Does my model work? Evaluation abstractions of cognitive modelers

Christopher Bogart[1], Margaret Burnett[1], Scott Douglass[2], David Piorkowski[1], Amber Shinsel[1]
[1]*Oregon State University* and [2]*Air Force Research Laboratories*
{bogart, burnett, piorkoda, shinsela}@eecs.oregonstate.edu
scott.douglass@mesa.afmc.af.mil

Abstract

Are the abstractions that scientific modelers use to build their models in a modeling language the same abstractions they use to evaluate the correctness of their models? The extent to which such differences exist seems likely to correspond to additional effort of modelers in determining whether their models work as intended. In this paper, we therefore investigate the distinction between "programming abstractions" and "evaluation abstractions". As the basis of our investigation, we conducted a case study on cognitive modeling. We report modelers' evaluation abstractions, and the lengths they went to in evaluating their models. From these results, we derive design implications for several categories of persistent, first-class evaluation abstractions in future debugging tools for modelers.

1. Introduction

Solving a problem can be a very different task from evaluating the correctness of your solution, potentially requiring a different set of abstractions. However, program comprehension and debugging tools are often built with the implicit assumption that the abstractions programmers use to evaluate a program (*evaluation abstractions*) are the same as the abstractions they use to create a program (*programming abstractions*).

The central question we investigate in this paper is whether distinct evaluation abstractions are an important aspect of programmers' evaluation and debugging practices. We are particularly interested in this issue for programmers who do not already have evaluation tools (test suite management tools, etc.) that they perceive to be well-suited to their needs. As Segal shows, one such population is scientific modelers [20].

The possibility of modelers building and using evaluation abstractions that are different from their programming abstractions raises several issues. If there are important evaluation abstractions that do not match a model's programming abstractions, what are they? What do modelers currently have to do to construct, use or reuse such evaluation abstractions? Are there implications for evaluation-time tools on how to support such abstractions?

In this paper, we shed some light on these issues through a case study on six cognitive modeling projects. We spent a month listening to a group of cognitive modelers at the Air Force Research Labs in Mesa, Arizona as they debugged their models and discussed them with their colleagues. Our goal was to gather cross-cutting commonalities in the ways these cognitive modeling cases work with abstractions in evaluating their models.

2. Cognitive Modelers' World

In this paper, we use the term *modeler* to refer to anyone who builds a computational model to simulate and understand phenomena in the world. One example is cognitive modelers modeling cognitive activities in the human brain.

The cognitive modelers in our study used ACT-R [2]. This modeling language is a particularly appropriate platform for studying the differences between evaluation and programming abstractions, because ACT-R models are (even) more unpredictable than traditional imperative programs.

Unpredictability is useful for investigating evaluation abstractions because an unpredictable system has a large gulf of evaluation [17]—there is a large distance between telling the system what to do and determining the correctness of its response. For example, modelers often do not force production rules to fire in a particular order, but instead attempt to set rule preconditions such that they will become available at appropriate times in a task flow. Verifying that this in fact happened is a non-trivial subtask for modelers. The difficulties modelers have arising from such unpredictability provides a useful magnification. This is because, as Ljungblad and Holmquist point out, studying the practices of marginal communities can give insights into effects that still apply, but are harder to spot, in a more general population [14].

Modeling in ACT-R features unpredictability in two ways. First, model behavior critically depends on the firing of *production rules*, and the storage and retrieval of data structures called *chunks*. The selection and timing of both of these ACT-R entities are governed by calculations involving many factors, and the results are often difficult to predict. Second, because the human cognition being modeled is flexible and adaptive, cognitive modelers often write models whose decision-

978-0-7695-4206-5/10 $26.00 © 2010 IEEE
DOI 10.1109/VLHCC.2010.16

making is highly reactive to the environment, rather than writing models to carry out fixed plans.

3. Related Work

Evaluation abstractions relate to expectations about what programs will do. Other researchers have considered the idea of eliciting information about users' expectations for use in debugging. WYSIWYT [4] for spreadsheets, Woodstein [23] for web transactions, and Declarative Debugging [16] for Prolog support Boolean expectations. That is, they let users flag values as right or wrong. (These systems use backward slicing through a growing set of these user judgments to narrow down possible causes of program errors.) Other systems have elicited expectations about correct values (either explicitly or in the form of a "Why not?" question) but only use them for a single recommendation, then discard them: these include the Whyline [12] for Java, the ACT-R debugger [3], GoalDebug [1] for spreadsheets, and Kulesza et al.'s technique [13] for end-user debugging of Bayesian classifiers.

