Cognitive Architectures: A Way Forward for the Psychology of Programming

Michael E. Hansen
Indiana University
mihansen@indiana.edu

Andrew Lumsdaine
Indiana University
lums@indiana.edu

Robert L. Goldstone
Indiana University
rgoldsto@indiana.edu

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 lack quantitative definitions of their internal components and processes. To ease the burden of quantifying existing models, we recommend using the ACT-R cognitive architecture: a simulation framework for psychological models. In this paper, we provide a high-level overview of modern cognitive architectures while concentrating on the details of ACT-R. We review an existing quantitative program comprehension model, and consider how it could be simplified and implemented within the ACT-R framework. Lastly, we discuss the challenges and potential benefits associated with building a comprehensive cognitive model on top of a cognitive architecture.

Categories and Subject Descriptors H.1.2 [*Information Systems*]: Software Psychology

General Terms Human Factors, Experimentation

Keywords cognitive architectures, psychology of programming, cognitive complexity

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 therefore necessary to ensure that the proposed design matches expectations [14]. Poor designs can be often be ruled out in manner, but what if we must decide between two or more good designs? As an example, consider the removal of Concepts (a language feature) from the upcoming version of C++. In addition to implementation concerns, leading experts in the field disagreed over two proposed designs for Concepts [39]. With no objective way of deciding which design would be better for the average C++ programmer, the feature was removed pending further study. While debates over usability are not unusual in computer science, it is rare that any scientific evidence is provided for design choices [26]. Despite at least four decades of research on the psychology of programming, quantifying design trade-offs in terms of usability is still out of reach.

Over the past thirty years, researchers in the psychology of programming have proposed many different 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 [11]. These models provide explanations for the results of empirical studies on programmers[1], and suggest ways in which languages and tools could be improved [32, 38]. 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 its components, we cannot hope to use a model 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 [9]. If we could operationalize its components, however, it would be possible to simulate a programmer's behavior under given model for different design conditions.

Providing operational definitions for the components and processes in a cognitive model is difficult, but we believe the burden can be lessened significantly 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 [12]. In Section 4, we describe a particular cognitive architecture (ACT-R) in detail, and cite examples of its success in other domains. In order to argue for the plausibility of modeling program comprehension in

[1] Studies of non-programmers solving programming problems have also been done, yielding interesting results [31].

ACT-R, we review Cant et al.'s Cognitive Complexity Metric [13]. Section 5 describes the components of this complexity metric in detail, and considers how the underlying cognitive model might be simplified within the ACT-R framework. Section 6 discusses some the technical and social challenges of our approach, as well as potential benefits to computer science and beyond. Finally, Section 7 concludes.

2. Psychological Studies of Programming

Psychologists have been studying the behavioral aspects of programming for at least forty years [15]. In her book *Software Design - Cognitive Aspects*, Françoise Détienne proposed that psychological research on programming can be broken into two distinct periods [19]. The first period, spanning the 1960's and 1970's, can be characterized by the importing of basic experimental techniques and theories from psychology into computer science. With a heavy reliance on overall task performance metrics (e.g., number of defects detected), results from this period are limited in scope and often contradictory. 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.

Without a cognitive model, however, experiments in Détienne's first period were unable to explain some puzzling results. Early studies whose goal was to measure the effect of meaningful variable names on code understanding, for example, were inconclusive. Two such studies found no effect [37, 45] while a third study found positive effects as programs became more complex [36]. It was not until years later that a reasonable explanation was put forth: experienced programmers are able to recognize code *schemas* or *programming plans* [38]. 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 cannot be measured independent of the programmer.

Cognitive Models. Détienne characterizes the second period of psychological studies of programming 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 [11]. Developing cognitive models of program comprehension requires an analysis of the entire activity of programming, not just the end result. A programmer's verbal reports, code changes, response times, and eye-gaze patterns throughout an experiment are examples of metrics that can capture the entire activity. These metrics fill in the blanks left by overall task performance metrics, such as number of errors detected or the number of correctly answered questions on a post-test. In addition to finer-grained metrics, empirical studies during Détienne's second period began using experienced programmers as experimental subjects instead of relying solely on 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 [16].

