Cognitive Architectures: A Way Forward for the Psychology of Programming

Michael Hansen (mihansen@indiana.edu)

School of Informatics and Computing, 2719 E. 10th Street

Bloomington, IN 47408 USA

Andrew Lumsdaine (lums@indiana.edu)

School of Informatics and Computing, 2719 E. 10th Street

Bloomington, IN 47408 USA

Robert L. Goldstone (rgoldsto@indiana.edu)

Dept. of Psychological and Brain Sciences, 1101 E. 10th Street

Bloomington, IN 47405 USA

March 22, 2015

Abstract

Programming language and library designers often debate the usability of particular design choices. These choices may impact many developers, yet scientific evidence for them is rarely provided. Cognitive models of program comprehension have existed for over thirty years, but the lack of quantitative (operational) validations for their internal components limits their utility for usability studies. To ease the burden of quantifying these existing models, we recommend using the ACT-R cognitive architecture – a simulation framework for psychological models. In this paper, we review the history of cognitive modeling in the psychology of programming. We provide an overview of the ACT-R cognitive architecture, and show how it could be used to fill in the gaps of an existing, yet incomplete, quantitative model (the Cognitive Complexity Metric). Lastly, we discuss the potential benefits and challenges associated with building a comprehensive cognitive model on top of a cognitive architecture.

1 Introduction

As programming languages and libraries grow in complexity, it becomes increasingly difficult to predict the impact of new design decisions on developers. Controlled studies of developers are necessary to ensure that proposed designs match expectations, and that usability has not been compromised [13]. Poor designs can be often be ruled out in this manner, but there are cases where we must choose between several good designs. As an example, consider the removal of Concepts (a language feature) from the upcoming version of C++. There were two proposed designs for Concepts, and experts were in disagreement over which was better for the average programmer [36]. With no objective way of choosing one design over the other, Concepts was removed from the C++ standard pending further study. Debates over usability are not unusual in computer science, but it is rare that any scientific evidence is provided for how design choices actually impact usability (e.g., Java was originally claimed to be "simple" and "familiar") [24]. New languages and programming paradigms are often advertised as more "natural" or productive for programmers, but the evidence is largely anecdotal or deduced from outdated principles.[1] [17] Despite at least four decades of research on the psychology of programming, it is not common to see computer scientists quantify design trade-offs in terms of usability.

Starting around 1980, researchers in the psychology of programming began proposing cognitive models of *program comprehension* – how developers understand programs. *Cognitive models* seek to explain basic mental processes, such as perceiving, learning, remembering, and problem solving [10]. These models (detailed in Section 2) provide explanations for the results of empirical studies on programmers.[2], and suggest ways in which languages and tools could be improved [35, 29]. Unfortunately, today's cognitive models of program comprehension are verbal-conceptual in nature, i.e., they are defined using natural language. Without quantitative or *operational* definitions of their components, we cannot hope to use these models to quantify design trade-offs and objectively inform usability debates. At best, we are forced to rely on natural language rubrics when evaluating alternative designs [8]. Given a quantitative cognitive model of program comprehension, however, it would be possible to simulate a programmer's behavior for different design conditions and make objective judgements about usability.

Providing operational definitions for the components and processes in a cognitive model is difficult, but we believe the burden can be decreased by building on top of a cognitive architecture (Section 3). *Cognitive architectures* are software simulation environments that integrate formal theories from cognitive science into a coherent whole, and they can provide precise predictions of performance metrics in human experiments [11]. These architectures can be used to simulate cognitive models, which must be formalized in a domain-specific language (different for each cognitive architecture). In Section 4, we describe the ACT-R cognitive architecture in detail, and give examples of its success in other domains. Section 5 reviews the Cognitive Complexity Metric [12], an attempt by Cant et al. to bring together and provide operational definitions for existing theories of program comprehension. Many of the definitions in Cant et al.'s model are placeholders for future empirical studies, and we believe ACT-R could help fill in the gaps. Section 6 discusses some the technical and social challenges of our approach, as well as potential benefits to computer

[1] For example, researchers of working memory in humans have long since abandoned the simple "7 ± 2 items" model of short-term memory [4].

[2] Studies of non-programmers solving programming problems have also been done, yielding interesting results [28] For example, 95% of non-programmers designing a Pac-Man game preferred set/subset specifications for operations on multiple objects (turn all the ghosts blue) to explicit iteration (for each ghost, turn the ghost blue).

science and beyond. Finally, Section 7 concludes.

2 Psychological Studies of Programming

Psychologists have studied the behavioral aspects of programming for at least forty years [14]. In her book *Software Design - Cognitive Aspects*, Françoise Détienne proposed that psychological research on programming can be broken into two distinct periods [17]. The first period, spanning the 1960's and 1970's, is characterized by the importing of basic experimental techniques and theories from psychology into computer science. Early experiments looked for correlations between task performance and language/human factors – e.g., the presence or absence of language features, years of experience, and answers to code comprehension questionnaires. While this period marked the beginning of scientific inquiry into software usability, results were limited in scope and often contradictory.

The problem is simple: if task performance depends heavily on internal cognitive processes, then it cannot be measured independent of the programmer. Early psychology of programming studies relied on statistical correlations between metrics like "presence of comments" and "number of defects detected," so researchers were unable to explain some puzzling results. For example, there were multiple studies in the 1970's to measure the effect of meaningful variable names on code understanding. Two such studies found no effect [41, 34] while a third study found positive effects as programs became more complex [33]. Almost a decade later, Soloway and Ehrlich suggested an explanation for these findings: experienced programmers are able to recognize code *schemas* or *programming plans* [35]. Programming plans are "program fragments that represent stereotypic action sequences in programming," and expert programmers can use them to infer intent in lieu of meaningful variable names. This and many other effects depend on internal cognitive processes, and therefore require a cognitive modeling approach.