Beyond Boolean judgments, some systems allow more elaborate expectations. Some programming IDEs such as Eclipse and Visual Studio allow users to enter arbitrary expressions in a "watch window" to display calculated values while debugging. For example a user debugging a program involving heights and weights could monitor body mass index (weight/height^2), even if the source code contained no such calculation.

More elaborate evaluation abstractions are those used for runtime verification, in which complex statements of temporal logic are continuously checked against the state of a running program. Colin [5] gives examples of runtime verification used to monitor the geometric shape of groups of autonomous flying robots and the properties of routing protocols. This specialized technique is not yet widely used; Colin suggests this is because runtime verifiers slow programs down, but usability may also be a factor.

In the domain of modeling, Zeigler [26] defines an *experimental frame* as the set of manipulations and measurements that a modeler chooses to adopt as a standard of validity. He distinguishes *replicative validity*, in which a model mimics a real system, *predictive validity*, in which models match new data they were not specifically adjusted for, and *structural validity*, in which models internal parts match the parts of the real-world system. The modelers we observed were primarily concerned with *replicative validity* and to some extent with *predictive validity*.

For cognitive modeling specifically, there is ongoing research into abstractions for modelers, but it is specifically directed at creating new languages for cognitive modelers [18], not at ways of evaluating and debugging them. These include HLSR [9], a high-level cognitive modeling language; Hank [15], a GUI interface for the SOAR cognitive modeling language; G2A [19] and HTAmap [7], both of which translate high level task descriptions into ACT-R; and CogTool [8], an ACT-R-based visual language for simulating user interface interaction. Finally, SimTrA [7] creates summary statistics of eye tracking data from cognitive models and outputs them into convenient tables in R.

In contrast to these works, we do not present a new tool or language, but rather aim to harvest abstractions and expectations directly from the intended population in order to make design recommendations for better debugging and program comprehension tools.

4. Case Study Design and Methodology

Our investigation method was the case study, the method of choice for investigating a contemporary set of events over which the investigator has little or no control [24]. Our study included six cases, each of which was a modeling project. Participants were six cognitive modelers working on these projects, with advanced degrees in Psychology, Computer Science, or Cognitive Science. These participants were civilian scientists with the Air Force Research Laboratories. We studied these modelers over the course of a month.

The elements of interest were evaluation abstractions. *Evaluation abstractions* are judgments, intentions, or beliefs about model behavior that, like other kinds of abstractions, ignore or hide details, usually to capture some kind of commonality among different instances. Given this definition, our research questions were:

RQ1: What kinds of evaluation abstractions do modelers have?

RQ2: How do modelers currently create, use, and reuse their evaluation abstractions?

RQ3: What operations do modelers need to be able to perform on evaluation abstractions?

4.1 The Models and Modelers

The first four cases were the following projects (anonymized here):

VISLANG was Steve's doctoral thesis work to demonstrate the impact of visual scenes on language comprehension. It models eye movements when a participant listens to a description of an airplane's location while looking for the plane on the screen. VISLANG's source code contained about 64 production rules. It can learn more production rules over the course of a run.

Gary was in charge of PILOT, a large component of a project to build a cognitive model that simulates flying an Unmanned Aerial Vehicle (UAV). Gary's focus at the time of the study was on the question of how

PILOT should determine when to check the dashboard controls as it flew the plane. PILOT had about 160 production rules and 30 chunk types.

John and Ellen were linguists working on LANGCOMP, a language comprehension model for a UAV pilot. The model interpreted incoming text chat from human teammates, and updated the model's understanding of what destination, airspeed, and altitude the teammates were requesting. LANGCOMP had about 540 rules and 360 chunk types.

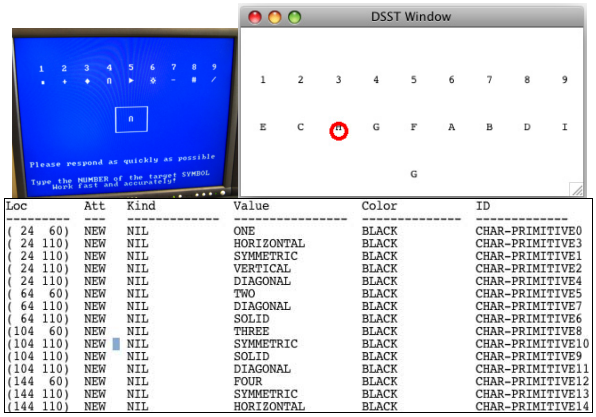
The SCANTYPE model (Figure 1) had just been handed from Mitch to Matt. It modeled humans performing a simple task: given a symbol, search for it on a screen, then press the right key on a keyboard. The model had alternate strategies for scanning and typing, and learned to use the more efficient strategies over time. SCANTYPE had 19 rules at the beginning of the study, and by the end, Matt had added 6 more. It had four chunk types.

