorithm requires have all been completed, the model redirects to the next unit task of the calling planning unit. For example, the *calculate_average* planning unit has as its first unit task to call upon the *initialize_variables* planning unit. Thus, the model is redirected to first complete the unit tasks of the *initialize_variables* planning unit; those unit tasks include requesting the variables to be initialized and implementing the step (writing the initialization of the variables to the Python solution file). Once the *initialize_variables* planning unit is complete, the model is redirected back to the *calculate_average* planning unit, and to its next unit task, which is to call upon the *iterate_loop* planning unit (see Figure 4.2). The *calculate_average* planning unit also has implementational unit tasks that request the variables used by the program (*sum* and *count*) and implement the step to calculate the average using those variables.

Program generation begins by calling upon 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). The sub-algorithm includes unit tasks for both high-level (requests to other planning units) and implementational-level (requesting variables/conditions and implementing steps) productions. 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 upon 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 redirects the model to the next unit task in the planning unit that called it. Program generation ends when the highest-level planning unit implements its final unit task; for the rainfall problem this would be writing the calculate average expression in its Python program.

By using planning units to organize information, it is possible to give an ordered structure to program generation without generating an entire algorithm in advance. In this way the SGOMS model more closely mimics the behavior shown by experts and the more successful novices in my study. By relying on planning units instead of an algorithm, the model is also given the capacity to recombine the planning units to generate different solutions, whereas the algorithm-driven models are forced to implement the order of goals and steps they are initialized with. This reflects the ability of the SGOMS model to be more 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, and it can do so using only the planning units and productions currently present.

The key differences between this model and models 1-4 are as follows. Similar to the SGOMS model, model 4 also did not have a complete algorithm. Model 4 relied on associations in its declarative memory to generate the goals, but could not specify the exact relationship between associated goals. Consequently, model 4 had problems 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. While SGOMS has a structure and hierarchy to the goals, it does not have a *pre-existing* complete algorithm (like models 1-3) that defines the implementation of the total solution. For example, in Figure 4.2, the planning unit to *calculate average* defines the need to initialize the planning units for *iterating the loop* and *initializing the variables*. However, the *calculate average* planning unit does not define which planning units need to be initialized later on by the other planning units. Therefore each planning unit functions semi-independently, and can be called upon by any number of other planning units, so long as 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 algorithm-driven models would struggle with.

4.5 Model Output

All of the algorithm-driven models and the SGOMS driven model were able to produce a correct python program (see Figure 4.3). The *goal-expansion* model was not able to produce a program that could solve the problem (Figure 4.3 b).

a) Model 1/2/3/5

```
sum = 0
count = 0
for x in list:
    if x == -999:
        break
    if x>= 0:
        sum += x
        count += 1
average = sum/count
```

b) Model 4

```
average = total/count
total= 0
count= 0
    if x  >=0:
        count+=1
        sum+= x
for x in rains:
    if x == -999:
        break
```

Figure 4.3 Solutions generated by the models. The three algorithm-driven models and the SGOMS model produced the same, correct solution (a), while the *goal-expansion* model did not produce a correct solution (b)

4.5.1 Output for Model 1 (Proceduralized Model)

1.Goal: **[calculate average]** I am starting the Problem to calculate the average of the positive numbers
2.Step: < *initialize variable* > sum=0
3.Step: < *initialize variable* > count=0
4.Step: < *iterate loop* > for x in list:
5.Step: < *stop loop* > if x == -999: break
6.Step: < *condition* > if x>=0:
< *increment variables* > sum+=x, count += 1
7.Step: < *calculation* > average = sum/count

Figure 4.4 Chronotranscript output of *proceduralized* model's goals and steps.

The chronotranscript output for the *proceduralized* model is in Figure 4.4. Recall that this model represents a problem-solver who does has the algorithm embedded in the productions and these automatically produce the solution with no additional reflection (e.g., retrieval of goals from memory). This is why the solution does not include goals. This model's output did not match expert or novice performance, as in both cases participants needed to engage in goal identification to identify the necessary steps and implement them. Thus this demonstrates how a proceduralized approach to problem-solving would be insufficient to describe expertise in programming problem solving.

4.5.2 Output for Model 2 (Algorithm Retrieval)

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

Figure 4.5 Chronotranscript output of *algorithm-retrieval* model's goals and steps.

The chronotranscript output for the *algorithm-retrieval* model is in Figure 4.5. Model 2 represents a problem solver that has the algorithm committed to its declarative memory but not proceduralized. Hence upon encountering the problem statement it begins retrieving the goals of the algorithm and implementing them. For example, it retrieves the goal to initialize variables for the total and count (Figure 4.5 line 1) and then immediately implements it (Figure 4.5 line 2). The solution path it followed was the same as for the canonical, hence its chronotranscript most closely replicated the canonical solution (more so than any of the novices, experts or other models), with goals recalled in the order of the canonical and implemented as they are identified (just as was done in the canonical).

4.5.3 Output for Model 3 (Algorithm Generation)

The output for the *algorithm-generation* model is in Figure 4.6. The *algorithm-generation* model represents a problem solver that identified all of its goals by reading the problem statement, generated the complete algorithm, and then implemented it. Unlike with the retrieval and implementation only model, this model needed to generate

all of its goals (by adding them to its declarative memory) into a solution plan (Figure 4.6 lines 1-5) and only then was it able to implement them (Figure 4.6 lines 6-10). Like the prior algorithm-driven models, the solution produced by this model also followed the canonical solution.

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

Figure 4.6 Chronotranscript output of *algorithm-generation* model's goals and steps.

As noted in Section 4.3.3, this model closely resembles the style of novice programmer identified by Castro and Fisler (2020) as one-way. This style was used by novice programmers, who first formulated a high-level plan (algorithm), with all of their goals, and then implement the steps to resolve it. Unlike the one-way programmers, who had difficulties correctly connecting their code steps together for a cohesive solution, model 3 managed to solve the problem and recreated the canonical solution. This was because model 3 was similar to model 2 in that all of the steps the model could write had no incompatibilities with the other steps the model could write. Model 3 more closely matched the cognitive process of one-way programmers. Model 3 could be made to better match the novice performance of one-way programmers, by incorporating more general productions for implementing the basic programming expressions. With a set of more

general productions, that produce code less specified to the target problem (compared to the productions the model relies on now), the model, just like the one-way style programmers, would have more difficulty correctly connecting its steps.

4.5.4 Output for Model 4 (Goal Expansion)

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 4.7 Chronotranscript output of *goal-expansion* model's goals and steps.

The output for the *goal-expansion* model is in Figure 4.7. Unlike the other models, model 4 was able to deviate from the canonical solution but was unable to correctly program a solution (possibly with sufficient runs it would accomplish it by random chance). Specifically, it had difficulties in correctly ordering its steps in the Python file. For example, it identified the goal to calculate the average (Figure 4.7 line 1) and then immediately wrote the line to the top of the Python file to accomplish the goal (Figure 4.7 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 4.7 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.

The model was capable of producing some of the behavior Rist (1989) attributed to novices. It identified goals from the problem statement, and engaged in program expansion to fill in additional goals and steps. In this way it was able to connect together stopping the loop, and iterating through the list (Figure 4.7 lines 7-10). By expanding from the program focus of iterating the list, it correctly connected the expressions together, but was unable to connect both expressions to the broader problem statement of calculating the average.

This model did not reflect the behavior shown by most of the novices who solved the rainfall problem in my study. This is likely due to the fact that the rainfall problem is very simple, and this type of buggy behaviour, as in the model, would only be observed in novices with a problem of sufficient difficulty. In general, the *goal-expansion* model at present generates solutions based on the presentation order of keywords it extracts from the problem statement. Thus, adding more refined NLP functionality may be needed to appropriately assess its validity as a model of novice programming.

4.5.5 Output for Model 5 (SGOMS)

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 4.8 Chronotranscript output of SGOMS model's goals and steps.

The output for the SGOMS model is in Figure 4.8. The SGOMS model was able to successfully construct the canonical solution without having a complete algorithm specific to the problem (like models 1-3). Additionally, it was able to replicate the expert behavior of identifying multiple goals before implementing them (Figure 4.8 lines 1 and 2) though this was restricted to the main problem goal of calculating the average, which it identifies at the start but does not implement until the very end (Figure 4.8 lines 1 and 10).

For the future it would be beneficial to incorporate a capacity to construct a GAP tree as described by Spohrer et al., (1985) and 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 in relation. At present, while there is a hierarchical structure to the planning units, the overall problem hierarchy is unknown. For problems more complex than the rainfall problem, the current iteration of the SGOMS programming model may have difficulties managing more complex arrangements of goals/planning units and determining the best implementation order. One possible way of resolving this could be to add an additional buffer that constructs and tracks a hierarchy tree of goals, and the addition of declarative knowledge of how to best arrange multiple subgoals during implementation (such as tracking variables always happen within a loop) to the declarative memory.