Cognitive Models. Détienne characterizes the second period of psychological studies of programming (1980 to the present) by the use of *cognitive models*. A cognitive model seeks to explain basic mental processes and their interactions; processes such as perceiving, learning, remembering, problem solving, and decision making [10]. Developing cognitive models of program comprehension requires measuring (directly or indirectly) what is happening in the programmer's head. Verbal reports, code changes, response times, and eye-gaze patterns are common ways of measuring a programmer's behavior. These measurements can help fill in the cognitive gaps left by earlier task performance metrics (e.g., number of errors detected, number of correct post-test questions). Besides better measurements, empirical studies in Détienne's second period also began experimenting on experienced programmers rather than just students. This allowed for more real-world problems to be investigated, and prompted researchers to consider the educational aspects of programming from a new perspective (i.e., teaching expert programming plans and Soloway's rules of discourse explicitly [15]).

In the past thirty years, a number of interesting cognitive program comprehension models have been proposed. These models focus on specific cognitive processes, and have incorporated new discoveries in cognitive science over the years. Tracz's Human Information Processor model (1979), for example, argued that the capacity limits on short-term memory constrain the complexity of readable programs [39]. Von Mayrhauser's Integrated Metamodel (1995) combined existing top-down and bottom-up comprehension

Rules of Discourse

1. Variable names should reflect function.

2. Don't include code that won't be used.

3. If there is a test for a condition, then the condition must have the potential of being true.

4. A variable that is initialized via an assignment statement should be updated via an assignment statement.

5. Don't do double duty with code in a non-obvious way.

6. An `if` should be used when a statement body is guaranteed to be executed only once, and a `while` used when a statement body may be repeatedly executed.

Figure 1: *Rules proposed by Soloway et al. in 1984. These "unwritten" rules are internalized by experienced programmers. [35]*

models while including the programmer's knowledge base as a major component [40]. More recently, Chris Parnin (a cognitive neuroscientist) has proposed the Task Memory Model (2010) to show how knowledge of the different types of memory in the brain can be used to design better programming environments [29].

While the models above focus on different aspects of the programming process, researchers have agreed on the following key features of any comprehensive cognitive model:

- **Knowledge** - Experienced programmers represent programs at multiple levels of abstraction: syntactic, semantic, and schematic. Conventions and common programming plans allow experts to quickly infer intent and avoid unnecessary details (see Figure 1).

- **Strategies** - Experienced programmers have many different design strategies available (e.g., top-down, bottom-up, breadth-first, depth-first). Their current strategy is chosen based on factors like familiarity, problem domain, and available language features [17].

- **Task** - The current task or goal will change which kinds of program knowledge and reading strategies are advantageous. Experienced programmers read and remember code differently depending on whether they intend to edit, debug, or reuse it [16].

- **Environment** - Programmers use their tools to off-load mental work and to build up representations of the current problem state [23]. The benefits of specific tools, such as program visualization, also depend on programming expertise [6].

The cognitive models mentioned above give a high-level, qualitative explanation of how these key features are realized in the programmer's head. However, language and library designers need specific answers. User studies can help distinguish good designs from bad after the fact, but precisely predicting the impact of design decisions in advance requires more detail. What we need is a *quantitative* cognitive

model whose components and processes are formally specified. We could use such a model to simulate a programmer's behavior under different design conditions, and make usability judgements based on the results. To reasonably approximate a real programmer, the model would also have to include a variety of low-level psychological models, e.g., of memory, learning, perception, etc.

Building a quantitative cognitive model of program comprehension is a daunting task. One of the most comprehensive attempts is Cant et al.'s *Cognitive Complexity Metric* (CCM) [12]. This metric includes a dozen factors from the psychology of programming literature that are thought to influence code comprehension. Unfortunately, many of the mathematical definitions in the CCM are simply placeholders for future empirical work.

The tools available to cognitive scientists have advanced considerably since Cant et al. proposed the CCM in 1995. A new version of their model could be built on top of a cognitive architecture like ACT-R – a software simulation environment that contains special-purpose modules for low-level psychological models. This would simplify the CCM, since it includes factors for familiarity, recognizability, and other psychological phenomena presently modeled in ACT-R. The next section introduces cognitive architectures (Section 3), and is followed by an overview of ACT-R (Section 4). Section 5 reviews the CCM in detail, and considers how it could be implemented within the ACT-R framework.

3 Cognitive Architectures

In his book *How Can the Human Mind Occur in the Physical Universe?*, Anderson defines a cognitive architecture as

> . . . a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind [2].

In other words, a cognitive architecture is not so abstract as to ignore the brain completely, but it does not necessarily emulate the low-level details of the brain either. For example, consider a computer program that solves algebra problems. While it may produce the same high-level answers as a human, it is likely doing something very different under the hood. A low-level neural simulation of the brain is impractical, however, and would make it virtually impossible to separate the simulation details from the relevant aspects of the task. A cognitive architecture sits somewhere in between, allowing researchers to deal with the abstract nature of specific tasks, but also enforcing biologically plausible constraints where appropriate.

A cognitive architecture is executable. The inputs to this software are a description of the model (written in a domain-specific and architecture-specific language) and the relevant stimuli for the task (e.g., text, images, sound). The output is a timestamped trace of the model's behavior, which can be used to get performance metrics like response times, learning rates, and even virtual fMRI data. In some cognitive architectures, models are able to interact with the same user interfaces as human subjects, reducing the number of assumptions the modeler must make.