We added two cases that were exercises from the ACT-R tutorials [3]: ZBRODOFF and SIEGLER. These cases served as sources of normative modeling expectations because they each contained a set of stated expectations to guide new modelers into building a new model or enhance an existing model. SIEGLER predicted the distribution of answers 4-year-olds made [22] when asked to add small integers. ZBRODOFF modeled a "letter addition" experiment [25]. For example, given "A+4" it should respond with "E", which is four letters later in the alphabet.

4.2 Data and Coding Procedure

The data about the models were model source code, model runs, model output, and model visualizations. The data about the modelers were recordings, notes, and transcripts from two presentations by modelers describing their work to other cognitive modelers in the group; from three working group meetings; from three interviews; and from three one-on-one job shadowing sessions with modelers in the style of [11].

Using these data, two researchers working together coded transcript samples from each of the projects into the evaluation abstractions shown in the next section's tables. For modeling projects, we coded the first ten minutes of each transcript, starting where the modeler began concretely discussing a model or behavior. For the tutorials, we coded about 200 lines from the "problem" section of the lesson where a model was described with the reader asked to modify it in some way.



Loc	Att	Kind	Value	Color	ID
(24 60)	NEW	NIL	ONE	BLACK	CHAR-PRIMITIVE0
(24 110)	NEW	NIL	HORIZONTAL	BLACK	CHAR-PRIMITIVE3
(24 110)	NEW	NIL	SYMMETRIC	BLACK	CHAR-PRIMITIVE1
(24 110)	NEW	NIL	VERTICAL	BLACK	CHAR-PRIMITIVE2
(24 110)	NEW	NIL	DIAGONAL	BLACK	CHAR-PRIMITIVE4
(64 60)	NEW	NIL	TWO	BLACK	CHAR-PRIMITIVE5
(64 110)	NEW	NIL	DIAGONAL	BLACK	CHAR-PRIMITIVE7
(64 110)	NEW	NIL	SOLID	BLACK	CHAR-PRIMITIVE6
(104 60)	NEW	NIL	THREE	BLACK	CHAR-PRIMITIVE8
(104 110)	NEW	NIL	SYMMETRIC	BLACK	CHAR-PRIMITIVE10
(104 110)	NEW	NIL	SOLID	BLACK	CHAR-PRIMITIVE9
(104 110)	NEW	NIL	DIAGONAL	BLACK	CHAR-PRIMITIVE11
(144 60)	NEW	NIL	FOUR	BLACK	CHAR-PRIMITIVE12
(144 110)	NEW	NIL	SYMMETRIC	BLACK	CHAR-PRIMITIVE13
(144 110)	NEW	NIL	HORIZONTAL	BLACK	CHAR-PRIMITIVE14

Figure 1: A SCANTYPE task screen as the participants saw it in the original experiment, as the model sees it (the red circle is where ACT-R is attending), and as ACT-R's visual location buffer sees it.

5. Results

Our first research question was to identify and categorize the different types of evaluation abstractions in the different modeling projects. We categorized them as *Data Structure Abstractions*, describing relationships among data, *Time Abstractions* describing the sequencing, choosing, and grouping of events over time, and *Statistical Abstractions* with descriptive statistics about model behaviors.

As Figure 2 shows, modelers used all of these abstraction categories in all projects, although the mix of categories varied from project to project. The figure also shows patterns of co-occurrence both within and across the categories.

5.1 Data Structure Evaluation Abstractions

As we pointed out in the introduction, existing pro-

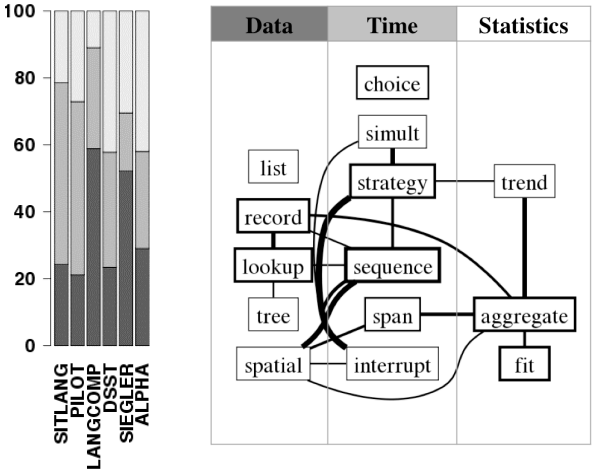


Figure 2: (Left:) Percentage of evaluation abstractions in projects' transcripts. Dark=Data; medium=Time; light=Statistical. (Right:) Co-occurrence of evaluation abstractions within and across categories. Nodes with thick borders occurred most frequently, and edge thickness indicates co-occurrence frequency. Low co-occurrences are not shown.

gramming environments are built on the assumption that programmers evaluate their programs using their program's data structures. Our modelers did take advantage of these, e.g., using the debugger to explore the chunk data structures that existed in their models. However, the difference between the data structures in the model and the five data structures (Table 1) modelers needed to evaluate was large, requiring translation.

