AN ABSTRACT OF THE DISSERTATION OF

Christopher Bogart for the degree of Doctor of Philosophy in Computer Science presented on June 7, 2013.

Title: Eliciting Informal Specifications from Scientific Modelers for Evaluation and Debugging

Abstract approved: _____

Margaret M. Burnett

Professional software engineers have an arsenal of techniques such as unit testing and assertions to check their specifications, but these techniques require tools, motivation, experience and training that programmers without professional software engineering training may not have. As a result, professionals in other fields, such as scientific modelers, face greater hurdles in debugging and validating the programs they write. This thesis introduces the concept of "evaluation abstractions" as a framework for tool designers to think about this kind of support. Evaluation abstractions are the patterns of data in program traces and outputs that programmers examine in order to evaluate software behavior. The thesis provides two intellectual contributions aimed at helping tool designers: (1) A theory of evaluation abstraction support (EAST) that describes at a granular scale the factors contributing to a modeler's decision to use or not use an evaluation abstraction support feature; (2) a new user-centered design methodology, Natural Programming Plus (NP+), specialized for the design of interactive languages aimed at experienced users, in a way that allows for validation early in the process. Using EAST and NP+ I built and evaluated an evaluation abstraction support tool for cognitive modelers (psychologists who study human cognition by writing simulations of cognition), with features that (1) elicit and persist a database of a modeler's evaluation abstractions, in a piecemeal, just-in-time fashion as their questions about model behavior arise, and (2) use the modeler's unique set of evaluation abstractions to structure visualizations, listings, and regression tests, as the modeler continues to maintain and develop the project. Using this tool modelers were able to repeatedly answer questions about model behavior that would have been time-consuming and error-prone to check in state-of-the-art cognitive modeling tools. This dissertation includes formative investigation of modelers' evaluation abstractions, iterative development and testing of interaction designs for elicitation and use of evaluation abstractions, a description of a domain-specific language for representing and transforming evaluation abstractions, and two summative studies showing the usability and generalizability of the technique.

Eliciting Informal Specifications from Scientific Modelers for Evaluation and Debugging

by

Christopher Bogart

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 7, 2013
Commencement June 2014

Doctor of Philosophy dissertation of Christopher Bogart presented on June 7, 2013.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Christopher Bogart, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor, Margaret Burnett, for her mentorship, friendship, and patience, and for training me to be a scientist and a better writer; Scott Douglass, for being a mentor, friend, and supporter, and especially for all the conversations about cognition and science that helped me think bigger thoughts about the context of the work we do; and Andrew Krizman, for uprooting himself to move across the country with me and teaching me not to worry too much.

I would also like to thank: The rest of my committee, for their feedback and encouragement: Martin Erwig, Carlos Jensen, Alex Groce, Christopher Sanchez, and Karen Dixon. My other coauthors on papers from which Chapters 5, 6 and 8 were drawn: Rachel White, Hannah Adams, David Piorkowski, and other collaborators who helped with data analysis: Jarrod Jackson, Damian Kulp, Jilian LaFerte, Irwin Kwan and Charles Hill. The Palm team for hosting me for repeated internships, and the modelers on the Palm team, for their patience in being my Guinea pigs. John Anderson's lab at Carnegie Mellon for the fantastic experience of the ACT-R Summer School, and the modelers there who also participated in my study. Other students, postdocs, and professors for their collaboration and advice: Scott Fleming, Irwin Kwan, Jill Cao, Valentina Grigoreanu, Todd Kulesza, Faezeh Bahmani, Joey Lawrance, Laura Beckwith, Kyle Rector, Chris Chambers, Eric Walkingshaw, Chris Scaffidi.

Finally I would like to thank the Air Force Office of Scientific Research for providing part of my support through grant AFOSR FA9550-10-1-0326.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

Figure Page

LIST OF FIGURES (Continued)

Figure Page

LIST OF FIGURES (Continued)

Figure Page

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

LIST OF TABLES (Continued)

Table Page

Chapter 1 – Introduction

Professional programmers have at their disposal a wide array of software engineering practices and tools for evaluating properties of their programs, but until recently, non-professional programmers have not had these advantages. However recent research has begun to yield a growing arsenal of interaction techniques that bring some of these benefits to non-professionals as well [2, 21, 61, 70, 82].

One theme that has not been thoroughly explored in this domain is what abstractions programmers use when they evaluate their programs, and how tools could capture and make use of those abstractions. In this thesis, therefore, we develop a language to represent the evaluation abstractions employed by a particular audience of scientific modelers, and describe our methodology so that other researchers and tool designers can apply it to other kinds of users. We also investigate how evaluation abstractions are deployed over time by modelers to answer complex questions about model behaviors, and we evaluate a theory of evaluation abstraction tool support, which predicts, at the level of individual GUI navigations, what tool properties and features must be present for a modeler to choose to use such features.

In Section 1.1 we define "evaluation abstractions" and argue that programming environments should support them more explicitly. In Section 1.2, we narrow the focus to the population of programmers we intend to work with, and in Section 1.3 we list the contributions of this thesis.

In this thesis we will use the term "professional programmers" to refer to software developers whose primary job is software development, and will assume that they already have sophisticated tools and practices at their disposal for evaluating software. "Non-professional programmers" will refer to anyone else who does programming. Some of the latter may in fact be professionals in other fields, and may be adept programmers, but the term is meant to emphasize that their professional focus is not on the tools and practices of programming per se. We will use "scientific modelers" or just "modelers" to refer to non-professional programmers who try to deepen understanding of phenomena by writing programs to simulate those phenomena. Finally, "cognitive modelers" are scientific modelers specifically in the field of psychology, trying to understand the process of human cognition.

1.1 Evaluation Abstractions

The Oxford English Dictionary defines *abstraction* as "the process of considering something independently of its associations, attributes, or concrete accompaniments" [3]. Applying this

definition to software development, abstractions are representations that omit details to allow for reasoning and comparison at a broader level than would otherwise be cognitively possible. When a programmer tries to comprehend a program's behavior, they form abstractions, because there is in principle an unbounded amount of information potentially at their disposal: the source code, the program inputs and outputs, the internal program states that could be printed, and the analyses that could be run. A programmer must ignore some of this possible data: i.e. must abstract it.

The abstractions this thesis investigates are "evaluation abstractions", which we define to be the patterns of data in program traces and outputs that programmers examine in order to evaluate software. This is in contrast to "programming abstractions", such as classes, functions, and objects that go into the writing and design of software. There is some overlap between the categories: a programmer may code a sequence of commands as a procedure, and may check to see if that sequence of commands was executed by looking to see if the procedure was executed. But the two sets are not the same: a programmer can check for patterns that are not explicitly-named entities in source code, and conversely a programmer might choose not to check the behavior of every construct they have defined.

For example, suppose a programmer writes a program that attempts to navigate through a maze by following one wall from the entrance, continually testing for the exit. In this example, the "programming abstractions" might be objects representing routes, walls or exits. Then, on running the program, the programmer plots the solution and notices that the program solves the maze, but that it does not take the fastest possible route. The programmer might describe what they see using notions like "fastest route", or "good enough route", or even "going in circles forever". Assuming the code did not happen to have a class, object or function directly corresponding to these notions, then the programmer's mental concepts of these possible emergent behaviors would be "evaluation abstractions", and not "programming abstractions". They are abstractions because they are simplified descriptions of a class of possible behaviors (e.g. many different cyclical routes could be described as "going in circles forever"), and they would be evaluation abstractions because the programmer would be using them to evaluate the behavior of a program under development.

"Expectations" are closely related to evaluation abstractions; if a programmer expects something to be true of a program's behavior, the abstractions with which they express that expectation would be evaluation abstractions. In other words, an evaluation abstraction describes some abstract way of summarizing of a program trace, and an expectation that employs that abstraction is a true/false claim that a programmer (or a tool) could check using that summary information.

There are several possible benefits from having programming tools which can explicitly elicit and store representations of users' expectations or their underlying evaluation abstractions. Tools

that have such a capability can collect information from the programmer at a more *convenient time*, use more *convenient abstractions*, and offer more *useful analyses*.

Collecting at a convenient time It stands to reason that an expectation a programmer has of their program is likely to be more mentally available to them at certain times and less available at other times. For example when a programmer is looking at output and sees a particular value that seems out of place, that expectation will probably be easier for them to take notice of and express when that value is present, in sight and in context, than at some other time in the development process. If the expectation or the evaluation abstraction(s) underlying it would be useful information for a tool to keep, now would be the time for the tool to collect it.

Unfortunately, other considerations typically drive the circumstances under which tools elicit expectations. For example consider the expectations a programmer might have about the return value of a Java function. If the programmer expects the value to be an integer, they might express this expectation as a type signature; if they expect the value to be even, they might express it as an assertion. The former must be written before the function is compiled; the latter can be written after the function has first been written, compiled, and executed. The reason programmers test these expectations at different stages is technological; it relates to the place in the tool chain where these conditions are easiest for the tool to check. It would be better for a tool to elicit expectations at a time dictated instead by the programmer.

Providing more convenient abstractions Why the need for supporting evaluation abstractions? A debugging tool might technically be said to provide "complete" debugging information if it exposed every atomic unit of hidden state: variable values, memory contents, and call stack, at every point in time. With this information, the programmer could in principle answer any question about what happened during a run. But providing "complete" information from an information theory standpoint may not be sufficient from a cognitive processing standpoint. Programmers might have to do a significant amount of further mental work to process that low-level information into the higher-level abstractions that constitute the domain-specific concepts the programmer actually cares about. This could be enough extra work that a programmer would choose to skip the step of evaluating some of their expectations.

For example, in Chapter 5, we describe a situation where a modeler "Matt" verbally stated (while looking for a bug) that a data structure he saw in a model behavior visualization should contain X,Y coordinates that would fall within a certain region if plotted in a window. An algorithm for doing this check would be trivial, but Matt decided the potential mental effort in finding and retrieving the data item's contents, figuring out the scale of coordinates on the window, and doing the back-of-envelope calculation necessary to check it was not worth the benefit. He decided to assume that it was correct and look elsewhere for his bug. If he had been

wrong, it would have been an expensive decision for him. Matt's experience demonstrates both how modelers can have expectations they do not encode explicitly into their program, and that they do not check some of them because of the time cost.

Providing more useful analyses Finally, there are good reasons to believe that tools can provide programmers with more useful analyses when informed of the modeler's expectations:

- Shared knowledge between human and computer could help disambiguate queries and limit the amount of information that needs to be shared. For example if a programmer was writing a program to process bank records, a tool that had acquired a few facts about accounts and how the programmer thought about them might make it easy for the programmer to list errors alongside the names of accounts that triggered them, rather than, say, the function names where the errors occurred.

- In some situations, programmers repeatedly check and recheck the same expectations because they have no way to save their expectations along with the code. Setting up a regression testing framework can solve this problem, but it is a significant upfront investment, and requires maintenance, both of which efforts a non-professional programmer might not choose to make [106]. Violations of modelers' expectations could be used not only as an opportunity for low-cost elicitation of evaluation abstractions, but also to generate unit tests or runtime verification checks.

- A model of what should have happened in a program can make it easier for a tool to detect when something goes wrong. Ko's Whyline [64] let programmers ask why-not questions, tracing backwards through an event trace to find out why one thing happened instead of another. Whyline accomplishes this by having users explicitly choose from a menu of "why not" questions (which can imply an expectation). But a more persistent model of programmer expectations could avoid the need for the user to ask the question, since the tool would already know when the program ran off the rails. Conversely, an interaction design like the Whyline could remember the presuppositions of "why not" questions, and save them up to help build a model of programmer expectations.

- Saved evaluation abstractions could be reused for multiple tools. Type declarations, unit tests, contracts, assertions, and debugger watch expressions are all statements of expectation that might in principle refer to the same expectations, but must be entered by a programmer differently for each tool.

- In the domain of cognitive modeling there is growing interest in generative modeling, in which models are generated from high-level descriptions. Although such research is in the

beginning stages still, there is potential for tools to use modeler expectations collected during model evaluation or debugging as suggested added constraints in code generation tools.

A better understanding of evaluation abstractions would lay the groundwork for reaping some of these benefits, by giving tool designers clear guidelines on when and how to elicit expectations needed for the above points, that might otherwise not be practical. This thesis develops a theory describing how to elicit expectations from programmers and a language for representing them and keeping track of them in connection with a program. Finally, it scratches the surface of the many possibilities for exploiting evaluation abstractions to help programmers, by showing that even the simplest visual representation of them – tabular summaries of abstracted model traces – can help programmers quickly answer complex questions about program behavior.

1.2 Scientific modelers

We have chosen scientific modelers as an appropriate programming population for this research, specifically cognitive modelers using ACT-R. Scientific modelers create models to test hypotheses about systems that they cannot test directly on those systems; their focus and professional interest is on doing science, not on building software [106]. Cognitive modelers are scientists who work to explain human cognition by building executable computer models of such phenomena as language, perception, and problem solving.

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, which provides the context for most of our investigation, although we also generalize beyond this language. 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.

We have chosen to investigate scientific modelers because, for these modelers, the Gulf of Evaluation [91] is wide. Norman defines the Gulf of Evaluation as the difficulty involved in assessing the state of a system. A wide gulf means the transformations necessary to check expectations are large. Although this may be inconvenient for the user, the large transformations

may be easier for us as researchers to observe. Furthermore, with a wide gulf, efforts to ameliorate the gulf should have a more pronounced effect.

Why is this gulf so wide for scientific modelers? Primarily because in trying to account for the behavior of systems in a particular situations, they must restrain themselves from programming that behavior in straightforwardly: since they are attempting to demonstrate how that behavior arises, explicitly causing it to happen would defeat the model's purpose. That is, if behavior happened because they simply told it to, then they would gain nothing from the modeling effort.

For cognitive modelers specifically, their scientific interest is not only in modeling correct human behavior, but also human mistakes. So in evaluating a model, modelers must distinguish between mistakes that are human-like in form and frequency, and mistakes that are implausible for humans. They must even reject behavior that is too correct – a model that does not make common human errors is not a good model. Although a C program to compute factorial can be proven wrong with a single bad test case, a cognitive model may require statistical evaluation to be shown incorrect. All of this means that a single execution of a probabilistic model might not in itself be evidence of a correct or incorrect model, without considering how typical that execution happened to be of all the possible runs.

Another reason the Gulf is wide for cognitive modelers is simply that the human cognition being modeled is flexible and adaptive. Modelers are likely to write models whose decision-making is highly reactive to the environment, rather than models carrying out fixed plans. This means that clear-cut expectations may be hard to unequivocally state, and therefore hard to write good tests for.

For all these reasons, cognitive modelers are a fruitful audience among which to observe programmers struggling with problems related to the expression and verification of expectations, and to design and test ways to solve their problems.

This thesis investigates cognitive modelers explicitly, but our theory of evaluation abstraction support in Chapter 4 is framed as having a broader scope of scientific modelers. Although we cannot claim to have evaluated the theory for this wider audience, the propositions in the theory are not specific to the study of cognition. Instead, the propositions are tied to other common features of scientific modelers: their focus on answering hypotheses about simulated systems without presupposing the answers, and their lack of focus on software development and maintenance as an end in itself.

1.3 Contributions

- An empirically evaluated theory of evaluation abstraction support for scientific modelers. This theory is aimed at tool designers, and predicts what navigations modelers will take in an interface to build evaluation abstractions, based on properties of the tool's features.

- Empirical evidence about evaluation abstractions used by cognitive modelers: a taxonomy, their relative frequencies, the operations modelers perform with them, and how they are sequenced when evolving over time in the exploration of model traces.
- An empirical validation and two summative studies of a domain-specific language, Lea3, for representing cognitive modelers' evaluation abstractions, and a related language, Lea3/T for representing transformations between abstractions.
- A new user-centered design methodology, Natural Programming Plus (NP+), specialized for the design of interactive languages aimed at experienced users, in a way that allows for validation early in the process.

Chapter 2 – Background and Related Work

"Evaluation abstraction" is a new term introduced in this thesis, but the set of phenomena the term represents are not new. There is a broad range of research that aims to help users make sense of complex and voluminous data by eliciting from them, in one way or another, some inkling of what matters to them about the data, then abstracting the data in a way intended to satisfy their need. The target users for the research projects surveyed below range from users of complex data sets, to professional programmers, to scientific modelers and end-user programmers, and specifically to cognitive modelers.

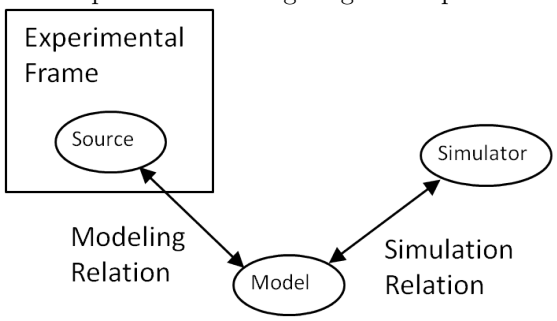
This chapter surveys prior work that elicits or caters to a user's specific evaluation abstractions. Cognitive and usability research central to the EAST theory will be covered in Chapter 4.

2.1 Evaluation Abstraction as scientific model validation

Zeigler [126] lays out a paradigm for scientific modeling that describes simulation in terms of three entities (Figure 2.1): a real-world system (*source*), a mathematical model (*model*), and a computer simulation of that model (*simulation*). He calls the relation between "source" and "model" the *modeling relation*, and the relationship between the model and the simulation the *simulation relation*. In these terms, *validation* is the process of checking the modeling relation, i.e. the correspondence between source and model, and *verification* checks the simulation relation, i.e. the correspondence between model and simulation. He further defines an *experimental frame* as the specification of the scope of modeling: what will be measured in the source, and what aspects of those measurements are to be modeled and validated. 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 cognitive modelers we discuss in Chapter 5 were primarily concerned with replicative validity and to some extent with predictive validity.

As Chapter 5 will show, modelers did both validation and verification by this definition: verification when debugging their models (to see that the software was behaving the way the modeler intended), and validation when checking model results against human data (to see that the theory encoded into the software was matching the real-world source data). Evaluation abstractions are part of the experimental frame, in a sense, because modelers must express their validation criteria in terms of such abstractions, whether that validation is against real-world source data, or against the modelers' own mental model of that source data.

Figure 2.1: Zeigler's conception of modeling. Figure adapted from his textbook [126].



Zeigler's view of modeling has only recently begun to be referenced within the cognitive modeling community, but there have been independent efforts in that community to formally describe their process. In particular, Ritter [103] analyzed the cognitive modeling process in his effort to assess the need for better tool support. He described the modeling process with the term "Trace-based protocol analysis" (TBPA). In TBPA, a modeler constructs an initial model that attempts to replicate the protocol of a human experiment, systematically compares the model trace with the human protocol, hypothesizes about the causes of any differences, and iteratively refines the model.

Segal [106] did field studies of development among professionals that included financial modelers, and biology, space, and planetary scientists. She described, among other things, how this population copes with evaluation abstractions in the absence of tools they perceive as appropriate. For example they often find ad-hoc solutions such as custom programming, repetitive manual procedures, and may test or validate inadequately. The only ones in her study doing modeling were the financial professionals, but in Chapter 5 we will describe cognitive modelers' attitudes and toolsets, which were markedly different Segal's financial professionals. For example, the financial professionals' tendency to underemphasize validation differed markedly from the cognitive modelers we observed, for whom validation was a central concern.

2.2 Empirical background: mental models, barriers, and information needs

In developing ways to help modelers evaluate their models, we used a variant of the Natural Programming [93] approach. In this methodology, researchers harvest concepts and relationships already in use by candidate users, and use them to inform the design of programming languages and tools. Although this technique was originally proposed for designing programming languages, Myers' group is also using it to design other aspects of the programming process: for example Ko used it to design a debugging interaction approach [61].