Three prominent cognitive architectures today are Soar [27], EPIC [22], and ACT-R [2]. All three architectures are based on *productions*, which are sets of `if-then` rules. A *production system* checks each production's pre-conditions against the system's current state, and fires the appropriate actions if they match. The constraints imposed on productions, and the system as a whole, differ between individual architectures.

ACT-R, for example, only allows a single production to fired at a time while EPIC allows any number of productions to simultaneously fire.

A more detailed review of Soar and EPIC can be found in [11]. For the remainder of the paper, we concentrate on ACT-R. While we believe a successful model of program comprehension could be implemented in any of these architectures, we have chosen to focus on ACT-R for three reasons. First, it is widely-used, and is actively being developed by cognitive scientists [1]. Second, a number of fMRI studies have been done to associate ACT-R modules with particular brain regions [3]. While this is not crucial for computer scientists, it provides an additional means of model verification. Lastly, ACT-R contains perception and motor modules, allowing models to combine cognition and interaction with the environment (called *end-to-end modeling*).

4 ACT-R: A Cognitive Architecture

ACT-R is "*a cognitive architecture: a theory about how human cognition works.*" [1] It is a domain-specific programming language built on top of LISP, and a simulation framework for cognitive models. There are eight *modules* in ACT-R, which represent major components of human cognition (Figure 2). Modules exist at two layers: (1) the *symbolic* layer, which provides a simple programmatic interface for models, and (2) the *subsymbolic* layer, which hides real-world details like how long it takes to retrieve an item from declarative memory. ACT-R models formalize humans performance on tasks using production rules (see Section 3) that send and receive messages between modules. Models are simulated and observed along psychologically relevant dimensions like task accuracy, response times, and simulated BOLD.[3] measures (i.e., fMRI brain activations) These simulations are reproducible, and can provide precise predictions about human behavior. We discuss the pieces of ACT-R in detail below, and provide examples of its success in other domains.

Buffers, Chunks, and Productions. The ACT-R architecture is divided into eight *modules*, each of which has been associated with a particular brain region (see [2] for more details). Every module has its own *buffer*, which may contain a single *chunk*. Buffers also serve as the interface to a module, and can be queried for the module's state. Chunks are the means by which ACT-R modules encode messages and store information internally. They are essentially collections of name/value pairs (called slots), and may inherit their structure from a parent chunk type. Individual chunk instances can be extended in advanced models, but simple modules tend to have chunks with a fixed set of slots (e.g., two addends and a result for a simple model of addition). Modules compute and communicate via *productions*, rules that pattern-match on the slot values of a chunk or the state of a buffer. When a production matches the current system state (i.e., all chunks in all module buffers), it "fires" a response. Responses include actions like inserting a newly constructed chunk into a buffer and modifying/removing a buffer's existing chunk.[4]

When it is possible, computations **within** a module are done in parallel. Exceptions include the serial fetching of a single memory from the *declarative* module, and the *visual* module's restriction to only focus on one item at a time. Communication **between** modules is done serially via the *procedural* module (see Figure 2). Only one production may fire at any given time[5], making the procedural model the central

[3]Blood-oxygen-level-dependent contrast. This is the change in blood-flow for a given brain region over time.

[4]Chunks that are removed from a buffer are automatically stored in the declarative memory module.

[5]This is a point of contention in the literature. Other cognitive architectures, such as EPIC [22] allow more than one production to fire at a time.

bottleneck of the system.

The *visual*, *aural*, *vocal*, and *manual* modules are capable of communicating with the environment. In ACT-R, this environment is often simulated for performance and consistency. Experiments in ACT-R can be written to support both human and model input, allowing for a tight feedback loop between adjustments to the experiment and adjustments to the model. The *goal* and *imaginal* modules are used to maintain the model's current goal and problem state, respectively.

The Subsymbolic Layer. Productions, chunks, and buffers exist at the *symbolic* layer in ACT-R. The ability for ACT-R to simulate human performance on cognitive tasks, however, comes largely from the *subsymbolic* layer. A symbolic action, such as retrieving a chunk from declarative memory, does not return immediately. Instead, the declarative module performs calculations to determine how long the retrieval will take (in simulated time), and the probability of an error occurring. The equations for subsymbolic calculations in ACT-R's modules come from existing psychological models of learning, memory, problem solving, perception, and attention [1]. Thus, there are many constraints on models when fitting parameters to human data.

In addition to calculating timing delays and error probabilities, the subsymbolic layer of ACT-R contains mechanisms for learning. Productions can be given *utility* values, which are used to break ties during the matching process.[6] Utility values can be learned by having ACT-R propagate rewards backwards in time to previously fired productions. Lastly, new productions can be automatically *compiled* from existing productions (representing the learning of new rules). Production compilation could occur, for example, if production P_1 retrieves stimulus-response pairs from declarative memory and production P_2 presses a key based on the response. If P_1 and P_2 co-occur often, the compiled $P_{1,2}$ production would go directly from stimulus to key press, saving a trip to memory. If $P_{1,2}$ is used enough, it will eventually replace the original productions and decrease the model's average response time.

Successful ACT-R Models. Over the course of its multi-decade lifespan, there have been many successful ACT-R models in a variety of domains (success here means that the models fit experimental data well, and were also considered plausible explanations). We provide a handful of examples below (more are available on the ACT-R website [1]).

David Salvucci (2001) used a multi-tasking ACT-R model to predict how different in-car cellphone dialing interfaces would affect drivers [31]. This integrated model was a combination of two more specific models: one of a human driver and another of the dialing task. By interleaving the production rules of the two specific models[7], the integrated model was able to switch between tasks. Salvucci's model successfully predicted drivers' dialing times and lateral deviation from their intended lane.