5.1.1 The types of evaluation data structures

Table 1 shows the five types of data structure evaluation abstractions we found, and Figure 3 shows their frequency. We coded *record*, *lookup* (like a hash map or lookup table), *list*, and *tree* when the structures (as described by modelers verbally) suggested resemblance to traditional programming data structures of these names, and *spatial* when modelers related data to locations in visual space.

Spatial evaluation abstractions were particularly interesting because they cut across programming abstraction boundaries, relating things to each other geometrically in visual space. Screen regions, goals, remembered chunks of knowledge, and even production rules all potentially related to regions of the visual space. Figure 1 shows, at top right, a screen that was shown to SCANTYPE, and at bottom, how SCANTYPE perceived it as lists of items with coordinates. However, when modelers copy those numbers into other locations, the debugger does not know the numbers are meant by the modeler as coordinate pairs, so modelers can view them only as numbers. Thus, as in other languages, if modelers want to know how items relate spatially, they must do the work to graph them.

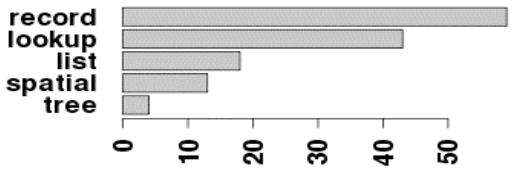


Figure 3: Counts of Data Structure codes.

Definition and *Example*
Record: Item made up of multiple parts *e.g., Gary: So there are productions that make this deduction, and stick it into the situation superchunk.*
Lookup Table: Items retrieved by key or matching content *e.g., Ellen: If you say that the third letter is too important, then that's going to mess up what is retrieved.*
List: Info structured as first, next, next, last *e.g., Matt: In the original task they're always presented 1,2,3,4,5,6,7,8,9 in the exact order every time*
Spatial maps: Information tied to visual space *e.g., Matt: In other words **this** one [pointing to the screen], [...] it would find it very quickly.*
Tree: Hierarchical Knowledge *e.g., Ellen: I have a feeling that "meet" was retrieved; it just didn't make it into the tree.*

Table 1: Data Structure Evaluation Abstractions, in order of frequency.

5.1.2 The translation problem

The modelers' work to translate from model data structures to evaluation data structures was hard, but the mismatch leading to the translation is necessary.

First, consider the work to do such translations. For example, John wanted to know why one word in LANGCOMP's large lexicon had been retrieved instead of another. In the model, each word in the lexicon was stored as an ACT-R chunk. But John treated this mass of chunks and the properties of ACT-R's chunk retrieval system as a *lookup* table, in which the choice of chunk to retrieve depended on the contents and computed "activation values" of all the chunks that were candidates for retrieval.

Since John did not have evaluation support for this table-like *lookup* perspective on the data, his recourse, if he had decided to pursue answering his question, would have been to scroll though a long list of chunks by name, and click on each individually to view and compare their activation levels.

The mismatch generating such translation work is a *necessary* abstraction mismatch. Because the goal of cognitive modeling is to model in terms of cognitive theory, evaluation data structures cannot be programming abstractions *inside* the model unless some cognitive theory proposes them. Instead, these data structures can exist only in tools outside the model.

5.1.3 Abstractions of Abstractions

The examples discussed so far each examined a single kind of evaluation abstraction in isolation, but as Figure 2 shows, these abstractions were sometimes compounded together into more elaborate structures.