Unfortunately, "harvesting concepts and relationships" may mean eliciting something like a mental model, but eliciting users' mental models is notoriously tricky for researchers. Norman [90] warned that researchers cannot simply ask users what concepts they have and how they relate: when participants in such studies do not have clear concepts already, they are likely to simply invent them on the spot. This is why researchers must design mental model elicitation studies carefully. One way to elicit useful information is to make the demand characteristics of the experiment match the eventual use case of the tool being designed. Pane [94] did this by asking children to describe how they would program a computer to play Pac Man, and then harvesting constructs from the children's responses. Another approach is to harvest users' concepts from their communications produced for purposes other than the study. Blackwell [12] used a technique called Conceptual Metaphor Research to study metaphors in Java library documentation, as a way of surveying the different ways the Java programming community conceives of relations between programming constructs.

Little is known specifically about how cognitive modelers build evaluation abstractions and what these abstractions are like, although much research has been done more generally to try to describe what programmers' mental models are like, and how programmers construct them. Among the more general population of end-user programmers, for example, studies of the questions [60] and barriers [62] programmers face have categorized their information needs in various ways, but these studies do not provide detailed enough data for this dissertation. Evaluation abstractions seem to fit largely within a single category in the Kissinger study [60] ("Oracle/specification") and two in the Ko study [62] ("understanding" and "information"), so these taxonomies are not especially useful for analyzing evaluation abstractions more deeply.

Haynes' study [51] of users (not programmers) of cognitive models suggests that the needs of cognitive modelers may be different in important ways from those of the end-user programmers studied in Ko's and Kissinger's studies. Haynes described four categories of explanation required: ontological (what entities exist), mechanical (how do the entities work), operational (how do I use the entities), and design (what did the entities' designers intend them to do). He was surprised to find that users asked more questions about how the model worked than how to use it, and

speculated that this apparent deviation from minimalist learning theory [25] might be particular to users of cognitive models. If his speculation is correct, it may also apply to not only users, but also creators of cognitive models.

Outside the field of program expectations, others have created taxonomies of kinds of questions people want to answer about complicated structures. For example Lee et al. [75] created a task taxonomy of questions about large graphs, and this has been fruitful in designing new graph visualizations in principled ways.

Since none of these codesets were suitable for our research questions, we developed our own codeset, which we used in the case study in Chapter 5.

2.3 Formal methods for checking expectations

Professional programmers in highly secure or failure-intolerant domains sometimes use formal specification languages such as Z [99] to rigorously specify required program behavior, check its internal consistency, then use the checked specification as a guide when building the actual program. In contrast, we were interested in investigating just-in-time, just-enough-information specifications. Our investigation was motivated by the idea (informed by Attention Investment Theory [11] and Segal's observations [106]) that, without a lot of software engineering training under their belts, modelers who are not professional programmers are only likely to be willing to state their expectations about a program in situations where there is likely to be an immediate payoff.

Two appropriate software engineering techniques for providing relatively quick information benefits in response to specification statements are execution trace analysis and runtime verification, because they both rely on execution of the program – something the modeler is already likely to be asking the computer to do. Other techniques for analyzing specifications, such as model checking, require significant extra preparation by the programmer (such as building an abstract model of the program to check specs against). Execution trace analysis and runtime verification, on the other hand, are relatively flexible, a low investment to implement, can often check properties in linear time (which is important for interactivity), and can answer a wide range of programmer questions.

Much work has been done to help programmers capture and make sense of execution traces. Hamou-Lhadj's SEAT [50], for example, presented "usable" visualizations relying on much of his group's prior work in summarizing execution traces using Compact Trace Format (CTF). Users could specify filters that indicated what elements of the trace interested them, and the tool would eliminate irrelevant information from the trace and from traces it subsequently collected. These filters were not only lists of event types to collect or ignore, but also patterns that could be abstracted to save space: for example if a user wanted to ignore the order and number of methods

invoked on some particular object, SEAT would just record a list of which methods were called, instead of the full sequence of calls. It is possible that aspects of CTF's compaction methods may be adaptable to a stackless language like ACT-R, despite the fact that CTF's compaction methods depend in part on the structure that stack-based execution imposes on a trace. However the evaluation abstractions inherent in CTF's compaction are much simpler than what cognitive modelers need (according to the empirical study in Chapter 5), and there is little information available about the visualizations (they are not described or depicted in his published work, which focuses on the trace compression method itself).

Although execution trace analysis can explore temporal patterns by capturing what happens in a program over a segment of time, it is also possible to look for some temporal patterns on the fly while a program is running. Runtime Verification [29] was originally proposed as a way of monitoring specification adherence in deployed code, but a tool can use the same technique to provide a programmer feedback on temporal patterns without the need for a lengthy trace-collection step. For example AnaTempura [127] checks Interval Temporal Logic statements against a running program, verifying whether they are maintained, and if not what the expected values should have been.

The common formalism behind runtime verification and execution trace analysis techniques is temporal logic [29]. Temporal logics are modal logics that extend predicate logic with temporal operators, in order to formalize propositions about program behavior over time. In temporal logic, a proposition without one of these temporal operators is claimed to be true only at a single point in time. Modal operators are introduced to add claims about the (relative) future and past. The particular modal operators and their semantics depend on the flavor of temporal logic. For example in Future Linear Temporal Logic, $\Box x$ means "always x", meaning that x will always hold, $\Diamond x$ means x will hold at least once sometime in the future, and $\bigcirc x$ means x will hold at the next time step. Interval temporal logic adds time bounds to these operators: for example $[t_1, t_2]x$ means that x will always hold between times t_1 and t_2. MaC logic makes a distinction between "conditions" with duration and instantaneous "events", and provides operators to convert between them and to make Boolean claims about them [29].

Properties expressed as temporal logic propositions can be continuously checked in linear time by runtime verification systems by a recursive rewriting scheme: a proposition can be rewritten as an instantaneous claim about one moment in the trace, and a temporal claim about the rest of the trace. For example, $\Diamond A$ (sometimes A is true) can be shown to be equivalent to $(A \vee \bigcirc\Diamond A)$: Either A is true now, or "sometimes A is true" will be true in the next step of the computation [29]. The instantaneous part of the claim (A) can be checked, and if it is not true, the modal part of the claim can be postponed to the next execution step.

Modal logic formalisms have also been used outside software engineering contexts. For example Bauer [10] used an extension of LTL to represent patterns in financial transaction histories to

represent analysts' characterizations of "good" and "bad" buyers and sellers. Kok [67] defined a modal logic for expressing user interests in a data retrieval application, and inferring other likely interests from past searches using this formal language.

Statements of modal logic can be difficult to correctly construct for some intuitive notions like "between" that, according to Dwyer et al. [39] are quite commonly of interest to programmers. Dwyer's group compiled a catalog of "specification patterns" that list common propositions people have wanted to check, and how they can be expressed across different logics. Konrad et al. [68] extend Dwyer's patterns to logics over real time intervals with patterns they classify as "duration", "periodic" and "real-time order".

Barringer et al.'s LOGSCOPE specification language [8] is similar to Dwyer's patterns in that it used temporal constructs judged to be more accessible to software engineers than temporal logic's traditional operators would be. The language allowed its users, NASA engineers running simulations of Mars probe software, to specify and check temporal properties in log files by specifying lists of ordered or unordered events, or the absence of events. LOGSCOPE's goals differ from those of our research in that Barringer et al.'s users were highly motivated to write test suites, and simply needed better tools to do so. Most of the cognitive modelers we have observed did not try to build test suites as such. Enticing modelers to create specifications requires going further than LOGSCOPE in making the syntax accessible and the payoff immediate.

Some of the elements of modal logic are similar to some of the abstract relationships that are of interest to cognitive modelers. One important difference, however, is that logical propositions return Boolean results, but in our case study we observed that cognitive modelers looked for much richer summaries of model behavior that they could use to understand their models, not just test simple statements of fact. Nonetheless, some of the relationships and practical evaluation techniques from the temporal logic literature inspired elements of our language for evaluation abstraction support.

2.4 Tools for professional programmers

There are many research projects aimed at helping professional software engineers understand program behavior in execution traces, but many of these techniques are unsuitable for scientific modelers because either they require significant training to use (e.g. a knowledge of UML), or require a professional software engineering mindset (e.g. willingness to do a lot of up-front specification work).

Two subfields from the field of program comprehension research, however, *trace analysis* and *behavior analysis*, are of potential interest because they relate to understanding program behavior, rather than design intent or program structure. Within both of these subfields there has been work that elicits and takes advantage of programmers' particular expectations in some way. These

were two of the four categories of paper that Cornellisen et al. [30] found when they surveyed more than four thousand papers on the use of dynamic analysis for program comprehension. Trace analysis includes activities such as creating visualizations of UML sequence diagrams, and an example of behavior analysis would be tools that infer protocols or grammars from traces.

For example, De Pauw et al. [95, 33] have done several projects using trace analysis to help with program comprehension. Their Zinsight tool lets users indicate an interest in the sequential relationship of a pairs of event types in an execution trace (of processes in an IBM SystemZ machine), and the tool shows instances and violations of that sequence as clickable tick marks. In another project, his group built visualizations for behavioral analysis [33], i.e. characterizing a trace in terms of common behavior patterns. Their tool detected and depicted patterns in a trace of web services transactions (a kind of execution trace, if cooperating web services are thought of as threads in a distributed program).

An example of behavior analysis is Koskinen et al.'s [69] behavior patterns, snippets of UML sequence charts that can be used as a template to check against an execution trace. Their tool only reports those parts of the trace that match, or nearly match, the template. Their evaluation did not include a user study, but they claim that unfiltered traces are simply too large for a human to analyze, so they argue persuasively that this filtering should make the diagrams easier to understand. Kosinken's pattern representation is similar to De Pauw's visualization of web services, but DePauw's is descriptive (what patterns are found in the trace) while Koskinen's is prescriptive (how the trace conforms to intended design).

Another relevant behavior analysis technique is that of Lencevicius et al [76], who created debugging techniques using queries akin to Koskinen's behavior patterns, but which search for data structure relationships rather than sequential relationships. Dynamic query-based debugging and on-the-fly query-based debugging give programmers a query language for seeking out fairly sophisticated, but non-temporal, data structure relationships.

A recent very successful tool for professional programmers that falls outside the domain of trace/behavior analysis is Code Bubbles [19, 35], in which researchers saw large reductions in programmers' task performance and navigation time, by allowing programmers to assemble editable snippets of source code task-relevant ways in "bubbles" on a large zoomable canvas. Bragdon et al. [19] showed that not only did it reduce navigation time, but helped them do other tasks more efficiently since they did not need to expend working memory on back-and-forth navigation. Code Bubbles allow programmers to create custom views of code, not of program behavior, but the results show the benefit that can come from letting people customize their view of complex structures.

2.5 Frameworks for professional programmers to build on

Some tools simply provide programmers a framework that can identify patterns of interest in program behavior, and let programmers write their own code to analyze or visualize data about those patterns. Marceau et al. [80] gave a good summary of such trace analysis frameworks. The common feature most of these share that differs from the work in this thesis (except Bruegge's; see below) is that they require the programmer to do more programming; the programmer must understand the debugger's API, language, and manner of operation in order to extract and reformat the information they need to check their expectations.

The approach in this thesis, in contrast, elicits modelers' expectations in a form close to how they might naturally state them, and not make them worry about the details of how the data are collected and manipulated. The point is to allow them to check their expectations with as little fuss as possible, not to give them another programming language to learn.

A straightforward example of a trace analysis framework for visualization is Wang's EVolve [120]. EVolve makes the programmer responsible for supplying trace data in a record-like format. At runtime the programmer can select fields from these records to plot in standard ways like histograms or scatterplots; the programmer can also write and plug in their own generic visualizations. Wang demonstrated how these plots could be used for comparing numerical and categorical data items' relationships over time, but does not claim to provide a way of comparing non-numerical relationships, such as varying event sequences or data whose structure differs between instances.

A programmer can encode more sophisticated evaluation abstractions in Reiss's VELD [101]. It provides an XML syntax that lets programmers describe events of interest, an extended finite state automaton (with variables and condition testing) over those events, and a mapping from automaton states and properties to dimensions like color and location. Reiss and his colleagues stated their intention to add a graphical automaton editor and a point-and-click way of adding events, but a discussion of VELD in their later work [102] suggests that they have abandoned that line of research.

Even more powerful than VELD's FSAs are the evaluation abstractions embodied in Martin et al.'s PQL [81]. A PQL query is actually a context-free grammar for recognizing program events, extended with constructs for intersection (the ability to require a program to match multiple rules at once), and dataflow. The dataflow construct would make it very simple, for example, to write a query to show where a string returned from a web form is eventually passed to a SQL execute method. The action part of a PQL query can either name a function to execute when a match is found, or it can replace a function call with a substitute function (which the programmer is responsible for writing). PQL can attempt to prove queries statically, or check them dynamically by inserting just enough code to perform the queries the programmer has requested.

Goldsmith et al.'s [45] PTQL is similar in some ways to PQL, but uses SQL syntax, treating the stream of method invocations and object creations and deletion as tables. This tool analyzes the query and the program code to decide where to inject its monitoring code and what data to collect. According to Martin [81], PTQL's query language is better for expressing value constraints among variables, but is more awkward than PQL for expressing complex ordering constraints.

Bruegge's generalized path expressions [20] used a simple regular expression syntax over an alphabet of code regions. The regular expressions were extended with the ability to restrict matches according to the number of times a region had been entered. When they matched, an associated list of debugger commands would be executed. Unlike in the systems mentioned above, programmers could specify and match these evaluation abstractions right in the debugger. In contrast to Bruegge's work, the approach described in this thesis does not attach debugger actions to patterns, but instead uses pattern matches to structure interactions and visualizations to help evaluate program behavior. We also introduce a richer expectation language, not limited to just sequences of code visits.

The frameworks mentioned so far used separate languages for the pattern to be matched, and the action to be taken. Other frameworks blend the two functionalities into unified languages. Ducassé [38], like Martin, also worked with streams of tuples from an execution trace, but used Datalog, a logic programming language, to match them. With this full-fledged programming language, she could also write visualizations directly in Datalog. Over the course of her PhD work she built visualizations for debugging languages as diverse as C, Prolog, Mercury, and CSL, a constraint satisfaction language, using a common framework called LSD. Outside the realm of program traces, a class of similar languages for processing event streams is emerging, called Complex Event Processing (CEP) [34]. Esper [42], for example, is aimed at business data applications and Post et al. [98] describe a CEP tool for clinical data querying. Marceau et al. [80] provided a functional reactive programming solution in MzTake, allowing programmers to write powerful abstractions in the Scheme-based FrTime language. Similarly to Bruegge, this was a full-fledged scripting language available to the programmer from within the debugger, with persistent scripts. The authors implement abstractions over data structures, temporal relationships, and statistical claims in their examples in the paper, all three of the categories we identified among cognitive modelers [13]. In contrast to MzTake, this thesis describes a domain-specific pattern language (Lea3) to succinctly elicit modelers' expectations, not a full-fledged programming language designed for execution trace analysis by professional programmers. Lea3's semantics could have been defined in terms of MzTake, but the language as presented to its target audience is very different from ours.

One final tool worth mentioning, although it does not strictly speaking involve evaluation abstractions, is Maoz et al.'s visualization [79] for execution traces produced by Live Sequence

Chart (LSC) programs. LSCs are like UML sequence diagrams, but divided into an initial match sequence and later action sequence. Using LSCs, programmers specify action sequences that should occur after matched sequences are recognized. The specified LSCs are analytically combined and compiled into a program. Maoz's visualization simply shows horizontal bars over time showing when various LSC's sequences were in play in the running program; it also animates depictions of LSCs as the program traverses them. Thus these are straightforward visualizations of programming abstractions, not evaluation abstractions, but they could be equally useful as depictions of expected sequences as performed by a program written in a traditional paradigm.

2.6 Tools for end-user programmers

In the emerging field of end-user software engineering, a number of research projects have involved eliciting very simple evaluation abstractions from end-user programmers, in the form of correctness judgments and expected values, and using these judgments to help end users improve their programs. Abraham and Erwig's Goaldebug [2] demonstrated the utility of collecting a relatively simple expectation: the value a user believed should have been calculated in a single cell of a spreadsheet. Using an empirically derived model of likely errors in spreadsheet formulas, Goaldebug worked backwards from the value a user said a calculated cell should have contained, and proposed a list of formula changes to cells in the spreadsheet that would result in the value meeting the user's expectation. Kulesza et al. [70] let users specify many expected values: category tags for multiple lines of transcribed text. Their tool used that set of desired results to generate corresponding changes to a machine-learned text classification program. Ko's Whyline [61] let users ask "why not" questions which presuppose an expected value (e.g. "Why isn't this line black" presupposes an expectation that the line is black), or "why" questions presupposing a judgment of incorrectness (e.g. "Why is this total zero?" suggests a presupposition that the total should not be zero) and used backward dynamic slicing on a stored execution trace to answer the "why not" question. Finally, WYSIWYT (What You See Is What You Test) [21] let users mark the values produced by formula cells in a spreadsheet as "right", "wrong", "maybe right", or "maybe wrong". It used a database of these judgments to predict likely error locations by collecting user judgments of correctness of spreadsheet cells over many test conditions. Woodstein [119] similarly captured user hypotheses about bugs in web-based processes.

Researchers have also created end-user programming tools for understanding more elaborate evaluation abstractions, but rather than assisting with understanding program behavior, these tools assisted with understanding other streams of data. Two languages, Querymarvel and PROPEL, provide a contrast demonstrating a difficult tradeoff in this domain: on one hand, an apparently simple sequence language may in fact have a lot of subtle semantic assumptions buried in it which might prove counterintuitive or insufficient for users' needs; on the other hand, a full-blown temporal logic might be too complex for some users' needs.

Querymarvel [53] was a query language for finding matches in a database of patients' medical histories. It used visual conventions from comics to express sequence, absolute and relative time spans, simultaneity, and negation. It also added an "or" construct and conventions for indicating that an object was the same or distinct from objects in neighboring frames of the query comic. In contrast to the query transformation approach of this thesis (Section 7.4), the user interacts directly with a visual query syntax in a query editor pane. The medical event sequences it describes do seem to be analogous to a subset of the kinds of program behavior events of interest to cognitive modelers.

Where Querymarvel was limited but easy to understand, Cobleigh et al.'s [28] PROPEL was powerful but complex, although still aimed at end users. They supplied a "query tree" wizard to step users through the process of creating sequence queries, which in the end were expressed as "disciplined natural language" English sentences, as well as finite state automata diagrams. Their query language was formally equivalent to temporal logic, describing patterns of workflow in medical record processing. Their design goal seemed to be an error-free complete automation of a classification task. Because of this, the language was extremely verbose and precise. For example, to say that something must be true between when an order is received and the documents are prepared required three paragraphs of text to define "between". Empirical validation of the language merely showed that end users (nurses) who were experts in this domain could make sense of the text most of the time, not that they could independently generate these queries with the query tree tool.

Another approach to end-user sequence searching was characterized by wizard-style configuration of machine learning techniques, followed up with further interaction as the data changed or user needs changed. Raz [100] used an arsenal of machine learning techniques to look for violations of a user's expectations in a data stream. Her tool elicited expectations by showing the user what a variety of pattern recognition algorithms could handle, letting the user build candidate expectations, then letting the user refine the expectations after seeing how they matched the data. Raz's users, transportation workers monitoring traffic patterns, had fairly stable sets of expectations over time compared to cognitive modelers pursuing a variety of analytical questions about models; modelers thus needed a quicker process for specifying evaluation abstractions than Raz's iterative training process.