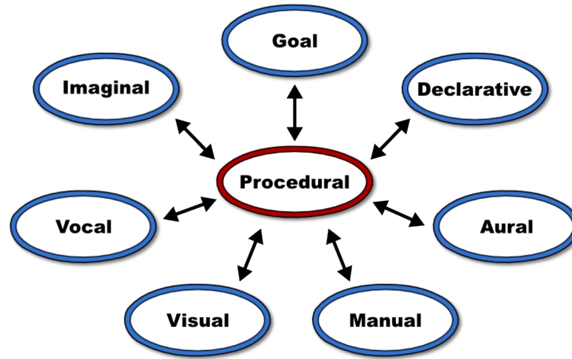
Brian Ehret (2002) developed an ACT-R model of location learning in a graphical user interface [19]. This model gives an account of the underlying mechanisms of location learning theory, and accurately captures trends in human eye-gaze and task performance data. Thanks to ACT-R's perception/motor modules, Ehret's model was able to interact with the same software as the users in his experiments.

[6]This is especially useful when productions match chunks based on *similarity* instead of equality, since there are likely to be many different matches.

[7]Salvucci notes that this required hand-editing the models' production rules since ACT-R does not provide a general mechanism for combining models. See [32] for a more advanced model of multi-tasking in ACT-R.

Lastly, an ACT-R model created by Taatgen and Anderson (2004) provided an explanation for why children produce a U-shaped learning curve for irregular verbs [38]. Over the course of early language learning, children often start with the correct usage (I went), over-generalize the past-tense rule (I goed), and then return to the correct usage (I went). Taatgen and Anderson's model proposed that this U-shaped curve results from cognitive trade-offs between irregular and regular (rule-based) word forms. Retrieval of an irregular form is more efficient if its use frequency is high enough to make it available in memory. Phonetically post-processing a regular-form word according to a rule is much slower, but always produces something. Model simulations quantified how these trade-offs favor regular over irregular forms for a brief time during the learning process.

The success of these non-trivial models in their respective domains gives us confidence that ACT-R is mature enough for modeling program comprehension. In the next section, we discuss how ACT-R might serve as a base for an existing quantitative cognitive model of code complexity.



Module	Purpose
Procedural	Stores and matches production rules, facilitates inter-module communication
Goal	Holds a chunk representing the model's current goal
Declarative	Stores and retrieves declarative memory chunks
Imaginal	Holds a chunk representing the current problem state
Visual	Observes and encodes visual stimuli (color, position, etc.)
Manual	Outputs manual actions like key-presses and mouse movements
Vocal	Converts text strings to speech and performs subvocalization
Aural	Observes and encodes aural stimuli (pitch, location, etc.)

Figure 2: *ACT-R 6.0 modules. Communication between modules is done via the procedural module.*

Term	Definition
chunk	representation for declarative knowledge
production	representation for procedural knowledge
module	major components of the ACT-R system
buffer	interface between modules and procedural memory system
symbolic layer	high-level production system, pattern-matcher
subsymbolic layer	underlying equations governing symbolic processes
utility	relative cost/benefit of productions
production compilation	procedural learning, combine existing productions
similarity	relatedness of chunks

Figure 3: *ACT-R terminology*

5 The Cognitive Complexity Metric

Developed in the mid-nineties, Cant et al.'s cognitive complexity metric (CCM) attempts to quantify the cognitive processes involved in program development, modification, and debugging [12]. The CCM focuses on the processes of *chunking* (understanding a block of code) and *tracing* (locating dependencies). Cant et al. provide mathematical definitions for factors that are believed to influence each process. Some of the factors in the CCM are quantified by drawing upon the existing literature, but many definitions are simply placeholders for future empirical studies.

We believe that a working implementation of the CCM could be developed on top of ACT-R. Many of the underlying terms in the CCM equations could be simplified or eliminated within the ACT-R framework. This is because the subsymbolic layer in ACT-R already contains equations for low-level cognitive processes like rule learning, memory retrieval, and visual attention (see Section 4 for details). Below, we describe the CCM in more detail. In addition to the high-level equations, we briefly discuss each factor in the metric, and consider how it could be subsumed within an ACT-R model.[8]

Chunking and Tracing. The cognitive processes of chunking and tracing play key roles in the cognitive complexity metric (CCM). *Chunking* is defined as the process of recognizing groups of code statements (not necessarily sequential), and recording the information extracted from them as a single mental symbol or abstraction. In practice, programmers rarely read through and chunk every statement in a program. Instead, they *trace* forwards or backwards in order to find relevant chunks for the task at hand [12]. Cant et al. define a chunk as a block of statements that must occur together (e.g., loop + conditional).[9] This definition, however, is intended only for when the programmer is reading code in a normal forward manner. When tracing backwards or forwards, a chunk is defined as the single statement involving a procedure or variable's definition.

In the remainder of this section, care must be taken to disambiguate Cant et al.'s use of the word "chunk" and the ACT-R term (Figure 3). We will clarify instances where there may be ambiguity between these two distinct terms.

Chunk Complexity (C) To compute the complexity C_i of chunk i, Cant et al. define the following equation:

$$C_i = R_i + \sum_{j \in N} C_j + \sum_{j \in N} T_j$$

where R_i is the complexity of the immediate chunk i, C_j is the complexity of sub-chunk j, and T_j is the difficulty in tracing dependency j of chunk i. The definitions of R and T are given as follows:

$$R = R_F(R_S + R_C + R_E + R_R + R_V + R_D)$$

$$T = T_F(T_L + T_A + T_S + T_C)$$

Each right-hand side term stands for a particular factor that is thought to influence the chunking or

[8]We have not implemented an ACT-R model of the CCM yet; this is planned for future work (Section 6).