For example, in explaining SCANTYPE's behavior, Matt identified a visual attention shift by composing a *spatial* comparison (between the model's gaze and a landmark he pointed to on the screen), with a *time sequence* abstraction (three events in sequence: a shift, an arrival, and a read; explained in Section 5.2):

> *Matt: OK, now it's gonna attend a probe, ... it's gonna shift visual attention there, its visual attention arrives, we're gonna read it.*

These compound abstractions took more work for modelers to evaluate because they sometimes required extra navigation among different logs and visualizations. For example whenever the SCANTYPE model "saw" a symbol, it logged the creation of a chunk with a name like `VISUAL-OBJECT1` (in the second `VISION` line in Figure 4, for example). The trace

```
0.216 PROCEDURAL    CLEAR-BUFFER VISUAL-LOCATION
0.216 PROCEDURAL    CLEAR-BUFFER VISUAL
0.216 PROCEDURAL    CONFLICT-RESOLUTION
0.290 VISION        Encoding-complete CHAR-PRIMITIVE2-0-0 NIL
0.290 VISION        SET-BUFFER-CHUNK VISUAL VISUAL-OBJECT1
0.290 PROCEDURAL    CONFLICT-RESOLUTION
0.359 IMAGINAL      SET-BUFFER-CHUNK IMAGINAL PAIR0
0.359 PROCEDURAL    CONFLICT-RESOLUTION
0.395 PROCEDURAL    PRODUCTION-FIRED ENCODE-INCORRECT-SYMBOL...
```

Figure 4: Part of the event trace from a run of the SCANTYPE model. Columns indicate the simulation clock time, the module responsible for the event, and a description of the event.

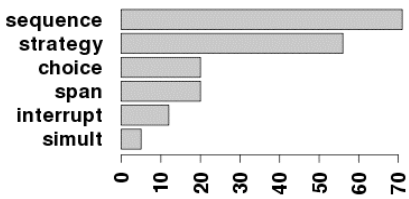


Figure 5: Time evaluation abstractions.

shows *when* this object was created, but to find out *where* it was, Matt would have had to run the debugger, tell it to skip forward to the appropriate time stamp, and open a chunk listing to see the coordinates of this object.

5.2 Time Evaluation Abstractions

Time clearly mattered to our modelers when they evaluated their models. Recall from Figure 2 that all six projects used time evaluation abstractions. Time constraints were not explicit in any of the models' source code; instead, modelers used time abstractions to check high-level patterns as *emergent* behavior. Gary explained why he did not program time explicitly into his model during a Q&A after his talk:

> *John:… you can have a declarative memory chunk that's actually a sequence of goals that allows you to prefer—*
>
> *Gary (interrupts): Yeah, but that's the type of thing I want the model to* ***learn*** *though, this sequence of goals; I don't want to build that in.*

Definition and *Example*
Sequence: B will occur after A *e.g., Matt: It starts at the far left, it shifts attention to [the] square, [to the] plus, to the three, to the U; just left to right serially, until it finds the one it's looking for.*
Strategy: Joint activity of related rules *e.g., ZBRODOFF: subjects have to engage in counting.*
Choice: Either A or B will happen *e.g., John: I have a dual-path capability. I can either retrieve this thing from memory […], assuming I've already created one and I can just retrieve it. Or I can create it.*
Span: Time interval *e.g., Steve: And then after a short interval there's an indication of the correct or actually described reference*
Interruption: Stopping or pausing a strategy *e.g., Gary: You can build very generic productions. Things like interruption productions. So if there is a task goal, then change goals, and so you could be in the middle of a goal and this thing could fire, and you'd cut out in the middle of the goal you're working on and you're starting something new.*
Simultaneous: Interleaved strategies *e.g., Steve: I could have the two separate threads in the model, and then basically the contest for resources would take care of all of the interleaving.*

Table 2: Time Evaluation Abstractions in order of frequency

5.2.1 Manually sifting the sands of time

When evaluating even simple sequences of events, the time abstractions of interest to our modelers were often buried in the logs and visualizations, so modelers had to do manual pattern matching work to find them. We saw all three of Matt, John, and Ellen reading through event logs like the one in Figure 4. The traces were very long, and all three modelers used a combination of scrolling and textual search to find items of interest. Modelers sometimes lost their place, because the interesting events were not always close enough together in the log to see on the screen at the same time.

These abstract sequences of interest had *structure*: they could not be gleaned by simply filtering one concrete event type of the vast number of events that occurred in the model. Gary for example described how PILOT changed airspeed:

> *Gary: You change airspeed using this particular piece of the interface, and you hit enter when the value is at the level where you'd like it to be.*

Changing the airspeed, then hitting enter, was a short sequence of model actions that Gary expected to occur many times throughout a model run. For a tool to have helped Gary check this, it would have needed to support the notion of a sequence abstraction (defined in Table 2), so that it could find events that mattered, but only if they occurred in sequence with all intervening but unrelated details abstracted away.