In the past thirty years, a number of interesting cognitive program comprehension models have been proposed. These models tend to 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 [43]. Von Mayrhauser's Integrated Metamodel (1995) combined existing top-down and bottom-up comprehension models while including the programmer's knowledge base as a major component [44]. Drawing on in-house empirical studies, Douce's Stores Model of Code Cognitive (2008) emphasized the visual, spatial, and linguistic abilities of programmers [20]. 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 [32].

While the models above tend to focus on different aspects of the programming process, an overall picture has emerged in the field. In general, 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 language features [19].
- **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 [18].
- **Environment** - programmers use their tools to off-load mental work and to build up representations of the current problem state [25]. The benefits of specific tools, such as program visualization, also depend on programming expertise [7].

A comprehensive cognitive model of program comprehension must explain the features above in terms of the programmer's underlying cognitive processes. This is a non-trivial task, requiring the combination of many different

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. [38]

psychological theories. In addition, the interaction between these theories must be specified and given some kind of biological plausibility. Such a task is a massive undertaking, but a great deal of the groundwork has already been done. In the next section, we define and describe *cognitive architectures*. These software simulation environments provide a base upon which we believe a future cognitive model of program comprehension could be built.

3. Cognitive Architectures

What is a cognitive architecture? We use the following definition given by Anderson in his book *How Can the Human Mind Occur in the Physical Universe?* [2]:

> A *cognitive architecture* is a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind.

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 answers as a human, it is likely doing something very different than the brain under the hood. A complete neural simulation of the brain, however, is not only impractical but would make it virtually impossible to separate the details of the simulation 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 major advantage of cognitive architectures over the traditionally narrow modeling in psychology is the generalizability of the resulting theory. By integrating specific psychological theories of memory, learning, attention, problem solving, and other phenomena into a coherent whole, cognitive architectures become foundations for a broad class of models. This is especially important for modeling program comprehension, since existing models have hand-picked theories from all over psychology [19]. Cognitive architectures have also been applied to the study of human-computer interaction [12], suggesting that they are well-suited for modeling complex, inter-disciplinary tasks.

A cognitive architecture is also a piece of software. The inputs to this software are a description of the model (usually written in a domain-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 of the most prominent cognitive architectures today are Soar [29], EPIC [24], 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 preconditions 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 [12]. For the remainder of the paper, we concentrate on ACT-R. While we believe a successful model of program comprehension could be implemented in all of these architectures, we have chosen to focus on ACT-R for several reasons. First, it is widely-used around the world, and is actively being developed [1]. Second, it contains perception and motor modules, allowing for models that combine cognition and interaction with the environment (called *end-to-end modeling*). Lastly, a number of fMRI studies have been done to associate ACT-R modules with particular brain regions [3]. While this last fact is not crucial, it provides an additional means of verification for any future program comprehension model developed in ACT-R.

4. ACT-R: A Cognitive Architecture

ACT-R is "*a cognitive architecture: a theory about how human cognition works.*" [1] Built on top of LISP, it is a domain-specific programming language and simulation framework for cognitive models. ACT-R models embody assumptions about how humans perform particular tasks, and can be observed along several psychologically relevant dimensions such as task accuracy, response times, and simulated BOLD measures (i.e., fMRI brain activations). Because ACT-R models generate quantitative results, they can be used to make precise predictions about behavior. These predictions can inform future experiments with human subjects, or suggest changes to the prevailing theory.

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. Modules compute and communicate via *productions*, rules that pattern-match on the state of one or more buffers including the chunks they may contain. When a production matches the current system state, it "fires" a response. Responses include actions like inserting a newly constructed chunk into a buffer and modifying/removing a buffer's existing chunk[2].

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[3], making the procedural model the central bottleneck of the system.

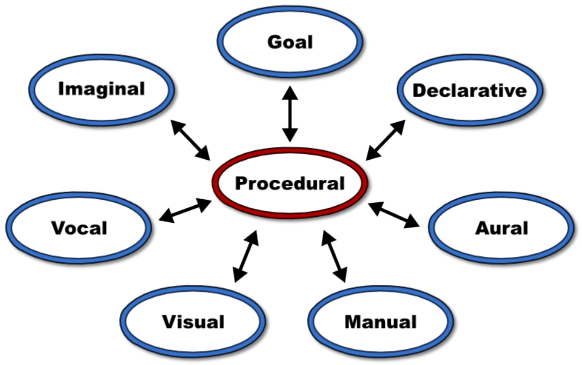

Figure 2. ACT-R 6.0 modules. Communication between modules is done via the **Procedural** module.

