Designing a Debugging Interaction Language: An Initial Case Study in Natural Programming Plus

Christopher Bogart[1], Margaret Burnett[1], Scott Douglass[2], Rachel White[1], Hannah Adams[1]
[1]Oregon State University
Corvallis, Oregon 97331 USA
[2]Air Force Research Laboratory
Dayton, OH 45433 USA
{bogart, burnett, white, adamshan}@eecs.oregonstate.edu, Scott.Douglass@wpafb.af.mil

ABSTRACT
In this paper, we investigate how a debugging environment should support a population doing work at the core of HCI research: cognitive modelers. In conducting this investigation, we extended the Natural Programming methodology (a user-centered design method for HCI researchers of programming environments), to add an explicit method for mapping the outcomes of NP's empirical investigations to a language design. This provided us with a concrete way to make the design leap from empirical assessment of users' needs to a language. The contributions of our work are therefore: (1) empirical evidence about the content and sequence of cognitive modelers' information needs when debugging, (2) a new, empirically derived, design specification for a debugging interaction language for cognitive modelers, and (3) an initial case study of our "Natural Programming Plus" methodology.

Author Keywords
Natural Programming; Evaluation Abstraction; Cognitive Model; End-user software engineering

ACM Classification Keywords
D.2.5 [Software Engineering]: Testing and Debugging; H.1.2 [Information Systems]: User/Machine Systems—Human Factors

General Terms
Experimentation

INTRODUCTION

Although the needs of both professional and end-user programmers have become popular topics in HCI research, the HCI needs of people who program in order to build scientific models have received relatively little attention—especially in the realm of debugging. Tools and languages exist to enable them to *write* programs, but relatively little research investigates how to support them in the debugging phase of programming.

A population of modelers in the very core of HCI research is cognitive psychologists working with cognitive models. Cognitive models have contributed important foundations to HCI, such as GOMS, information foraging theory, and cognitive tutoring (e.g., [1, 6, 8, 15, 18]). A few practical tools for modeling have emerged from the modeling community itself (e.g. [20]), but HCI research into how to support the population doing this important work is sparse. This paper aims to help fill this gap.

We believe that cognitive modelers mentally construct *evaluation abstractions*—abstractions they work with when evaluating a model's runtime behaviors [3]. These abstractions range from simple reflections of a model's internal data structures (e.g., content of simulated short-term memory as a model runs), to much more complex abstractions (e.g., some particular recurring pattern of short-term memory changes). When modelers' abstractions do not coincide with the model's internal data structures, today's debugging tools do not support them well.

To overcome this lack of support, cognitive modelers sometimes write separate programs just to debug their models. Modelers use these *secondary programs* to examine their models' (i.e. *primary programs'*) logs and outputs, to understand, debug, and validate the model's behavior. [3, 19]. The abstractions captured by these secondary programs are often the same abstractions that modelers verbalize as information goals while debugging [3], so we set out to design a useful design specification for an interactive "debugging language" that would support both kinds of evaluation abstractions – one-off debugging questions and persistent secondary programs – within a single unified tool.

To accomplish this, we needed to choose a methodology to investigate the constructs, relationships, and interaction sequences that modelers used to assess and fix model behavior, at a finer-grained level of detail than previous work [3]. We were faced with the choice between a task analysis, (appropriate for interactivity, and for one-off debugging questions) and a language design methodology (appropriate for the secondary programs described above). Since modelers' information queries in both areas seemed to have a complex internal structure, we chose to adapt a programming language design methodology: Natural Programming.

Natural Programming [17] is a user-centered design approach in which researchers observe how people try to naturally express programming intentions, and use these observations to devise programming tools whose conceptual models fit as closely as possible to the participants' expressions. This technique was first introduced to design the children's programming language Hands [16]. It has since

CHI '12, May 05 - 10 2012, Austin, Texas, USA

been used to design numerous tools that support programming, scripting, and debugging (e.g., [11, 13, 14, 21]). The Natural Programming methodology, however, leaves implicit exactly *how* designers should connect their empirical observations of people to a new language that can serve those people well.

Therefore, for this work we extended NP (henceforth, "Natural Programming Plus" or NP+), by adding new steps for precisely and accountably treating interactive sequences of naturally expressed verbal "programs" and their results (in our case, modeler's evaluation abstractions) as cases of a language specification. This precision helped us by providing both scaffolding for our effort to design an interactive debugging language based on empirical evidence, and ongoing analytical measures of how well the emerging language matched that evidence.

The contributions of this paper are:

- Empirical evidence about the evaluation abstractions requested by cognitive modelers, and how those abstractions were sequenced over time.
- An empirically derived and validated design specification for a debugging interaction language for cognitive modelers.
- An initial case study of Natural Programming Plus, an extension of the Natural Programming methodology to more precisely capture and validate the structure and flow of ideas expressed by the participants.

The first two contributions also serve as initial data points towards an understanding of the potential of NP+ as a methodology of wider interest, a prospect which we will discuss at the end of this paper.

METHODOLOGY

In this section we present our methodology at a high level. The remaining sections then illustrate how we used it.

Pane and Myers [17] defined the Natural Programming methodology as four steps (applied iteratively, as needed):

A. Identify the target audience and domain
B. Understand the target audience
C. Design the new system (e.g., a language or programming tool's interaction language)
D. Evaluate the new system

We expanded on Step B to provide a *process* for designing and validating a specification of the new language. The essence of our process was to use a pipeline of two experiments, which together allowed refining the results to derive a precise specification, and finally empirically validating three properties of the derived specification. Thus, as Figure 1 summarizes, we replaced Step B with the following four steps:

Step B1/Formative: As is implied by the original Natural Programming process, we conducted a formative study (see Study N, below) to harvest the constructs and relationships that modelers used to describe what they were looking for when debugging. B1's results are the constructs and relationships the participants used. *Why:* These constructs and relationships are the basis of a software tool for administering the experiment in B2.