[9]When operationalizing the definition of a chunk, Cant et al. admit that "*it is difficult to determine exactly what constitutes a chunk since it is a product of the programmer's semantic knowledge, as developed through experience.*"

tracing processes (Figure 4). The underlying equations for each factor can be found in [12], but are not needed for the discussion below.

5.1 Immediate Chunk Complexity (R)

The chunk complexity R is made up of six additive terms (R_S, R_C, R_E, R_R, R_V, R_D) and one multiplicative term R_F. These represent factors that influence how hard it is to understand a given chunk.[10]

$\mathbf{R_F}$ (chunk familiarity) This term captures the increased speed with which a programmer is able to understand a given chunk after repeated readings. In ACT-R, the subsymbolic layer of the declarative memory module would handle this, as repeated retrievals of the same ACT-R chunk will increase its retrieval speed. Declarative chunks are not stored independently, however, so the activation of similar chunks will potentially cause interference. This means that increased familiarity with one chunk will come at the cost of slower retrieval times for similar chunks. Production compilation could also underly familiarity. Once a compiled production's utility exceeds that of its parents, it will be fired instead and result in speed gains.

$\mathbf{R_S}$ (size of a chunk) This term captures two notions of a chunk's "size": (1) its structural size (e.g., lines of code) and (2) the "*psychological complexity of identifying a chunk where a long contiguous section of non-branching code must be divided up in order to be understood.*" In other words, R_S should be effected by some notion of short-term memory constraints. An ACT-R model would be influenced by a chunk's structural size simply because there would be more code for the visual module to attend to and encode (i.e., more sequential productions fired). The additional "psychological complexity" could be modeled in several ways. ACT-R does not contain a distinct short-term memory component, relying on long-term memory to serve as a short-term and working memory. According to Niels Taatgen, however, the decay and interference mechanisms present in ACT-R's subsymbolic layer can produce the appearance of short-term memory constraints [37].

$\mathbf{R_C}$ (control structures) The type of control structure in which a chunk is embedded influences R because conditional control structures like `if` statements and loops require the programmer to comprehend additional boolean expressions. In some cases, this might involve mentally iterating through a loop. We expect that boolean expressions would be comprehended in much the same way as for the R_E factor (see below). Modeling the programmer's mental iteration through a loop could draw on existing ACT-R models for inspiration. For example, a model of children learning addition facts (e.g., $1 + 5 = 6$) might "calculate" the answer to $5 + 3$ by mentally counting up from 5.[11] After many repetitions, the pattern $5 + 3 = 8$ is retrieved directly from memory, avoiding this slow counting process. Likewise, an ACT-R model of program comprehension could start out by mentally iterating over loops, and eventually gain the ability to recognize common patterns in fewer steps.

$\mathbf{R_E}$ (boolean expressions) Boolean expressions are fundamental to the understanding of most programs, since they are used in conditional statements and loops. According to Cant et al., the complexity of boolean expressions depends heavily on their form and the degree to which they are nested. To incorporate boolean

[10]More general forms for R and T are discussed in [12], but Cant et al. suggest starting with simple additive representations.
[11]The model would likely use the subvocalize feature of ACT-R's vocal module to simulate the child's inner voice.

expressions into an ACT-R model, it would be helpful to record eye-gaze patterns from programmers answering questions based on boolean expressions. A data set with these patterns, response times, and answers to the questions could provide valuable insight into how programmers of different experience levels evaluate boolean expressions. For example, it may be the case that experienced programmers use visual cues to perform pre-processing at the perceptual level (i.e., they saccade over irrelevant parts of the expression). The data may also reveal that experienced programmers read the expression, but make efficient use of conceptual-level shortcuts (e.g., `FALSE AND ... = FALSE`). These two possibilities would result in very different ACT-R models, the former making heavy use of the visual module, and the latter depending more on declarative memory.

$\mathbf{R_R}$ **(recognizability)** Empirical studies have shown that the syntactic form of a program can have a strong effect on how a programmer mentally abstracts during comprehension [20]. An ACT-R model would show such an effect if its representation of the program was built-up over time via perceptual processes. In other words, the model would need to observe actual code rather than receiving a pre-processed version of the program as input (e.g., an abstract syntax tree). Ideally, the ACT-R model would make errors like real programmers do when the code violates Soloway's unwritten rules of discourse (e.g., the same variable is used for multiple purposes) [35]. ACT-R has the ability to partially match chunks in memory by using a model-specific similarity metric. This ability would be useful for modeling recognizability, since slight changes in code indentation and layout should not confuse the model entirely.

$\mathbf{R_V}$ **(visual structure)** This term represents the effects of visual structure on a chunk's complexity, and essentially captures how visual boundaries influence chunk identification. While Cant et al. only describe three kinds of chunk delineations (function, control structure, and no boundary), more general notions of textual beacons and boundaries have been shown to be important in code [42]. For example, Détienne found that advance organizers (e.g., a function's name and leading comments) had a measurable effect on programmers' expectations of the code that followed [18]. Biggerstaff et al. have also designed a system for automatic domain concept recognition in code [7]. This system considers whitespace to be meaningful, and uses it to bracket groups of related statements. As with the recognizability term (R_R), it would be crucial for an ACT-R model to observe real code instead of an abstract syntax tree. ACT-R's visual module is able to report the xy coordinates of text on the screen, so it would be simple to define a notion of whitespace in the model.