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 goal and problem state, respectively. Production rules are usually written to fire only when the system is in a particular state (based on the goal buffer chunk) and when relevant pieces of the problem are available (based on the imaginal buffer chunk).

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, percetion, and attention [1]. Thus, there are many constraints on models when fitting model parameters to match human data.

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

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 when multiple productions successfully match the current state[4]. 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 using a set of well-defined rules. Production compilation could occur, for example, when production P_1 retrieves stimulus-response 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 $P_{1,2}$

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

[3] This is a point of contention in the literature. Other cognitive architectures, such as EPIC [24] allow more than one production to fire at a time.

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

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[5]. We provide a handful of examples here, though many 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 [34]. 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[6], the integrated model was able to switch between tasks. Salvucci's model successfully predicted drivers' dialing times and lateral deviation from the intended lane.

Brian Ehret (2002) developed an ACT-R model of location learning in a graphical user interface [21]. 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.

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 (i.e., starting with correct usage - *went*, over-generalizing - *goed*, and returning to correct usage - *went*) [41]. This model proposed that the U-shaped curve results from 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.

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.

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 [13]. The CCM focuses on the processes of *chunking* (understanding a block of code) and *tracing* (locating dependencies). Cant et al. provide operational definitions for the factors that are believed to influence each process. Some of the factors in the CCM are operationalized by drawing upon the existing literature, but many definitions are simply placeholders for future empirical studies. The complexity metric (C) depends on a dozen factors, many of which contain one or more "empirically determined constants".

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). These processes would not need to be reimplemented by the CCM, and would simplify the existing model significantly. Below, we describe the Cognitive Complexity Metric 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.[7]

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. Of course, 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 [13].

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.*" For the purposes of the CCM, however, they define a chunk as a block of statements that must occur together (e.g., loop + conditional). This definition 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 do our best to 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 [8] as follows:

[5] Success here means that the models fit experimental data well, and were also considered plausible explanations.

[6] Salvucci notes that this required hand-editing the models' production rules since ACT-R does not provide a general mechanism for combining models. See [35] for a less naive model of multi-tasking in ACT-R.

[7] Note that we have not implemented an ACT-R model yet; this is planned for future work (Section 6).

[8] Note that Cant et al. discussed more general forms for the equations of R and T, but suggested starting with simple additive representations.

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

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

Each right-hand side term stands for a particular factor that is thought to influence the chunking or tracing processes (Figure 4). Below, we provide a high-level overview of each factor (more details can be found in [13]). In addition, we conjecture about how an ACT-R model of the CCM would incorporate and implement these factors.

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.

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 are believed to influence how hard it is to understand a given chunk. Cant et al. cite examples from the literature to support the inclusion of each factor.

$\mathbf{R_F}$ ***(chunk familiarity)*** This term is meant to capture 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 nicely, 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 should come at the cost of slower retrieval times for similar chunks.

$\mathbf{R_S}$ ***(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. The additional "psychological complexity" could be modeled in several ways. ACT-R does not contain a distinct short-term memory component, and instead relies on long-term memory to serve as a 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 [40].

$\mathbf{R_C}$ ***(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 (using the subvocalize feature of ACT-R's vocal module to simulate the inner voice). 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 in a chunk depends heavily on their form and the degree to which they are nested. To incorporate boolean 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 [22]. An ACT-R model would likely 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 also make errors similar to real programmers when the observed code violates specific rules of discourse [38]. ACT-R has the ability to partially match chunks in memory by using a similarity metric in place of equality during the matching process. This ability would be useful for modeling recognizability, since slight changes in code indentation and layout should not necessarily result in a failure to retrieve a known pattern.

$\mathbf{R_V}$ ***(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 [46]. 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 [17]. Biggerstaff et al. have also designed a system for automatic domain concept recognition in code [8]. 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 in order to produce these effects. 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 distance threshold above which text is considered to be separated by whitespace.