Step B2/Wizard: We performed a Wizard of Oz study (see Study W, below), using a query tool in which the constructs and relationships of B1 were supported, but without a concrete syntax or GUI yet. Participants were asked to seek information similar to queries observed in the course of B1's tasks, by asking the experimenter (Wizard) who would manually query the results and show the participant a table of the results. The results of this step were (1) any constructs and relationships missed or misunderstood in B1, and (2) the way participants naturally *sequenced* their interactions in response to the feedback of executing each query. *Why:* This step revealed how modelers responded when the capabilities derived from B1 actually executed. It also allowed us to validate the B1 results with the target audience itself.

Step B3/Derive a precise specification: We refined the results of B1 and B2 into a language specification. When possible, we structured the specification such that small differences from one user request to the next were mirrored by small differences in the language used to represent modelers' queries. *Why*: The language specification is a precise form of "Implications for Design". Because it is precise, it was auditable, and this facilitated evaluation and kept us accountable.

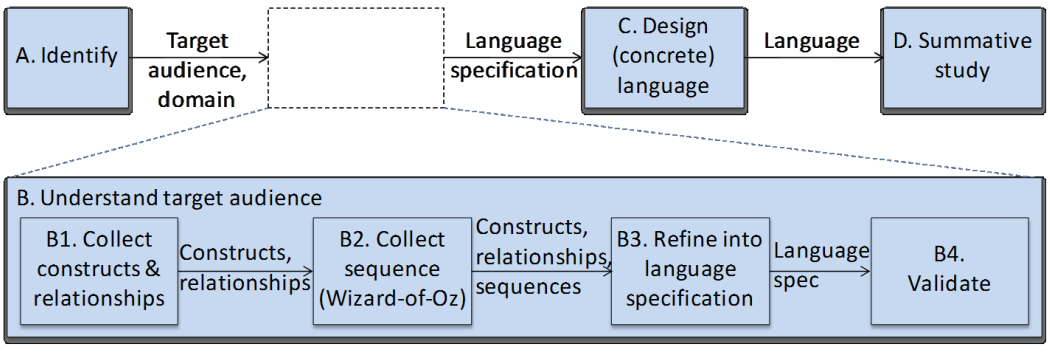

Figure 1: Natural Programming Plus replaces NP's Step B with Steps B1-B4. Arrows show what results feed from one step to the next. The "Language" from step C to D may mean an interaction language or programming tool, not necessarily a programming language.

Step B4/Validation: We measured *coverage* of the language specification (how many of B2's requests it could execute), its *soundness* (correct-

ness, i.e., the responses it did produce are what the participants asked for, in the context of the available data), and its *viscosity* (the effort required in the new language as specified to change from one request to the next, if the requests were related.)

The rest of this paper shows how we applied Steps B1-B4 of NP+ to the problem of designing a debugging interaction language for cognitive modelers.

OUR POPULATION: COGNITIVE MODELERS

Cognitive modelers try to model cognitive functioning of the human mind. They often have backgrounds in psychology or linguistics; some also have backgrounds in computer science, but many do not (as our empirical data in later sections will show). To build their models, they sometimes use rule-based languages specifically designed for cognitive modeling, such as ACT-R, and ACT-R provided the context for our investigation. ACT-R is both a theory of human cognition, and a simulation language that implements the theory. In ACT-R, modelers specify rules that move "chunks" of information among cognitive subsystems such as vision, memory, goal, and motor modules. The *chunk* is ACT-R's primary data structure, which consists of a varying number of named slots, and simulates a grouping of mental information in short-term memory (*buffers*) or long-term (*declarative*) memory. ACT-R builds in current assumptions from the cognitive science community about how these subsystems work.

A common task of cognitive modelers is to simulate a human subject participating in a psychological experiment. In the simulated experiments, the modeler manipulates something and the model (i.e., the simulated human) responds. This stimulus/response pattern happens multiple times, and each instance is called a *trial*. Yaremko et. al. define a trial as "a single instance or event from which a datum is collected" [22].

Time passes during a trial, and many events may occur between the stimulus and response. Data that could in principle be collected about a single trial include things such as: a start and end time as per a simulation clock, the timing and attributes of stimuli presented and responses observed, and the timing and attributes of the model's (simulated human's) internal mental events. Thus, trials are composed of data, some or all of which a cognitive modeler may find interesting when evaluating or debugging a model.

STEPS B1-B2: TWO EMPIRICAL STUDIES

Informed by a taxonomy of evaluation abstractions and operations we identified [3], we conducted two studies to identify the ways cognitive modelers went about a debugging task. The combined goal of these two studies was to identify the concepts and relationships behind modelers' information requests in debugging, and how they were sequenced in time, as required by Steps B1 and B2

Study N: Participants and Methods

Study N ("N" for native environment) was a talk-aloud study whose aim was to elicit modelers' information-seeking language and approach for evaluating an ACT-R simulation's runtime behavior.

We recruited 8 cognitive modelers at the Air Force Research Laboratory and Carnegie-Mellon University. The modelers' experience (primarily in the ACT-R language) ranged from a few months to 20 years. Five were Ph.D.s, two had masters degrees, and one was a Ph.D. candidate. Their degrees were in psychology (3 modelers), computer science (3), and linguistics (2).

The participants worked to debug the models "Zbrodoff" and "Paired" from the standard tutorials [4] distributed with ACT-R 6.0. The Zbrodoff model we gave them was an early attempt by one of the experimenters to build this model, in which the author's rule design was flawed. The Paired model's bug was a timing problem we introduced into a correct solution written by one of the experimenters; we chose that bug in order to provide a contrasting bug where the rules appeared to be correct, but the behavior was wrong.

Participants had 30 minutes to work on each model. Three of the participants spent an hour and worked on both models, and the remaining five spent a half hour and worked on just one model. Participants used the ACT-R 6.0 tool set, and chose for themselves whether to use a textual or GUI environment, elements of which are shown in Figure 2. Participants talked aloud as they worked, and we video-recorded their sessions.