Chapter 5: Discussion, Limitations and Future Work

The aim of this thesis was to design computational models of programming capable of solving programming problems and modeling differences between expert and novice problem solvers. I begin by discussing the study conducted that helped inform the models.

5.1 Use of Goals and Steps in Programming

All experts and many of the novices were capable of generating correct solutions. However, neither group followed the exact solution process of the canonical solution, nor did their solution reflect the same order of goals and steps. This means that neither group was limited to a single algorithm and that participants were capable of identifying and implementing diverse goals and steps at a given point. The order of goal identification was highly variable though some general trends existed. Their behaviors are not out of line with some prior proposals. Specifically, Soloway's (1986) and Spohrer et al.'s (1985) GAP trees provide a framework that supports variability in goal identification and step implementation. Problem goals are decomposed into multiple branches of subgoals that can be addressed in variable order (see Figure 5.1 for the GAP tree for the rainfall problem). However, there were irregularities between the goals identified by the participants themselves and the goal decomposition structure used by Soloway (1986) and Spohrer et al., (1985). For instance, the GAP tree for the rainfall problem divided the problem into goals based on the calculation for the problem (computing the sum, computing the total and computing the division - see Figure 5.1) while participants in my study used more programming focused goals (iterating through the list using a loop and initializing the variables).

Soloway (1986) focused on analyzing buggy programs written by novices and used the GAP tree framework to address the potential sources of their bugs. However, his analysis involved final solutions only and thus did not include process data. By adding the talk-aloud and process analysis, I found that programmers (expert and novices) did not seem to identify the same goals that Soloway's GAP trees used. For example, while programmers in my study would identify the need to track the sum and count of the positive numbers in the list, they would also separately identify the need to iterate through the provided list, often before even implementing tracking the sum and count. Soloway did not identify iterating through the list as a separate goal but rather as part of separate plans to track the sum and the count, respectively. Soloway's (1986) goal decomposition structure also made it difficult to identify where common programming expressions may be used for the goals, such as for example knowing that incrementing the variables for computing the sum and count could be done within a single loop structure.

Similar to what was reported in Soloway (1986), I found that novices had additional sources of variability in their solutions (see Chapter 3 Section 3.7.3). By collecting the participant's talk alouds, I had direct access to the goals they were attempting to address, and thus had a clearer understanding of where and how they deviated from a correct programming solution.

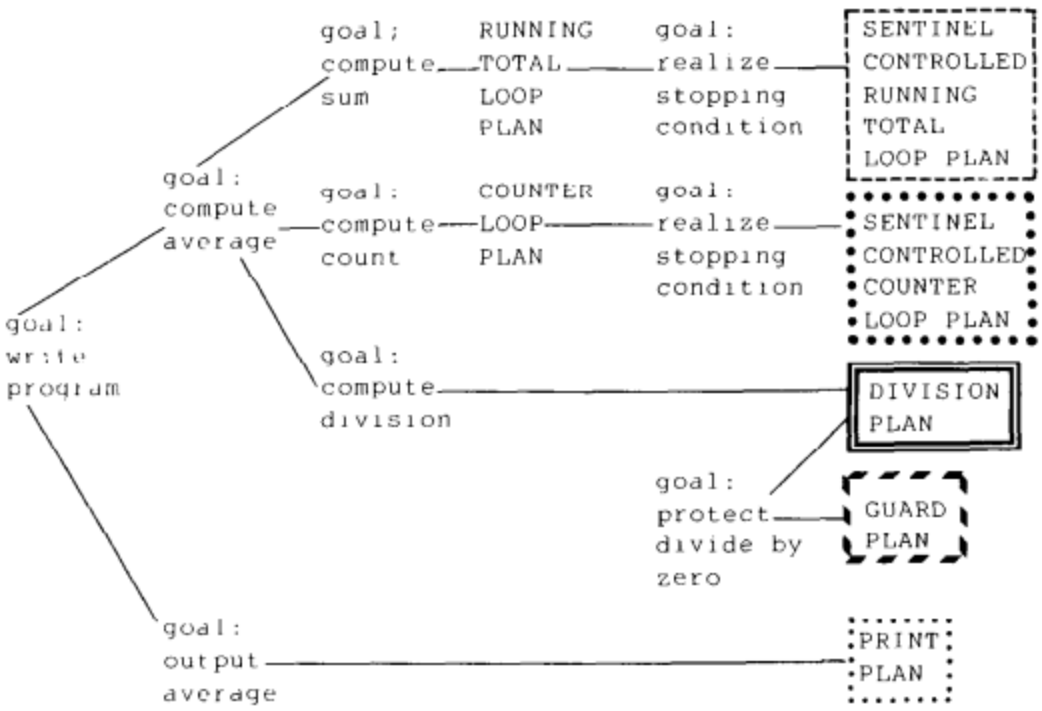

Figure 5.1 Soloway's (1986) depiction of a GAP Tree for the rainfall problem.

5.2 Modelling Programming Problem Solving

The aim of conducting the study was to identify the goals and steps used by experts and novices, in order to facilitate the development of models of programming problem solving. Specifically, the common goals and steps identified in both expert and novice solutions were used to create the declarative knowledge chunks (goals) and productions (steps) that would implement the solutions. Earlier models of programming problem solving 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 coherent solution. My models are quite limited in scope, only being capable of handling one

problem at present (the rainfall problem). However, 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 model 4).

As expected, the *proceduralized* model did not accurately reflect novice or expert behavior, as both novices and experts broke down the primary problem goal into numerous goals and did not simply implement a complete solution plan automatically. The *algorithm-retrieval* model best replicated the canonical solution (as it was directly based on it) but it required the participants to have a full solution algorithm in declarative memory. None of the participants demonstrated having such a declarative memory structure, as they took extra time to understand how to best tackle the conditions of the problem, rather than just retrieving a known solution from their memory. It was surprising that none of the novices used the one-way programming style, as identified by Castro and Fisler (2020) and modelled by the *algorithm-generation* model (model 3). Both novices and experts interwove their goal identification and step implementation, and none seemed to plan out their whole solution process in advance.

Novices were expected to be best modelled by the *goal-expansion* model (model 4), that reflected Rist's (1989) framework of a novice approach to problem solving. Rist argued that 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. While some novices programmed around identified code fragments, such as participant P109 who programmed around a statement that defined the name of a list averaging function (Figure 3.11 lines 2-6), these novices also tended to struggle more and lacked knowledge or held misconceptions regarding the programming

steps and functions used themselves. More competent novices seemed to produce code and chronotranscripts that were more similar to the experts and the SGOMS model – albeit with some errors and misconceptions such as participant P100 (see Chapter 3 section 3.7.3). Hence it did not seem that either the Rist framework or model 4 accurately modelled all of the novices.

Differences between the performance of some novices and models of novice behaviors (i.e., model 4) may be due to the simplicity of the rainfall problem not necessitating a more generative problem-solving process by not requiring many goals and steps rather than inaccuracies in the Rist (1989) model. Additionally, since the models lacked a revision and reflection function, it was difficult to engage in program expansion, as the model was not able to write steps that would precede other already written steps in the Python file. For example, the models were not able to initialize variables at the top of the file if they 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 canonical 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 model that best represented the performance of both the experts and competent novices in my study was the SGOMS model, as it produced a chronotranscript that showed the greatest degree of similarity to the chronotranscripts produced by experts (and some novices) as determined by a qualitative analysis. It replicated some of the interesting behaviours apparent in the expert chronotranscript, such as identifying goals

before they needed to be addressed and identifying multiple goals in a row (without step implementation between them), thus demonstrating some pre-planning capabilities.

5.3 Limitations and Future Directions

More work is needed to identify the library of goals that programmers rely on. None of the goals identified by either novice or expert programmers in my study matched the goal decomposition provided by Soloway (1986) for the rainfall problem (with the exception of the goal to calculate average). Identifying these goals would require more talk-aloud studies of programming problem solving.

I implemented five computational models that differed in their approaches to program generation. While models were informative, there are many future improvements that could be made to facilitate further research. Models would benefit in terms of validity, if they had productions capable of reflecting on and revising the programs written. This problem with the lack of reflection and revision is most apparent in the inability of the models to capture the novice and expert behaviour of iterating through the loop (and often implementing other steps) before initializing the variables. Reflection and revision productions would allow the models to understand what they have already written, and insert new code before already written code, thus being able to initialize variables after implementing the loop iterate function and still producing a correct solution. Given that there were no reflection and revision productions, the models were limited in needing to follow the canonical solution in regards to certain functionalities (e.g. initializing the variables, as otherwise a coherent program would not be written). However, variability of implementation of expressions within the loop

(stopping the loop and tracking the variables) could be captured, by changing statements within the declarative memory (representing the programmer's knowledge base).