$\mathbf{R_D}$ ***(dependency disruptions)*** There are many disruptions in chunking caused by the need to resolve dependencies. This applies not only to remote dependencies, but also to local ones such as nested loops and decision structures. Cant et al. cite the small capacity of short-term memory as the main reason for these disruptions, and propose that both the complexity and tracing difficulty of dependent chunks be included in R_D. As mentioned previously with the chunk familiarity term (R_F), ACT-R does not have a distinct "short-term memory" module with a fixed capacity. Instead, the declarative memory module serves as a long-term and working memory, with short-term capacity limits being 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 (i.e., the number of disruptions that can be handled without issue is not fixed). This is a case where the psychological theory underlying the cognitive architecture can help to suggest new experiments on humans.

Tracing Difficulty (T)

Most code does not stand alone. There are often dependencies that the programmer must resolve before chunking the code 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 may be opened up in a new tab or bookmarked within the programmer's development environment. We expect that a comprehensive ACT-R model will need to interact with the same tools during an experiment as human programmers in order 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, while a local dependency is within modular boundaries (e.g., within the same function). This classification might 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. 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.

$\mathbf{T_S}$ ***(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).

$\mathbf{T_C}$ ***(level of cueing)*** This binary term represents whether or not a reference is considered "obscure." References that 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, so an ACT-R model may need to maintain multiple whitespace thresholds for the different processes. In other words, the threshold above which two pieces of text are considered separate blocks may be higher during tracing.

6. Discussion

Our understanding of the psychology of programming has come far in the last several decades, but there is still a long ways to go. While many cognitive models of program comprehension are available today, the majority are verbal conceptual theories. Because the components and processes of these models are specified using natural language, it is not possible to quantify programming language and library design trade-offs in terms of usability. The biological plausibility of existing cognitive models is also quite limited. While some recent models have begun to embrace cognitive science research on working memory from the last thirty years [4], many simply assume a fixed size for short-term memory and cite Miller's Magic Number [27].

We believe that operationalizing models of program comprehension on top of a cognitive architecture will solve many of the above problems. Cognitive architectures require models to be well-specified, and contain quantitative implementations of modern psychological theories. Simulations within a cognitive architecture are reproducible, and results can often be directly compared to human data collected for the same task. Parameters within a cognitive architecture also have psychological meaning, so fitting a model to human data is not an unconstrained search through the parameter space.

ACT-R is a well-known, actively developed cognitive architecture that has be successfully used in a variety of domains. In addition to traditional cognitive components, it provides perception and motor modules that allow for end-to-end modeling of a task (the model interacts with the same environment as human subjects). Studies of fMRI data have also associated ACT-R modules with specific brain areas, providing an additional means of model verification [3]. We believe that these reasons make a strong case for the use of ACT-R in future program comprehension modeling efforts.

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. Providing reasonable operational definitions of a cognitive model's components is not an easy task for a complex activity like programming. Our review of Cant et al.'s Cognitive Complexity Metric hinted at how ACT-R would facilitate a complex model, but many details were left out. Because of this difficulty, there will be the tendency for modelers to pursue a "divide-and-conquer" approach. Like traditional psychology, the temptation is to model only narrow phenomena with the intention of putting the individual pieces back together again someday[9]. As Alan Newell said in 1973 regarding the state of cognitive psychology, however, "*you can't play twenty questions with nature and win.*" [28]. We believe the same applies to the psychology of programming, and that using a cognitive architecture would encourage modelers to build on top of each other's work instead of splintering efforts to study isolated phenomena.

Interpretation issues. Even if an ACT-R model of program comprehension were to exist today, interpreting its simulation output would not necessarily be straightforward. This output might 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

[9] This is often referred to as the "Humpty Dumpty Problem".

may include additional assumptions). Extreme care must be taken by the psychology of programming community to avoid making claims for "better" or "worse" designs based on only a single dimension of a model's output (e.g., design X is better than design Y because the model takes more time to complete its task with Y).

Social rejection. It is always possible that designers will not want to cede judgement to a cognitive model, even if there are substantial benefits. Computer scientists in general have a reputation for making usability claims with virtually no hard evidence to back them up [26]. While this may be due to reasons of education or funding, there are surely some language/library designers who simply believe they know what is "best" for the average developer. 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 will still be a major challenge to get designers to see it as a useful tool.

Potential Benefits

Despite the challenges above, 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 focusing the community on human factors.