Study W: Participants and Methods

Although Study N gave us a good sample of relatively natural debugging behavior, the data was sparse for more complex evaluation abstractions. Traditional debugging tools such as ACT-R's do not allow for automated extraction of complex evaluation abstractions, and on several occasions we saw modelers ask themselves complex questions, but either guess at their answers based on scant evidence, or set

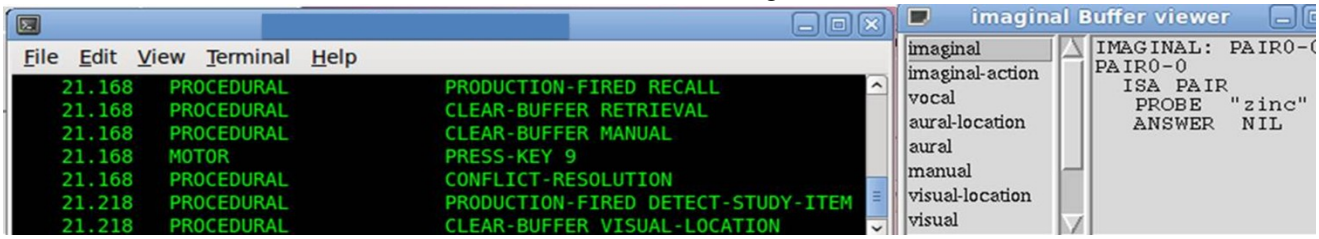

Figure 2: Elements of the standard ACT-R environments. (Left): Trace showing events and their properties. (Right): Buffer viewer showing a chunk in the "imaginal" buffer of the model's "short-term memory".

them aside because they were too expensive to pursue. We wondered what information seeking strategies modelers *would* use if such a tool existed.

Therefore, for Study W ("W" for Wizard-of-Oz), we designed an experiment to observe just one aspect of the debugging process: seeking runtime information in a model trace. We built an experimental tool to execute queries similar to the more difficult questions modelers asked during Study N. To focus users on this subtask alone, we had them answer specific questions, and we denied them access to other tools or information that might support the habitual workarounds we had already studied. For example we did not show them the model's source code, to prevent them using it to guess or infer model behavior. Note that this highly constrained design limits the validity of Study W results to pure trace inspection behaviors, and the results should be interpreted in conjunction with more natural observations, such as Study N and our prior work in this domain [3].

We recruited 7 cognitive modelers at the Air Force Research Laboratory, with experience (primarily in the ACT-R language) ranging from six months to 10 years. Five were Ph.D.s and two had masters degrees. Their degrees were in psychology (4 modelers), computer science (2), and linguistics (1). Three of these participants had previously participated in Study N. For clarity, we will prefix each participant ID with "N" for Study N and "W" for Study W.

In selecting a model for the task, our criterion was that it should present challenges similar to questions we saw Study N modelers pose, but that they failed to easily answer with existing tools. This let us observe how modelers would take on these challenges in areas where the existing toolset is weakest.

To satisfy this criterion, the model we used in Study W was a defective solution to one of the modeling exercises in the ACT-R 6.0 tutorial [4], simulating how a child learns regular and irregular verbs. A bug was chosen that was not trivial to spot: the model's rules produced a mix of right and wrong verbs, as real children do, but not in the right proportions, and it failed to follow a child's typical learning curve. We chose this task because it was heavily dependent on complex runtime behavior over a long time span, and we believed it would provide a rich context for exactly the kinds of questions modelers found difficult to answer with existing ACT-R tools. We ran a single simulation of 500 trials, then loaded the trace data into our tool. Figure 2 (Left) shows a few events of that trace. We set our tool's initial display to the same output as the ACT-R tutorial.

Study W's participants' tasks were to find answers to the following questions, designed to be similar to questions that had caused participants difficulty in Study N. (T1): In trial around 54000 seconds, the model produced "HAD" as the past tense of "HAVE". Was that the first time that happened? (T2): What kinds of verbs are counted as regular and irregular? (T3): Which rules, if any, ONLY fire when the model is about to produce an "-ed" ending? (T4): In the trial that starts about 21017 seconds, Production70[1] fires. Is that typical? If so, what's special about trials that don't do this? If not, what's special about this trial? (T5): Under what circumstances (if any) does the model write a chunk to declarative memory that is grammatically incorrect?

To perform these tasks, participants verbally told the experimenter what information they wanted from the program's runtime trace. The experimenter (the Wizard) used the tool to produce the information the participant had requested. Participants were allowed to point out errors in the Wizard's interpretation of their queries, and the Wizard fixed them until the participant was satisfied. Audio, video, query text, and screenshots were recorded for all sessions. All participants performed Tasks T1, T3, and T4, five performed T5, and three performed T2. We allowed participants to work on the tasks as long as they liked, but cut off the sessions at 1 hour, regardless of the number of tasks completed.

The study produced 149 episodes of participant queries and experimenter responses. Twelve were requests to look at previous queries, and four were garbled or incomprehensible, leaving 133 distinct queries. We analyzed these data in an iterative process that ultimately led to the language spe-

Abstraction (instance count) and Participant request example	(Small portion of) result of the request
Trial (75): The begin and end time of a trial, and several model-specific attributes. *W413a: All the trials where the verb is HAVE […] I would like to see what the stem is*	`trialnum: 2` `start_time:200.155` `word: "HAVE"` `stem: "HAD"` `end_time: 400.383` `[…other trials…]`
Event (26): A value with a time stamp. *W412a: I'll do a list of when Production70 fires.*	`time: 15814.232` `rule_name:` `"PRODUCTION70"` `[…other events…]`
State (20): A value with a begin and end time. *W415a: So it executes a retrieval […] I want to see the details of that chunk.*	`type: past-tense` `buffer: retrieval` `verb: use` `stem: use` `suffix: ed` `start_time:1802.268` `end_time: 2002.512` `[…other states…]`
Rules (12): The text of a production rule. *W413a: Can I search for rules that […] affect the suffix slot?*	`(Wizard refused; experiment prohibited use of rule text)`
Total (133)	

Table 1: Types and counts of abstractions that Study W modelers queried in Study W as they worked. (In Study N, *all* modelers drew on all four categories of data.)