In spite of these limitations, the models presented in this thesis matched novice and expert performance better than earlier developed models such as PROUST, as they relied on goals provided expert and novice programmers (via talk-aloud). Additionally, they were capable of capturing various problem solving strategies from both the study conducted and prior research. The models developed for this thesis provide a good starting point for future work. Future work on the SGOMS model specifically should look at implementing some higher level GAP tree style frameworks, and allow the model to track a high-level level schema of their planning and implementation – though the goals used should reflect goals identified by programmers, and not the goals used by Soloway (1986). This would allow it to handle more complex nested planning units. In doing this it would be better able to capture the cyclic programming style demonstrated by good student programmers (Castro and Fisler, 2020) and the programming shown by experts in my study. Additionally, models should be expanded to solve multiple problems. At present in only being able to solve one problem, mechanisms to organize the same goals and steps for multiple problems have not been formalized and future models built should address this.

Appendices

Appendix A

Contains the pretest and pretest grading scheme.

A.1 Pretest

Participant Id : _____ **Date: _____**

Note: Pretest Questions where you are asked to write a program you may use python or pseudocode. Do not worry about perfect syntax, you will not be asked to run them.

1. Please list all university computer science or related courses you have taken:
2. List the names of 2 different sorting algorithms (no need to describe, just list).
3. Write a program that processes numbers from 1 to 50 as follows: (1) if the number is a multiple of three AND five, the program prints “FizzBuzz”; (2) if the number is a multiple of three (but not five), the program prints “Buzz”, (3) if the number is a multiple of five (but not three), the program prints “Fizz and (4) otherwise the program prints the number.
4. Suppose you have two variables, x and y. Write code to swap their value.
5. A standard method for finding a number in a list of sorted numbers is to start at the beginning and check each number until we get to the number we are looking for.
    a. Write code that would find a desired number.
    b. Can you think of a faster way to solve this problem? Explain in plain English (no need to write code).

A.2 Grading Scheme

Pretest

Question 1: At least 1 1000 level course for novices / Extensive experience for experts.

Question 2: Any 2 sorting algorithms (common ones include bubble, merge, quick)

Question 3: A Fizzbuzz program

Example:

```
for i in [1:50]:
    if i%15==0:
        print("FizzBuzz")
    elif i%3==0:
        print("Buzz")
    elif i%5==0:
        print("Fizz")
    else:
        print("Fizz")
```

Question 4:

Z = X

X = Y

Y = Z

Question 5:

a) For i=1:length(list)

    if list[i] == desired_number

        return i

b) Description of a search algorithm (common one is binary search)

Appendix B

Contains introduction to problem solving component and the rainfall problem that participants solved.

B.1 Introduction to Problem Solving

```
Welcome to the Problem Solving Component of the Experiment!

The Tabs above will lead you to the 4 Programming Problems we would like y
ou to solve. Please solve each problem in order.
For each problem you are given 10 minutes to write a small python program
to solve it. DURING THESE 10 MINUTES YOU CANNOT RUN/TEST YOUR CODE.
Only once you have fully written a program that you believe solves the pro
blem, can you run/test it.
Should any error or bugs be revealed, you will have 2 MINUTES to DEBUG bef
ore running the code one final time.
Even if the code still does not work please move on to the next problem.

To summarize, you will have a 10 minute writing period, a test run, a 2 mi
nute debugging period and a final run, for each programming problem.
In addition to stopping you at the 10 min and 2 min mark for program writi
ng and debugging respectively, we will also warn you at the 5 min and 8 mi
n mark for the program writing component.

As you are problem solving, please TALK-
THROUGH your thought process (as you did during the warm-
up) as we would like to know what you are thinking.

Finally if you have any questions in regards to navigation or clarificatio
n please do not hesitate to ask, but we cannot provide syntax or problem-
solving relevant help.
Thank you for participating! :)
```

B.2 Rainfall Problem

```
"""
Design a program that has as a first line in it a list of numbers represen
ting daily rainfall amounts.
The list may include zero or more -
999 values – these indicate an error and as soon as one is encountered, th
e program stops processing the list.
The program outputs the AVERAGE of the NON-
NEGATIVE values in the list up to the first -999 (if it shows up).
There may be negative numbers (as although there cannot be negative rainfa
ll there may be erroneous inputs)  other than -999 in the list.
The program must work with a list of any length

==================================

Here is an example. Suppose we have the list [50, 400, 300, -27, 50, -
999]. The program outputs:

Problem_1[]: 200

Where -27 is ignored in the list and -999 is the stop signal.

Write your code using the input provided below.
"""

from Give.Givens import rains
# this is your input list, you will not see it until you run your code
y = rains
```

Appendix C

General model components, common to all models.

C.1 Motor Module

```
class MotorModule(ccm.Model):
    def type(self, text):
        with open('output_file.py', 'a') as out:
            print (text, file = out
    def talk(self, text):
        with open('output_talk.txt', 'a') as chrono:
                print (text, file = chrono)

class MyAgent(ACTR):
    motor=MotorModule()
```

This is the motor module common to all models. The motor module can type and talk. When using the type function, the motor module adds code to its Python output file. When using the talk function, the motor module logs its retrieved goals and Python code to a transcript file.

C.2 Environment

```
class Problem_Sheet(ccm.Model):
    rain_problem=ccm.Model(isa='problem', name='rainfall',
status='unsolved', text_exp='Given list, sum positive numbe
rs, stop at first -999')

Agent=MyAgent()
env=Problem_Sheet()
env.agent=Agent
ccm.log_everything(env)
env.run()
ccm.finished()
```

These are the expressions that define the environment. Problem_Sheet is the name of the environment in which problem solving occurs, and the rainfall problem text is defined

within the Problem_Sheet. MyAgent() initializes the problem solving productions (often referred to as the agent) within the environment.

C.3 Declarative Memory Module

```
1.DMbuffer=Buffer()
2.DM=Memory(Dmbuffer)
3.DM.add("step:initialize_variable name1:sum name2:count
 lstep:None nstep:initialize_loop")
4.DM.request("step:? lstep:?lstep")
```

The declarative memory module is accessed by the problem solving productions through the DMbuffer. Information can be either added to the declarative memory using the command DM.add(“”) or retrieved from the declarative memory using the command DM.request(“”).

Appendix D

Below are annotated code samples of the models built. Models were built in Python ACT-R (version CCMSuite3)

Can be found at: https://github.com/CarletonCognitiveModelingLab/CCMSuite3

D.1 Model 1: Proceduralized

```
1.class MyAgent(ACTR):
1a.    focus=Buffer()
1b.    motor=MotorModule()
1c.    DMbuffer=Buffer()
1d.    DM=Memory(DMbuffer)

2.     def init():
3.         focus.set("start")

4.     def start_problem(focus='start', rain_problem= 'status:unsolved' ):
5.         motor.talk('1.Goal:I am starting the Problem to calculate the average
 of the positive numbers')
6.         focus.set('initialize sum' )
7.     def ini_sum(focus = 'initialize sum'):  #initializes sum
8.         motor.talk('3.Step: sum=0')
9.         motor.type_first("sum = 0")
10.        focus.set('initialize count')
11.    def ini_count(focus='initialize count'): #initializes count
12.        motor.talk('4.Step: count=0')
13.        motor.type("count = 0")
14.        focus.set('iterate loop')
15.    def looper(focus = 'iterate loop'):
16.        motor.talk('5.Step: for x in list:')
17.        motor.type("for x in list:")
18.        focus.set('loop stop at -999')
```

Figure D.1 Problem Solving Productions of the Proceduralized Model

The proceduralized model has the algorithm of the problem embedded within the problem-solving productions of the model (productions are defined using the def statements). The first part of the model (Figure D.1 line 1) specifies its buffers and the modules they communicate with (if any). If the buffer does not specify any outside modules it is connected to, it is only used within the problem-solving portion of the model (like the focus buffer see Figure D.1 line 1a). The problem-solving process starts

with the model being initialized into the problem-solving environment (see Figure D.1 line 2) and the focus buffer is set to start the model's problem solving (Figure D.1 line 3). Then the next production (start_problem) fires based on the conditions that a) the focus buffer contains the chunk "start" and b) the rainfall problem in the environment has its status set to unsolved (see Figure D.1 line 4). The model then logs the problem goal using the talk function of the motor module, accessing the motor module via the motor buffer (Figure D.1 line 5); the focus buffer is set to the first step in the algorithm, "initialize sum"(Figure D.1 line 6). Based on the content of the focus buffer, the next production fires, ini_sum, which has as its condition the current contents of the focus buffer (Figure D.1 line 7) and the content of the focus buffer acts as the only condition for the production firing. The production accesses the motor module to implement the step of typing "sum = 0" that will initialize the sum in the Python program in addition to including it as a step in the log (Figure D.1 lines 8 and 9). Finally, the production sets the contents of focus buffer to contain the chunk that will act as the condition for the next step, in this case "initialize count", as can be seen in Figure D.1 line 10.