$\mathbf{R_D}$ **(dependency disruptions)** There are many disruptions in chunking caused by the need to resolve dependencies. This applies both to remote dependencies (e.g., variable definitions), and to local dependencies (e.g., nested loops and decision structures). Cant et al. cite the small capacity of short-term memory as the main reason for these disruptions. As mentioned earlier, ACT-R does not have a distinct "short-term memory" module with a fixed capacity. Instead, short-term capacity limits are an emergent property of memory decay and interference. Given the biological plausibility of ACT-R's architecture, we should expect to find empirically that R_D effects in humans are actually context-dependent. In other words, the number of disruptions that a programmer can handle without issue should depend on the situation. This is a case where the psychological theory underlying the cognitive architecture can help to suggest new experiments

on humans.

5.2 Tracing Difficulty (T)

Most code does not stand alone. There are often dependencies that the programmer must resolve before chunking the code in memory and ultimately understanding what it does. The Cognitive Complexity Metric (CCM) operationalizes the difficulty in tracing a dependency as $T = T_F(T_L + T_A + T_S + T_C)$. For these six terms, the definition of a chunk is slightly different. Rather than being a block of statements that must co-occur, a chunk during tracing is defined as a single statement involving a procedure's name or a variable definition.

$\mathbf{T_F}$ (familiarity) The dependency familiarity has a similar purpose to the chunk familiarity (R_F). As with R_F, ACT-R's subsymbolic layer will facilitate a familiarization effect where repeated requests for the same dependency information from declarative memory will take less time. The effect of the available tools in the environment, however, will also be important. An often-used dependency (e.g., function definition) may be opened up in a new tab or bookmarked within the programmer's development environment. A comprehensive ACT-R model will need to interact with the same tools as a human programmer to produce "real world" familiarization effects.

$\mathbf{T_L}$ (localization) This term represents the degree to which a dependency may be resolved locally. Cant et al. proposed three levels of localization: embedded, local, and remote. An embedded dependency is resolvable within the same chunk. A local dependency is within modular boundaries (e.g., within the same function), while a remote dependency is outside modular boundaries. This classification would fit an ACT-R model that tries to resolve dependencies by first shifting visual attention to statements within the current chunk (embedded), switching then to a within-module search (local), and finally resorting to a extra-modular search (remote) if the dependency cannot be resolved. It is not clear, however, what the model should consider a "module," especially when using a modern object-oriented language and development environment. It might be more useful to derive a definition of "module" empirically instead. An ACT-R model in which the effects of dependency localization were emergent from visual/tool search strategies could be used to define the term (i.e., how the language and tools make some chunks feel "closer" than others).

$\mathbf{T_A}$ (ambiguity) Dependency ambiguity occurs when there are multiple chunks that depend upon, or are effected by, the current chunk. The CCM considers ambiguity to be binary, so a dependency is either ambiguous or not. Whether ambiguity increases the complexity of a chunk is also dependent on the current goal, since some dependencies do not always need to be resolved (e.g., unused parameters can be ignored). We expect an ambiguity effect to emerge naturally from an ACT-R model because of partial matching and memory chunk similarity. If the model has previously chunked two definitions of the variable x, for example, then a future query (by variable name) for information about this dependency may result in a longer retrieval time or the wrong chunk entirely.

T_S (spatial distance) The distance between the current chunk and its dependent chunk will affect the difficulty in tracing. Lines of code are used in the CCM as a way of measuring distance, though this seems less relevant with modern development environments. Developers today may have multiple files open at once, and can jump to variable/function definitions with a single keystroke. The "distance" between two chunks in an ACT-R model may be appropriately modeled as how much time is spent deciding how to locate the desired chunk (e.g., keyboard shortcut, mouse commands, keyword search), making the appropriate motor movements to interact with the development environment, and visually searching until the relevant code has been identified. This more complex version of T_S would depend on many things, including the state of the development environment (e.g., which files are already open), and the programmer's familiarization with the codebase.

T_C (level of cueing) This binary term represents whether or not a reference is considered "obscure." References that are embedded within large blocks of text are considered obscure, since the surrounding text may need to be inspected and mentally broken apart. This term appears to be related to the effect of visual structure on a chunk's complexity (R_V). Clear boundaries between chunks (e.g., whitespace, headers) play a large role in R_V, and we expect them to play a similar role in T_C. Tracing is presumed to involve a more cursory scan of the code than chunking, however. An ACT-R model may need to be less sensitive to whitespace differences during tracing than during chunking.

Term	Description
R_F	Speed of recall or review (familiarity)
R_S	Chunk size
R_C	Type of control structure in which chunk is embedded
R_E	Difficulty of understanding complex Boolean or other expressions
R_R	Recognizability of chunk
R_V	Effects of visual structure
R_D	Disruptions caused by dependencies
T_F	Dependency familiarity
T_L	Localization
T_A	Ambiguity
T_S	Spatial distance
T_C	Level of cueing

Figure 4: *Important factors in the Cognitive Complexity Metric. R_x terms affect chunk complexity. T_x terms affect tracing difficulty.*

6 Discussion

6.1 Potential Benefits

There are many potential benefits to having an ACT-R model of program comprehension. We briefly discuss the potential gains for computer science, including optimizing designs for usability, lowering barriers to user evaluation, and providing a common vocabulary for experiments.