[1] Production70 was a rule that the model learned.

cification of Step B3. We describe the ways we validated the analysis in a later section, but first we describe the empirical results and implications (labeled as I-*).

B1/B2 RESULTS: THE MODELERS' ABSTRACTIONS

The modelers' abstractions that we observed in Studies N and W consisted of constructs that fell into four categories: trials, events, states, and rule text.

The "trial"

The experimental "trial" is a staple in the practice of cognitive modeling, but it is not well-supported in ACT-R's standard tool set. The only abstractions supported by the debugging tools are simply the ACT-R *programming abstractions,* such as chunks and buffers (recall the section about our population). As a result, modelers can point and click to see chunks, but to see trials, they would have to write Lisp code to show them, or use some manual process. For example, Participant N706 spent 7% of his time trying to find a way to do a textual "find" in an ACT-R log file, just so he could step through and find out how many boundaries, and thus how many trials, were in the run.

Although modelers struggled when comparing entries that were far apart in a lengthy trace, four of the eight participants in Study N nonetheless chose debugging strategies that involved explicitly comparing behaviors between trials. This suggests that trials were critically important to modelers, despite their lack of support.

We therefore introduced support for trials in the tool we built for Study W, in the form of a two-paned window that let participants choose trials in one pane, and see the details in the other (Figure 3). Study W modelers made heavy use of them: trials were at the root of about half (75 of 133) of all requests in Study W (Table 1). This detailed view enabled Study W modelers to click on different trials and immediately see the rule sequences, which reduced minutes of searching down to a single request.

This design was still not ideal, however, because multiple sequences were not visible at once, as several of the modelers pointed out. W412b worked around the limitation by remembering one sequence while he looked at another in seeking patterns. W415d had the Wizard add summarized facts about each sequence as attributes to each row of the trial listing (e.g., Figure 3, top), such as how many rules fired, whether event or state properties were present, or the identity of the last rule that fired. W412a on the other hand asked for a new feature:

> W412a: [...to] visualize the sequence of productions fired so that I can make a visual comparison, because going through a list is a bit tedious.

Implications for supporting trials

Although the native ACT-R tools faithfully reflect the model's continuous view of time, modelers need support for a segmented view of time *(I-TRIAL)*, defined by identifying some event type as a boundary between trials. Modelers also needed support for viewing and comparing details within those segments *(I-VISUALIZE, I-DETAIL)* as well as collecting summaries or visualizations of critical features of those details *(I-COLLECT)*.

Events vs. States

Modelers' second and third most common constructs were model events (behaviors) and model states (data). Study N modelers made extensive use of both event logs and displays of variable contents (particularly ACT-R's chunks and buffers) to learn about the models' behavior and state. Study W modelers also showed strong interest in events and states (Table 1), asking 26 queries about attributes of momentary events (primarily stimuli, responses, and rule firings), and 20 about state (working memory buffers and long-term memory chunks).

Interestingly, even when modelers talked about state, they tended to use event-oriented language, referring to some event *during* the state, or *marking a change* of state:

> W415a: What I would look for are *events* where a chunk with an ED ending *was put in the imaginal buffer* then look backward from that to find the […] imaginal action that put the chunk there. (Italics added for emphasis.)

Implications for Supporting Events and States

Modelers evaluated in terms of both instantaneous events, and states that persisted over time. Thus, these types of evaluation abstractions are needed. The ways in which they worked with these suggests that their query language should allow referencing events as events, but referencing states as *attributes* of the events at their boundaries *(I-EVENT)*, or during their lifespans *(I-DURING)*.

All about rules

All participants in Study N kept a window open all the time showing rule text. To avoid gathering redundant data on code inspection, we allowed Study W participants to see only the names and dynamic behavior of rules, but not their text. Still, in twelve (9%) of Study W's 133 episodes, modelers asked to read rule text, sometimes quite adamantly:

> W415a: I'd really like to see the production. May I see the production? ... It seems natural that you'd want to look at the production.

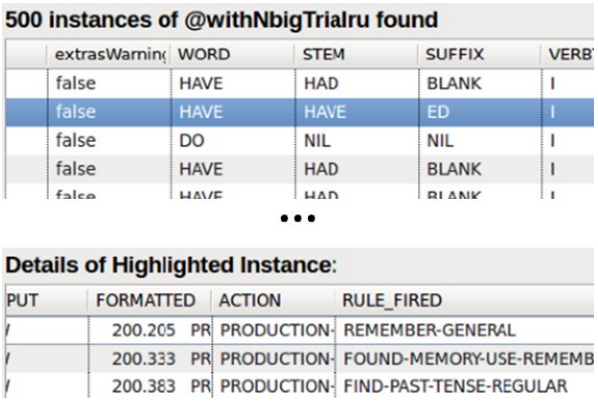

Figure 3: W412b asked for "the production firing sequence" (below) "…within this trial" (above, highlighted).

Not only did modelers want to see specific rules, but they wanted to find rules having some attributes, in order to identify causes of events, or to compare rules to each other. For example, Participant N701 noticed in the log an error in which the model was trying to press a non-existent key called "rope" (the Paired model was supposed to press a digit key in certain circumstances). That participant then searched the rules' text for "press-key", to find candidate rules that may have been immediately responsible for this erroneous action. Similarly, in Study W, a modeler asked:

> W415c: What productions do retrievals?

These can be time-consuming questions to answer in the native ACT-R environment, as the information is scattered in several places. For example the trace shows only the *names* of rules that fired (e.g., in Figure 2 (Left), the second row from the bottom shows that rule DETECT-STUDY-ITEM fired). The model source file contains rules' content, but only rules written by the modeler, not rules the model learns itself (through "production compilation").