Researchers in Programming-by-demonstration (PBD) have also had reason to find ways to elicit desired sequences from end-user programmers, even though these were programming abstractions, not evaluation abstractions. Researchers have proposed solutions to PBD challenges such as generalization, negation, and manipulation of abstract entities. Generalizing a sequence description is a hard, context-dependent problem. Kurlander [71] built a macro language for a graphical editor, using programming-by-demonstration, that helped users find snippets of their own interaction history and generalize those snippets into reusable macros. To generalize reasonably, the tool has to guess why the user chose objects to act on, and why they chose the action they did, in order to perform an analogous action on analogous objects when the macro is run in a new context. The authors found that they needed to create specialized heuristics for each of the object and action types in their environment, and present lists of possible generalizations for users to select from. They depict the generalizations as English text, because they claim the possibilities for generalization are too varied and complex to make a concise easily learnable graphical language.

The inclusion of negation in sequences (event B should not happen after event A) and abstract entities (event A should be followed by a complicated situation B) is a challenge for mouse-based interaction, because negative and abstract entities are harder to depict, and therefore harder to refer to with a mouse gesture. One PBD system that explored solutions to this problem was Gamut [82], which learned rule-based programs for playing games by observing the user's demonstrations of correct behavior. It learned negative rules with a "Stop that" feature where the user could mark a behavior as wrong at the moment that it happened. Handling of negative rules has been shown to be a difficult problem in general [56, 114], and the Gamut research suggested that the right time to collect negative rules may be in the context of an example, not out of context when building rules in principle.

Gamut also added visual guide objects and ghost objects to represent abstract entities, or what the authors termed "hidden entities", that is, triggers for behavioral rules that were not simple presence of on-screen objects. The researchers had some difficulty getting participants comfortable with these guide objects, however. This suggests that rules complicated enough to require manipulation of such hidden factors should probably be avoided when possible; but the Gamut work also provides design ideas for cases where use of hidden objects are unavoidable.

2.6.1 Techniques from Visual Analytics

Visual analytics is an emerging discipline of software design for understanding large data sets. Thomas [117] defines it as the effort to "support multiple levels of data and information abstraction, including integration of different types of information into a single representation."

LifeFlow [122] is a visualization that summarizes a large number of event sequences (hospital events such as admission, drug delivery, or discharge, for clinical analysts), using one dimension to depict how often different orderings of the events occurred, and a second dimension to show the average time between events within each ordering. Lifeflow's query language [85] is not unlike that of QueryMarvel: the user lays out icons representing events in a query window to abstractly represent sequences they wish to search for, with special conventions for query conditions like intervals and required absences of events.

Cardiogram [105] lets automotive engineers examine the diagnostic logs generated by automotive computers. For a variety of straightforward graphs of event occurrences, fluctuating measurements, and changing states, Cardiogram lets engineers configure panels of graphs and decide what to display out of many possible data items. More interestingly, the engineers build finite state automata based on disparate events throughout the car, and display visualizations based on the automata's transitions. These automata are custom-built by programmers, at the engineers' request, however, so Cardiogram is a good source of inspiration about display techniques, but not about capturing evaluation abstractions from its end-user engineers.

The approach in this thesis is similar in some ways to Wrangler [49, 57]. Like Wrangler, our prototype tool EAST-Env lets users interactively transform data. Both systems tailor the set of offered transformations to the current context. However, Wrangler targets taking arbitrarily formatted data into a relational format for further processing, which is not the goal of EAST-Env. (In fact, EAST-Env uses data already in relational format, which is done as a pre-processing step.) Unlike Wrangler, EAST-Env focuses on helping scientific modelers find and understand complex data patterns, among multiple time-oriented data tables representing multiple streams of model trace data.

For scientific model debugging, Grimm [48] advocates "visual debugging", by which he means making a variety of visualizations available at every step of simulation in order to make it easier to catch mistakes in models. His example is marmot populations, in which marmots went extinct "from south to north" instead of randomly–an effect that might only be noticed in a visual display, not in tables of numerical output. He also lists as advantages reproducibility (researchers can download, run, and observe each others' models) and explainability to people in related fields. He does not address the question of how the evaluation abstractions underlying visualizations should be chosen or programmed, but his scheme would certainly let modelers see transitory runtime variables alongside scientific visualizations, which could give modelers the power to expose some relationships not previously available to them.

2.7 Tools for cognitive modelers

The cognitive modeling community has contributed important foundations to HCI such as GOMS, information foraging theory, and cognitive tutoring (e.g., [6, 44, 54, 89, 96]). However, although a few tools to assist with model evaluation have emerged from the modeling community itself (e.g. [118]), HCI research into how to support the population doing this important work is sparse.

Within the field of cognitive modeling, there is active ongoing research specifically directed at creating languages for modelers that allow them to specify their models at a higher level of abstraction. Crossman et al. [31] recently described current modeling languages as akin to assembly languages, and are working to build a higher-level cognitive modeling language on abstractions derived from advice from cognitive modelers that they surveyed, and their own cognitive modeling experience. For cognitive modeling specifically, there is ongoing research into abstractions for modelers, but it is specifically directed at creating new languages for cognitive modelers [104], not at ways of evaluating and debugging models. These include HLSR [104], a high-level cognitive modeling language; Hank [86], a GUI interface for the Soar cognitive modeling language; G2A [113] and HTAmap [52], both of which translate high-level task descriptions into ACT-R; and CogTool [55], an ACT-R-based visual language for simulating user interface interaction. Finally, SimTrA [52] creates summary statistics of eye tracking data from cognitive models and outputs them into convenient tables in R. In contrast to these projects, this thesis develops an evaluation abstraction language against which behaviors of existing models can be abstracted and described, rather than a programming language from which models are compiled.

Ritter's research [103] into cognitive modeling methodology, mentioned in Section 2.1, resulted in a tool for allowing a modeler to align a model's trace with a human protocol consisting of a mixture of verbalizations with interactions like mouse clicks and keypresses. His tool can propose an alignment between mouse clicks, after which the cognitive modeler checks this alignment and then attempts to align the verbalizations with in-the-head events in the model. The tool assists with a very specific and demanding model-building methodology that Ritter recommends, but which not all modeling projects seem to include. Ritter's tool does not attempt to address modelers' broader need to understand model behavior when the model is not yoked to a particular experiment's transcript.

For more general evaluation purposes, modelers generally have two main devices available for checking their expectations against model behaviors: standard listings and visualizations that are designed to meet the needs of cognitive modelers generally; and frameworks allowing modelers with programming experience to do their own visualization or analysis.

Cognitive model development environments tend to be supplied with both of these options. ACT-R allows users to choose the level of detail in a textual trace, or pick which modules'

activity to trace in a simplified sequence diagram [17, 18]. Its debugger provides sequence diagrams that show patterns of activity in the model over time, but these are only tailorable in the limited sense of choosing which of the dozen available modules to show. For more sophisticated evaluation, modelers can (and do: see Chapter 5) embed Lisp code to intercept, log, or process events. Soar [72], another common cognitive modeling language, has a similar set of visualizations, and a somewhat more elaborate command-line argument language for controlling logging, letting modelers describe exactly which productions and memory changes to log. In both these languages, therefore, elaborate evaluation abstractions can be monitored, but may require significant amounts of programming, training and experience to use effectively.

A tool specifically for evaluation of both Soar and ACT-R cognitive models is Tor et al.'s CaDaDis [118], or Categorical Data Display. CaDaDis is a tool for multiple cognitive modeling platforms that generates Pert charts (i.e. dependency graphs of subtasks), Pert sequence charts (Pert charts laid out along a timeline), and Gantt charts from Soar or ACT-R model runs. These are tailorable only in small ways: for example the user can select which items to place in a chart.

CaDaDis is built on top of VISTA [115], a flexible framework intended to help modelers build facilities for explaining the behaviors of agents in large multi-agent simulations. VISTA allows a programmer to create mappings between a cognitive model's native data structures and a standardized representation of the model's notions of such things as goals, milestones, and other agents. VISTA requires a modeler to do significant amounts of programming, both to do the representation mapping, and to create custom visualizations within VISTA specific to the modelers' needs.

Between the extremes of standardized visualizations out of the box, and fully-open frameworks for expert programmers, this thesis attempts to fill a middle ground by providing the modeling community the best of both, of creating model-specific and even question-specific representations of behavior without requiring a lot of programming time or expertise.

Chapter 3 – Approach and Methodology

3.1 Methodology

To address the research questions of Chapter 1, we followed a user-centered design process, and used problems we encountered along the way to identify the features that might be unique and interesting to this domain (i.e., providing support for evaluation abstractions). The user-centered design approach we used to structure our research was Natural Programming Plus (NP+), a variant of Natural Programming my coauthors and I devised for Study 3W and Study 4 [14].

3.1.1 Why a new methodology?

First, we believed that neither a pure task analysis approach nor a pure language design approach was appropriate for this research, because the process of evaluating a model is an iterative, *interactive* task where questions evolve in response to lessons learned (suggesting a task analysis), yet the questions themselves involve complex abstractions (suggesting a language design approach). To accomplish our research goals, then, we needed to investigate the constructs, relationships, and interaction sequences that modelers used to assess and fix model behavior. We chose Natural Programming (NP) as an appropriate methodology for studying evaluation abstraction queries because, as Chapter 5 will show, evaluation abstractions (EAs) are in some ways like programs. EAs have complex internal structures that are persistent and composable. On the other hand, it was valuable to extend NP to thoroughly investigate the interactive aspects of EA construction, since EAs may also exist as fleeting one-off debugging questions, that may arise in cognitively demanding debugging situations in which modelers need answers to questions quickly and with minimal disruption.

Another reason that we needed to extend NP was related to our audience's prior experiences. We hypothesized that long experience with particular cognitive modeling tools would give modelers a learned set of habits for evaluation strategies for evaluation that were particularly well supported by those tools. These habits might interfere with their use of a new tool supporting new strategies. A study of such a new tool might therefore say more about how difficult it is to unlearn old habits than about the effectiveness of the new tool. For example as Chapter 6 will show, ACT-R modelers sometimes examine data structures remaining in memory at the end of the run in order to infer what happened during the run, and they sometimes tried to do this using

our prototype tool, EAST-Env, even though that strategy was poorly supported by EAST-Env and they had more direct information available.

Finally, we also needed a way to validate results earlier in the process. Building and evaluating experimental prototypes is expensive, and we found this to be especially true for an evaluation abstraction tool, since the work involved not only user-interface design and implementation, but also extracting data from model runs and manipulating it efficiently enough to provide interaction fast enough to be a reasonable test. Unfortunately, low-fidelity prototyping would be especially difficult in this domain, since an evaluation abstraction tool's behavior in the face of complex questions and complex model behavior would be hard for a researcher to predict and simulate on the fly with a paper prototype. Thus, we added 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 helps by providing both scaffolding for our effort to design an interactive evaluation abstraction language based on empirical evidence, and ongoing analytical measures of how well the emerging language matched that evidence. NP+ was jointly developed by me and my coauthors in [14].

3.1.2 Natural Programming Plus

Natural Programming Plus makes explicit a method for mapping the outcomes of NP's empirical investigations to a language design. Its aim is to provide a concrete method for computer scientists uncomfortable with the design leap from empirical assessment of users' needs to a language.

Natural Programming [93] 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 [92]. It has since been used to design numerous tools that support programming, scripting, and debugging (e.g. [65], [87], [88], [121]).

Pane and Myers [93] 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

What makes NP+ distinct from Natural Programming is that we expand on Step B to provide a process for designing and validating a specification of the new language. The new process breaks

down the process of "understanding the target audience" into steps aimed at first observing the concepts and relationships they use, exploring how their use of the concepts responds to interactive feedback over the course of a task, defining a language specification, then checking to see that it matches the empirical data collected.

Thus, as Figure 3.1 summarizes, we replaced Step B with the following four steps:

Step B1/Collect constructs and relationships Use a case-study or field study methodology to harvest the constructs and relationships found in the target audience's domain of interest.

Result 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/Collect sequence Use a Wizard of Oz methodology to learn how the users employ those constructs and relationships in a task cycle with rapid feedback. To do this, the experimenter must develop an executable language that is an approximation or superset of the concepts and relationships from B1, and practice using it until he or she can translate natural language "programs" into this language on the fly, and interpret the results. Thus in the study the participant can very approximately experience use of the interactive and informational structure of a hypothetical new system, without the expensive of UI development or the confound of choosing a user interface.

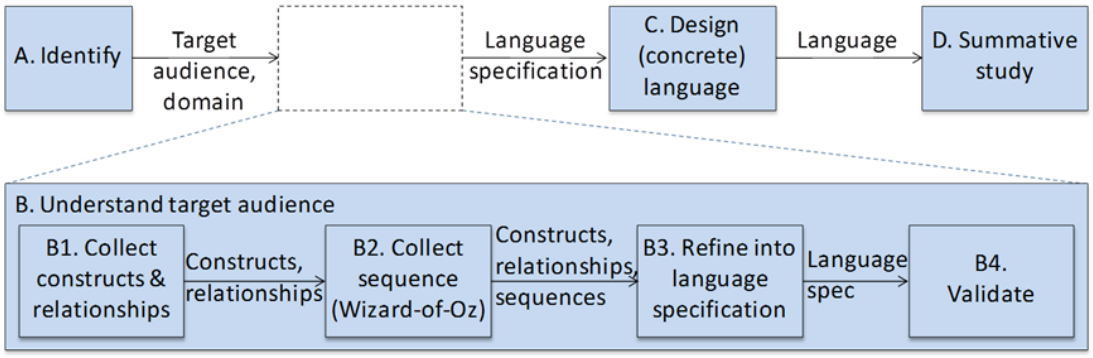
Result (1) any constructs and relationships missed or misunderstood in B1, in particular things that may arise uniquely in the interactive environment that were not observed before; and (2) the way participants naturally sequence their interactions in response to the feedback of executing each query.

Why This step reveals how users respond when the capabilities derived from B1 actually execute. It also allows B1's results to be validated with the target audience itself.

Step B3/Refine into a language specification Refine the results of B1 and B2 into an interactive language specification. This consists of syntactic and semantic definitions of a language, as well as some way of capturing the sequencing of syntactic forms that users are likely to make over time. (This could be done in many ways. For this thesis, we attempted to define the language such that changes to the root of the abstract syntax tree corresponded to likely next queries that users would make, then augmented that by defining a set of deeper tree transformations based on the kinds of queries modelers tried to perform.)

Result Syntax, semantics, and sequencing specification for an interactive language.

Figure 3.1: 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.



Why The language specification is a precise form of "Implications for Design". Because it is precise, it is auditable, and this facilitates evaluation and keeps the researcher accountable.

Step B4/Validate Measure coverage of the language specification (how many of B2's programs it could execute), its soundness (correctness, i.e., the responses it did produce are what the participants asked for, in the context of the available data), and *root viscosity* with respect to B2's data. (In Section 7.3.3 we define *root viscosity*, an estimated upper bound on the effort a user will require for the most likely follow-up requests.)

Result Measures of coverage, soundness, and root viscosity.

Why To ensure the language designs serve at least the needs observed to date.

3.2 Outline of Research Activities

The research activities outlined below (see also Table 3.1), and described in subsequent chapters, were driven by the NP+ process and Evaluation Abstraction Support Theory (EAST), described in Chapter 4. EAST is a theory we have been developing which describes the factors behind a modeler's decision to use an evaluation abstraction support feature. The theory posits that modelers will use and benefit from evaluation abstraction support features to the extent that the features help them build composable, reusable abstractions about their model's behavior, and that the interface to these features should provide new information about the model with almost every click, rather than requiring elaborate set-up and planning. If a modeler does not perceive

Table 3.1: The major research activities that this thesis comprises.

Activity	**NP+ Step**	**Chapter**	**Product of Research**
Study 1/Modelers	A	5	Taxonomy of EAs and their operations
Study 2N/ACT-R Use	B1	6	Modelers' ACT-R usage
Analysis of findings from A and B1, and prototype build	B2		Clarified understanding of EAs; foundation for studies 3,4
Study 3W/EA Sequencing	B2	6	Sequences of EA use in answering evaluation questions
Lea3 and Lea3/T design	B3	7	A domain specific language of EAs
Study 4/Validation	B4	7	Validation of findings so far
2nd prototype build	C	8	EAST-Env: A prototype sufficient for independent use
Study 5/ Summative: Lab	D	8	Support for EAST
Study 6/ Summative: Field	D	9	Delineation of factors influencing generalizability of approach

the immediate visible result of evaluation abstraction support to be useful, they will probably not be persuaded to use the facility in the long run. But in order to derive useful representations from a repository of evaluation abstractions elicited from a modeler, then, we first had to know (1) what evaluation abstractions they had, and (2) what they were using them for.

To investigate the identity and uses of modelers' evaluation abstractions, we did a case study (NP+ step A) in order to develop a taxonomy of evaluation abstractions, and a list of operations modelers performed with/on them. We analyzed how modelers described the behaviors of their models to each other in meetings and presentations (Chapter 5). A case study [124] is appropriate for qualitative study of existing processes over which the researcher has little control. The methodology involves systematic comparison of observed properties across different cases, and the use of multiple sources of evidence to bolster the claim that the observed phenomena are real and do not just stem from the researchers' biases. The evaluation abstractions and their operations revealed by this study motivated many of the constructs in EAST, as described in Section 5.7.

This taxonomy of evaluation abstractions came mostly from human-human communication among modelers, such as conversations and presentations, but this raised the question of how these forms might differ from how they appeared in human-computer interaction when actually doing the task of evaluating model behavior. Thus we did another study, Study 2N, with cognitive

modelers, to see how they went about debugging two ACT-R models, seeded with bugs. We used that data to understand what questions modelers have when debugging a model, and what techniques they use to answer the questions. Analyzing these debugging sessions using the same codeset as Study 1 revealed that the distribution of evaluation abstraction types was different, notably that the "statistical" category of evaluation abstractions was disproportionately rare in debugging.

Modelers answered some of their questions in Study 2N by doing manual work that could in principle be automated, and in some cases this took the bulk of the time spent in Study 2N's session. But we also needed to know how modelers would proceed over time if given the ability to more quickly answer such questions. Studying this would require software that could quickly carry out such queries, so we built a prototype evaluation abstraction tool to make an executable representation of evaluation abstractions, as a foundation for further experiments, and as a first cut at a L_{EA} language as described by EAST. This prototype captured an execution trace and allowed users to examine it. Some practice with the prototype gave us the capability to quickly answer questions like the ones modelers asked in Study 2N. This allowed us to perform a Wizard-of-Oz study (Study 3W; NP+ step B2), in which we examined the sequence of questions modelers would ask, if they were presented with scenarios like those they encountered in Study 2N, but had rapid feedback after each query. This study allowed us to validate the taxonomy of evaluation abstractions found in Study 1, and learn about the *sequencing* of modelers' evaluation abstractions in the context of problem-solving. This work is described in Chapter 6.

In order to check that the evaluation abstractions we had learned about from studies so far broadly covered the range of modelers' needs, and that our understanding of their sequencing was correct, we encoded the results from Study 3W into a revised and reimplemented version of EAST's Lea3 language (Chapter 7; NP+ step B3). Lea3 captured a range of evaluation abstractions and had an explicit "transformation language" describing how modelers used it over time. It was designed to be compositional in such a way that each additional subquery would be both immediately useful, but also be a stepping stone to further queries likely to be of interest, aiming at maximizing the *M4 Perceived Usefulness for composition* and *M2 Representation Usefulness* constructs of EAST. The language served as a coding scheme for a more formal analysis of Study 3W's transcripts, allowing for several kinds of validation (Section 7.3; NP+ step B4), as well as serving as a spec for a new prototype (NP+ step C, described in Chapter 8 and Appendix 9).