Quantifying and optimizing. A major goal of this work is to quantify design trade-offs for usability in an objective manner, which could help resolve disagreements between experts speaking on behalf of "Joe Coder" [36]. There is also the potential for semi-automated design optimization. Given a cognitive model and optimization criteria (i.e., higher accuracy on X, fewer keystrokes to do Y), a design could be automatically optimized by trying many different syntaxes, tool layouts, etc. This kind of approach has been considered by Air Force researchers for the placement of instruments on the heads-up display (HUD) provided to expert pilots [25]. If anything, these "optimal" designs could suggest new and interesting directions for future research.

Lower barriers to user evaluation. There have recent been calls for more rigorous empirical methods in computer science, particularly when evaluating the benefits of new languages and programming paradigms [21]. User evaluation is rarely done to back up claims of increased productivity or a more "intuitive" design. Computer scientists see several barriers to user evaluation, including recruiting subjects and getting approval from an institution review board (IRB) [9]. An ACT-R model would not replace real user evaluation, but it could serve as a preliminary step. Early experiments could be tried and refined with the model as a subject, saving effort otherwise spent on recruiting and running multiple rounds of human subjects. This will not eliminate the barriers to user evaluation, but it may help lower them.

A common experimental vocabulary. Another barrier to user evaluation is the time and effort taken to design experiments [9]. For *qualitative* usability studies, a common vocabulary has existed for more than a decade: the Cognitive Dimensions of Notation [8]. These dimensions (e.g., consistency, hidden dependencies) and their trade-offs allow usability researchers to design and interpret experiments within a common framework. An ACT-R model might serve a similar purpose for quantitative usability studies. For a given experimental task, the ACT-R model will explicitly define the expected inputs and outputs (e.g., syntax-highlighted code in, eye-gaze patterns and question/answer response times out). Using the model as a guide, researchers will be able to build up a common set of tasks, and know which experimental variables have previously been observed.

6.2 Challenges

There are many challenges to overcome with our approach, both technical and social. We focus here on the difficulties of model implementation, output interpretation, and designer adoption.

Implementation difficulties. Our review of Cant et al.'s Cognitive Complexity Metric hinted at how ACT-R would facilitate a program comprehension model, but many details were left out. Programming is a complex activity, and quantifying a cognitive model of it is no easy task. Because of this difficulty, there will be the tendency for modelers to pursue a "divide-and-conquer" approach. Like traditional psychology, it is tempting to model narrow phenomena with the intention of putting the pieces back together again someday. As Alan Newell said in 1973 regarding the state of cognitive psychology, however, "*you can't play twenty questions with nature and win.*" [26] We believe the same applies to the psychology of programming. Using a cognitive architecture would encourage modelers to build on top of each other's work instead of splintering efforts and studying isolated phenomena.

Interpretation issues. Even if an ACT-R model of program comprehension can be built, interpreting the output of simulations may not be straightforward. This output would likely consist of inter-mixed response times, eye-gaze patterns, button presses, and answers to test questions. Modelers or laypersons who must make informed judgements based on simulation output (directly or indirectly) will require some knowledge of ACT-R and traditional statistical analysis tools. Specialized packages for analyzing particular kinds of output, such as eye-gaze patterns, may also be required to pre-process data into a more useful form (which may include additional assumptions). The psychology of programming community should also focus on design *trade-offs* rather than how the model performs along a single dimension (e.g., time to complete the task).

Social rejection. It is possible that designers would not want to cede judgement to a cognitive model, even if there were substantial benefits (e.g., quantifiable trade-offs). Computer scientists in general have a reputation for making usability claims (regarding the average user) with little evidence to back them up [24]. In most cases, this is likely due to a lack of education in psychological methods and limited funding for experiments. There are surely some language/library designers, however, who simply believe they know what is "best" for the average user. We agree that expert opinion on usability is a valuable resource, but it is not sufficient on its own to make objective, informed design decisions. Should a functioning ACT-R model be built, it may be a major challenge to persuade designers to use it.

6.3 Future Work

In future work, we plan to use Cant et al.'s Cognitive Complexity Metric (CCM) as a starting point for building an ACT-R model of program comprehension. The components of the CCM are based on decades of empirical studies prior to 1995, and there has been significant work since. The use of eye-tracking in coding experiments, for example, has increased in recent years [5] making it possible to compare the visual search strategies of different programmers. There have also been extensions to Soloway and Ehrlich's work on programming plans, such as classifying the different roles variables play in a program [30]. We are also encouraged by the success of ACT-R models in other complex domains (e.g., students learning to solve simple linear equations [3]).

To begin with, we will study particular cases. A model of how novices and experts evaluate boolean expressions (R_E in the CCM), for instance, would incorporate eye-gaze data, verbal reports, and answers to test questions. As more cases are added (e.g., understanding loops), the model will grow in complexity. It

will be important to combine cases at each stage into a single task (e.g., a loop inside of nested `IF` statements) and verify the model against human data.

7 Conclusion

Predicting how design changes to a language or library will affect developers is a challenge for the psychology of programming field. Over the past few decades, cognitive models have helped explain data from empirical studies of programmers. These models are almost all verbal-conceptual in nature, however, making them difficult to combine and reason about precisely. In order to accurately quantify usability trade-offs between language/library designs, we believe modelers should build on top of a cognitive architecture. Specifically, we recommend the ACT-R cognitive architecture because of its large user base, active development, associated fMRI studies, and perceptual/motor modules.

There are many challenges to our approach. Operationalizing the components of existing program comprehension models is a massive undertaking. Modelers must also become fluent in a new language, and learn how to properly interpret the output of their model simulations. Despite these challenges, we believe the potential benefits are worth pursuing. A quantitative model could serve as an objective ground truth in usability debates. Additionally, common barriers to user evaluation may be lowered as researchers build a common experimental vocabulary.