Implications for Rule Information:
(I-RULETEXT): Unsurprisingly, the modelers needed to see rule text. However, an interesting nuance is that modelers wanted to query the text of rules, both human-authored and model-generated, according to their attributes. This suggests the need to query rule text in the same ways as states, events, and trials.

B1/B2 RESULTS: RELATIONS AND SEQUENCES

Modelers made elaborate queries that composed, filtered, selected, or summarized the simpler references to events, states, trials, and rule text discussed above. Table 2 lists the operations that made up these queries.

Operations for composing queries

Time-based and Dataflow/Slice composition
When modelers had questions relating to the sequencing of events in the trace, they often needed multiple navigations to answer them. For example, Participant N702 restarted a run and painstakingly stepped forward to an "earlier" time he was curious about. By the time he found it, his previous run was no longer in the scroll buffer:

> N702: Oh, great, now I've lost the previous trial and I'm doubting my memory... did this one fire? it was the next one that didn't fire?

Study W modelers also asked for events with temporal relationships, usually starting with a known "anchor" event and adding a related event before, after, or simultaneous with some other event of interest. For example:

> W412a: I want to see what productions fired at these times. […] or should we go back 50ms to see who produced these?

Events or states connected by dataflow and/or control-flow relationships were regularly of interest to modelers. Some of these requests were data centric:

> W415c: So the chunks that were in declarative memory… what buffer were they stored in [before they were in declarative memory]?

Other, more intricate requests sought rules that had particular effects on data over time. For example to determine why a particular chunk was retrieved, Study N modelers worked backwards through the code to determine what had triggered its retrieval. They essentially had to construct by hand a backward slice of code that affected the output of interest.

Implications for Facilitating Composition
(I-TIME): Modelers used a variety of temporal relationships: next, previous, simultaneous, and "in the same period". Such operations need to start with all "anchor" events, then either *include* or *exclude* instances where the non-anchor event is missing, in order to answer, respectively, *whether* or *what kind of* events happened at nearby times. (*I-DATAFLOW):* Modelers also needed operators to understand how data moves from one variable to another through various dataflow and control flow relationships.

Operations for summarizing

Study N participants were drowning in data. Modelers spent a great deal of time scrolling through ACT-R's very detailed logs and clicking through the debugger. Our purpose in pursuing support for evaluation abstractions is to allow modelers to hide extraneous information, leaving just the relevant information accessible.

Filtering
One "fire hose" of data was the declarative memory dump. In the Paired task, most of the Study N modelers listed all the chunks in long-term memory to see if they were being created correctly. They drew wrong conclusions about the distribution of chunks in at least half the cases because chunks of the same type could not easily be made visible at the same time.

Motivated by the flood of information overwhelming the Study N modelers, we provided a more general filtering

Operators for composition	
Time constructors: Next, previous, simultaneous, within-trial items	Produce all items with the specified time relationships, starting with an "anchor event", and including/excluding items with no secondary event.
Slicing and dataflow constructors	Produce a backward dynamic slice through code or a backward flow of data through data structures.
Operations for summarizing, filtering, or rearranging	
Filter	Limit the items shown.
Distinct	List and count distinct values of some attribute.
Set	Do set operations on *distinct* results.
Sort	Rearrange items in order by some property.
Operations for comparing details	
Any, First, Last	Produce any, the first, or the last, respectively, single item with the specified properties.
Visualize	Produce a graphic (e.g., a bar chart) of all items with the specified properties.

Table 2: Query-building operations in Study W. "item" means an event, trial, or any other abstraction.

capability in Study W, and the modelers used it extensively. Modelers filtered data in 122 of 133 episodes, and actively changed the way they were filtering in 32 of them.

Ranges of values, unique values, and sets

Filtering rows of data is not the only way to summarize it. Modelers often asked what range of values an attribute could take on, and sometimes the relative frequency of those values:

> W415c: Can you show me the firing rates for the productions? Uh, not rates, but the number of times a production was used?

> W413a: What percentage of these verbs are irregular?

After seeing the result and listing the distinct verbs involved, W413a then asked for *set* operations to find values unique to one or the other set:

> W413a: Now I want to [...] subtract the irregulars from the regular. I want to do a diff between the [...] set of regular rules and the set of irregular rules *[in the trace]* and see if there's any rule that is unique to regular.

Looking for things that are not there

Abstracting away information can even be a way to directly test a hypothesis. Modelers sometimes asked for counterexamples to their hypotheses, treating an empty result as a confirmation:

> W412b: Is this rule firing when the trial is irregular […] we're looking for an empty set.

The result was indeed an empty table, but this exposed an interesting problem with such queries: its lack of data left no context to verify that the query had run correctly, confusing both experimenter and participant. A related problem also appeared when modelers asked to list distinct attribute values and their counts: in some situations modelers expected them to be listed with a count of zero, but our experimental tool omitted such values.

Implications for summarizing

(I-FILTER): Modelers needed to be able to filter data in flexible and task-specific ways, without having to rerun the program. *(I-DISTINCT)*: Modelers needed to find value ranges and list distinct values. They often applied these to filtered lists. They sometimes needed set operations. *(I-ZEROES)*: Counts of distinct items in filtered lists should include zero counts for items that did not pass through the filters, rather than simply omitting them. This requires interoperation between "distinct" and filtering features.

B3/B4: ABSTRACT SYNTAX AND ITS VALIDATION

Drawing from the implications in the last two sections, in step B3 we created a language specification in the form of an abstract syntax to represent modelers' queries. In this section we describe and validate it for *soundness*, *coverage*, and *viscosity* relative to *our participants'* data in Study W.