With a language specification and prototype in hand, two summative questions remained (NP+ step D). The first was a qualitative study evaluating some portions of EAST: whether *M4 Perceived Usefulness for composition*, *M2 Representation Usefulness* and *M5 Context-relevant Cueing* would let modelers quickly check complex evaluation abstractions, without a Wizard of Oz by their sides. We address this question in Chapter 8, describing a summative, observational

lab study, in which modelers were indeed able to check a list of expectations, with accelerating speed over in multiple runs of a model, that would have been time-consuming and error-prone to check using ACT-R tools. The study also gathered evidence about what kinds of *M5 Context-relevant Cueing* would be most appropriate for lowering modelers' perceived costs to navigate, and what influenced modelers' perceptions of risk and cost when considering exploratory navigations.

The second summative question was to understand in what ways our findings to date depended on the particulars of the modeling language that we had chosen to support. We answered this question with a field study to find out what issues would arise in generalizing to other cognitive modeling platforms and projects. Chapter 9 outlines the issues we encountered in adapting the evaluation abstraction tool to four other modeling paradigms; none of them related directly to EAST's predictions, but the findings nonetheless suggest useful implications for designers of similar systems.

Chapter 4 – Evaluation Abstraction Support Theory (EAST)

Software engineering tools in general, and tools specifically tailored to cognitive modelers, already provide some evaluation abstraction support features. Such features may be underused, "advanced" features, however. We hypothesize that modelers[1] will be more likely to use such features if several facilitating conditions are present in the design of the tool. In this chapter we propose EAST (shown in Figure 4.2) to explain why modelers use or do not use evaluation abstraction features. The research in this thesis is an attempt to collect evidence supporting or refuting parts or the whole of this theory.

To design evaluation abstraction support features which can successfully elicit evaluation abstractions from modelers, the designer must choose the domain of evaluation abstractions to support, the visual representation(s) of the evaluation abstractions, and the topology for navigating among them. EAST helps by offering a prediction of what aspects of that design will influence modelers to use features. It says that the design should demonstrate the benefit to the modeler of each navigation by rewarding every navigation with some improvement to the currently displayed representations, and the design should make the abstractions underlying the resulting display reusable, shareable, and composable. It recommends that the design should lower the perceived cost of navigation by offering a small set of contextually-relevant and contextually-cued navigations to features, but also allow ad hoc access to features independent of context. Finally the design should lower the perceived risk of taking navigations by making their effects predictable and reversible.

The theory is described at the level of detail of the individual clicks and button presses that modelers make while using a tool, to allow for a fine-grained analysis and repair of interaction problems with modeling tools whose evaluation abstraction features might be underused.

We present this theory below using Sjøberg's template [111], listing first constructs, then propositions about how they relate. Sjøberg advocates the use of theory in software engineering as a better way of communicating, organizing facts, and generalizing to new situations. Shaw [108] also advocates for theory in software engineering, saying that scientific theory historically has been the key to transforming the rules of thumb and customs of artisanal and commercial practice into professional engineering disciplines. We have used EAST as a conceptual map for choosing research questions, designing experiments, designing prototypes, and analyzing data

[1] In this chapter we will refer to "modelers" for consistency, but we believe these principles will apply beyond scientific modelers to programmers in general, even professional programmers. In Section 4.1.3 we will discuss the theory's scope.

over the course of this research, and we hope that other researchers will use it to help analyze or guide design of evaluation abstraction features in modeling and software engineering tools. The theory contains many detailed constructs and propositions, as opposed to a few generalized ones, because it aims to serve in part as a concrete checklist for tool designers looking for guidance in building evaluation abstraction support into their tools.

4.1 Constructs

4.1.1 Primary Constructs

***EA*$_{Modeler}$: Modeler's Evaluation Abstraction** the set of in-the-head notions a modeler has about what a model does, could do, should do, or might do; including both correct behaviors and potential misbehaviors. Ex: "I bet it will fail in trial 6", "The activation level of this chunk shouldn't increase as it learns this task".

Evaluation Task The modeler's current in-the-head evaluation goal; what they're trying to find out. This may be ill-defined.

Evaluation A phase of modeling that happens during debugging, verification, validation, and testing, in which a modeler assesses just what it is that a model is doing. It does not necessarily have to entail a judgment of correctness.

4.1.2 Language/Tool Constructs

***Tool*: Tool that supports Evaluation Abstractions** Any tool that assists modelers with the task of debugging or evaluating models. Note that although this thesis makes use of a prototype tool designed to better support evaluation abstractions, the theory is also meant to apply to existing tools with less systematic EA support.

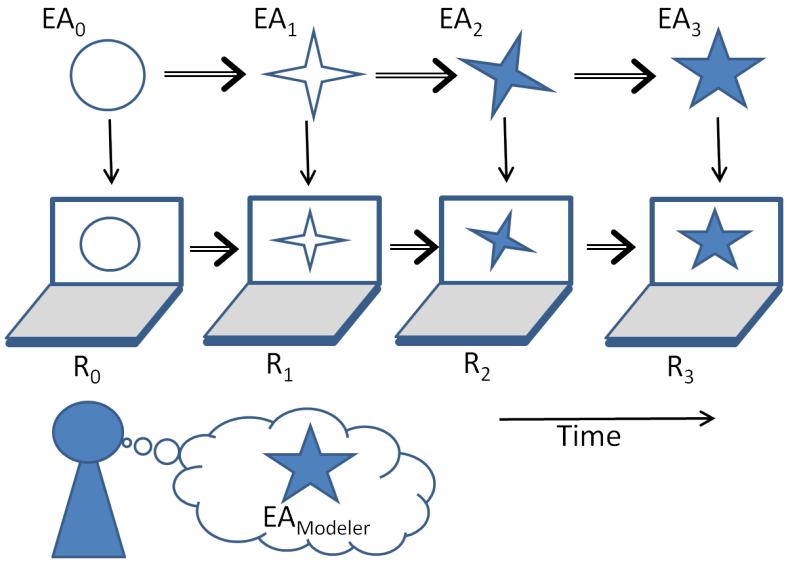
***EA*$_{Tool}$: Tool's Evaluation Abstractions** A L_{EA} expression, elicited from a modeler, and represented somehow (explicitly or implicitly) within a *Tool*.

Representation Any output of a *Tool*, visible on the modeler's screen, including visualizations as well as textual outputs like listings and error messages

Navigation An action that a Modeler takes in a *Tool* that alters the currently visible *Representation*. Ex: Clicking, picking from a menu, typing a command, closing a window.

Transformation The difference between two EA_{Tool} s that underly Representations between which a modeler has navigated.

Figure 4.1: A modeler looking at a *Representation* R_0 of some underlying EA_0 might have a question involving EA_3. To get the *Tool* to show a useful representation of EA_3, such as R_3, the user might need to perform a series of *Navigations* $Nav_1 \rightarrow Nav_2 \rightarrow Nav_3$ by using the affordances of the *Tool* (clicking, typing commands, etc); the corresponding *Transformations* $T_1 \rightarrow T_2 \rightarrow T_3$ are the corresponding changes to the EA_{Tool} underlying each current *Representation*.



Cue Words, pictures, or positional clues associated with a *Navigation* affordance in the *Tool*'s interface. Ex: the words labeling a menu option

***L_{EA}*: Language for Evaluation Abstractions** An abstract language to represent the space of possible EA_{Tool}s within a *Tool*. This may or may not be *explicitly* part of the implementation: in a *Tool* that is not built with this theory in mind, L_{EA} refers abstractly to the space of *Tool* settings or modes that cause it to produce different abstract summaries of model behavior. Distinct from a "specification" language in that specifications are prescriptive, while a L_{EA} expression (an EA_{Tool}) may denote descriptive, speculative, or hypothetical expectations. (Note: in Chapter 7 We will introduce an example of a L_{EA}, called Lea3: L_{EA} is the theoretical construct, and Lea3 is an implemented example).

Example In ACT-R's command-line interface, a displayed model trace shown in a text window (after typing a Lisp command to run the model) is an abstract representation of the model's behavior. The space of possible options for how to display that trace (e.g. settings about level of detail) comprise the L_{EA} of ACT-R-textual.

L_{EA}/Transform The concrete language for user interaction with EA_{Tool} that influences what evaluation abstractions are revealed in its visual representations. This may be a typed, textual language, an "interaction language" of menu choices, mouse gestures, and button presses, or a complex set of procedures involving multiple tools. $L_{EA}/Transform$ may be a subset of the full command set of a *Tool*. Instances of the $L_{EA}/Transform$ language describe Transformations from one EA_{Tool} to another. (Note: in Chapter 7 we will introduce an example of a $L_{EA}/Transform$, called Lea3/T: $L_{EA}/Transform$ is the theoretical construct, and Lea3/T is an implemented transformation language)

Example In ACT-R's command-line interface (ACT-R's $L_{EA}/Transform$), typing `(sgp :trace-detail 'medium)` changes the level of detail shown in the trace—in other words it requests a change to the evaluation abstraction underlying the trace representation. On the other hand, the ACT-R command `schedule-event` changes the behavior of the model itself, but not how model behavior is displayed, so it would not be part of $L_{EA}/Transform$ for ACT-R.

***EA_{Reuse}*: Evaluation Abstractions Available for Reuse** The set of EA_{Tool}s that are made available, through a user interface, for a modeler to reuse directly or to build upon.

Example For example, in WYSIWYT [21], the Forms3 spreadsheet stored judgments of validity (right, wrong, maybe right, or maybe wrong) for different definition-use pairs of cells in a spreadsheet, calculated from the users' judgments of a cell value's correctness. Each such judgment would be an EA_{Reuse} because WYSIWYT keeps these judgments and helps the user build a more complex model of correct behavior. The space of possible such correctness judgments would make up the L_{EA} of WYSIWYT. A contrasting example would be another spreadsheet tool, Goaldebug [2], which elicited and made use of a single modeler-expected value for a single cell, but did not need to save that value after its analysis was presented to the user. A Goaldebug's judgments would be an EA_{Tool}, but not an EA_{Reuse}.

***EA_{Vis}*: Evaluation Abstraction Elements Available and Visible** The subset of EA_{Reuse} that are currently visible on the screen and clickable, draggable, copyable, or in some other way directly available for reuse in constructing new abstractions. $EA_{Vis} \subset EA_{Reuse} \subset EA_{Tool}$.

Example In WYSIWYT, not all stored judgments about correctness are visible at any one time. Only a currently displayed EA_{Tool} is an EA_{Vis}.

4.1.3 Supporting Constructs

Modeler A creator or maintainer of the model

Modeling Project A model, all its changes and versions, and the potentially changing expectations modelers have about it over its lifetime.

4.1.4 Measure Constructs

The following constructs are things that could in principle be operationalized and measured quantitatively in the context of a modeler making the choice to use a particular navigational affordance of *Tool* to step from R_0 to R_1 (see Figure 4.1). Figure 4.2 shows how these relate to each other and the Propositions.

M1 Representation Ease of Use Modeler's perception of the closeness of mapping [47] between a representation and a modeler evaluation abstraction ($EA_{Modeler}$).

M2 Representation Usefulness Modeler's perception of how sufficient the information in a *Representation* is towards fulfilling their $EA_{Modeler}$.

M3 Perceived Usefulness for reuse/sharing The degree to which the *Modeler* believes a *Navigation* will produce a *Representation* that the modeler will return to more than once to get information about model behavior.

M4 Perceived Usefulness for composition The degree to which the modeler believes a representation will be a useful intermediate next step in building an EA_{Tool}, or will be a useful place to start in future *Navigation*s.

M5 Context-relevant Cueing The degree to which the set of *Navigation*s currently available to the *Modeler* are both contextually appropriate (i.e. lead to expressions of L_{EA} that are syntactically valid and likely to be useful) and contextually labeled (i.e. in a way that communicates a preview of the *Navigation*'s output in the current context.)

M6 Context-independent Cueing The presence of *Tool* affordances for building EA_{Tool}s using stable affordances that are always available and labeled the same way regardless of context.

M7 Low outdegree Outdegree refers to the number of *Navigation*s possible from an EA_{Tool}; this measure varies inversely with outdegree; in other words, the measure is high when there are few navigation paths.

M8 Visual Linkage The presence of *Tool* features that help *Modeler* understand the relationship between the *Representation* they are looking at and the one they arrive at after *Navigation*.

M9 Familiarity with transform The degree to which the transformation underlying each *Navigation* corresponds to transformations the *Modeler* is already familiar with.

M10 Experience with transform The amount of past experience a *Modeler* has with a *Tool*'s features.

M11 Undo The presence of a feature allowing a *Modeler* to retract a *Navigation*.

M12 Evaluation Thoroughness The thoroughness (e.g. frequency or variety of checks made) with which a *Modeler* evaluates a model's behavior .

M13 Probability of Navigation The likelihood that a *Modeler* performs a particular *Navigation*.

M14 Perceived Benefit of Navigation The overall amount of benefit a *Modeler* predicts/ perceives that would come from performing a *Navigation* in the *Tool*.

M15 Perceived Cost of Navigation How costly (in terms of time and working memory load) a *Modeler* perceives it to be to perform an available *Navigation* within a *Tool*'s interface.

M16 Perceived Risk of Navigation How risky (in terms of expected time and working memory load cost, if things go wrong) a *Modeler* perceives it to be to perform a *Navigation* within a *Tool*'s interface.

M17 Comprehension of Model The breadth and correctness of a *Modeler*'s understanding of a model's behavior.

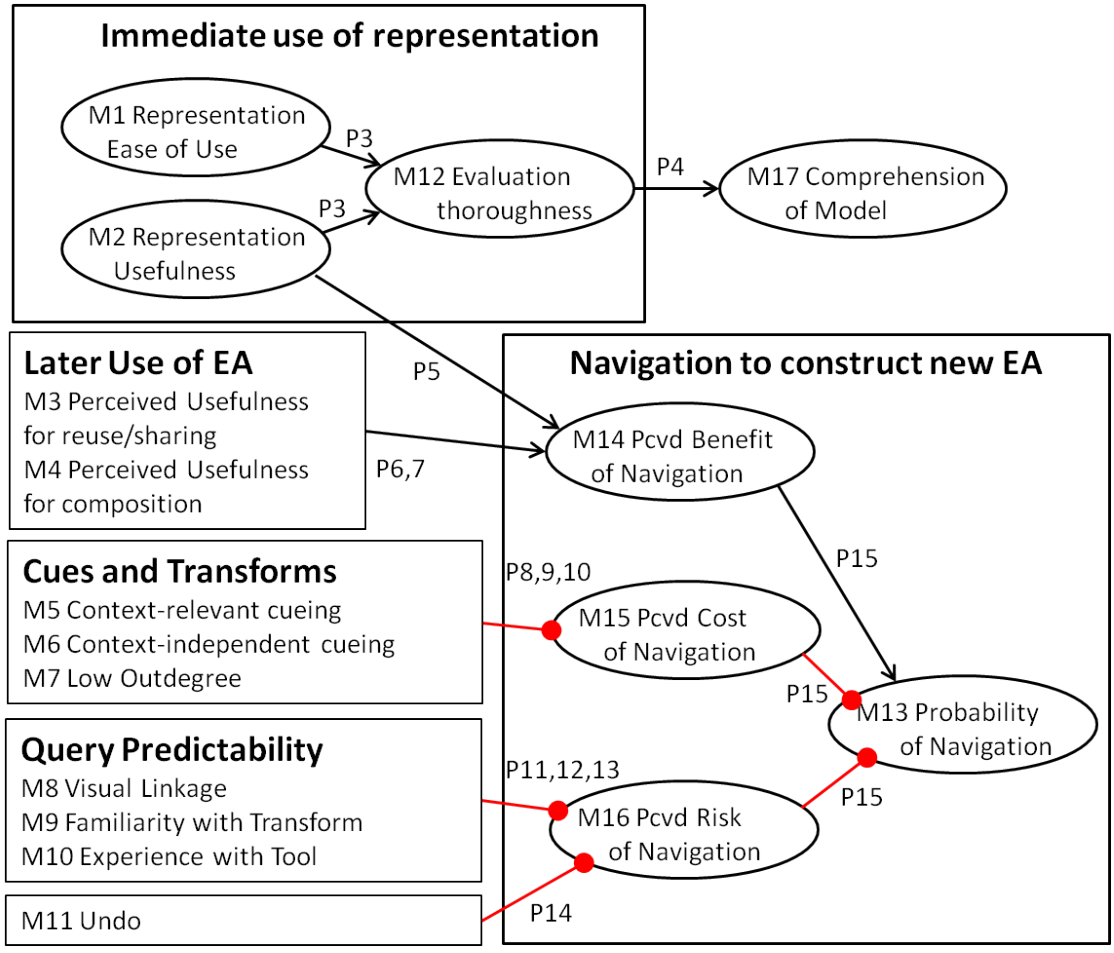
4.2 Propositions

Propositions are testable claims about the constructs. Figure 4.2 shows how the propositions and measure constructs interrelate.

Some of the following propositions rely only on psychology of programming theory for their support, and might therefore apply to programmers in general, not just modelers. In the points below, when the arguments below apply specifically to some particular class of users, we will state a (hypothesized) scope for the proposition.

P1 An evaluation task entails one or more modeler evaluation abstractions ($EA_{Modeler}$).

Figure 4.2: Relationships among measure constructs: the diagram depicts the situation around a modeler's decision to perform a navigation that creates a new evaluation abstraction. Arrowheads indicate positive influence; circular connectors indicate inhibition. Of particular interest is the contrast between the "Immediate use" and "Later use" motivations for making a navigation: these predict that modelers will navigate because they expect to learn something immediately, and/or they are planning ahead to either build something more complex or reuse the resulting representation in some future situation.



Explanation Modelers may employ evaluation abstractions above and beyond the abstractions evident in source code, since they write models with the hope of drawing some conclusions that are not self-evident from the mere statement of what the model is. Those hoped-for conclusions would be evaluation abstractions.

Scope P1 may also be true for professional programmers, since artifacts associated with evaluation abstractions exist, such as unit tests, assertions, and type declarations.

P2 Some modeler evaluation abstractions ($EA_{Modeler}$) persist across evaluation tasks, across modelers on a modeling project, and even across modeling projects (i.e. they have differing scopes).

Explanation EAs may persist for several reasons. First, modelers may have scientific hypotheses they are addressing, that they aim to answer by running code. It follows that part of the hypothesis might only answered by looking at the behavior of the model, not the raw code. Thus it makes sense that the evaluation abstractions that the hypotheses embody should be of enduring interest to the modelers.

Secondly, modelers learn about their models' behavior: they do not ask the same questions over and over, but form new ones based on what they have learned. The fact that they do not repeat the same questions could be evidence that they remember the answers to earlier questions, and have incorporated those answers into their persistent understanding of the model, as persistent evaluation abstractions.

Thirdly, modelers talk to each other about their hypotheses as they collaborate – both explicit research questions, and subsidiary hypotheses that they develop along the way. Those hypotheses are evaluation abstractions, and so it is likely that evaluation abstractions will be shared among a team.

Scope We believe the arguments above will apply to anyone doing programming to answer hypothetical questions, so this would imply that P2 is at least true of scientific modelers. Professional programmers also deal with hypotheses [73] but it is less clear how persistent these are: perhaps a programmer forgets hypotheses after completing a task and writing a unit test or assertion about it. If so, evaluation abstraction support might be useful to professional programmers, but P2 might not be true for them.

P3 A modeler will evaluate a model more thoroughly and check its properties more often (*M12 Evaluation Thoroughness*) if representations provided are perceived to be useful (*M2 Representation Usefulness*) and easy to use (*M1 Representation Ease of Use*).