The setting of the focus buffer in the prior production triggers the firing of the next production in the algorithm ini_count (Figure D.1 line 11). This production initializes the variable count within its Python script and adds that completed step to its log, following that is sets the focus buffer to the next step in the algorithm "iterate loop". The ini_count production uses the exact same process as the ini_sum production (D.1 lines 12-14). The next production in the algorithm, looper, fires once the focus buffer contains "iterate loop" (Figure D.1 line 15) and similarly to the last two productions it fulfils its step (initializing the loop that will iterate through the list), records the step

within its log file and changes the focus buffer to trigger the next production "stop loop at -999". This process continues until the full solution is produced, at which point the problem solving stops and the model turns off.

D.2 Model 2: Algorithm Retrieval

```
DM.add("step:initialize_variable var1:sum var2:count lstep:None nstep:initialize_loop")
DM.add("step:initialize_loop var1:x var2:rain lstep:initialize_variable nstep:stop_list")
DM.add("step:calculate_average var1:sum var2:count lstep:track_variables nstep:stop")
```

Figure D.2 Sample of Goals used to Initialize the DM of the *Algorithm-Retrieval* Model

The *algorithm-retrieval* model stores the entire algorithm for the rainfall problem in its declarative memory. A fragment of the algorithm is shown in Figure D.2, with each line representing a goal, and the slots tracking: (a) the step, which resolves the goal of the declarative memory line; (b) the lstep, which is the previous step; (c) the nstep, which is the step that follows the current target step; and (c) the variables necessary for the target step (var1-var4). The model is shown in Figure D.3.

```
    1.DM.add("step:track_variables name1:x name2:0 name3:count name4:sum lstep:stop_
list nstep:initialize_variable_2")
    2. def init():
    3.     focus.set("request lstep:None")
    4. def requests(focus = "request lstep:?lstep"):
    5.     DM.request("step:? lstep:?lstep")
    6.     focus.set("goal")
    7. def stateGoal(focus = "goal", DMbuffer = "step:?step"):
    8.     motor.talk('Goal:I am going to:' + step)
    9.     focus.set("step")
    10.def ini_variable(focus = 'step', DMbuffer = "step:initialize_variable name1:?
name1 name2:?name2 lstep:None" ):
    11.    motor.talk("Step:"+name1+"=0," + name2 + "=0")
    12.    motor.type(name1 + "= 0")
    13.    motor.type(name2 + "= 0")
    14.    focus.set('request lstep:initialize_variable')
    15.def ini_loop(focus = 'step', DMbuffer = "step:initialize_loop name1:?
name1 name2:?name2 lstep:initialize_variable" ):
    16.    motor.talk('Step: for'+ name1 + " in " + name2 + ":")
    17.    motor.type('for ' + name1 + " in " + name2 + ":")
    18.    focus.set('request lstep:initialize_loop')
```

Figure D.3 Sample of the *Algorithm-Retrieval* Model's Problem Solving Productions

Problem solving begins when the model fires the *requests* production (Figure D.3 line 4) to retrieve the first goal to be implemented from the declarative memory (Figure D.3 line 5), the first goal being initializing the variables. Additionally, the *requests* production sets the focus buffer of the model to "goal", which is the condition for the production state_goal (Figure D.3 line 7). The state_goal production is what allows the model to log its goals and add them to the chronotranscript. After this the focus buffer is set to “Step” so that the model will locate a production whose precondition matches both the contents of the focus buffer of the model (step) and the contents of the DMbuffer (the goal). An example of that is illustrated by the production ini_variable (Figure D.3 line 10) that fires when the focus buffer is set to step and the content of the DMbuffer contain the goal to implement the step “initialize variables”. Then the appropriate production (in this case ini_variable) implements the step that resolves the current goal. The production has the motor module write the step to the model’s output file, and adds it to its chronotranscript (Figure D.3 lines 11-13). Then ini_variable sets the focus to “request lstep: “ and sets the chunk lstep to the name of the step just implemented (ini_variable) (Figure D.3 line 14). This sets the focus buffer to contain the conditions for firing the requests production again, so that the requests production retrieves the next goal from declarative memory, in this case it would be to iterate through the list using the ini_loop production (Figure D.3 lines 15-18). The cycle repeats until all of the goals in the algorithm are resolved, and all of the steps implemented.

D.3 Model 3: Algorithm Generation

```
1.class MyAgent(ACTR):
    focus=Buffer()
    motor=MotorModule()
    DMbuffer=Buffer()
    DM=Memory(DMbuffer, finst_size=5,finst_time=300.0)

2.    DM.add("calculate_average average keywords")
3.    DM.add("calculate_average initialize_variables sum count ")
4.    DM.add("initialize_variables track_variables ")


5.    def textparse(focus = 'read', rain_problem = "status:unsolved text_exp:?text_exp"):
6.        self.text_list = text_exp.split()
7.        focus.set("check")

8.    def read_list(focus = "check"):         if len(self.text_list) != 0:
9.            i = self.text_list.pop(0)
10.           try:
               int(i)
11.               DM.add("stop_list ?i no_keyword")
12.               self.focus.set('related step:stop_list')
13.           except ValueError:
14.               DM.request("? ?i keywords")
15.               self.focus.set("identify")
16.       else:
           self.focus.set('request lstep:None')

17.    def no_id(focus = "identify", DM = 'error:True'):
           focus.set("check")
```

Figure D.4 Initialization of *Algorithm-Generation* Model and Productions that Process the Problem Statement

The *algorithm-generation* model generates the goals of the algorithm in its declarative memory and then implements it. It begins by reading through the problem statement and translating it into the goals of the algorithm. To accomplish the translation step the model relies on a very rudimentary keyword search of the problem statement. Specifically, the model is initialized with chunks in its declarative memory representing keyword-goal associations. To start, the production (text_parse) converts the problem statement in the model's environment into a list of words that the model can iterate through to check for keywords (Figure D.4 lines 5-7). This is not analogous to human cognitive performance (humans do not tend to make lists composed of the words we are

reading), but as our focus is not on the NLP aspects, I have simplified the processing in this way. Currently, the productions that manage this process are very specific to the rainfall problem, and thus the model would have difficulty generating an algorithm for a different problem statement even if the python expressions needed were the same.

Once the text_parse production converts the problem statement into a list of the words of the problem statement, the read_list production checks each word in the list. If the current word in the list is not a keyword, or the declarative memory fails to retrieve it, the no_id function fires to redirect the focus back to the read_list production to check the next word in the list (Figure D.4 line 17). If the word a keyword the appropriate production fires based on the focus buffer contents and the retrieved keyword – goal chunk from memory.

```
1.    def related_goals_search(focus = "related goal:?goal"):
2.        DM.request("?goal ? ")
3.        focus.set("identify")

4.    def average_id(focus = "identify", DMbuffer = "calculate_average average keywords"):
5.        motor.talk('Goal: I should calculate the average of the positive numbers')
6.        DM.add("step:calculate_average name1:sum name2:count lstep:track_variables nstep:st
op")
7.        focus.set("related goal:calculate_average")

8.    def initialize_var_id(focus = "identify", DMbuffer = "calculate_average initialize_vari
ables sum count"):
9.        motor.talk('Goal: I should initialize the variables sum and count to track the posi
tive numbers')
10.        DM.add("step:initialize_variable name1:sum name2:count lstep:None nstep:initialize
_list")
11.        focus.set("related goal:initialize_variables")
```

Figure D.5 Sample of the *Algorithm-Generation* Model's Productions that Control Goal Identification and Step Implementation

When read_list successfully retrieves a keyword association from declarative memory, a goal-specific production fires that adds the appropriate goal into the declarative memory. For example, when the keyword "average" is encountered, the keyword-goal association stored in the model's declarative memory (see Figure D.4 line

2 for the model being initialized with that fact) is retrieved and stored in the DMBuffer by the read_list production. Once the focus buffer contains the chunk "identify" (letting us know the model will need to identify the goal) and the DMBuffer contains the calculate_average keyword association, the average_id function fires (Figure D.5 line 4). The average_id function then adds the goal to calculate the average to its declarative memory (Figure D.5 line 6) and adds the calculating the average goal to its log (Figure D.5 line 5). The goal added to the declarative memory is the same as calculate average goal used in the algorithm by model 2, and uses the same slot values.