From Implications for Design to Abstract Syntax

Our abstract syntax specifies a "natural" deep structure for the final concrete language, which we will eventually design in Step C in accordance with these specifications. (Recall Figure 1.) We left the syntax of this language specification *abstract* in order to avoid conflating modelers' needs for model information with their need for help with query syntax—an important but orthogonal issue.

An overview, by example

Our abstract syntax defines a *query* as a function that takes a program trace and returns a table that represents all the situations in the trace that matched the query. These "tables" are actually data structures that could be used as the basis for a variety of visualizations, although we chose to depict them simply as tables in Study W.

For example, this query:

```
Rules_fired filter (name="PRODUCTION70")
        next Buffer_goal
```

begins with the term Rules_fired, which is itself a subquery. It returns all the rule-firing events, their times (on a simulation clock), and their attributes. Study W's implementation of this was simply a table stored within a database representing the trace. (Study W had 37 such tables stored in the trace database.)

The *filter* and *next* keywords in the query are *operators*. (Recall them from Table 2 in the results section.) Here, *filter* transforms Rules_fired into a new query that returns only the rules that fired named "PRODUCTION70". *Next* then adds information to each row about the next change to the "goal" buffer—but only if that change happens before the next time PRODUCTION70 fires. *Next* does so by merging corresponding rows from two result tables (*filter*'s output table, and Buffer_goal), based on constraints on their timestamps. Table 3 shows formal definitions of a sample of common operators, including *filter* and *next*, along with study results' implications they satisfy.

Validation

For purposes of analysis, we divided Study W transcripts into 149 episodes representing participant queries and the

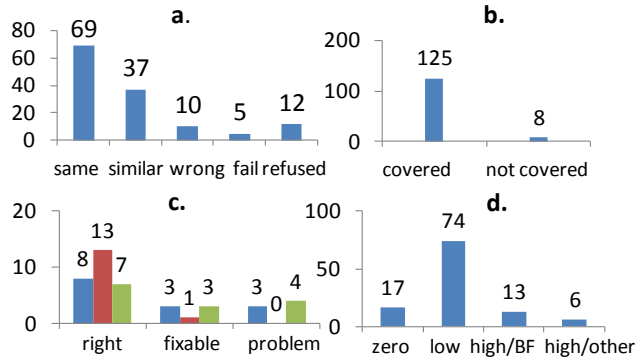

Figure 4: Validation against Study W: (a) Recoded queries vs wizard's live queries. "Wrong" and "Failed" were experimenter errors (b) Episodes covered by the abstract syntax (c) Three panel members' ratings for 14 sample episodes. 83% were "right" or "fixable". (d) Move depth: 91 moves (83%) were low or no viscosity. BF="buried filters" (see text)

eventual satisfaction of that query by the experimenter. A single "episode" began when the participant asked the wizard to produce some output, and continued, sometimes with several query attempts, until both parties were satisfied that the output an adequate representation of the participant's request. Thus, each episode had either a final output produced in the form of a table or visualization, or none when a query could not be satisfied.

We coded each episode twice: once as an "as requested" code and once as an "as provided" code. The "as provided" code was an objective, direct translation of the query string the experimenter typed during the study (in Study W's query language) to the final abstract syntax. The "as requested" code is a subjective coding—but using the same abstract syntax—of what we in retrospect believed the participant actually requested.

Validation of Coverage

Although it is not possible to validate coverage of the language for the *universe* of modelers, we could objectively validate it for our Study W participant data: our abstract syntax was able to represent 125 (94%) of the 133 usable and non-repetitive episodes from Study W (Figure 4b). Of the remaining eight episodes, three were vague or logically incoherent, four required extra complexity but had easier substitutes (for example a "set difference" operation on two small groups of items), and one would have required a special operator that we doubted would be widely used (a co-occurrence matrix).

Validation of Soundness

To validate the soundness of the "as-requested" recoding, we asked a panel of experienced modelers (drawn, with some overlap, from the same population as Studies N and W) to review the *output* that as-requested codes *would have* generated, for a random selection of anonymized episodes, and asked them to find any mistakes in our post-experiment analysis of what the participants had actually requested.

We gave the panel a random sample of 14 episodes out of the 125 episodes that our abstract syntax aims to cover (see the "coverage" subsection above). For context we also gave them relevant transcript segments and prior screenshots to establish context, and a summary sheet giving statistics about the model run. In each case, the panelists were asked to indicate whether the query had been carried out correctly per the participant's wishes.

The panelists were given five options, and a free text area to explain their answer. The five options were "right", "fixable" (only rearrangement or simple arithmetic would be needed to fix it), "some missing", "right assuming…" (panelist did not have enough information about the model to be sure), and "wrong".

After checking modelers' assumptions in the "right, assuming" category and changing the code based on the assumption when possible, on average the panelists rated 11.6 (83%) of the 14 queries as either "right" or "fixable", as shown in Figure 4c, and all but one episode was rated as "right" or "fixable" by at least 2 panelists.

We also compared the "as-requested" codes directly to the "as-provided" codings, which participants helped refine during the study. As Figure 4a shows, in 106 (88%) of the 121 non-refused episodes (12 were requests for rule text), the two codings were substantially the same, in the sense that the as-provided query produced at least enough information that a modeler could in principle use it to produce the as-requested query's results by rearranging data or doing simple arithmetic. 10 episodes (7%) were clear experimenter errors, and in 5 (4%), the experimenter could not produce a response.

Validation of Low Viscosity

In Study W, modelers often evolved their queries incrementally rather than invent them from whole cloth. We would like to avoid the situation where a modeler adds and adds to a query, then wants to change an earlier decision, and has to undo all those layers to make the change; in other words, we want low *viscosity* [7].