Explanation Usability and usefulness contribute to behavioral intention to use, as implied by the Technology Acceptance Model [32].

P4 If modelers evaluate their models more thoroughly (*M12 Evaluation Thoroughness*) they will comprehend their models' behavior better (*M17*) over the course of a modeling project.

Explanation If more aspects of a mental model are verified against reality more often, there are more chances for contradictions to come to light. More exposure to visualizations that make manifest the abstractions programmers are interested in might make it more likely that they will catch a discrepancy.

P5 *M14 Perceived Benefit of Navigation* is increased by *M2 Representation Usefulness* of the *Representation* that constitutes the *Tool*'s immediate response to the *Navigation*.

Explanation It seems plausible that getting an immediately useful visualization as a byproduct of entering an evaluation abstraction into a tool would help motivate a user, even if their *primary* expected benefit in entering the abstraction was something else (for example, if they were just doing one step of construction of a more complicated abstraction, or if they were experimenting with the tool to learn how to use it).

P6 A modeler's perception that a representation will be useful later for themselves or other modelers (*M3 Perceived Usefulness for reuse/sharing*) will contribute to their perception that the representation is beneficial (*M14 Perceived Benefit of Navigation*).

Explanation Evaluation abstractions are likely to persist over time and between modelers. It is likely that modelers are aware of this, and will explicitly value and plan for the reuse and sharing of abstractions as they are navigating and building evaluation abstractions.

P7 A modeler's perception that an EA_{Tool} will be composable (*M4 Perceived Usefulness for composition*) will contribute to their perception that the representation is beneficial (*M14 Perceived Benefit of Navigation*).

Explanation Modelers may be willing to take multiple navigations to get to an inherently useful representation; if so, it is plausible that they see the steps themselves along the way as also beneficial.

Additionally, by analogy with bottom-up programming practices in which programmers value small programs for their ability to be assembled into larger programs, it may be that modelers would value an evaluation abstraction if they knew it to be a part of many likely future evaluation abstractions.

Compositions that physically bring related data together into the same display can be extremely valuable to modelers: Plumlee and Ware [97] showed that users use less working memory and make fewer errors when they can move their eyes back and forth between related data rather than having to navigate a GUI to move back and forth.

P8 *M15 Perceived Cost of Navigation* is inversely related to *M5 Context-relevant Cueing* in the current *Representation*. In other words, modelers are more likely to enter evaluation abstractions that are about things they can see represented on the screen.

Explanation Attention investment theory uses time as its currency unit, when stating that people make an economic-like decision in balancing perceived risk, benefit, and cost of action. We hypothesize that a similar process might also be at work regarding the scarce resource of working memory. Working memory is known to be limited [84], and rational analysis theory suggests that people manage their cognitive resources in a way that is boundedly optimal [5]. Together these imply that while performing a working-memory-intensive task, people will be averse to starting a subtask that requires putting unrelated chunks into working memory.

This is why we hypothesize that people will perceive menu options as less costly if they recognize the cues on those options as concepts they already have in working memory. In other words, they might find the operation they need without having to mentally shift gears and think about it. Designing for recognition rather than recall is commonly cited interface design advice (e.g. [107], p. 686).

Additionally, Rosson and Carroll's [25] minimalist learning theory says that active learners are reluctant to set their task aside to make a time investment learning something new.

Finally, in models and tools based on information foraging theory, the constructs of scent (a person's assessment of how useful information at the end of a navigation is) and cues (words associated with a navigation used to assess scent) have often been modeled using metrics of similarity between cues and the words taken from the content at the destination, since people were often interested in pursuing specific words. This has been true in in modeling users' behavior on the web [27, 44] and while debugging code [74].

Scope These arguments would apply to anyone doing a working-memory-intensive task

P9 *M15 Perceived Cost of Navigation* is also inversely related to *M6 Context-independent Cueing*; modelers are more likely to enter EA_{Tool}s when there are stable affordances for them that are *not* dependent on context.

Explanation This does not contradict P8; the theory claims that both context-relevant and context-independent access to evaluation abstraction support features are necessary. Context-independent access may be necessary for situations in which the modeler's context is not available to the tool; for example if they are working from documents external to the

tool. In that case, a modeler may perceive a lack of context-independent cues as costly, because they might have to artificially create an appropriate context in order to access a feature.

Cao et al. [23] noted this need in their design for the Idea Garden, which offered contextualized tool tips based on user data. They found that they needed to also place the entire set of decontextualized tool tips in a permanent menu, so that users could consult them without having to reconstruct an appropriate context.

P10 *M7 Low outdegree* can reduce *M15 Perceived Cost of Navigation*; in other words, modelers are likely to perceive cost of *Navigation*s as lower if there are fewer of them available from the context of the *Representation* of some EA_{Tool}.

Explanation Attention investment theory includes a notion of *prospecting cost*: that users take into account the perceived cost of exploring the available options when weighing a decision. More available navigations imply higher prospecting costs, and so EAST proposes that offering too high an outdegree from nodes in the *Tool*'s topology may increase perceived cost and discourage exploration of unfamiliar features.

P11 A modeler will come to perceive a *Navigation* as less risky (*M16 Perceived Risk of Navigation*) if, on taking it, the interface does something to show how the new *Representation* displayed relates to the previous one (*M8 Visual Linkage*), thereby helping them predict its behavior in the future.

Explanation This is a common and helpful technique in information visualization; for example the Eclipse debugger highlights values in yellow that change in the variable inspection window at each step. The Protege data visualization toolkit has facilities for generating animations morphing between different *Representation*s. It may be that the design of a L_{EA}/*Transform* will lend itself to easier demonstration of such linkages.

We associate unpredictability with risk because an unknown, unpredictable transformation of existing data may seem risky to a modeler, if there are many such items, and the time to find the correct one could add up if they have to try them all; or if they believe that it will take a significant amount of time to puzzle out what has happened after an unknown transformation.

P12 If a transformation is similar to ones modelers are familiar with from experience with the same or analogous domains (*M9 Familiarity with transform*) they will be more able to understand and predict the outcome of the transition, thus perceiving it as less risky (*M16 Perceived Risk of Navigation*).

Explanation Modelers are likely to have other tools for working with data, such as spreadsheets, databases, and text editors. They also likely have habits and practices in working with data manually, such as visual scanning, scrolling, or jotting down intermediate results of calculations. It seems likely that modelers would understand transformations and activities transposed to a new tool more readily than they would understand novel transformations.

P13 A modeler who gains experience using a particular *Navigation* and L_{EA}/*Transform*s in a tool (*M10 Experience with transform*) will perceive the effects of the *Navigation* in the future as less risky (*M16 Perceived Risk of Navigation*).

Explanation Modelers learn from experience to predict operations they have perceived before, and their understanding of the costs and benefits become more certain.

P14 A *Modeler* will perceive an undoable feature to be less risky (*M16 Perceived Risk of Navigation*).

Explanation Providing undo to mitigate risk is well-established usability advice. [24].

P15 *M13 Probability of Navigation* is a function of *M14 Perceived Benefit of Navigation*, *M15 Perceived Cost of Navigation*, and perceived risk of *Navigation*.

Explanation This is implied by TAM, the technology adoption model [32]. Attention investment theory [11] also suggests that modelers will use evaluation abstraction support features if they perceive it to benefit them.

Chapter 5 – Understanding the Modelers[1]

What kinds of evaluation abstractions did cognitive modelers employ in real-life projects? What were these abstractions like, and what roles did they play within the projects? My coauthors and I [13] did a case study of six cognitive modeling projects. We spent a month listening to a group of cognitive modelers at the Air Force Research Laboratory (AFRL) in Mesa, Arizona as they debugged their models and discussed them with their colleagues. The goal was to gather commonalities across projects in the ways modelers use abstraction to evaluate and explain their models.

5.1 Cognitive modelers' world

The cognitive modelers in our study used ACT-R [5]. Modelers using this language are a particularly appropriate population in which to study 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 [91]; that is, 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 wish to force production rules to fire in a particular order, but instead attempt to set rule preconditions such that the rules 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 of the difficulties scientific modelers face generally when evaluating models. 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 [78].

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

[1] This chapter contains material previously published in VL/HCC, 2010, as "Does my model work? Evaluation abstractions of cognitive modelers" [13].

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

5.2 Case study design and methodology (NP+ Step A)

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 [124]. 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 Laboratory. We studied these modelers over the course of a month. The elements of interest were evaluation abstractions. Evaluation abstractions, as defined in Chapter 4, 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?

5.3 The Models and Modelers

The projects we used as cases are listed in Table 5.1. The first four cases were past or ongoing projects at AFRL; the modelers involved are referred to with pseudonyms. The last two cases drew on tutorial materials from the ACT-R documentation.

VISLANG was "Steve's" doctoral thesis work to demonstrate the impact of visual scenes on language comprehension. It modeled eye movements of a person listening 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 was able to 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 5.1) 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.

Finally, we included as cases two projects that were exercises from the ACT-R tutorials [16]: **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 [110] when asked to add small integers. ZBRODOFF modeled a "letter addition" experiment [125]. For example, given "A+4=E" it should respond by pressing a key indicating "true", because "E" is four letters beyond "A" in the alphabet.

Table 5.1: Projects and Modelers used in the case study

Modelers	Model Name	Size
Steve	VISLANG	64 rules, 26 chunk types
Gary	PILOT	160 rules, 30 chunk types
John and Ellen	LANGCOMP	540 rules, 360 chunk types
Mitch and Matt	SCANTYPE	19-25 rules, 4 chunk types
ACT-R Tutorial	SIEGLER	7 rules, 2 chunk types
ACT-R Tutorial	ZBRODOFF	9 rules, 3 chunk types

5.4 Data and coding procedure

The data for each model consisted of 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 [66]. 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,

Figure 5.1: A SCANTYPE task screen as human participants saw it in a human psychological study performed prior to modeling (upper left), as the SCANTYPE ACT-R model, meant to mimic those human participants, saw it (upper right; the red circle is where the ACT-R is "looking"), and as ACT-R's visual location buffer saw it (bottom).

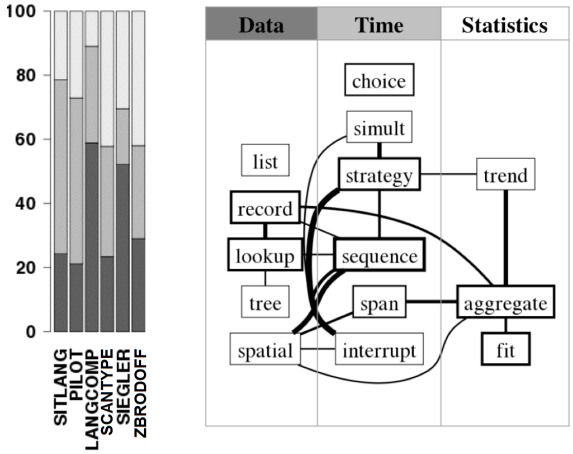
Loc	Att	Kind	Value	Color	ID
(24 60)	NEW	NIL	ONE	BLACK	CHAR-PRIMITIVE0
(24 110)	NEW	NIL	HORIZONTAL	BLACK	CHAR-PRIMITIVE3
(24 110)	NEW	NIL	SYMMETRIC	BLACK	CHAR-PRIMITIVE1
(24 110)	NEW	NIL	VERTICAL	BLACK	CHAR-PRIMITIVE2
(24 110)	NEW	NIL	DIAGONAL	BLACK	CHAR-PRIMITIVE4
(64 60)	NEW	NIL	TWO	BLACK	CHAR-PRIMITIVE5
(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

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.

5.5 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.

Figure 5.2: (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.



As Figure 5.2 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.

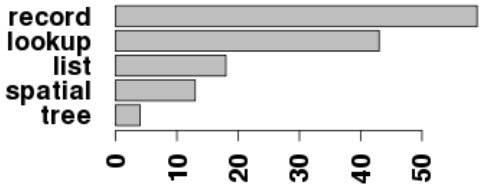
5.5.1 Data structure evaluation abstractions

First, we discuss modelers' *Data Structure Abstractions*—relationships between individual data items at a single point in time. As mentioned in Chapter 1, ACT-R and other modeling environments mostly only support modelers' efforts to evaluate their models in terms of their model's low-level data structures. Modelers in this study did take advantage of such affordances, 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 kinds of data abstractions (Table 5.2) modelers needed to evaluate was large, requiring that modelers do some kind of mental or programmatic translation.

5.5.1.1 The types of evaluation data structures

Table 5.2 shows the five types of data structure evaluation abstractions we found, and Figure 5.3 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.

Figure 5.3: Counts of Data Structure codes.



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 5.1 shows, at top right, a screen that was shown to the SCANTYPE model, and at bottom, how SCANTYPE represented it internally it as lists of items with coordinates. However, when the model's code copies those numbers into other locations, the ACT-R debugging tool no longer associates the values with coordinates. So even if modelers still think of them as coordinates, the tool only lets them view the values as numbers. Thus, as in other languages, if modelers want to know how items relate spatially, they must do the work to graph them.

5.5.1.2 The translation problem

The modelers' work to translate from model data structures to evaluation data structures was hard, but the mismatch leading to the need to translate is a necessary consequence of the task of evaluating a model.

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 a named 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 not on the name, but on the contents and computed "activation values" of all the chunks that were candidates for retrieval.

Table 5.2: Data Structure Evaluation Abstractions, in order of frequency.

Abstraction	Definition	Sample Quote
Record	Item made up of multiple parts.	*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	*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	*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	*Matt: In other words* ***this*** *one [pointing to the screen], [. . .] it would find it very quickly.*
Tree	Hierarchical Knowledge	*Ellen: I have a feeling that "meet" was retrieved; it just didn't make it into the tree.*

The mismatch we refer to is the fact that in ACT-R's GUI a user must click on a chunk name to see the contents, but John needed to look up chunks by content and by activation value. 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.

5.5.1.3 Abstractions of Abstractions

The examples discussed so far each examined a single kind of evaluation abstraction in isolation, but as Figure 5.2 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

Figure 5.4: 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.

```
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-SLOWLY
```

pointed to on the screen), with a *time sequence abstraction* (three events in sequence: a shift, an arrival, and a read; *sequence abstractions* are discussed in Section 5.5.2):

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 5.4, for example). The trace 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.

5.5.2 Time Abstractions

Time clearly mattered to our modelers when they evaluated their models. Recall from Figure 5.2 that all six projects used time evaluation abstractions. Although time constraints were not explicit in any of the models' source code, modelers used time abstractions to check high-level patterns as emergent behavior. Gary explained the importance of not programming sequences explicitly into his model during a Q&A after a talk he gave:

John:. . . you can have a declarative memory chunk that's actually a sequence of goals that allows you to prefer—

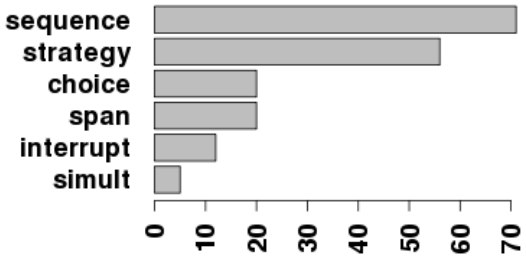
Gary (interrupts): Yeah, but that's the type of thing I want the model to learn though, this sequence of goals; I don't want to build that in. . .

5.5.2.1 Manual evaluation of sequences

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 5.4. 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.

Figure 5.5: Time evaluation abstractions.



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 5.3), so that it could find events that mattered, but only if they occurred in sequence with unrelated intervening details abstracted away.

5.5.2.2 Tracking models as they strategize

Although Figure 5.5 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.

Table 5.3: Time Abstractions in order of frequency

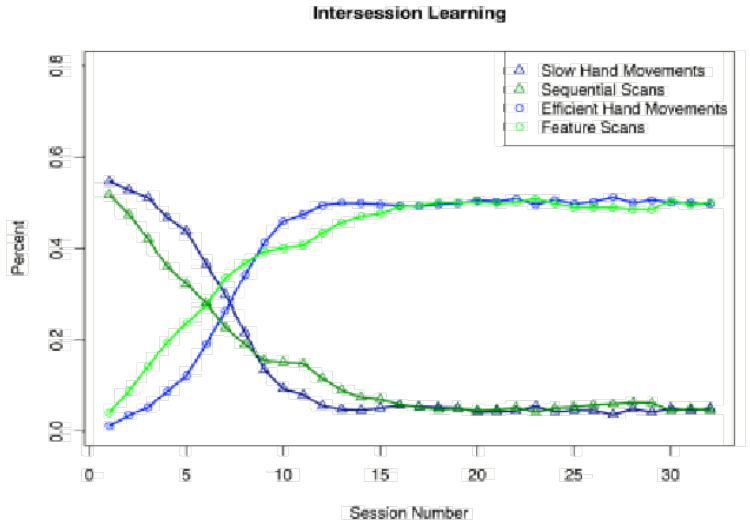
Abstraction	Definition	Sample Quote
Sequence	B will occur after A	*Matt: It starts at the far left, it shifts attention to [the] square, [to the] plus, to the three, to the U; just left to right serially, until it finds the one it's looking for.*
Strategy	Joint activity of related rules	*ZBRODOFF: subjects have to engage in counting.*
Choice	Either A or B will happen	*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	*Steve: And then after a short interval there's an indication of the correct or actually described reference.*
Interruption	Stopping or pausing a strategy	*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	*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.*
Timing	A will occur at time B	*Steve: About 300 ms into hearing "leftmost," you can see the precipitous drop-off in red.*

Strategies could be interrupted by other rules preempting them, could be simultaneous when rule firings were interleaved, or could 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. Figure 5.6 shows a graphic he made by plotting firing frequencies of four rules, representing four strategies.

5.5.2.3 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 told us that he originally designed PILOT using a formal task description language, NGOMSL [59]. Unfortunately, Gary's NGOMSL description existed only as documentation; the only way to check that it was

Figure 5.6: Proportions of different strategies in SCANTYPE: The graph tracks *strategies* by using instances of representative rules firing; it also demonstrates modelers' interest in *trends*.



being followed by PILOT was very detailed manual 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.

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.

5.5.3 Statistical Abstractions

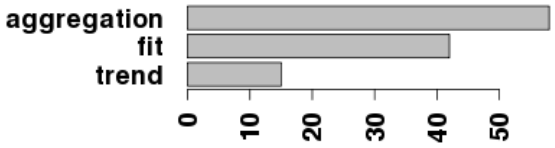
Perhaps the most distinct from traditional programming abstractions were the statistical evaluation abstractions. These were ways of evaluating model performance in terms of aggregation, trend, or fit to human data (Table 5.4 and Figure 5.7). For all but one of the projects, at least 20% of the evaluation abstractions coded were in the statistical category.

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 turn to other software. Specifically, they had

Table 5.4: Statistical Abstractions in order of frequency

Abstraction	Definition	Sample Quote
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.*

Figure 5.7: Counts of Statistics Evaluation Abstractions



to write Lisp code to collect numeric data, export the data to external files, and then use or write other software to process those files. 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.

Mitch had depicted this change over time of "getting more familiar" by graphing frequency of certain rules firing over time, and showing that one kind of rule increased while the other decreased (Figure 5.6). This was produced by 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.