Quantifying and optimizing. A major goal of this work is to be able to quantify design trade-offs in an objective manner for usability purposes. This might be used to resolve disagreements between experts who are speaking on behalf of "Joe Coder" [39], but there is potential for semi-automated design optimization as well. Given a cognitive model and optimization criteria (i.e., higher accuracy on X, fewer keystrokes to do Y), a program could attempt to optimize a given design. This kind of approach has been considered by Air Force researchers in order to optimize the heads-up display (HUD) provided to expert pilots. If anything, the suggested "optimal" designs from a program comprehension model's perspective could be used in new and interesting human experiments.

Lower barriers to user evaluation. There have recent been calls for the use of more rigorous empirical methods in computer science, particularly when researchers are evaluating the benefits of new languages and programming paradigms [23]. Buse et al. investigated the main barriers to user evaluation perceived by computer scientists, and found them to be (1) the recruiting of subjects, (2) the time and effort taken to design experiments, and (3) the challenges of getting approval from their local institution review board (IRB) [10]. While a comprehensive ACT-R model would not replace user evaluation, it could serve as a preliminary means of evaluating new ideas. Early experiments could be refined using the model as a subject, saving time and money that would otherwise be spent on recruiting and running human subjects. Some expertise with ACT-R would needed to properly code experiments and interpret the results of simulations, but this might help overall to lower common barriers to real user evaluation.

Spotlight on human factors. A successful and widely-used quantitative cognitive model could help dissuade some computer scientists from believing that research with human subjects is merely "soft science". As Newell and Card said over twenty-five years ago, "*. . . hard science (science that is mathematical or otherwise technical) has a tendency to drive out softer sciences, even if softer sciences have important contributions to make.*" [30] We do not want to see human factors kept out of mainstream computer science research merely because it is considered "soft" (see [42] for additional excuses that computer scientists make to avoid experimentation). This is important not only for computer scientists, but also for the increasing number of programmers from non-computer science fields [5].

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, though there has been significant work since 1995 that we plan to incorporate. The use of eye-tracking in coding experiments, for example, has increased in recent years [6] making it possible to compare the visual search strategies of different programmers. There have also been extensions to the work done by Soloway and Ehrlich on programming plans and the different roles that variables play in a program [33]. We are encouraged by the success of ACT-R models in other complex domains, such as a model of students learning to solve simple linear equations [3].

7. Conclusion

Predicting how design changes to a language or library will affect developers is a challenge for the psychology of programming field. Cognitive modeling efforts over the last several decades in this field have produced many useful models that provide explanations for existing empirical data. These models are almost all verbal-conceptual in nature, however, making them difficult to combine and reason about precisely. In this paper, we argue that modelers should use a cognitive architecture in order to operationalize existing models and ultimately quantify the usability trade-offs between language/library designs. Specifically, we recommend the ACT-R cognitive architecture because of its large user base, active development, perceptual/motor modules, and associated fMRI studies.

Quantitative cognitive modeling with a cognitive architecture is not without its challenges. Operationalizing the components of existing program comprehension models is a massive undertaking. Modelers must also become fluent in a new language, and also learn how to properly interpret the output of their model simulations. Despite these challenges, we believe that 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 might be lowered, promoting more focus on human factors in computer science.

The psychology of programming brings together researchers in 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 for a wide audience.

References

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

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

[3] J. 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] S. Baxter, S. Day, J. Fetrow, and S. Reisinger. Scientific software development is not an oxymoron. *PLoS Computational Biology*, 2(9):e87, 2006.

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

[7] 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.

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

[9] 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.

[10] R. 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.

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

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

[13] S. Cant, D. 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.

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

[15] 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.

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

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

[18] F. Détienne. What model (s) for program understanding? 1996.

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

[20] C. Douce. The stores model of code cognition. 2008.

[21] B. 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.

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

[23] 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.

[24] D. E. Kieras and D. 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, Dec. 1997. ISSN 0737-0024.

[25] A. Ko, B. Myers, M. Coblenz, and 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.

[26] 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.

[27] G. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[28] 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.

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

[30] A. Newell and S. Card. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1(3):209–242, 1985.

[31] J. 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] S. Sheppard, B. Curtis, P. Milliman, M. Borst, and T. Love. First-year results from a research program on human factors in software engineering. In *afips*, page 1021. IEEE Computer Society, 1979.

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

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

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

[41] N. Taatgen, J. 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.

[42] W. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

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

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

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

[46] S. Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6): 697 – 709, 1986. ISSN 0020-7373. doi: 10.1016/S0020-7373(86)80083-9.