To measure viscosity, we tightened our focus to *moves* rather than episodes. In the realm of strategy literature, Bates [2], defined *moves* as "an identifiable thought or action that is part of information searching". Thus, in this study, we defined each *move* to be a single addition, deletion, or

Operator: Implication ID (see Results sections) Definition
filter: (I-FILTER) Given a query and some criterion, returns a more limited variant of the original query that matches the criterion. (See text for example) Given: query **Q** returning a set of n tuples of k tagged attributes, $\{ (a_1{:}Q.v_{j-1},\ldots a_k{:}Q.v_{jk}) \mid j=1..n\}$, criterion $\mathbf{a_s}$ the s^{th} attribute tag of Q **filter (Q,a_s=v_s)** returns: $\{ (a_1{:}\ Q.v_{j1}, \ldots a_k{:}\ Q.v_{jk}) \mid j=1..n$ such that $Q.v_{js} = \mathbf{v_s}\}$.
next/prior/simul: (I-TIME) Pair up two event queries into a single query returning data about *pairs* of events, one after the other. (See text for example) Given: event queries **E** and **F** returning $\{(a_1{:}\ E.v_j,$ time: $E.t_j) \mid j=1..n_E\}$ and $\{(a_1{:}\ F.v_k,$ time: $F.t_k) \mid k=1..n_F\}$ **next(E,F)** returns: $\{($time:$E.t_j$, $a_{1E}{:}E.v_j$, $time_E{:}E.t_j$, $a_{1F}{:}v_k$, $time_F{:}F.t_k) \mid E.t_j<F.t_k \le E.t_{j+1}$ and $F.t_{k-1} \le E.t_j$ when $k>1\}$.
distinct: (I-DISTINCT) Produces a list of the distinct values that some attribute of a query result took on, and a count of each value's occurrences. Given: query **Q,** whose attributes include $\mathbf{a_{1\ldots k}}$ then **distinct(P,$\mathbf{a_{1..k}}$)** returns: $\{ (a_{1..k}{:}Q.v_{1\ldots k}, count_Q{:}q) \mid$ for each distinct tuple of values $Q.v_{1\ldots k}$ that co-occur together, exactly q times. $\}$
segment: (I-TRIAL) Breaks a trace into subtraces (usually, trials) using an event query to specify the boundaries between each subtrace (trial)" Given: event query **E** returning $\{(a_1{:}E.v_k,$ time:$E.t_k) \mid k=1..n\}$ **segment(E)** returns: $\{(seg_E{:}k,$ time:$E.t_k,$ endtime:$E.t_{k+1}) \mid k=0..n\}$ where $t_0=0$ and t_{k+1} is the last timestamp in the trace.

Table 3: A sample of the most common query operators. The set notation shows the table that the query returns when applied to a trace. $a_k{:}v_k$ refers to a value v_k in a column titled a_k in the query's output table

change of operators in the formal codings, and we dissected each pair of adjacent episodes into atomic moves necessary to explain the difference between them.

Of the 149 episodes, 55 were not analyzable as moves from the previous query: either one or the other had no coding, or the requests had so little in common that it seemed unlikely a modeler would want to transform one to the other in this way. Of the remaining 94 episode pairs, we decomposed 6 into 3 moves, 19 into 2 moves, and 66 were single moves, for a total of 110 moves.

We operationalized viscosity by classifying moves as *shallow*, low-viscosity moves, when only the outermost layer of the abstract syntax tree was modified; and *deep*, high-viscosity moves otherwise. As shown in Figure 4d, 91 of these 110 moves (83%) were shallow. In fact 17 of those (15%) required no changes at all. 19 were deep moves. Our intention is that language designer tasked with building a usable, fluid debugging interaction language could rely on the abstract syntax to drive the affordances offered: e.g., menu options to add, remove, or change the "outermost" layers of the query could map to the most common ways Study W participants sequenced their queries. Our 83% viscosity score, while not ideal, seems reasonable for at least providing a good basis for such interaction design.

Although we did not find an elegant abstract syntax that could improve viscosity further, an analysis of the 19 deep queries reveals that 13 of them fell into an information-seeking strategy in which modelers repeatedly modified filters *underlying* distinct or detail operators to see how the query results changed. Because of this strategy, and other interactions between these operators (see I-ZEROES and I-DISTINCT above), some of the viscosity could be further reduced in the user interface design by providing affordances to directly manipulate this class of "buried" filters.

DISCUSSION

We expect our debugging language to be most useful for reactive systems like ACT-R, RML, and EPIC, in which behavior is tightly (and often, probabilistically) driven by the timing and content of environmental inputs. In such systems, modelers' analysis of runtime behavior in concert with environmental conditions is critical. We designed our language to be relatively modeling-language agnostic, and are now starting a field study with a prototype debugger for RML [5] based on the language. (RML is a cognitive modeling language being developed at the Air Force.)

A methodological question we wrestled with was how to validate replicability of the complex coding of Study W episodes. It seemed unlikely that an inter-rater reliability scheme would work: two researchers would not likely use the abstract syntax in precisely the same way to represent modelers' informal programs. Our choice was to validate soundness, coverage, and viscosity instead of replicability.

In surveying other researchers' work, we found the question of validating replicability in NP studies to be a common problem. As Table 4 shows, NP researchers have been solving this by validating other properties of their coding schemes. Ko et al., for example [10], validated the *distribution* of codes by conducting a survey of expert programmers to check the work a single coder had done. Pane et al. [16] also validated distribution by averaging ratings from a small panel of domain experts (experienced programmers) tasked with assessing aspects of the language and structure in children's handwritten solutions to programming problems. Little et al. [12] checked *usability* and *ease of production* with a summative validation of whether users with little training could produce Chickenfoot queries and accomplish tasks with the tool. One contribution of this paper, then, is the identification of the *choices of language properties* that different NP validation methods can evaluate.

NP+ may be useful beyond our particular case, and we hope to use it in other design projects in the future. We envision it as being particularly appropriate for language designers uncomfortable making the leap from NP's Step B, understanding the target audience, to Step C, designing the new system. Such a leap requires language designers to have a level of user-centered design experience and a comfort with the target domain that may not always be practical. For example, we envision NP+ as an aid to programming language specialists who know how to build an abstract syntax, but who are looking for some user-centered basis for making technical choices. For our purposes, and perhaps for other researchers, the path from Step B's "implications for design" to Step C's concrete language design seemed to rely on too much "magic" to translate into correct and fluid designs with broad coverage, and Step D seemed too long to wait to spot this kind of problem. However, we have not validated the methodology beyond this initial case study, and future research is needed to evaluate its generality.