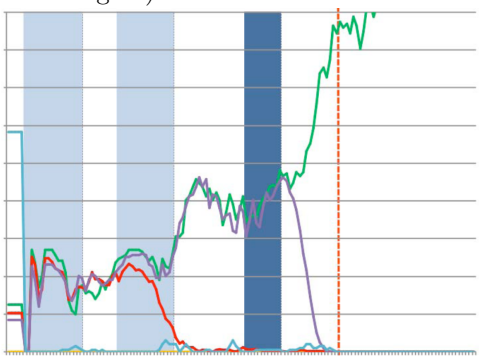
Compared to the other types of evaluation abstractions, statistical evaluation abstractions seemed to exist in later phases of model development. We observed modelers discussing them in their presentations more often than when they were working directly with the models. Our interpretation is that modelers wanted to evaluate in terms of individual data structures and time behaviors at first, when debugging the detailed behavior of models; then later evaluating in

terms of aggregates, trends, and fit after they had some confidence that the model was working as they intended.

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 debugging to do, and thus fewer data structure and sequence details to evaluate. At the other end of the size spectrum, the largest project, LANGCOMP, talked very little about statistics. Modelers were still working to get individual sentences to parse correctly, rather than exploring the broader implications of a stable model of language parsing. (Refer back to Figure 5.2 for project-by-project use of the different abstractions.)

Statistical evaluation abstractions were built on other evaluation abstractions. For example, the caption in Figure 5.8 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* over that *aggregation*. Figure 5.2 shows that statistical evaluation abstractions were also linked to the time and data abstractions *record*, *span*, *spatial*, and *strategy*.

Figure 5.8: Steve's VISLANG graph combines evaluation abstractions of time (*sequence*: blue stripes and red dashed line show when words were heard and mouse was clicked; *span*: width of blue bars), data structure (*spatial*: colors of trend lines indicate the screen region where eye fixations occurred), and statistics (*trend*: colored lines; *aggregation*: vertical axis represents frequency of eye visits per screen region).



5.6 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 their models 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 modeling tools such as IDEs and 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 modeling 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 5.5). These operations correspond fairly well to the kinds of operations advocated for abstractions in other settings (e.g., Shneidermans proposals for information visualization research [109]), which suggests that modelers evaluation maneuvers are consistent with other situations in which full-fledged support for abstractions is accepted as being desirable.

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 5.5.2.3 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 informally specified 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 even in the case of ad hoc questions about model behavior, modelers also repeatedly looked for the same type of information. The regularity with which they did so suggests that their habitual manual 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 program comprehension tools for modelers. Without such support, modelers ex-

Table 5.5: Operations on evaluation abstractions.

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.

pended considerable effort to perform these operations manually or with tools they had to create themselves.

5.7 Implications for Theory

The evaluation abstractions we identified and the operations on them are the basis of several constructs of the EAST theory (Chapter 4) and the Lea3 and Lea3/T languages (Chapter 7). The case study as a whole addressed EAST's proposition P1 that evaluation abstractions exist in evaluation tasks; but additionally:

Compare Modelers interest in comparing instances of evaluation abstractions motivated the basic database-like design of Lea3's semantic domain (Section 7.1). Storing multiple matches side-by-side in a data structure makes it easier to display them side-by-side in visualizations for easy comparison. *M8 Visual Linkage* relates to the visual transitions between visualizations before and after a navigation occurs; these can also help with comparison.

Visualize EA_{Vis} and *Representation* are central to EAST theory, and the *M2 Representation Usefulness*, *M1 Representation Ease of Use* measure constructs reflect the fact that visual inspection of evaluation abstractions is the primary use we saw for them (although there were other uses as well, such as regression testing).

Navigate The navigation-related constructs in EAST (*Transformation*, *Navigation*, *M14 Perceived Benefit of Navigation*, *M15 Perceived Cost of Navigation*) specifically refer to navigations that define new evaluation abstractions. This case study identified other kinds of navigations

that would be matters of design in an environment but not specifically prescribed by EAST: navigation among instances of a particular abstraction, and among already existing abstractions.

Compose EAST's measure construct of *M4 Perceived Usefulness for composition* is actually broader than the *compose*' operation described in this chapter. In this case study composition refers to drawing two sources of information together; but in the theory we extend the notion to also include composing built-in operators with queries to achieve new queries.

Filter L_{EA}'s `filter` operation allows for the most common, but very limited, kind of filtering. The `filter` and `calc` operators together can perform the same filtering capabilities as SQL's where clause allows.

Persist The persistence of modelers' evaluation abstractions over time motivated the *M3 Perceived Usefulness for reuse/sharing* construct of EAST, and its proposition P2 that evaluation abstractions persist across tasks, modelers, and projects.

Other motivations of EAST theory constructs and propositions were explained in Section 4.2.

5.8 Conclusion

The case study revealed a richly interconnected network of evaluation abstractions involving data structures, time sequences, and statistical aggregation. The most important findings were 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. It takes us a step closer to filling that gap by supplying enough detail about the constructs and relationships in their descriptions of evaluation abstractions to satisfy step B1 of the NP+ process. In the next chapter we will move on to step B2, the collection and analysis of detailed sequences of evaluation abstractions in the context of a task.

Chapter 6 – Empirical language design[1]

Chapter 5's case study identified the content of evaluation abstractions; now in this chapter we turn to the question of how modelers build up and apply these abstractions to the task of evaluating models.

In order to design evaluation support appropriate to the needs of cognitive modelers, we set out to investigate how modelers created and manipulated abstractions while they were doing evaluation and debugging tasks. Study 1 (Chapter 5) characterized such abstractions at a high level [13], but we wanted a precise description of the constructs, relationships, and interaction sequences a model evaluation tool would need to support such abstractions (i.e., a useful design specification for a new model evaluation interaction language).

This chapter presents:

- Empirical evidence about the evaluation abstractions requested by cognitive modelers, and how those abstractions were sequenced over time.
- An empirically derived design specification for an evaluation abstraction interaction language for cognitive modelers.
- An initial case study of Natural Programming Plus (Section 3.1.2), used to precisely capture and validate the structure and flow of ideas expressed by the participants.

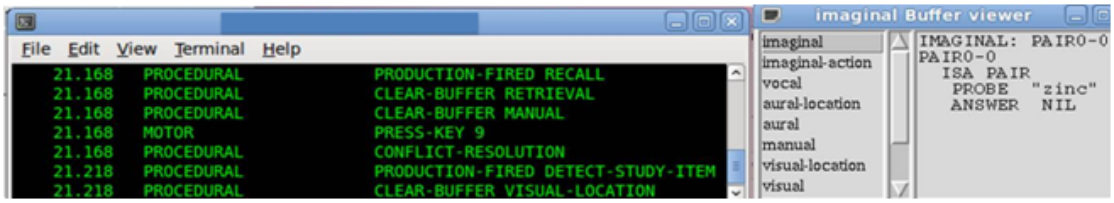
The first two contributions also serve as initial data points towards an understanding of the potential of NP+ as a methodology of wider interest, a prospect which we will discuss along with the validation of this specification in the next chapter (Section 7.5).

6.1 Methodology

Informed by a taxonomy of evaluation abstractions and operations identified in Chapter 5, we conducted two studies to identify the ways cognitive modelers went about evaluation tasks. The combined goal of these two studies was to identify the concepts and relationships behind modelers' information requests, and how they were sequenced in time, as required by Steps B1 and B2 of NP+.

[1] This chapter contains material previously published at CHI, 2012, as "Designing a debugging interaction language: an initial case study in Natural Programming Plus" [14].

Figure 6.1: 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".



6.1.1 Study 2N: Participants and Methods

Study 2N ("N" for native environment) was an observational lab study whose aim was to elicit modelers' information-seeking language and approach for evaluating an ACT-R simulation's run-time 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 [16] 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 6.1. Participants talked aloud as they worked, and we video-recorded their sessions.

6.1.2 Study 3W: Participants and Methods

Although Study 2N gave us a good sample of relatively natural debugging behavior, the data was sparse for some categories of evaluation abstractions that we had seen in Study 1.[2] 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

[2]See Section 6.4 for an analysis.

either guess at their answers based on scant evidence, or set 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 3W ("W" for Wizard of Oz), we designed an experiment to observe just one evaluative subtask 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 2N. 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 3W results to pure trace inspection behaviors, and the results should be interpreted in conjunction with more natural observations, such as Study 2N and Study 1.

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 2N. Participant IDs are prefixed with "N" for Study 2N and "W" for Study 3W.

In selecting a model for the task, our criterion was that it should present challenges similar to questions we saw Study 2N modelers pose, but that the modelers 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 3W was a defective solution to one of the modeling exercises in the ACT-R 6.0 tutorial [16], 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 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 6.1 (Left) shows a few events of that trace. We set our tool's initial display to the same output as the ACT-R tutorial.

Table 6.1: Tasks posed to modelers in Study 3W, designed to be similar to questions that had caused participants difficulty in Study 2N.

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 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?

Study 3W's participants' tasks were to find answers to the questions listed in Table 6.1, designed to be similar to questions that had caused participants difficulty in Study 2N, while at the same time emphasizing evaluation abstraction types that were underrepresented in Study 2N ("choice" and the "statistical EAs" category, particularly) to round out our coverage of the Study 1 codes, and to ensure that we would collect data around the evaluation abstractions that were the least supported with existing tools.

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 specification 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-*).

6.2 B1/B2 Results: The modelers' abstractions

The modelers' abstractions that we observed in Study 2N and Study 3W consisted of constructs that fell into four categories: trials, events, states, and rule text. In Study 1's taxonomy, trials and states fall into the category of "span", rule text was treated as a "record", since it was

a data item with constituent parts, and events were treated as an atomic item which "Time" abstractions tied together.

6.2.1 The trial

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 [123].

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 models (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.

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 2N 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 3W, in the form of a two-paned window that let participants choose trials in one pane, and see the details in the other (Figure 6.2). Study 3W modelers made heavy use of them: trials were at the root of about half (75 of 133) of all requests in Study 3W (Table 6.2). This detailed view enabled Study 3W 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 asked the Wizard to add summarized facts about each sequence as attributes to each row of the trial listing (e.g., Fig-

Table 6.2: Types and counts of abstractions that Study 3W modelers queried in Study 3W as they worked. (In Study 2N, all modelers drew on all four categories of data.)

Abstraction (instance count) and Participant request example	(Small portion of) result of the request

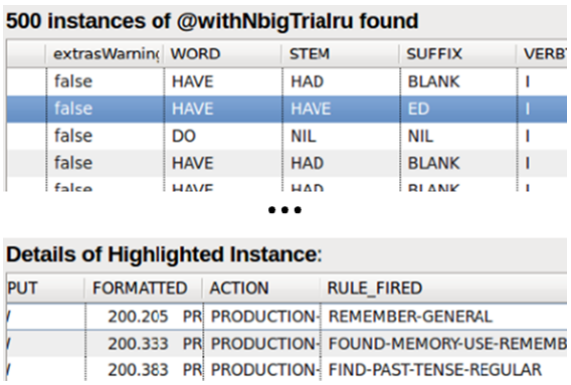
Trial (75): The begin and end time of a trial, and several model-specific attributes.
W413a: All the trials where the verb is HAVE [...] I would like to see what the stem is
	`trialnum: 2`
	`start_time:200.155`
	`word: ‘‘HAVE’’`
	`stem: ‘‘HAD’’`
	`end_time: 400.383`
	`[... other trials... ]`

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)	

Figure 6.2: W412b asked for "the production firing sequence" (below) "... within this trial" (above, highlighted).



ure 6.2, 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 needed 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).

6.2.2 Events vs. States

Modelers' second and third most common constructs were model events (behaviors) and model states (data). Study 2N 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 3W modelers also showed strong interest in events and states (Table 6.2), 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.

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).

6.2.3 All about rules

All participants in Study 2N kept a window open all the time showing rule text. To avoid gathering redundant data on code inspection, we allowed Study 3W participants to see only the names and dynamic behavior of rules, but not their text. Still, in twelve (9%) of Study 3W'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.

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, but because of a bug in the model a string for a different purpose was being interpreted as the name of the key to press). 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 3W, 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 ACT-R's standard trace output shows only the names of rules that fired (e.g., in Figure 6.1 (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 a mechanism in ACT-R called "production compilation").

Implications for Rule Information: (I-RULETEXT): Unsurprisingly, the modelers needed to see rule text. They also 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.

6.3 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 6.3 lists the operations that made up these queries, and their relationship to the constructs and operations described in Study 1.

Table 6.3: Query-building operations in Study 3W and their ties to Study 1 constructs. "item" means an event, trial, or any other abstraction.

Operators for composition		**Ties to Study 1**
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.	*Time* abstractions; *compose* operation
Slicing and dataflow constructors	Produce a backward dynamic slice through code or a backward flow of data through data structures.	*compose* operation
Operations for summarizing, filtering, or rearranging		
Filter	Limit the items shown.	*filtering* operation
Distinct	List and count distinct values of some attribute.	*aggregate* abstraction
Set	Do set operations on distinct results.	*compare* operation
Sort	Rearrange items in order by some property.	*list* abstraction
Operations for comparing details		
Any, First, Last	Produce any, the first, or the last, respectively, single item with the specified properties.	*list* abstraction
Visualize	Produce a graphic (e.g., a bar chart) of all items with the specified properties.	*Visualize* operation

6.3.1 Operations for composing queries

6.3.1.1 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 was stepping through a model in ACT-R's debugger, and wondered how an event related to something that had happened earlier in the simulation. He restarted a run and painstakingly stepped forward to the "earlier" time he was curious about. By the time he found it, his previous run was no longer in the scroll buffer of the window where his trace was displayed:

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 3W 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 2N 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.

6.3.2 Operations for summarizing

Study 2N 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.

6.3.2.1 Filtering

One "fire hose" of data was the declarative memory dump. In the Paired task, most of the Study 2N 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 2N modelers, we provided a more general filtering capability in Study 3W, 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.

6.3.2.2 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.

6.3.2.3 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: the lack of data in the query's output 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.

6.4 Discussion: Evaluation Abstractions in debugging tasks

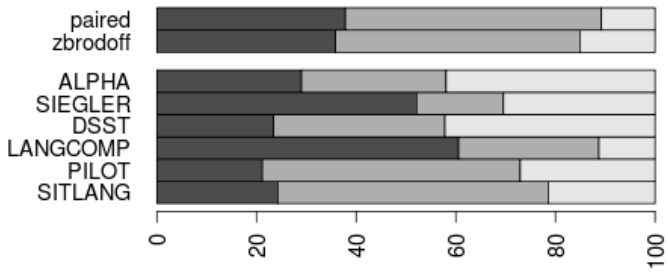
In this chapter we introduced Study 2N to deepen understanding of evaluation abstractions after the case study of Study 1, but how similar were the abstractions, really, between the two activities of scientific discussion among humans and the human-computer interaction of debugging? As a separate analysis we performed the same coding on Study 2N as we did with the Study 1 transcripts, using a random sample of 100 (out of 622) 30-second transcript segments from Study 2N transcripts .

What we found was that Study 2N had very little in the way of statistical evaluation abstractions (Figure 6.3). Figure 6.4 shows that when modelers in Study 2N did use statistical evaluation abstractions, it was almost entirely aggregation, not fit or trend that they talked about.

Interpretation The rarity of *fit* and absence of *trend* makes sense because Study 2N was conceived as a debugging task, not a model fitting task. That is, in preparing the experiment we altered the two models in a way that made the models carry out their tasks *incorrectly*, not just in a way that subtly mismatched empirical data.

One other interesting difference in Figure 6.4 was the near absence of discussion of "choice" among the data evaluation abstractions. In Study 1 the code came up when modelers talked about the overall design of their model, and how different mechanisms could apply in different situations. In Study 2N, in contrast, modelers talked most about what was happening at a particular time in a model run, thus only discussing the one road that was taken when two roads diverged.

Figure 6.3: (Top:) Percentage of evaluation abstractions in the two models used in Study 2N. Dark=Data; medium=Time; light=Statistical. (Bottom:) Percentage of evaluation abstractions in Study 1.



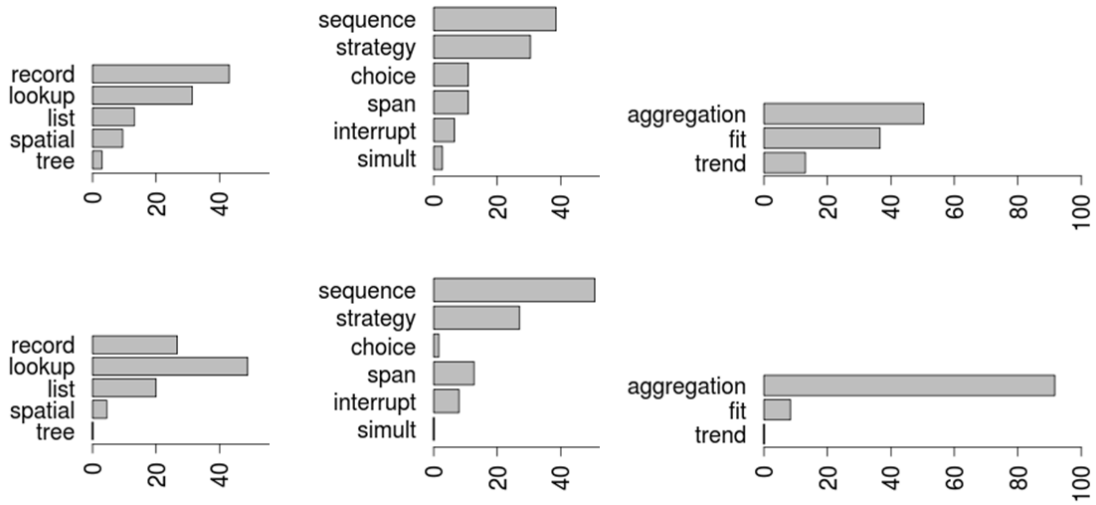
Overall, modelers used statistical abstractions less often when debugging than when talking about their models, which raises the question: did they avoid statistical abstractions because these evaluation abstractions were less *needed* while debugging; or have modelers simply learned to limit their statistical thinking about model traces because such questions are not well supported by current tools? We hypothesize that the latter is true. Eisenstadt [40] documented that professional programmers use such techniques; in his survey of bugs and debugging techniques among professional programmers, he identified some strategies for answering causative questions by comparison of contrasting executions: namely the "dump & diff" and "controlled experiments" strategies. Statistical abstractions could be used to follow comparative strategies: for example a modeler seeing a suspicious value at the time of an incorrect behavior could ask how broadly that suspicious value occurred in similar circumstances, rather than simply tracking down the causes and effects of the value in that particular instance.

In fact, all but one of the modelers in Study 2N explicitly asked "why" or "why not" questions, and sometimes they followed this kind of comparative strategy. For example after following through a trial in the Paired model's trace, N701A saw it make a mistake, and hypothesized that it had learned the pairing wrong. Rather than narrowly check that learned fact (in the form of an ACT-R "chunk"), she looked at *all* the chunks of that type:

N701a: So let's see what the pairs look like. [types command to output contents of all chunks, and scrolls to where some "pair" chunks are] Well, they're backwards.

Unfortunately for N701A, the chunks she looked at happened to be backwards (their two slots had their values swapped), but not all of them. The command she typed did not let her see all the "pair" chunks in an easily-compared layout, so she misgeneralized, and this eventually led her to a wrong hypothesis about the bug in the model.

Figure 6.4: Comparison of evaluation abstractions within each of the three categories; values are percentage of codes within category: Top: Data, Time, and Stats in Study 1 (talking about models to other people); Bottom: Data, Time, and Stats inStudy 2N (debugging models and talking aloud). Notable differences are that in the debugging sessions, *choice* and *fit* abstractions were relatively less frequent within their categories; and the *lookup* abstraction was more frequent relative to list and record.



Another modeler, N702A, tracing through the Zbrodoff model, asked about the firing of a rule called "read-first" using wording that suggested an interest in typical behavior, not just the behavior at that instant:

N702A: Does it always do "read-first"?

He did not have an easy way to check this broad question, and so simply moved on.