Since not all goals are directly readable in the problem statement the model is also capable of searching its declarative memory for goal – goal association. When the focus buffer is set to search for related goals, the related_goals_search production makes a request to the declarative memory to check whether it has a goal – goal association (Figure D.5 line 2) and sets the focus buffer to "identify". If there is no goal – goal association, the no-id production fires (just as in the case that there is no keyword association) and the model is redirected to the next word in the list. If there is a goal – goal association in memory (as there is for calculating average and initializing the variable) then the ID production for the non-keyword goal fires. In the example, the inititialize_var_ID production fires, as its conditions are "identify" in the focus buffer and the calculate_average – initialize_variables association in the DMBuffer (Figure D.5 line 8). It then adds the goal to its declarative memory (Figure D.5 line 10) and logs its goal (Figure D.5 line 9). When all the words in the list have been processed, the read_list production sets the focus buffer to trigger the problem solving process (Figure D.4 line 16). The problem solving process, which translates goals to steps, then proceeds

identically to the *algorithm-retrieval* model, as the algorithm the *algorithm-generation* model generates is identical to the one the *algorithm-retrieval* model is initialized with, and the step implementation productions are identical as well.

D.4 Model 4: Goal Expansion

```
1. class MyAgent(ACTR):
    focus=Buffer()
    motor=MotorModule()
    DMbuffer=Buffer()
    plan_step = Buffer()
    DM=Memory(DMbuffer, finst_size=5,finst_time=30.0)
    text_list = [] #represents the text of the problem being read
2.   def init():
3.       DM.add("step:calculate_average keyword:average request:variables costeps:Yes ")
4.       DM.add('step:stop_loop_iterate keyword:stop request:variables costeps:No ')
5.       DM.add('step:initialize_variables keyword:costep request:variables costeps:No costep:calculate_average')

6.       DM.add("keyword:average request:step variable1:total variable2:count variable3:None variable4:None")
7.       DM.add('keyword:stop request:when variable1:None variable2:None variable3:x variable4:None')

       focus.set("read")

8.   def textparse(focus = 'read', rain_problem = "status:unsolved text_exp:?text_exp"):
       self.text_list = text_exp.split()
       focus.set("check")

9.   def read_list(focus = "check"):
       if len(self.text_list) != 0:
           i = self.text_list.pop(0)
           print(i)
10.          try:
               int(i)
11.            DM.add('keyword:stop request:step variable1:None variable2:None variable3:x variable4:?i')
12.            self.focus.set('special')
13.          except ValueError:
14.            DM.request("keyword:?i request:variables")
15.            self.focus.set("understand")
16.      else:
           self.focus.set("stop")
```

Figure D.6 The Initialization of the *Goal-Expansion* Model and its Productions Which Process the Problem Statement

The Goal Expansion model is initialized with keyword associations for goals (Figure D.6 lines 3-5) and associated variables (Figure D.6 lines 6 and 7), as was the case for model 3. Keyword associations for the goals provide some information through the following slots: (a) step - is the step which would resolve the goal chunk; (b) keyword - the keyword associated with the goal; (c) request - identifies the information the goal needs to implement the step; (d) costeps - states if the goal requires the resolution of any other goals; (e) costep - states the goal that requires the current goal to resolve. If the

keyword is costep, that means it is not a focal goal identified from keyword association with the problem text, but one of the goals which is expanded around it; such as in the case of initialize_variables which is required for the resolution of calculate_average, so its keyword is “costep” and it has calculate_average listed as its costep (Figure D.6 line 5).

Additionally, the model is initialized with keyword associations for variables which would be used by a given step. This is done to keep track of the variables and conditions of the problem and which ones are relevant to which goal-step combo; the variables and conditions are treated this way to allow for expansion of the model for non-rainfall problems. The following information is tracked by the following slots: (a) keyword – the associated keyword for the variables, which is used when later productions request the variables to implement the step; (b) request - identifies what the variables need to make sense (will always be step, as only a goal-step combo can request variables currently); and (c) variable(x) - tracks the necessary variables for implementing the step-goal associated with the keyword.

```
1.    def variable_request(focus = "understand", DMbuffer = "keyword:?keyword request:variables step:?
step costeps:?costep"):
2.        plan_step.set("keyword:?keyword request:variables step:?step fire:Yes costeps:?costep")
3.        DM.request("keyword:?keyword request:!variables")
4.        focus.set("variable")

5.    def goal_initialize(focus = "variable", DMbuffer = "keyword:?word request:step variable1:?A variable2:?
B variable3:?C variable4:?D", plan_step = "keyword:?word request:variables step:?step fire:Yes"):
6.        DM.add("step:?step name1:?A name2:?B name3:?C name4:?D fire:Yes")
7.        motor.talk("I need to:" + step)
8.        focus.set("recall step")

9.    def step_recall(focus = "recall step", plan_step = "keyword:?word request:variables step:?
step fire:Yes"):
10.        DM.request("step:?step fire:Yes")
11.        focus.set("step")

12.    def ini_variable(focus = 'step', DMbuffer = "step:initialize_variables name1:?name1 name2:?
name2 fire:Yes" ):
13.        motor.talk("Step: " + name1 + "= 0, " + name2 + "= 0")
14.        motor.type(name1 + "= 0")
15.        motor.type(name2 + "= 0")
16.        focus.set('request costeps')
```

Figure D.7 *Goal-Expansion* Model's Productions Which Generate Goals and a Step Implementing Production

Language processing occurs in the same way as the *algorithm-generation* model, using text_parse and read_list, with text_parse generating a list of words of the problem statement and read_list checking each word to identify whether or not it is a keyword. Similarly to model 3, the read_list production adds the stop signal value (-999) to the declarative memory (Figure D.6 line 11), and if the list item is not an integer, the production sends a request to the declarative memory to check if it is a keyword.

If a word in the list made by text_parse and read by read_list matches a keyword - goal association then the variable_request production fires, to request any of the variables or conditions needed by the goal to implement the step (Figure D.7 line 1). This production moves the goal retrieved by the keyword – goal association in the read_list production, from the DMBuffer to the plan_goal buffer, so that the model tracks current goal (Figure D.7 line 2). The plan_goal buffer also tracks whether or not the current goal should fire using the slot "fire"; the slot should be set to yes, until the goal is resolved and the step implemented. The variable_request production then queries the declarative memory for

the variables associated with the step (using !variables in the request, because variables in the DM request step, while goals in the DM request variables) (Figure D.7 line 3) and focus is set to go to “variable” (Figure D.7 line 4) to trigger the production which combines the variables and the goal to trigger the step.

The focus buffer containing the chunk “variable”, the plan step buffer containing the current goal – keyword association, and the DMBuffer containing the variable names associated with the keyword in the model’s declarative memory are the conditions which trigger the firing of the goal_initialize production (Figure D.7 line 5). This production connects the goal – keyword association and the keyword – variable associations into a single actionable goal which it then adds to the declarative memory (Figure 5.10 line 6). It then logs its goal (Figure D.7 line 7) and sets its focus buffer to “recall step” to trigger the production which will request the complete goal from the declarative memory (step_recall). Step_recall requests the goal in the declarative memory that is ready to fire (slot fire set to yes) (Figure D.7 line 10) and sets the focus buffer to “step” (Figure D.7 line 11) to trigger the appropriate step to fire.

If for example the goal which had been added and retrieved from memory was the initialize variables goal (as you would need to initialize variables to solve the rainfall problem), it would then trigger the firing of the ini_variable production (Figure D.7 line 12). Step implementing productions (like ini_variable, calc_average and track_variable) then implement the step by writing the step to its Python file (Figure D.7 lines 14 and 15) and log that step (Figure D.7 line 13). Since piecemeal modelling models program expansion, the focus buffer is set to “request costeps” to trigger a production that will check if initialize variables requires other steps to be implemented to resolve.

```
1.    def request_costep_no(plan_step = "keyword:?word request:variables step:?
step fire:Yes costeps:No", focus = "request costeps"):
2.        plan_step.set("")
3.        focus.set("check")

4.    def request_costep_yes(plan_step = "keyword:?word request:variables step:?
step fire:Yes costeps:Yes", focus = "request costeps"):
5.        DM.request("step:? keyword:costep costep:?step")
6.        focus.set("costep")

7.    def ID_costep(plan_step = "keyword:?word request:variables step:?
step fire:Yes costeps:Yes", focus = "costep", DMbuffer = "step:?step2 keyword:costep costep:?step costeps:?
costeps" ):
8.        plan_step.set("keyword:?word step:?step2 fire:Yes costeps:?costeps request:variables")
9.        DM.request("keyword:?word request:variables")
10.        focus.set("understand_costep")

11.    def variable_request_for_costep(focus = "understand_costep", DMbuffer = "keyword:?
keyword request:variables step:?step2", plan_step = "keyword:?keyword step:?step fire:Yes costeps:?
costeps" ):
12.        DM.request("keyword:?keyword request:!variables")
13.        focus.set("variable")
```