Language/Tool	Validation method	Property validated
Whyline for Alice [9]	Triangulation; subjective inter-rater coding comparison	Support for cognitive breakdown theory
Contributed to Whyline for Java [10]	Survey of domain experts	Relative importance of information seeking goals
Chickenfoot for end-user web scripting [12]	Usability study	Usability and ease of production
Hands language for children [16]	Experts working independently, ratings averaged together	Reliability of researcher classification
This paper (debugging for cognitive modelers)	Compare codings, count codings, expert panel, depth-check moves	Coverage, soundness relative to dataset, and viscosity.

Table 4: Natural Programming practitioners have validated a variety of properties, using a variety of methods. This paper is at the bottom of the table.

CONCLUSION

In this paper we empirically investigated cognitive modelers debugging to derive a design specification for a debugging interaction language for them. Some interesting insights about this population's debugging needs were:

- Modelers needed to refer to states by their endpoints, or events that occur during the state.
- Modelers preferred to "anchor" temporal relationship queries starting from a known set of events.
- They needed results that integrated source code (original as well as learned rules) and runtime data together into the same query results.

We then validated our design specification in several ways, showing that, in the context of our collected data, our design was reasonably *complete* and *sound*, and was a *low-viscosity* representation of the query evolution paths that modelers followed.

Finally, we used our investigation as an initial case study in NP+. Replacing the leap of design expertise in the NP process with precise, explicit steps, NP+ helped us ground our language design in empirical evidence and validate along the way. Although some HCI researchers are comfortable moving directly from formative empirical results to a language design, for us the more explicit roadmap of NP+ helped us incrementally monitor our progress towards meeting the needs of our users.

ACKNOWLEDGMENTS

We thank our participants for their help, and AFOSR for support under FA9550-10-1-0326 and FA9550-09-1-0213.

REFERENCES

1. Anderson, J., Boyle, C. Corbett, A., and Lewis, M. Cognitive modeling and intelligent tutoring, In *Artificial Intelligence and Learning Environments,* W. Clancey and E. Soloway, (Eds.). MIT Press (1990), 7-49.
2. Bates, M. Where should the person stop and the information search interface start? *Information Processing and Management,* 26:5, (1990), 575-591.
3. Bogart, C., Burnett, M., Douglass, S., Piorkowski, D., Shinsel, A. Does my model work? Evaluation abstractions of cognitive modelers. *Proc. VLHCC*, IEEE (2010), 49-56.
4. Bothell, D. et al., *ACT-R Tutorial*. Distributed with ACT-R 6.0 version 1.3 r766. http://act-r.psy.cmu.edu/actr6/. Retrieved August, 2009.
5. Douglass, S., Mittal, S., Using domain specific languages to improve scale and integration of cognitive models, *Behavior Representation in Modeling and Simulation,* (2011).
6. Fu, W. and Pirolli, P. SNIF-ACT: A cognitive model of user navigation on the world wide web. *Human-Computer Interaction*, 22:4 (2007), 355-412
7. Green, T. and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7:2 (1996), 131-174.
8. John, B. and Kieras, D. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM TOCHI* 3:4 (1996), 320-351.
9. Ko, A., Myers, B. A framework and methodology for studying the causes of software errors in programming systems. *J. Vis. Langs. & Computing*, 16 (2005), 41-84.
10. Ko, A., DeLine, R. and Venolia, G. Information needs in collocated software development teams. *Proc. ICSE*, ACM (2007) 344–353.
11. Ko A., Myers, B. Finding causes of program output with the Java Whyline. *Proc. CHI*, ACM (2009), 1569-1578.
12. Little, G., Miller, R., Chou, V., Bernstein, M., Lau, T. and Cypher, A., Sloppy programming, In *No Code Required*, Cypher, A., Dontcheva, M., Lau, T. and Nichols, J. (Eds.). Morgan Kaufmann (2010), 289-307.
13. Myers, B., Weitzman, D., Ko, A., Chau, D. Answering why and why not questions in user interfaces. *Proc. CHI*, ACM (2006), 397-406.
14. Neumann, C., Metoyer, R. and Burnett, M. End-user strategy programming. *J. Visual Languages and Computing*, 20:1 (2009), 16-29.
15. Newell, A. and Card. S. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1:3 (1985), 209-242.
16. Pane, J., Ratanamahatana, C. and Myers, B., "Studying the language and structure in non-programmers' solutions to programming problems," *Intl. J. Human-Computer Studies*, 54:2 (2001) 237–264.
17. Pane, J., Myers, B., More Natural Programming Languages and Environments," In *End User Development, vol. 9 of the Human-Computer Interaction Series*, H. Lieberman, F. Paterno, V. Wulf, (Eds.). Springer (2006).
18. Pirolli, P., Card, S. Information foraging in information access environments. *Proc CHI*, ACM (1995), 51-58.
19. Segal, J. Some problems of professional end user developers. *Proc. VLHCC*, IEEE (2007), 111-118.
20. Tor, K., Ritter, F., Haynes, S. and Cohen, M., CaDaDis: A tool for displaying the behavior of cognitive models and agents. *Proc. Conf. on Behavior Representation in Modeling and Simulation*, (2004), 192–200.
21. Wong, J. and Hong, J. Making mashups with Marmite: Re-purposing web content through end-user programming. *Proc. CHI*, ACM (2007), 1435-1444.
22. Yaremko, R., Harari, H., Harrison, R., Lynn, E. *Reference Handbook of Research and Statistical Methods in Psychology for Students and Professionals*. Harper and Row (1982), New York.