The psychology of programming brings together computer science, human-computer interaction, cognitive science, and education. Computer programming is one of the most intellectually demanding tasks undertaken by so many people in industry and academia. As languages and libraries grow in complexity, human factors will become critical to maintaining program correctness, readability, and reusability. A scientific understanding of the cognitive processes behind programming is necessary to ensure that the languages and tools of tomorrow are usable by a wide audience.

References

[1] ACT-R Research Group. About ACT-R. `http://act-r.psy.cmu.edu/about/`, mar 2012.

[2] J.R. Anderson. *How can the human mind occur in the physical universe?*, volume 3. Oxford University Press, USA, 2007.

[3] J.R. Anderson. The algebraic brain. *Memory and mind: a festschrift for Gordon H. Bower*, page 75, 2008.

[4] A. Baddeley. Working memory. *Current Biology*, 20(4):R136–R140, 2010.

[5] R. Bednarik. *Methods to analyze visual attention strategies: Applications in the studies of programming*. University of Joensuu, 2007.

[6] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 125–132. ACM, 2006.

[7] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[8] A. Blackwell, C. Britton, A. Cox, T. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. Nehaniv, M. Petre, et al. Cognitive dimensions of notations: Design tools for cognitive technology. *Cognitive Technology: Instruments of Mind*, pages 325–341, 2001.

[9] R.P.L. Buse, C. Sadowski, and W. Weimer. Benefits and barriers of user evaluation in software engineering research. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 643–656. ACM, 2011.

[10] J.R. Busemeyer and A. Diederich. *Cognitive modeling*. Sage Publications, Inc, 2010.

[11] M.D. Byrne. Cognitive architecture. *The human-computer interaction handbook: Fundamentals, evolving technologies and emerging applications*, pages 97–117, 2003.

[12] SN Cant, D.R. Jeffery, and B. Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, 1995.

[13] S. Clarke. Measuring API usability. *Doctor Dobbs Journal*, 29(5):1–5, 2004.

[14] B. Curtis. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th international conference on Software engineering*, pages 97–106. IEEE Press, 1984.

[15] Simon P. Davies. Plans, goals and selection rules in the comprehension of computer programs. *Behaviour & Information Technology*, 9(3):201–214, 1990.

[16] F. Détienne. What model (s) for program understanding? In *Proceedings of the Conference on Using Complex Information Systems*, Poitiers, France, 1996.

[17] F. Détienne and F. Bott. *Software design–cognitive aspects*. Springer Verlag, 2002.

[18] Françoise Détienne. *La compréhension de programmes informatiques par l'expert: un modéle en termes de schémas*. PhD thesis, Université Paris V. Sciences humaines, 1986.

[19] B.D. Ehret. Learning where to look: Location learning in graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, pages 211–218. ACM, 2002.

[20] DJ Gilmore and TRG Green. The comprehensibility of programming notations. In *Human-Computer Interaction-Interact*, volume 84, pages 461–464, 1985.

[21] S. Hanenberg. Faith, hope, and love: an essay on software science's neglect of human factors. In *ACM Sigplan Notices*, volume 45, pages 933–946. ACM, 2010.

[22] David E. Kieras and David E. Meyer. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Hum.-Comput. Interact.*, 12(4):391–438, December 1997.

[23] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

[24] S. Markstrum. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, page 7. ACM, 2010.

[25] C.W. Myers, K.A. Gluck, G. Gunzelmann, and M. Krusmark. Validating computational cognitive process models across multiple timescales. *Journal of Artificial General Intelligence*, 2(2):108–127, 2010.

[26] A. Newell. *You can't play 20 questions with nature and win: Projective comments on the papers of this symposium*. Carnegie Mellon University, Department of Computer Science Pittsburgh, PA, 1973.

[27] A. Newell. Unified theories of cognition. *Cambridge. MA: Har*, 1990.

[28] J.F. Pane and C. Ratanamahatana. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2):237–264, 2001.

[29] C. Parnin. A cognitive neuroscience perspective on memory for programming tasks. In *In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG)*. Citeseer, 2010.

[30] J. Sajaniemi and R. Navarro Prieto. Roles of variables in experts programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, pages 145–159. Citeseer, 2005.

[31] D.D. Salvucci. Predicting the effects of in-car interface use on driver performance: An integrated model approach. *International Journal of Human-Computer Studies*, 55(1):85–107, 2001.

[32] D.D. Salvucci and N.A. Taatgen. Threaded cognition: An integrated theory of concurrent multitasking. *Psychological Review*, 115(1):101, 2008.

[33] B. Schneiderman. Interactive interface issues. *Software Psychology: Human Factors in Computer and Information Systems*, pages 216–251, 1980.

[34] S.B. Sheppard, B. Curtis, P. Milliman, MA Borst, and T. Love. First-year results from a research program on human factors in software engineering. In *Proceedings of the National Computer Conference*, page 1021. IEEE Computer Society, 1979.

[35] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, (5):595–609, 1984.

[36] B. Stroustrup. Simplifying the use of concepts. Technical report, 2009.

[37] N.A. Taatgen. Dispelling the magic: Towards memory without capacity. *Behavioral and Brain Sciences*, 24(01):147–148, 2001.

[38] N.A. Taatgen, J.R. Anderson, et al. Why do children learn to say "broke"? A model of learning the past tense without feedback. *Cognition*, 86(2):123–155, 2004.

[39] W.J. Tracz. Computer programming and the human thought process. *Software: Practice and Experience*, 9(2):127–137, 1979.

[40] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[41] L. Weissman. Psychological complexity of computer programs: an experimental methodology. *ACM Sigplan Notices*, 9(6):25–36, 1974.

[42] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697 – 709, 1986.