Figure D.8 Productions that Manage Goal Expansion in the *Goal-Expansion* Model

Costeps are tracked in the declarative memory goal – keyword associations (Figure D.6 lines 3-5), and the plan step buffer stores a given goal – keyword association until it is checked for costeps. If the plan step buffer does not have a costep in it, the request_costep_no production fires (Figure D.8 line 1), which clears the plan step buffer (Figure D.8 line 2) and sets the focus buffer to "check" which triggers the read_list production to look at the next word in the problem text list. This is what happens after the variables are initialized, as it was itself a costep for the calculate average goal. If the plan step buffer contains a goal – keyword association that includes a costep (which the goal in the plan step buffer needs for its own resolution) then the request_costep_yes production fires (Figure D.8 line 4) and the declarative memory is queried for a costep for the goal just implemented (Figure D.8 line 5) and the focus buffer is set to "costep". This then triggers the firing of the ID_costep production (Figure D.8 line 7), which sets the plan step buffer to the goal to implement the costep (Figure D.8 line 8) and requests the variables for the costep from declarative memory (Figure D.8 line 9). Once the production sets the focus buffer to "understand_costep" (Figure D.8 line 10), the

variable_request_for_costep production fire, to request the necessary variables for the new goal from the declarative memory (Figure D.8 line 12). This is different from the earlier variable_request production (Figure D.7 lines 1-4) as it does not set the plan step buffer, which has already been set by the ID_costep production. Once the variable_request_for_costep production sets the focus buffer to "variable" (Figure D.8 line 13) the goal with its variables get added to the declarative memory and logged using the earlier described goal_initialize production (Figure D.7 lines 5-8), and implementation proceeds the same. The goal to initialize variables gets added and implemented in this way, as a program expansion from the step to calculate average.

Goals are identified from the word list and via costep search until the model is done reading the list. At which point the Python program is considered finished.

D.5 Model 5: SGOMS

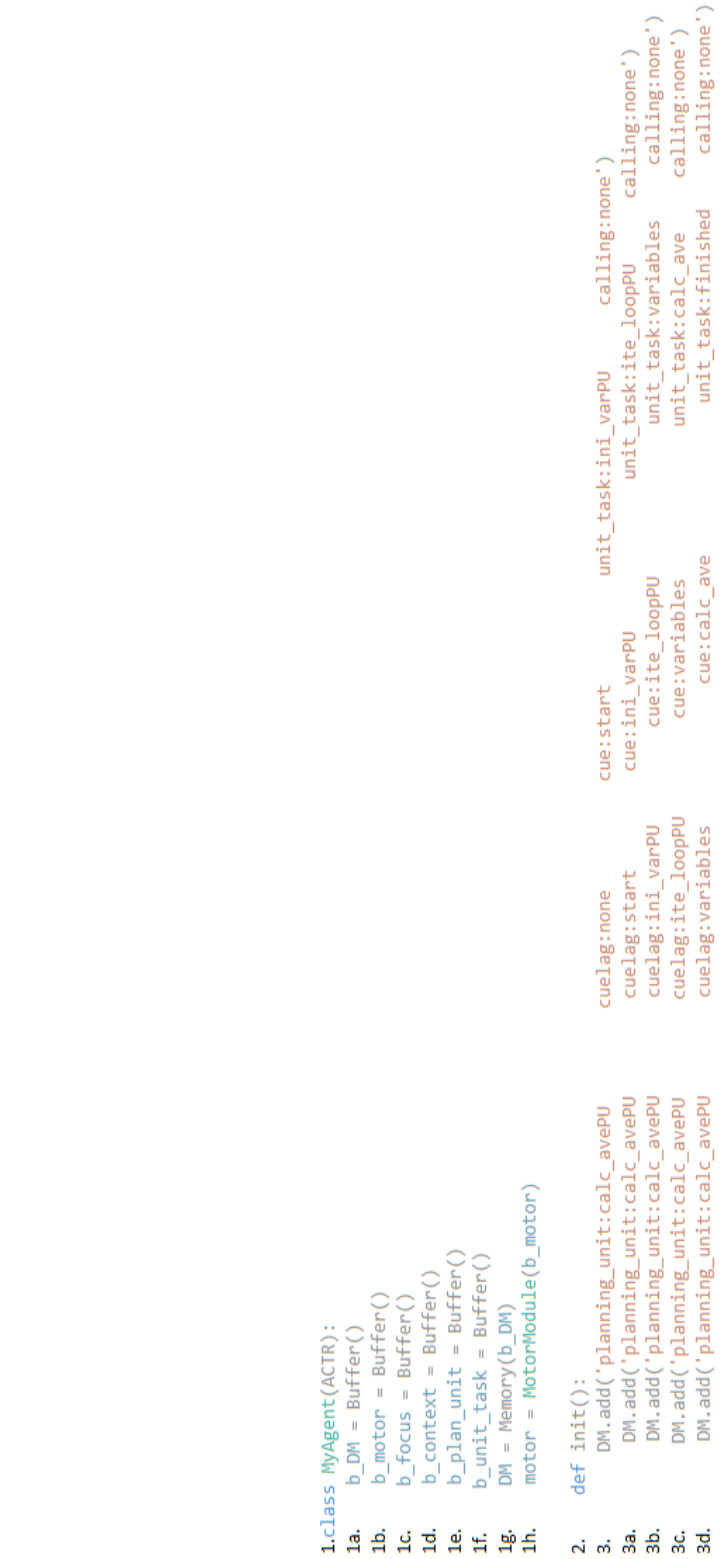

Figure D.9 Initialization of *SGOMS* Model and the Planning Units into its Declarative Memory

The SGOMS model uses an additional set of goal buffers (alongside the focus buffer) to track and shuttle information between productions. These buffers include a context buffer (see Figure D.9 line 1d), which tracks where the model is in the problem solving process (the focus buffer has a slightly different role in this model) and is initialized with the following slots: (a) finished – which contains the most recently finished unit task, and is initialized with none (as no unit tasks have been finished); (b) status – which tracks whether or not the model is currently occupied with a unit task and is initialized as unoccupied (as it does not start mid unit task); (c) conditions and variables – these track the conditions and variables as needed by the model, and are modified by different planning units (which will rely on different variables and conditions), storing them in the context buffer allows them to be accessible for the entire planning unit, and not just a single unit task (see Figure D.9 line 7).

The model's declarative memory is initialized with the planning unit chunks which represent the unit tasks necessary to complete the "calculate average" goal of the rainfall problem (calculate the average of the positive numbers in the list) (see Figure D.9 lines 3-3d). The slots represent the following information: (a) planning_unit - the name of the planning unit declarative memory chunk is part of; (b) cuelag - prior cue for unit task the model completed in planning unit; (c) cue – model's cue for the unit task (also the last unit task completed); (d) calling – which tracks which planning unit requires the current planning unit to resolve and the (e) unit task - the unit task to be implemented by the model. , The planning unit buffer (Figure D.9 line 1e) stores the current line of the planning unit being implemented by the model, the line being any one of the lines initialized in the declarative memory (as in Figure D.9 lines 3-3d). The unit task buffer

(Figure D.9 line 1f) tracks the progress of the model's current unit task (whether it is complete or not). The focus buffer is only used to track the model's process through a unit task, if there are multiple productions which must be implemented (See Figure D.10 lines 16 and 17).

Productions start a planning unit based on the contents of the context buffer. The production that initiates the calculate average planning unit (run_calc_ave; see Figure D.10 line 1) is able to fire as the initialized context buffer can act as a precondition of its fire (since the calculate average planning unit organizes many other planning units in hierarchical order below it, it must be able to fire after many different completed unit_tasks, and may therefore fire under multiple contexts). It then logs its goal to calculate the average (Figure D.10 line 2), sets the planning unit buffer to reflect the first planning unit line in the declarative memory (Figure D.10 line 4), and unit task buffer is set to reflect the first unit task to be completed as part of the overall planning unit (line 3). Additionally, the context buffer is modified to reflect that no unit tasks have been completed for this planning unit thus far (as the planning unit is just starting) (Figure D.10 line 5). Once a unit task is loaded in the unit task buffer, the first unit task in the planning unit will fire.

```
1.    def run_calc_ave(b_context='finshed:?unit_task status:unoccupied '):
2.        motor.talk('Goal: I should calculate the average of the positive numbers')
3.        b_unit_task.set('unit_task:ini_varPU state:running pu_type:ordered')
4.        b_plan_unit.set('planning_unit:calc_avePU cuelag:none cue:start unit_task:ini_varPU state:running')
5.        b_context.set('finished:nothing status:occupied condition:none variable1:none variable2:none')
        print('calculate average planning unit')

6.    def run_ini_var(b_context='finshed:nothing status:unoccupied '):
7.        motor.talk('Goal: I should initialize the variables sum and count to track the positive numbers')
8.        b_unit_task.set('unit_task:variables state:running pu_type:ordered')
9.        b_plan_unit.set('planning_unit:ini_varPU cuelag:none cue:start unit_task:variables state:running')
10.        b_context.set('finished:nothing status:occupied condition:none variable1:none variable2:none')
        print('initialize variables planning unit')

11.    def ini_varPU(b_unit_task='unit_task:ini_varPU state:running'):
12.        b_context.set('finshed:nothing status:unoccupied')
13.        b_unit_task.modify(state='stopped')
        print('initializing variables - PU')

14.    def variable1(b_unit_task='unit_task:variables state:running', b_plan_unit = 'planning_unit:?
PU unit_task:variables'):
15.        DM.request('planning_unit:?PU variable1:? variable2:?')
16.        b_focus.set("var2")
        print('what are the variables?')
17.    def variable2(b_unit_task='unit_task:variables state:running', b_focus = 'var2', b_DM = 'planning_unit:?
PU variable1:?var1 variable2:?var2'):
18.        b_context.modify(variable1 = var1)
19.        b_context.modify(variable2 = var2)
20.        b_unit_task.modify(state='end')
21.        b_focus.set('')
        print('variables retireved')

22.    def ini_var(b_unit_task='unit_task:ini_var state:running', b_context = 'variable1:?var1 variable2:?
var2'):
23.        motor.talk("Step: " + var1 + "= 0" + var2 + "= 0")
24.        motor.type(var1 + '= 0')
25.        motor.type(var2 + '= 0')
26.        b_unit_task.modify(state='end')
        print('intializing variables')
```