5.2.2 Tracking models as they strategize

Although Figure 5 shows that the sequence abstraction was the most frequently observed, the four strategy-related abstractions (*strategy, choice, interrupt, and simult*), were even more common if considered as a group. *Strategies* were activities of groups of rules that shared a

common purpose (although the rules were not grouped within ACT-R, which simply picks one rule at a time and fires it). Some modelers described strategies as threads that were "running" when the state of the model was such that their productions would happen to be triggered. Strategies could be *interrupted* by other rules preempting them, be *simultaneous* when rule firings were interleaved, or make *choices* when one rule was selected over another. Modelers confirmed that strategies were active by checking whether the rules fired. Mitch, for example, added a statement to SCANTYPE to print "Continuing search for (feature)" every time the "encode-incorrect-symbol-quickly" rule fired, so as to gather evidence that the "quick" visual scanning strategy was running.

5.2.3 Persistence

Some evaluation abstractions were so important that modelers formalized and kept them as part of their projects, in the form of tools or documentation. For example, Gary had a particularly formal way of describing such sequences. He told us that he originally designed PILOT using a formal task description language, NGOMSL [10]. Unfortunately, Gary's NGOMSL description existed only as documentation; the only way to check that it was being followed by PILOT was very detailed inspection of numerous model runs.

Steve devised an elaborate solution to the problem of evaluating high-level sequence patterns. His model generated x,y coordinates of eye movements at exact points in time, but Steve wanted to know about certain overall patterns of movement, such as looking at or near a particular region of interest, then looking away. So he created a custom visual finite state automaton language for recognizing sequences of eye movements, which he could apply to eye tracking data in his model.

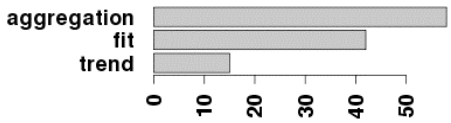


Figure 6: Counts of Statistics Evaluation Abstractions

Definition and *Example*
Aggregate comparison: Maxima, averages, deviations, count, frequency *Matt: So it has to search through on average half the symbols.* *Matt: So 12 productions are going to fire before you can find some reward.*
Fit/Validation: Comparison with human data *Gary: The number of clicks is almost identical between average human behavior and average model behavior.*
Trend: Change over time *Steve: The only difference is that they're starting to respond more rapidly.*

Table 3: Statistical Evaluation Abstractions in order of frequency

Gary and Steve went to considerable effort to construct these persistent, formal artifacts. This suggests that evaluation abstractions exist not just as ad hoc evaluations, but may be something modelers want to maintain and reuse over multiple runs.

5.3 Statistical Evaluation Abstractions

Perhaps the most distinct from traditional programming abstractions were the statistical evaluation abstractions. These were ways of evaluating model performance in terms of *aggregation*, *trend*, or *fit* to human data (Table 3 and Figure 6). Coded transcripts for all except one of the projects were at least 20% about statistics.

Unlike the other abstractions, in which modelers were able to use existing outputs to perform their evaluations (even if doing so this way was often very inefficient), evaluating in terms of statistical abstractions *required* the modelers to write and turn to other software. Specifically, they had to write Lisp code to collect numeric data and either process it in Lisp or export it to external files to process with other software.

For example Matt talked about how the SCANTYPE model worked in terms of *trends*:

> *Matt: I think [Mitch]'s hypothesis was that people get more familiar with what they're searching for and how to respond with the keyboard.*

Matt was referring to code Mitch had written to count executions of quick and slow versions of each searching and keyboarding strategy, in order to graph the shifting proportions of these events over time. The decrease of one line and increase in the other was how he determined that the model successfully modeled a *trend* from one strategic choice to another over time.

Compared to the other types of evaluation abstractions, statistical evaluation abstractions seemed to exist in later phases of model development. We observed modelers discussing them in their presentations more often than when they were working directly with the models. Our interpretation is that modelers wanted to evaluate in terms of individual data structures and time behaviors before trusting their models enough to evaluate in terms of aggregates, trends, and fit.

This interpretation is supported by how models' sizes related to the use of statistical abstractions. The smallest projects (ZBRODOFF, SIEGLER, and SCANTYPE) talked about statistics most often, perhaps because their smaller size meant they simply had less data structure and sequence detail to evaluate. At the other end of the size spectrum, the largest project, LANGCOMP, talked very little about statistics. (Refer back to Figure 2 for project-by-project use of the

different abstractions.)

Statistical evaluation abstractions were built on other evaluation abstractions. For example, the caption in Figure 7 details how Steve interrelated different evaluation abstraction types in VISLANG. A count of eye movement events was an *aggregation* evaluation abstraction, and their trend line from trial to trial amounts to a *trend* abstraction of that aggregation. Figure 2 shows that statistical evaluation abstractions were also linked to the time and data abstractions *record*, *span*, *spatial*, and *strategy*.