This thesis does not conclusively answer the question of whether, and exactly how, statistical abstractions are useful to modelers in debugging, but instead proposes it as an open question for future work. The empirical work in this thesis from Study 3W onward takes a microscope to the evaluation tasks themselves, as potential subtasks of *any* phase of a model's lifecycle.

6.5 Conclusion

In this chapter we empirically investigated cognitive modelers evaluating a model's behavior, and derived a design specification for an evaluation interaction language using the assisted Wizard of Oz methodology described as Step B2 of NP+. That design specification consists of not only the named I-* implications, but also the corpus of questions asked by modelers, and the sequence in which they asked them.

Some interesting insights about this population's evaluation 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.

In the next chapter, to carry out steps B3 and B4 of NP+, we will describe an evaluation abstraction language with features motivated by this design specification, and validate that it is able to convey most of the abstractions modelers asked for in Study 3W.

Chapter 7 – Lea3: A Domain-Specific Language for Evaluation Abstractions

In this chapter we describe the domain-specific language Lea3 (i.e. Language for Evaluation Abstractions), an experimental implementation of a L_{EA} as described in Chapter 4, that was the product of the research described in Chapters 5 and 6. We begin by describing high-level design of the language, then touching on each of the language's operators, and how they arose from the findings in the last chapter. Next we describe an empirical evaluation of the language, showing that it soundly and with low *root viscosity* meets the specification set by the corpus of evaluation abstractions in the transcript of Study 3W.

After discussing the language's viscosity, we describe in detail an implementation of the EAST construct, $L_{EA}/Transform$, called Lea3/T, a DSL for describing incremental changes allowed between Lea3 queries. Finally, we conclude by comparing our choice of properties to validate with the choices made by other Natural Programming researchers.

Lea3 and Lea3/T together form a system for extracting potentially useful information out of a program trace. Lea3 (Section 7.2) encodes complete queries over the trace, and Lea3/T (Section 7.4) encodes a modeler's incremental changes to Lea3 queries. Lea3/T is designed to be triggered by menu choices, so that a modeler can build up increasingly complex Lea3 queries one step at a time. Some Lea3/T changes represent the addition or removal of operators from the Lea3 query's abstract syntax tree; others represent more complex transformations. The relationship between Lea3/T, Lea3, and the semantic domain is depicted in Figure 7.1.

Figure 7.1: Two DSLs: Query changes are the semantic domain of the transformation syntax, Lea3/T (which describes menu options in the GUI), and program trace data is the semantic domain of the Lea3 queries.

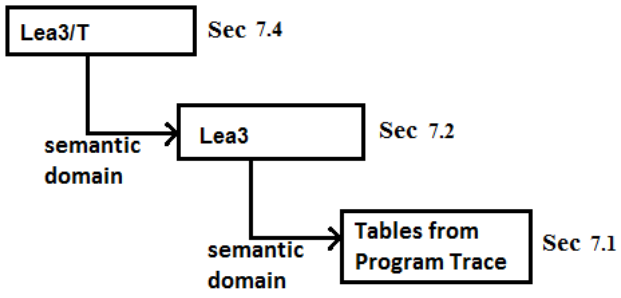
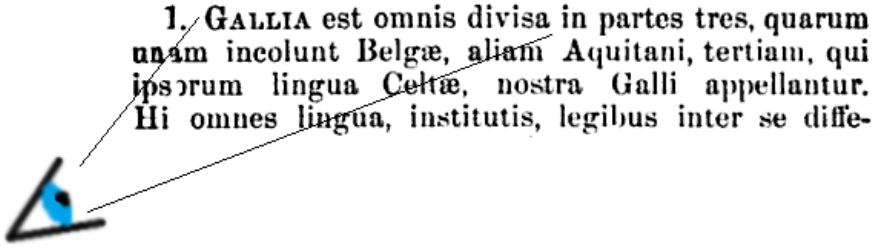


Figure 7.2: A model of reading simulates an eye working its way horizontally across text on a screen. (Figure adapted from [22])



Lea3 is intended to be a behind-the-scenes abstract syntax, with Lea3/T as its only user interface. However for our own convenience we also created a concrete syntax for Lea3 that the user can type (see Appendix A). Since the concrete syntax is not intended for the user to interact with, none of the user studies described in this thesis involved the concrete syntax.

7.1 Semantic Domain of Lea3: A model's trace as tables

Lea3 is a language to express modelers' queries about a cognitive model's behavior represented by a detailed trace of a model run. In this section we introduce a simple example scenario, which we use to informally describe the concepts of the semantic domain, and finally we give formal definitions.

7.1.1 Example scenario

In the example, a modeler has created a cognitive model that simulates reading text horizontally across a screen, and bringing the words it sees into a visual memory buffer, as shown in Figure 7.2. The "screen", the reading eye, and the visual memory buffer are all part of the model's simulation. In the captured trace of this hypothetical model's execution, the modeling simulation recorded the model's "eye" position in a table called `EyePosition` (Figure 7.3), with columns `eyeX` and `eyeY` along with a timestamp for each change in position. The modeling simulation also recorded the immediate, volatile memory of things the eye saw in a trace table called `Visual` (Figure 7.3) with content in an column `vBuf`, and also time information.

Suppose the modeler wants to know the horizontal position of each word the model saw. She could derive this information manually by looking up each word in `Visual` finding the time, then looking up the most recent previous time in the `EyePosition`. Cross-referencing these she could compile a table something like Figure 7.4.

Figure 7.3: `Visual` table (representing contents of a model's visual buffer) and `EyePosition` table (representing position of a model's eye gaze). The "timems" column is time in milliseconds.

Visual	
timems	vBuf
15	Gallia
415	est
715	omnis
1065	divisa
1320	in

EyePosition		
timems	eyeX	eyeY
0	32	15
400	54	15
700	89	15
1050	108	15
1300	125	15

Figure 7.4: Output of a hypothetical modeler's query matching up the `Visual` and `EyePosition` tables of Figure 7.3, taking the immediate next `Visual` row that followed each `EyePosition` row. We have prefixed `next` to the columns from the `Visual` table to indicate their time relationship to the `EyePosition` columns. The `eyeX` and `vBuf` columns together answer the question "Where were the words seen on the page".

timems	endtime	eyeX	eyeY	next.timems	next.endtime	next.vBuf
0	400	32	15	15	415	Gallia
400	700	54	15	415	715	est
700	1050	89	15	715	1065	omnis
1050	1300	108	15	1065	1320	divisa
1300	1500	125	15	1320	1420	in

Lea3 is intended as a behind-the-scenes trace query language underlying an evaluation abstraction support tool like EAST-Env described in Chapters 8 and 9. The environment would present the modeler with separate displays of `Visual` and `EyePosition` tables as in Figure 7.3, and would maintain, unseen by the modeler, the Lea3 queries that produced each displayed table (which in this case would be simple queries simply requesting retrieval of the table). The environment would generate a list of possible Lea3/T transformations and make them available to the modeler in the form of GUI affordances. The modeler would choose an affordance on an output table, and the tool would apply the associated transform to the query underlying that table, resulting in a new query. The tool would run this new query and display a new output table, looking like Figure 7.4.

7.1.2 Informal description of semantic domain

Lea3 expressions are queries that operate over tables of data from a model run, and return other tables. For example "`get(EyePosition, [eyeX, timems])`" is a simple query that returns just the `eyeX` column and associated timestamp information from the EyePosition table. `previous (Visual, EyePosition)` is a query that augments each row in the `Visual` table with the most

recent previous row in `EyePosition`. `raw(EyePosition)` is the simplest query, returning simply the table in the trace called `EyePosition`. A tool builder who wanted to build an evaluation abstraction support tool using Lea3 would need to create logging code in a modeling engine, generating and populating some default set of tables representing the raw trace data from each model run. Lea3 does not constrain what the names and schemas of those raw tables should be, except that timestamp columns should be named `timems` and `endtime` if they are to be treated as time by Lea3's temporal operators.

A *Table* is a collection of rows, like a SQL database table. A visual representation of a *Table* need not actually be tabular, however, so the "row" and "column" terminology only refers to the conceptual organization of the data, not necessarily how software using Lea3 might portray it on screen to a user.

Although querying tables and producing other tables is reminiscent of SQL, and in fact in the EAST-Env prototype Lea3 was implemented on top of an SQL engine, there are a few important conceptual differences from an SQL database:

- Each row may be timestamped. The timestamps are specially-named columns that the temporal operators rely on. All the operators could be defined in terms of the temporal extension to SQL, TSQL2 [112], although the semantics would differ in some corner cases because of the second point:

- Like an SQL database, the rows in the table are ordered, independent of whatever ordering is implied by the timestamp column(s). Unlike SQL, however, many Lea3 operators' semantics depend on the ordering of rows, while only a few do in SQL. For example the `makeState` operator copies a start timestamp from each row into the previous row's end timestamp column, an operation which would be difficult to express in SQL.

An example output of a query is shown in Figure 7.4. The query that has been kept simple for clarity, although it contains more information than the user wanted in this example (remember, the modeler just wanted the horizontal position of each word, i.e. the `eyeX` and `vBuf` columns). Fortunately there is an operator (`get`) that could simplify the table by dropping unwanted columns like `eyeY`.

Also notice the timestamps in the figure, `timems` and `endtime`. A table's timestamping consists of 0, 1, or 2 time values which, respectively, represent data independent of time (in which case it will have no time columns), event data that happened at a discrete time point (`timems` only), or state data that is relevant over a span from a beginning to an end time (`timems` and `endtime`).

Finally, note that the values in the cells themselves are represented as strings. In the formal definition (and in the prototype implementation), there are no value types other than strings

(except for the timestamps), so for example the horizontal eye positions in the example above are stored as strings, not integers. This choice was motivated by practical consideration – we did not investigate modelers' thinking about types, and the software we were using (ACT-R, its substrate, Lisp, and with traces stored in SQLite) were all dynamically typed. As a consequence, visible representations of the data that have particular type requirements are responsible for any type conversion, and must be robust in the face of violating data. Adding a richer type system is a possible enhancement for future work.

7.1.3 Formal definition of semantic domain

A *Trace* of a model run is simply defined as a map from table names to *Table*s. A *Table* is a list of rows, each of which is a data structure mapping a *ColName* (which is a list of *Tag*s) to a string or a timestamp.

$$
\begin{aligned}
\mathit{Trace} &= \mathit{Tablename} \to \mathit{Table} && \\
S, T \in \mathit{Table} &= (\mathit{Schema}, \mathit{Rowlist}) && \text{A table of data from a program trace.} \\
\mathit{Rowlist} &= \mathit{Row}* && \text{A list of rows} \\
r \in \mathit{Row} &= \mathit{ColName} \to \mathit{Value} && \text{A map from column names to values} \\
v \in \mathit{Value} &::= \mathit{String} \mid \mathit{Time} \mid \texttt{null} && \text{A value in the table} \\
\mathit{Time} &= \mathbb{Z} \cup \{\infty, -\infty\} && \text{Timestamps in ms, including infinities} \\
\mathit{Tablename} &= \mathit{String} && \\
s \in \mathit{Schema} &= \mathit{ColName}* && \\
c, d \in \mathit{ColName} &= \mathit{Tag}* && \\
\mathit{Tag} &= \mathit{String} &&
\end{aligned}
$$

Notational conventions are shown in Table 7.1. Note in particular that although *ColName*s are defined as lists of strings, we will denote one-tag *ColName*s as a word in fixed font, e.g. `timems`, and a multitag colname as dot-separated, e.g. `next.timems`. This should not lead to confusion since multi-tag names are rare in this chapter, and it will avoid profusion of brackets when we deal with lists of column names.

We will also define the special null *Row*, defined as $\texttt{Row}_{\texttt{null}} = \lambda\, \mathit{ColName}.\texttt{null}$, to handle special cases in the operator definitions; it is a row that returns a `null` for any *ColName*.

Table 7.1: Notational conventions in this chapter

::=	Definition of a nonterminal
$x \to y$	A mapping from x to y. For example, a *Trace* provides a way to look up a *Table* given a *Tablename*, and a *Row* provides a way to look up values from column names.
$r(c) = v$	means that v is the value in column c of row r
italics	Nonterminals, functions, and variables
`fixed-width`	Terminals
`++`, `--`	List concatenation, list subtraction
[ ]	List
`next.timems`, `xyz`	Column names are lists, but we will omit the brackets, and use "." as a separator if there is more than one *Tag* in the *ColName*.
`timems`, `endtime`	Special columns names indicating start and end time
⟦ ⟧	Evaluation function: evaluates a query and returns a table
q	Metavariable standing for a query
c, d	Metavariables standing for a column name
r	Metavariable standing for a row
S, T	Metavariables standing for a table
s	Metavariable standing for a schema
m	Metavariable standing for a renaming function

To describe these tables, first we define a schema for a table. A schema is a table definition, defining its column structure. The schema consists of a list of column names in a table.

The types of columns are not represented explicitly, but are implied by the last *Tag* of the column name: names ending in T_S and T_E are columns containing data of type $Time$, and all other columns contain strings.

`timems` and `endtime` are special fixed column names reserved for timestamps of start and end time, respectively. Tables with just a start time hold "events": instantaneous point events (like rules firing). Tables with a start and end time hold "states" that persist over some time span. Tables with no timestamp columns (static) hold information that is not tied to time, for example summary tables that compute averages of different values across a model run.

7.1.3.1 Ties to database theory

Because the contributions of this thesis are primarily HCI, not database theory, in this chapter we are using terminology that is non-standard for database theorists, but which we believe will be easier to follow for a wider audience.

However the mapping between our terms and the standard terms are straightforward, and are shown in Table 7.2:

Table 7.2: Correspondences between definitions in this chapter and standard database theory terms[83].

Lea3	Database Theory
Table	Relation
Row	Tuple
Rowlist	The tuples in a relation
Schema	Schema
Tag	(no equivalent)
ColName	Attribute name

Another difference is that in database theory a table is an unordered set of records. Lea3, like SQL, diverges from database theory by keeping the rows in a list, not a set, because some of the query operations depend on ordering. Records may have timestamps, but there is an ordering even among records with the same timestamp, or among records in tables without timestamps. That is why this chapter uses list comprehension notation for *Rowlist*s, not set builder notation.

7.2 Abstract Syntax and Informal Semantics of the Query Language

A Lea3 expression is a query which, when executed on a trace, produces as output a *Table* as described in the previous section. We will start with the high level description of the abstract syntax of the query language, define some convenience functions and notations about that syntax, then finally informally describe the semantics of each operator. In this thesis we will not define a formal semantics, but leave this for future work.

7.2.1 Operators

Operators are listed here in the order they are discussed in the text; the text is ordered in this way for clarity of explanation.

$q \in \mathit{Query} ::=$	`raw`(v)	Refer to raw trace: Section 7.2.2
|	`simul`(q, q)	Simultaneous events: Section 7.2.3
|	`nextRanged`(q, q, t, t)	One event after another: Section 7.2.4
|	`next`(q, q)	
|	`previousRanged`(q, q, t, t)	One event before another: Section 7.2.5
|	`previous`(q, q)	
|	`filter`(q, cond)	Rows that meet a criterion: Section 7.2.6
|	`addState`(q, q, c)	Add a state's current value: Section 7.2.7
|	`makeState`(q)	Treat event as a state boundary Section 7.2.9
|	`mergeRows`(q, a)	Make state from stuttering data Section 7.2.9
|	`collect`(q, q, g)	Summarize values during time periods: Section 7.2.8
|	`calc`(q, c, e)	Add calculated column: Section 7.2.10
|	`unpack`(q, c, v)	Extract from a packed column: Section 7.2.11
|	`join`(q, q, c, c)	SQL-like join operation: Section 7.2.12
|	`segment`(q, c)	Treat events as trial boundaries: Section 7.2.13
|	`get`$(q, c*)$	Hide columns: Section 7.2.14
|	`remove`$(q, c*)$	
|	`rename`(q, m)	Rename columns per a renaming function
|	`agg`$(q, c*, g*)$	Sum, count, min, distinct, etc. Section 7.2.15

$$
\begin{aligned}
cond &= (c, op, v) && \text{condition}\\
m \in renamer &= c \to c && \text{Column renaming function for } \texttt{rename}\\
op ::= &\ = \mid \neq &&\\
g = &\ c_1 \texttt{:=} f\,(c_2) && \text{Aggregate function expression}\\
f ::= &\ \texttt{count}|\texttt{sum}|\texttt{min}|\texttt{max}|\texttt{avg} && \text{Aggregate functions of arguments}\\
&\ |\texttt{countUnique}|\texttt{all}|\texttt{unique} &&\\
v \in \texttt{Value} &= String &&
\end{aligned}
$$

7.2.1.1 Convenience Functions

For convenience in the discussion of the individual transformations, we first define some functions and notations:

Evaluation Evaluating a query over a program trace returns a table; this function will be notated as $[\![q]\!]$

$$[\![\cdot]\!] : Query \to Table$$

List concatenation We use the symbol `++` to represent list concatenation, and `--` for list difference.

Schema information The $colnames : Query \to Schema$ function returns the schema of the table that a query would output; recall that the schema is simply the list of column names for the table.

$$\frac{[\![q]\!] = (s, rl)}{colnames(q) = s} \tag{7.1}$$

7.2.2 The Raw operator

The `raw` operator retrieves a table by name from the program trace. The schema of the table and its contents are determined by whatever infrastructure captures traces from the model run.

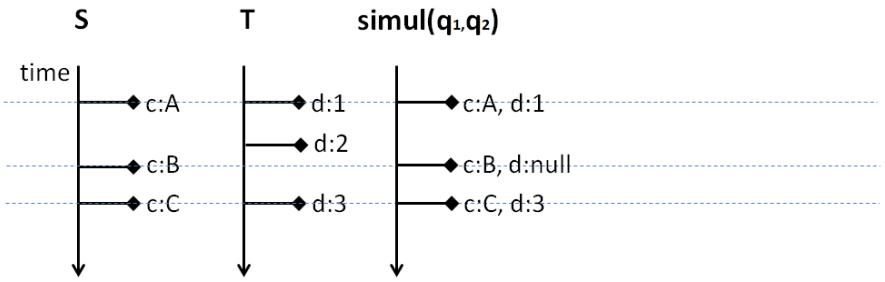
An example trace with two tables is shown in Figure 7.3; evaluating `raw(Visual)` would simply return the table on the left. We will denote "evaluating `raw(Visual)`" as ⟦`raw(Visual)`⟧ below. ⟦q⟧ means to execute a query q and return a table. We omit the formal definition of ⟦·⟧ for each query, which would constitute the formal semantics of Lea3.

7.2.3 The `simul` operator

The `simul` operator (see Figure 7.5) is relatively simple and demonstrates some features of the language.

`simul`(q_1, q_2) joins two tables (⟦q_1⟧ and ⟦q_2⟧) into one, matching up rows only where the rows have the exact same start timestamp value in column `timems`. The resulting table is a copy of the output of q_1, augmented with information from a matching row from q_2 when possible.

Figure 7.5: The `simul` operation, for a *Table*, $S = ⟦q_1⟧$ (containing a list of three events with values 1, 2, and 3 for a column c), and a *Table*, $T = ⟦q_2⟧$ (containing three events, each with a value for column d). All rows from S are preserved by `simul`; and it attaches rows from T if they have exactly the same timestamp. Notice that the second row of T (in which $d = 2$) does not appear in the output of the `simul` query.



`simul`'s output table matches q_1 rows to q_2 rows with the same timestamp, and produces a table for which each output row contains data from one row of q_1, plus data from exactly one row of q_2, or `null`s if no row in the second argument applies). Columns from q_2 are prefixed with the tag `"simul"` (or a variant like `"simul2"`, if there are already *ColName*s starting with `"simul"`).