Figure D.10 Sample of *SGOMS* Model's Productions that Control Planning Units and Implement Unit Tasks

According to the planning unit for calculating the average stored in the declarative memory (Figure D.9 line 3) the first unit task is to complete the initialize variables planning unit (ini_varPU), therefore the first unit task which fires is the ini_varPU unit task (Figure D.10 line 11), which changes the context buffer (Figure D.10 line 12) to the context which will act as the precondition to the production which starts the initialize variables planning unit (run_ini_var) (Figure D.10 line 6). From Figure 4.2 we know that that first unit task for the initialize variables planning unit is to retrieve the variables to initialize from the declarative memory. Those variables are sum and count (Figure D.9 line 5) and are retrieved by the unit task variables, which the run_ini_var sets its first unit task to in the unit task and plan unit buffer (Figure D.10 lines 8 and 9).

Variables are retrieved with the variable1 and variable2 productions. Variable1 fires when the unit_task is set to variables in the unit task and planning unit buffer (Figure D.10 line 14). It then sends a request to the declarative memory to retrieve the variables needed by the current planning unit (Figure D.10 line 15), and the focus buffer is set to “var2” (Figure D.10 line 16). This is done to ensure the next production (variable2) fires (Figure D.10 line 17) as it is the production which then adds the variables to the context buffer so that they are accessible to the next unit task. From there the unit task is set to implement the step of initializing the variables in the python program (Figure D.10 line 22). It states its step to the chronotranscript (Figure D.10 line 23), then types the code necessary to initialize the variables into its Python file (Figure D.10 lines 24 and 25). Since this completes the initialize variables planning unit, the model returns to its calling planning unit, calculate average, and the next unit task in calculate average is implemented.

```
1.  def request_next_unit_task(b_plan_unit='planning_unit:?planning_unit cuelag:?cuelag cue:?cue unit_task:?
unit_task state:running',
                               b_unit_task='unit_task:?unit_task state:end pu_type:ordered'):
2.      DM.request('planning_unit:?planning_unit cue:?unit_task:? cuelag:?cue')
3.      b_plan_unit.set('planning_unit:?planning_unit cuelag:?cuelag cue:?cue unit_task:?unit_task state:retrieve')

4.  def retrieved_next_unit_task(b_plan_unit='state:retrieve',
                                 b_DM='planning_unit:?planning_unit cuelag:?cuelag cue:?cue unit_task:?unit_task!
finished'):
5.      b_plan_unit.set('planning_unit:?planning_unit cuelag:?cuelag cue:?cue unit_task:?unit_task state:running')
6.      b_unit_task.set('unit_task:?unit_task state:running pu_type:ordered')

7.  def retrieved_last_unit_task(b_plan_unit='planning_unit:?planning_unit state:retrieve',
                                 b_unit_task='unit_task:?unit_task state:end pu_type:ordered',
                                 b_DM='planning_unit:?planning_unit cuelag:?cuelag cue:?cue unit_task:finished calling:?
calling',
                                 b_context = 'finished:?finished status:occupied' ):
8.      b_unit_task.modify(state='stopped')
9.      DM.request("planning_unit:?calling cuelag:? cue:?planning_unit unit_task:? calling:?")

10. def retrieved_last_unit_task_with_call(b_plan_unit='planning_unit:?planning_unit state:retrieve',
                                           b_unit_task='unit_task:?unit_task state:stopped pu_type:ordered',
                                           b_DM='planning_unit:?pu cuelag:?cuelag cue:?planning_unit unit_task:?ut calling:?
calling',
                                           b_context = 'finished:?finished status:occupied'):
11.     b_plan_unit.set('planning_unit:?pu cuelag:?cuelag cue:?planning_unit unit_task:?ut state:running')
12.     b_unit_task.set('unit_task:?ut state:running pu_type:ordered')
13.     b_context.set('finished:?planning_unit status:occupied condition:none variable1:none variable2:none')
```

Figure D.11 Sample of *SGOMS* Model's Production that Control Unit Tasks and Ending Planning Units

While I have given a general overview of how planning units and unit tasks are managed, there are separate productions that manage the retrieval of the unit tasks from within the planning unit and fire whenever a unit task has been completed. The request_next_unit_task production (Figure D.11 line 1) fires based on both the plan unit buffer and unit task buffer storing within them a planning unit with a completed unit task (state = end). Tasks are set as ended by unit task productions when they complete their unit tasks (see Figure D.11 lines 13, 20 and 26). With the request_next_unit_task production the model can then request the next unit task from memory. The next unit task is retrieved by the declarative memory buffer, and will contain the next line in the planning unit (Figure D.11 line 2). A reminder that the planning unit lines are stored in declarative memory (Figure 5.13 lines 3-3d). The planning unit state is also set to retrieve, to let the model know that it will be retrieving and beginning the next unit task (Figure 5.15 line 3). Once the planning unit state is set to retrieve, the retrieve_next_unit_task is fires. The retrieve_next_unit_task production's conditions are for the planning unit to have the state set to retrieve and the declarative memory buffer to have the next planning unit line (Figure 5.15 line 4). The production then sets the planning unit (Figure 5.15 line 5) and unit task buffer (Figure 5.15 line 6) with the next unit task in the planning unit, which is then executed.

When the last unit task in a planning unit is completed, the next unit task requested by request_next_unit_task will be "finished". When that occurs instead of retrieve_next_unit_task firing, the retrieved_last_unit_task will fire (Figure 5.15 line 7). The production sets the unit task state to stopped (Figure 5.15 line 8), to track that the unit tasks of the planning unit have ended. Reminder that the planning units in declarative

memory also store the calling planning unit, that is the planning unit which needs the current planning unit to be resolved to resolve itself (Figure D.9 line 3-3d). The retrieved_last_unit_task production then requests the next step of the calling planning unit, that is the calling planning unit, has the current planning unit as one of its unit task and this production requests the next unit task in calling planning unit. Once the DMbuffer has the next unit task of the calling unit the retrieved_last_unit_task_with_call production fires (Figure D.11 line 10). This production then sets the plan unit buffer to being a line within the calling planning unit (Figure D.11 line 11), the unit task buffer to reflect the next unit task in the calling planning unit (Figure D.11 line 12) and the context is cleared of the all the variables and conditions used by the last planning unit. For example, this set of productions fires when the initialize variables planning unit is completed, and the model returns to address the next unit task in the calculate average planning unit (which is the calling planning unit for initialize variables).