6. Implications for Design

As our results show, modelers used numerous evaluation abstractions that were often not the same as their programming abstractions. Further, they expended hours of effort to evaluate in terms of these evaluation abstractions. The complexity and pervasive use modelers made of these abstractions suggest a need for new powerful but low-overhead scripting capabilities within debuggers.

As one example illustrating this need, in ACT-R's Lisp environment, programmers can code ad-hoc analysis functions from scratch. However, although a few of our modelers used this device, they did not all have the expertise for this, and it still left many of their evaluation needs unmet.

If a debugging tool were to support such a language, what should it enable modelers to *do*? The evaluation abstractions we observed shared a common set of operations that modelers attempted to perform on them (Table 4). These operations correspond fairly well to the kinds of operations advocated for abstractions in other settings (e.g., Shneiderman's proposals for information visualization research [21]), which suggests that modelers' evaluation maneuvers are consistent with other situations in which full-fledged support for abstractions is accepted as being desirable.

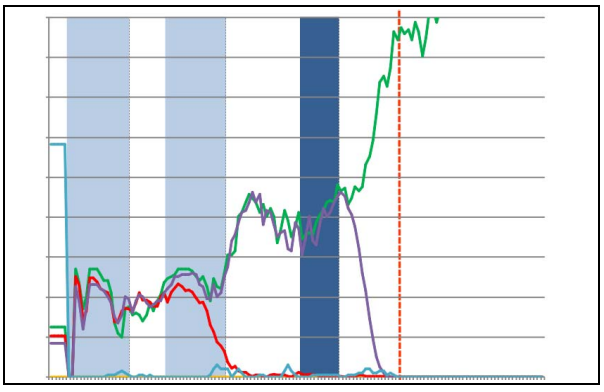


Figure 7: Steve's VISLANG graph combines evaluation abstractions of time (*sequence:* blue stripes and red dashed line are words heard and mouse clicked; *span:* width of stripes), data structure (*spatial:* colors of trend lines indicate screen regions of eye movements), and statistics (*trend:* colored trend lines; *aggregation:* vertical axis represents frequency of eye visits per region).

Compare: Modelers went to great lengths to compare evaluation abstractions, both within and between models. They did so by searching manually through traces and visualizations looking for expected patterns of events, by using "diff" tools for regression testing, and by using statistical packages to compare data for validation. Steve's automata language from Section 5.2.3 gives one possible direction for future support of comparing evaluation abstractions

Visualize and *Navigate:* Modelers created visualizations of abstractions in every presentation they gave, especially statistical abstractions. In debugging, modelers often used them to spot, and sometimes compare, phenomena that were unforeseen, too costly, or too informal to check more precisely. The modelers incurred high costs from attempting to navigate among visualizations and the abstractions themselves.

Compose and *Filter*: Modelers *composed* evaluation abstractions from combinations of other evaluation abstractions and programming abstractions. Conversely, modelers sometimes *filtered* to exclude irrelevant material. When their programming abstractions were not good matches for the modelers' desired composition and filtering, it became costly for modelers to check their expectations.

Persist: Persistence was a prerequisite of the regression testing modelers did, but modelers also repeatedly looked for the same type of information in ad hoc evaluations. The regularity with which they did so suggests that their evaluation practices were integral parts of their modeling projects.

These operations suggest a base set of functionality for designers to support when creating debugging or

Compare: Comparing evaluation abstractions within models, between models, and/or to their own expectations.
Visualize: Viewing visual patterns within and between their evaluation abstractions.
Navigate: Moving between different parts of an evaluation abstraction or between parts of different evaluation abstractions.
Compose: Composing evaluation abstractions using combinations of other evaluation abstractions and programming abstractions.
Filter: Removing irrelevant details of an evaluation abstraction, temporarily or permanently.
Persist: Saving and reusing the same evaluation abstraction repeatedly over multiple runs or multiple models.

Table 4: Operations on evaluation abstractions.

program comprehension tools for modelers. Without such support, modelers expended considerable effort to perform these operations manually or with tools they had to create themselves.

7. Conclusion

We have described a case study involving six cognitive modeling projects, investigating the expectations and corresponding abstractions that modelers have when they evaluate their models. We discovered a richly interconnected network of evaluation abstractions involving data structures, time sequences, and statistical aggregation. We further found that:

- Evaluation abstractions were varied in form; some mimicked common programming abstractions like sequences and trees, while others, like strategies and spatial layouts, were new.
- The abstractions were not just ad hoc descriptions of modelers' roving explorations, but patterns of persistent interest, as much a part of the modeling project as the code itself.
- Statistical analysis and debugging were separate phases of modeling, yet showed deep ties. The data on which modelers ran statistics for validation were the same entities they used for "up close" comprehension and debugging.