Appendix E

E.1 Sample Log File From Models

time	agent			
	DM	DMbuffer	focus	production
	busy	chunk	chunk	
0.000	False		request *lstep:*None	requests
0.050	True		goal	
0.100	False	*lstep:*None *name1:*sum *name2:*count *nstep:*initialize_loop *step:*initialize_variable		stateGoal
0.150			step	ini_variable
0.200			request *lstep:*initialize_variable	requests
0.250	True		goal	stateGoal
0.300	False	*lstep:*initialize_variable *name1:*x *name2:*rain *nstep:*stop_list *step:*initialize_loop	step	ini_loop
0.350			request *lstep:*initialize_loop	requests
0.400	True		goal	stateGoal
0.450	False	*lstep:*initialize_loop *name1:*x *name2:*-999 *nstep:*track_variable *step:*stop_list	step	stop_loop
0.500			request *lstep:*stop_list	requests
0.550	True		goal	stateGoal
0.600	False	*lstep:*stop_list *name1:*x *name2:*0 *name3:*count *name4:*sum *nstep:*initialize_variable_2 *step:*track_variables	step	track_var_ave
0.650			request *lstep:*track_variables	requests
0.700	True		goal	stateGoal
0.750	False	*lstep:*track_variables *name1:*sum *name2:*count *nstep:*stop *step:*calculate_average	step	calc_ave
0.800			stop	stop_production
0.850				

agent.DM.error	False
agent.DM.latency	0.05
agent.DM.maximum_time	10.0
agent.DM.record_all_chunks	False
agent.DM.threshold	0
agent.production_match_delay	0
agent.production_threshold	None
agent.production_time	0.05
agent.production_time_sd	None
rain_problem.isa	problem
rain_problem.status	unsolved
rain_problem.text_exp	Given list, sum positive numbers, stop at first -999 in list

Example of a log file produced by Python ACT-R built models, which depicts the cognitive process of the models. Buffer contents (in this case the focus buffer and declarative memory buffer) and productions fired are time-stamped.

Bibliography

Altmann, E. M., Cleeremans, A., Schunn, C. D., & Gray, W. D. (Eds.). (2001). A Model of Individual Differences in Learning Air Traffic Control. In *Proceedings of the 2001 Fourth International Conference on Cognitive Modeling* (0 ed., pp. 374–384). Psychology Press.

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

Bertels, K., Vanneste, P., & Backer, C. (1992). A cognitive model of programming knowledge for procedural languages. In I. Tomek (Ed.), *Computer Assisted Learning* (Vol. 602, pp. 124–135). Springer Berlin Heidelberg. http://link.springer.com/10.1007/3-540-55578-1_63

Bertels, Koen. (1994). A Dynamic View on Cognitive Student Modeling in Computer Programming. *Journal of Artificial Intelligence in Education*, *5*(1), 85–106.

Braithwaite, D. W., Pyke, A. A., & Siegler, R. S. (2017). A computational model of fraction arithmetic. *Psychological Review*, *124*(5), 603–625. https://doi.org/10.1037/rev0000072

Byckling, P., & Sajaniemi, J. (2006). A role-based analysis model for the evaluation of novices' programming knowledge development. *Proceedings of the 2006 International Workshop on Computing Education Research - ICER '06*, 85. https://doi.org/10.1145/1151588.1151602

Castro, F. E. V., & Fisler, K. (2016). On the Interplay Between Bottom-Up and Datatype-Driven Program Design. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 205–210. https://doi.org/10.1145/2839509.2844574

Castro, F. E. V., & Fisler, K. (2020). Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 487–493. https://doi.org/10.1145/3328778.3366847

Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science*, *13*(2), 145–182. https://doi.org/10.1207/s15516709cog1302_1

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and Representation of Physics Problems by Experts and Novices*. *Cognitive Science*, *5*(2), 121–152. https://doi.org/10.1207/s15516709cog0502_2

Chi, M. T. H., & VanLehn, K. A. (1991). The Content of Physics Self-Explanations. *Journal of the Learning Sciences*, *1*(1), 69–105. https://doi.org/10.1207/s15327809jls0101_4

Corbett, A. (2000). *Cognitive Mastery Learning in the ACT Programming Tutor*. 6.

de Kleer, J. (1990). MULTIPLE REPRESENTATIONS OF KNOWLEDGE IN A MECHANICS PROBLEM-SOLVER. In *Readings in Qualitative Reasoning About Physical Systems* (pp. 40–45). Elsevier. https://doi.org/10.1016/B978-1-4832-1447-4.50009-2

Détienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. In *Psychology of Programming* (pp. 205–222). Elsevier. https://doi.org/10.1016/B978-0-12-350772-3.50018-5

Fisler, K., & Castro, F. E. V. (2017). Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. *Proceedings of the 2017 ACM Conference on International Computing Education Research*, 12–20. https://doi.org/10.1145/3105726.3106183

Fisler, K., Krishnamurthi, S., & Siegmund, J. (2016). Modernizing Plan-Composition Studies. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16*, 211–216. https://doi.org/10.1145/2839509.2844556

Fleischman, E. S., & Jones, R. M. (2002). Why Example Fading Works: A Qualitative Analysis Using Cascade. In W. D. Gray & C. D. Schunn (Eds.), *Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society* (1st ed., pp. 298–303). Routledge. https://doi.org/10.4324/9781315782379-88

Frischkorn, G., & Schubert, A.-L. (2018). Cognitive Models in Intelligence Research: Advantages and Recommendations for Their Application. *Journal of Intelligence*, *6*(3), 34. https://doi.org/10.3390/jintelligence6030034

Gertner, A. S., & VanLehn, K. (2000). Andes: A Coached Problem Solving Environment for Physics. In G. Gauthier, C. Frasson, & K. VanLehn (Eds.), *Intelligent Tutoring Systems* (Vol. 1839, pp. 133–142). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45108-0_17

Gobet, F., & Simon, H. A. (1996). Templates in Chess Memory: A Mechanism for Recalling Several Boards. *Cognitive Psychology*, *31*(1), 1–40. https://doi.org/10.1006/cogp.1996.0011

Johnson, W. L., & Soloway, E. (1985). PROUST: Knowledge-based program understanding. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, *3*, 267–275.

Kahney, H. (1983). What do novice programmers know about recursion. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '83*, 235–239. https://doi.org/10.1145/800045.801618

Kennedy, W. G., & Trafton, J. G. (2007). *Long-Term Symbolic Learning in Soar and ACT-R:* Defense Technical Information Center. https://doi.org/10.21236/ADA478581

Koedinger, K. R., & Anderson, J. R. (1993). Reifying implicit planning in geometry: Guidelines for model-based intelligent tutoring system design. In *Computers as cognitive tools* (pp. 15–46).

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1986). Soar: An Architecture for General Intelligence. *Artificial Intelligence*, 72.

Lebiere, C., Anderson, J. R., & Bothell, D. (2001). *Multi-Tasking and Cognitive Workload in an ACT-R Model of a Simplified Air Traffic Control Task.* 8.

Lebiere, C., Wallach, D., & Taatgen, N. (1998). *Implicit and explicit learning in ACT-R*. 9.

Leonard, W. J., Dufresne, R. J., & Mestre, J. P. (1996). Using qualitative problem-solving strategies to highlight the role of conceptual knowledge in solving problems. *American Journal of Physics*, *64*(12), 1495–1503. https://doi.org/10.1119/1.18409

Li, N., Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2015). Integrating representation learning and skill learning in a human-like intelligent agent. *Artificial Intelligence*, *219*, 67–91. https://doi.org/10.1016/j.artint.2014.11.002

MacDougall, W. K., West, R. L., & Genovesi, C. (2014). *Modeling the Function of Narrative in Expertise* [Application/pdf]. 5 pages. https://doi.org/10.4230/OASICS.CMN.2014.116

Pirolli, P. (1986). A Cognitive Model and Computer Tutor for Programming Recursion. *Human–Computer Interaction*, *2*(4), Article 4. http://content.ebscohost.com/ContentServer.asp?T=P&P=AN&K=7309052&S=R&D=bth&EbscoContent=dGJyMMvl7ESeprA4y9fwOLCmsEiep7BSsKi4Ta%2BWxWXS&ContentCustomer=dGJyMPGutlC1qbVRuePfgeyx44Dt6fIA#page=1&zoom=auto,-128,654

Recker, M. M., & Pirolli, P. (1995). Modeling Individual Differences in Students' Learning Strategies. *Journal of the Learning Sciences*, *4*(1), 1–38. https://doi.org/10.1207/s15327809jls0401_1

Rist, R. S. (1989). Schema Creation in Programming. *Cognitive Science*, *13*(3), 389–414. https://doi.org/10.1207/s15516709cog1303_3

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

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

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*, 15.

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

Sun, R. (2008). *The Cambridge Handbook of Computational Psychology*. 768.

VanLehn, K. (1998). Analogy Events: How Examples are Used During Problem Solving. *Cognitive Science*, *22*(3), 347–388. https://doi.org/10.1207/s15516709cog2203_4

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

Weiser, M., & Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies*, *19*(4), 391–398. https://doi.org/10.1016/S0020-7373(83)80061-3

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. https://doi.org/10.1016/j.bica.2014.03.009

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., Ward, L., Dudzik, K., Nagy, N., & Karimi, F. (2018). Micro and Macro Predictions: Using SGOMS to Predict Phone App Game Playing and Emergency Operations Centre Responses. In D. Harris (Ed.), *Engineering Psychology and Cognitive Ergonomics* (Vol. 10906, pp. 501–519). Springer International Publishing. https://doi.org/10.1007/978-3-319-91122-9_41

Ye, N., & Salvendy, G. (1996). Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics*, *39*(3), 461–481. https://doi.org/10.1080/00140139608964475