The evidence reported here of mostly unsupported evaluation abstractions demonstrates a gap in support for evaluation abstractions needed by cognitive modelers. We suspect this gap is not unique to this population alone. We therefore hope this line of research will inform the development of modeling environments that allow a wide range of modelers to keep better tabs on whether, and how, their models work.

Acknowledgments

We thank the Air Force Office of Scientific Research for partial support of this work under FA9550-10-1-0326.

References

1. Abraham, R. and Erwig, M. GoalDebug: A spreadsheet debugger for end users. *ACM ICSE* (2007), 251-260.
2. Anderson, J. R. *Rules of the Mind.* Erlbaum, 1993.
3. 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.
4. Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., and Wallace, C. End-user software engineering with assertions in the spreadsheet paradigm. *International Conference on Software Engineering*, 2003, 93-103.
5. Colin, S. and Mariani, L. Run-time verification. In M. Broy, et al. (eds.) *Model-Based Testing of Reactive Systems, LNCS 3472*. Springer-Verlag, 2005, 525-555.
7. Heinath, M., Dzaack, J., Wiesner, A., and Urbas, L. Simplifying the development and the analysis of cognitive models. *EuroCogSci07*, (2007).
8. John, B., Prevas, K., Salvucci, D. and Koedinger, K., Predictive human performance modeling made easy. *ACM CHI* (2004), 455-462.
9. Jones, R. M., Crossman, J. A., Lebiere, C., and Best, B. J., An abstract language for cognitive modeling. *Intl. Conf. Cognitive Modeling* (2006), 160-165.
10. Kieras, D. A guide to GOMS model usability evaluation using NGOMSL. ftp://ftp.eecs.umich.edu/people/kieras/GOMS96guide.pdf. Retrieved Sept. 3, 2009.
11. Ko, A., Myers, B. A framework and methodology for studying the causes of software errors in programming systems. *J. Visual Langs. Computing* 16, 1-2 (2005).
12. Ko, A. J. Asking and answering questions about the causes of software behaviors, Ph.D. thesis, *Technical Report CMU-CS-08-122* (2008).
13. Kulesza, T., Wong, W., Stumpf, S., Perona, S., White, R., Burnett, M., Oberst, I., Ko, A. Fixing the program my computer learned: Barriers for end users, challenges for the machine. *ACM IUI* (2009), 187–196
14. Ljungblad, S., Holmquist, L., Transfer scenarios: grounding innovation with marginal practices. *ACM CHI* (2007).
15. Mulholland, P. and Watt, S. Learning by building: A visual modelling language for psychology students. *J. Vis. Langs. Computing* 11, 5 (2000), 481-504.
16. Naish, L., A declarative debugging scheme. *J. Functional and Logic Programming 3*, (1997).
17. Norman, D. *The Design of Everyday Things*. Basic Books, 1988.
18. Ritter, F., Haynes, S., Cohen, M., Howes, A., John, B., Best, B., Lebiere, C., Jones, R., Crossman, J. and Lewis, R. High-level behavior representation languages revisited. *Conf. Cognitive Modeling* (2006), 404-407.
19. St. Amant, R., Freed, A.R. and Ritter, F.E., Specifying ACT-R models of user interaction with a GOMS language. *Cognitive Systems Research* 6 (2005), 71–88.
20. Segal, J., Some problems of professional end user developers. *IEEE Symp. VLHCC* (2007), 111-118.
21. Shneiderman, B. The eyes have it: A task by data type taxonomy for information visualizations. *IEEE VL* (1996), 336-343.
22. Siegler, R. S. and Shrager, J. Strategy choices in addition and subtraction: How do children know what to do? In C. Sophian (Ed.), *Origins of Cognitive Skills.* Hillsdale, NJ: Erlbaum, (1984), 229-293.
23. Wagner, E. and Lieberman, H. Supporting user hypotheses in problem diagnosis on the web and elsewhere. *ACM IUI* (2004), 30-37.
24. Yin, R. *Case Study Research: Design and Methods,* Sage Publications, 2003.
25. Zbrodoff, N. J. Why is 9 + 7 harder than 2 + 3? Strength and interference as explanations of the problem-size effect. *Memory & Cognition, 23*:6 (1995), 689-700.
26. Zeigler, B. P., Kim, T. G., and Praehofer, H. *Theory of Modeling and Simulation*. 2nd. Academic Press, 2000.