This semantics differs from the analogous TSQL2 left join[1], which might find more than one match for a row from q_1 in q_2, and if so, would pair that q_1 row up with *every* match in q_2, increasing the number of rows in the result. Lea3 avoids this by taking the *first* match in the list of rows output from q_2. This difference from TSQL2 is an empirically motivated

[1] `SELECT * FROM` q_1 `LEFT JOIN` q_2`;`

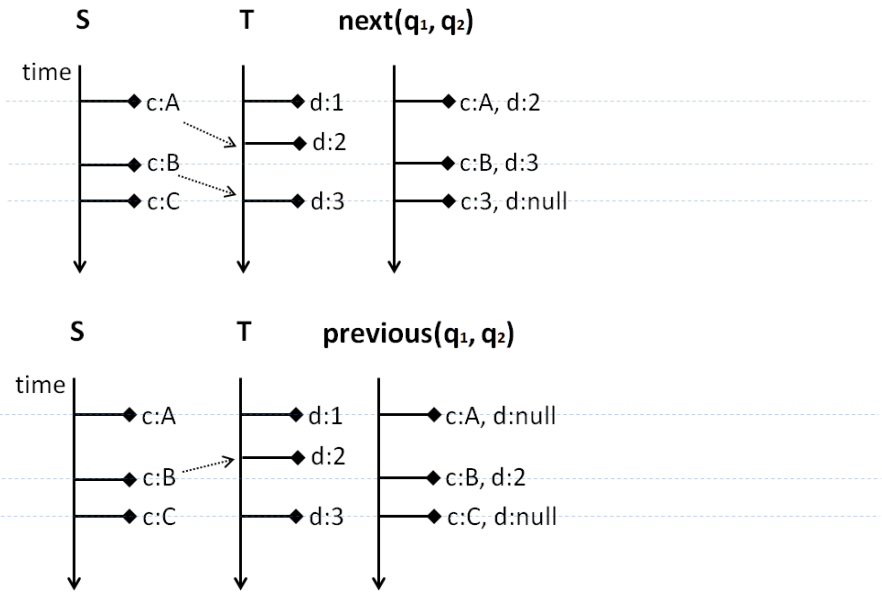
preference for tabular operations that change either rows, or columns, but not both, as described in Section 6.3.1.

Justification: The I-TIME requirement identified in Section 6.3.1.1 motivates this operator and the four that follow (`simul`, `next`, `nextRanged`, `previous`, `previousRanged`) Raw data from ACT-R traces is timestamped. Most modeler requests regarding the relationships of events in time could be adequately met by connecting an item to a simultaneous, next, or previous item meeting some particular criterion.

7.2.4 `next` and `nextRanged`

The `next` and `previous` operators (Figure 7.6) find temporally adjacent pairs of rows from two abstractions (i.e. rows from two tables). `next` and `previous` are not just equivalent operations with arguments reversed: for example $[\![\texttt{next}(q_1, q_2)]\!]$, will have the same number of rows as $[\![q_1]\!]$, but $[\![\texttt{previous}(q_2, q_1)]\!]$ will have the same number of rows as $[\![q_2]\!]$.

Figure 7.6: The "next" and "previous" operations where $S = [\![q_1]\!]$ and $T = [\![q_2]\!]$.



In $\texttt{next}(q_1, q_2)$, a row of $[\![q_2]\!]$ is attached to a row of $[\![q_1]\!]$ if it would be the very next event were the two tables sorted together by `timems`. So for example in Figure 7.6 if the q_1 event

marked $q_1 : 3$ had been just a tiny bit earlier, then it would be between the times of $q_1 : 2$ and $q_2 : 3$, and prevent `next` from joining them. That would leave the events of $\texttt{next}(q_1, q_2)$ as $(q_1 : 1, q_2 : 2)$, $(q_1 : 2, null)$, and $(q_1 : 3, q_2 : 3)$.

$\texttt{nextRanged}(q_1, q_2, t_S, t_E)$ is a more general operator that constrains the "next" event to be within a minimum/maximum time bracket, i.e. the next $[\![q_2]\!]$ must be no less than t_S and no more than t_E milliseconds after the $[\![q_1]\!]$ event for it to match.

`next` and `nextRanged` are related in this way:

$$\texttt{next}(q_1, q_2) \equiv \texttt{nextRanged}(q_1, q_2, 1, \infty) \tag{7.2}$$

7.2.5 `previous` and `previousRanged`

The `previous` operator (see Figure 7.6) is defined analogously, except that the most *recent* matching row is selected.

$$\texttt{previous}(q_1, q_2) \equiv \texttt{previousRanged}(q_1, q_2, 1, \infty)$$

7.2.6 `filter`

The filter operator is like SQL's `WHERE` clause: it applies a test to all the rows in a table and returns only those that pass the test. Unlike `WHERE`, its expressions are highly constrained to compare the equality, or inequality, of a single field with a single value. More complex filters can be constructed by using the `calc` operator to do a calculation, then filtering on its output. These simpler filters have several motivations:

- This simple one-column-one-value filter operation was far more common than more complicated filtering in our empirical observations, motivating a simple one-click operation for users. It made sense to us to map this minimalist filter to an operator, and define more complicated operations in terms of it, rather than design a more powerful filter operation that would not map cleanly to a simple transform and an associated GUI operation. For example EAST-Env provides a pair of icons on cells that allow filtering in or out of that value in that column; the two icons map directly to the choice of $=$ or $=$ for *op* in the definition of `filter`.

- `filter`'s interaction with the `agg` operator is also easier to reason about than a filter allowing expressions involving more variables would be; in particular see the `SeeInst` transformation in Section 7.4.3.

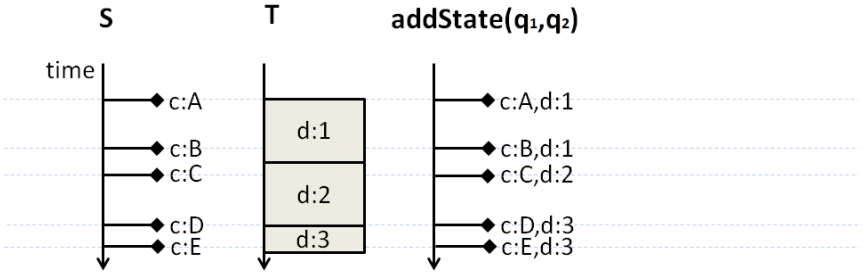
- It is easier to determine which other operators `filter` can commute with than if filters allowed expressions involving more variables; only the existence of a single column has to be checked, not an entire subexpression.

Justification: This is motivated by I-FILTER, described in Section 6.3.2.3.

7.2.7 `addState`[2]

The `addState`(q_1, q_2) operation (Figure 7.7) joins two tables, like `simul`, `next`, and `previous` do. It adds information to an event table describing what the state of the system was according to state table $[\![q_2]\!]$. Thus its first, anchor, argument (q_1) should be an event table, and its second argument (q_2) should be a state table. It augments each event in q_1 with information from the row in q_2 for which the event's `timems` falls between the start and end time (`timems`, `endtime`) of the state. Like `simul`, if there is more than one matching row in the second input argument, the first one is chosen; and if there are none, then the fields are filled in with null.

Figure 7.7: The `addState`(q_1, q_2) operation. $S = [\![q_1]\!]$ is depicted as spikes, and $T = [\![q_2]\!]$ as regions, because S is an event table (having just a `timems` time column) and T is a state table (having both `timems` and `endtime` columns).



Justification: This is motivated by I-DURING, described in Section 6.2.2.

[2] A variant of `addState` called "detail" was used in Section 7.3's validation study. "Detail" was identical to `addState` except that the order of the arguments were swapped, and the columns deriving from the two arguments were displayed by default in adjacent, linked, windows.

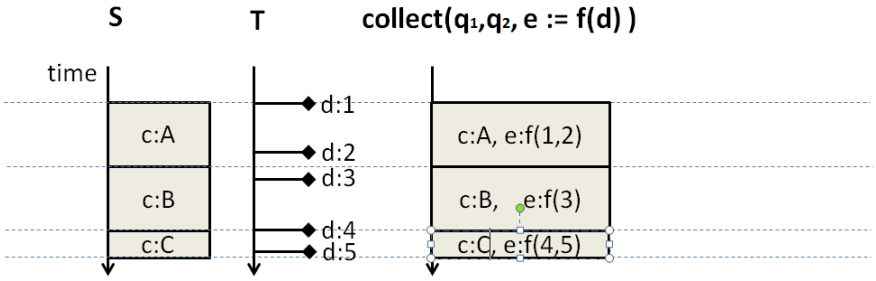
7.2.8 `collect`

The `collect`$(q_1, q_2, c_{new}$`:=`$f(c_{old}))$ operator (Figure 7.8), like `addState`, joins a state and an event table, but in the opposite sense: $[\![q1]\!]$ is the state table, and it collects information across *all* the events in $[\![q2]\!]$ that fall within that span of time, applying a function f to summarize a single column c_{old} in that event table: (e.g. counting them, averaging them, or listing all distinct values in a comma-separated list).

So, for example, if $[\![q_1]\!]$ was a table listing experimental trials, with start and end times in the trace, and $[\![q_2]\!]$ was a table of keypress events with the name of the key pressed stored in column `key`, then `collect`$(q_1, q_2,$ `numKeypresses := count(key)`$)$ would return a list of trials augmented with the number of keypresses in each trial.

Justification: This is motivated by I-COLLECT, described in Section 6.2.1.

Figure 7.8: The collect operation: $S = [\![q_1]\!], T = [\![q_2]\!]$



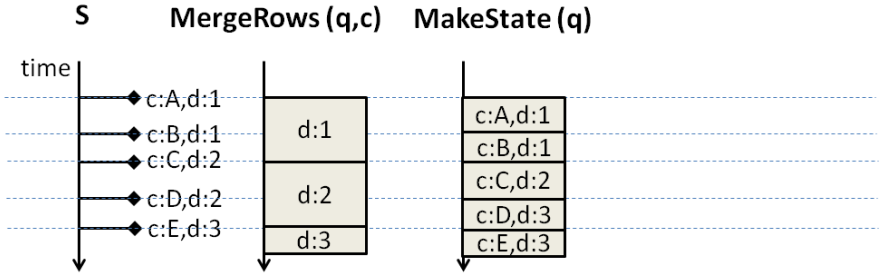
7.2.9 `makeState` and `mergeRows`

`mergeRows` operator (Figure 7.9) coalesces adjacent events or adjacent intervals into one interval, if they are the same (using a column as a standard for sameness).

For example, it would be sensible to apply `mergeRows`(`roomId`) to a table of repeated messages to an agent telling it what room it was in: the result would be a nice summary of the intervals during which it was in each room. It would likely not be sensible to apply it to a table of repeated messages telling the agent that it heard a gunshot: the result would be a single row with an interval bracketing the first and last gunshot ever heard.

This distinction between repeated *notifications* about a state, and repeated *discrete events* turns out to be an interesting semantic choice that in some cases has been silently "baked in"

Figure 7.9: The `mergeRows` and `makeState` operations, applied to a query q_1 whose output S has columns c and d.



to temporal databases' theory of meaning. This theoretical distinction is called *telicity*. *telic* events or intervals are ones which stand alone as discrete entities, while temporally adjacent *atelic* entities can be combined with no change in meaning. Terenziani et al. [116] give the example of a telephone company's database. If rows in a database represent phone calls, then a phone company's billing database should not merge adjacent calls between parties, because one long call may not cost the same as two shorter calls. But the same data for some other purpose may need to treat the combined interval as "on the phone" and merge the two conceptually. Terenziani and Snograss's solution [116] was to make a database where tables could be marked as either telic or atelic, to handle both situations; TSQL2, in contrast, assumes by default that data is telic, and provides syntax allowing data to be *coalesced* as part of a query.

In the case of Lea3, a data import facility from a model behavior trace cannot in principle know which intervals or events are telic or atelic; or whether repeated events are really reoccurrences or just repeated evidence of a continuous state. That is not merely a question of lack of knowledge about the semantics of the underlying data for a particular model; the telicity of the data also depends on the use the modeler intends to make of it. So marking data in this way is part of the interpretive work that users must do with the database, and this affects the kinds of operators that must be made available. See Section 9.3 for a discussion of how telicity relates to the problem of generalizing Lea3 for use beyond ACT-R.

`makeState`(q) turns telic events into states, by simply adding an end time, `endtime`, and filling it in with the `timems` of the subsequent row. The final row's `endtime` is a globally-defined constant of type *Time*, `Maxtime`, which may be ∞, or an implementation may set it to the known end time of data collection in a trace.

`mergeRows`(q, k) treats a single column k in a table as evidence about an underlying atelic state, and combines it into a simpler table with just columns k, `timems`, and `endtime`, throwing away all other columns (since, on combining adjacent rows with the same value in this column, it would not be clear what values to put in other columns which might have taken on multiple values in the mean time).

Justification: These operators together satisfy I-EVENT and I-DURING, described in Section 6.2.2.

7.2.10 `calc`

The `calc` operator adds a new column, calculated from the other columns. Operators include standard math operations `(+,-,*, int())`, and assume that string values are first converted to float. There is also a string concatenation operator `||`. In the EAST-Env implementation of Lea3, the expression language is implemented by translating the expressions directly into SQLite's expression syntax [1] in a `SELECT` statement.

Justification: This operator was not directly motivated by a specific requirement uncovered in Chapter 6, but instead serves several support purposes:

- Lea3 simplifies filtering by only allowing simple equality and inequality tests. `calc` juxtaposed with `filter` allows for more complex SQL-like filtering (i.e: `filter`(`calc`$(q, c, expr), (c, op, v)$)[3]), without making that complexity part of the basic `filter` functionality.
- Some evaluation abstractions in Study 1 involved summarizing data over blocks of a fixed number of smaller time units. This can be done with `calc` by using the *int* function, e.g. `agg`(`calc`$(q,$ `"block"`, *int*(`"trial"`$/10)),$ `"block"`, ...)
- Lisp code written by modelers in Study 1 in general used simple arithmetic, although we did not formally code those uses. Thus we predicted it would be useful to modelers in the future.

[3] Equivalent to `SELECT` *expr* `AS c ... WHERE c=` v

7.2.11 `unpack`

We list just one `unpack` operator, but it stands in for a set of related operators with varying arguments, which are conceptually similar to `calc` in that they create new columns whose values are calculated from others in the same row. An unpack operator's effect is to parse a string and extract a named or numbered substring into one or more new columns. Each variant differs in how it assumes the substrings are packed in the column it reads from, and produces null values in its output column(s) if the assumptions are not met.

For example, the version of EAST-Env used in Chapter 9 had the following variants:

Comma Delimited list `unpackCSV`(q, c, vs) treats each value in column c as a list delimited by commas. The vs argument is a list of integers indicating which items of the list to retrieve.

Space Delimited list `unpackSSV`(q, c, vs)

JSON `unpackJSON`(q, c, vs) JSON Array or Object notation. vs is a list of numerical indices or keys, respectively.

Erlang `unpackErlang`(q, c, vs) Erlang tuple or record syntax, treated the same way as JSON arrays and objects, respectively.

E.g. if column "js" contains the string "[a,b,c]" for some row, then the corresponding row in $[\![$`unpackJSON`$(q,$ `"js"`$, [$`"2"`$])]\!]$ would contain a new column with the value `"b"`.

> **Justification:** The `unpack` set of operators solve the problem of how to represent, in database-like tables, the contents of chunks in ACT-R buffers over time. Rather than creating a different table for each possible chunk type and linking them to the buffers, we pack the chunks' slots and values into a generic "contents" field, and allow the modeler to unpack them as needed for particular queries. This design choice is discussed further in Section 9.6.

7.2.12 `join`

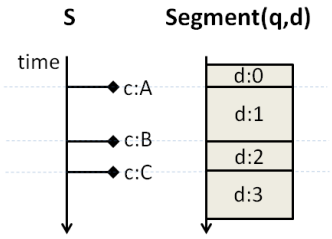
`join` is a more general version of `simul`, that combines two tables, except that it specifies some field other than the start time, `timems`, to coordinate the two input tables. It is equivalent to SQL's `JOIN`, or TSQL2's `JOIN` applied to non-temporal tables. Like the other joining operators, it too has the "anchoring" convention that only the first matching row of the second input table is used, so that the operator's output rows are an exact copy of the first input, augmented when possible with a row from the second input table (or nulls otherwise).

Justification: `join` is included in Lea3 by analogy with SQL's `JOIN` and as a non-temporal analog of Lea3's temporal joining operators: `next`, `previous`, `addState`, etc..

7.2.13 `segment`

The `segment` operator (Figure 7.10) throws away most columns of the input table, leaving only the start time, `timems`, and adding an end time, `endtime`, and a serially-assigned trial number. (The data is thrown away since there is not a clear interpretation of whether to assign the attributes to the preceding or subsequent trial of these edge events. Modelers who need other data can reattach them using other operations such as `next` or `collect`, or display it alongside other data in some visualization that compares data from multiple tables and coordinates their timestamps, such as the timeline depicted in the upper left of Figure 9.1.)

Figure 7.10: The `segment`(q_1) operation, where $S = [\![q_1]\!]$. The start time of the zeroth trial and the end time of the last trial in this example are an implementation-specific global setting, `Maxtime`, for the trace; they may be 0 and ∞, or they may be set specifically to known start and end times over which the trace was collected.



Justification: [I-TRIAL] (Section 6.2.1). In Study 2N, modelers often looked for start/end points of experimental "trials" in the trace in order to limit the amount of trace they had to consider at any one time, and to find analogous points earlier and later in the trace for comparison with the place they were looking at. The `segment` operator divides up the trace in exactly this way, using events as boundaries. We used it in Study 3W and the later reanalysis of Study 3W data in conjunction with other operators: for example a modeler can use `next`(`segment`(...), q) to decorate a plain list of trials with information from the first instance of q from that trial, or `addState`(q, `segment`(...)) to decorate all the events in q with the number of the trial in which they occurred.

7.2.14 `get`, `remove`, `rename`

The three related operators `get`, `remove` and `rename` explicitly remove or rename attributes. `remove` specifies which attributes to *remove*; `get` removes attributes by specifying which attributes to *keep*, and `rename` renames columns based on a function mapping some subset of the columns to new names, leaving column names alone if they are not in the domain of `rename`'s argument (its renaming function). `rename`'s name changes are performed simultaneously, so names can be swapped. It is an error to use `rename` to give two columns the same name, or one column two different names.

When used on `timems` or `endtime` they can change the basic temporal interpretation of columns. In the EAST-Env software, the GUI affordances of EAST-Env were designed to discourage removing timestamps.

Although `get` and `remove` are logically equivalent in the context of a single trace, they would not be equivalent if the same query were applied to a new trace with a different column structure. Additionally, one or the other can be more convenient if the user wants only to exclude a single row, or exclude all but a single row, without having to list the complementary list of columns.

> **Justification:** Many of Lea3's operators have the effect of adding columns to the table produced by the operator's first argument. `get` and `remove` provide the modeler ways of explicitly cleaning up columns that are no longer of interest. Modelers can use `rename` to denote some more contextually appropriate meaning that they associate with a column after processing it.

7.2.15 `agg`

The `agg` operator applies aggregate functions like counting and averaging, applied over groups of rows that share common features. Its output is a table of static (non-timestamped) rows, rather than timestamped state or event rows (unless one of the arguments happens to be a time column, `timems` or `endtime`).

Table 7.3: `agg` example: Applying `agg(raw(EyePosition), [eyeY], [(phraseMiddle := avg(eyeX))])` to the table `EyePosition` in Figure 7.3

eyeY	phraseMiddle
15	81.6

The first argument is a table to be analyzed, the second is a list of columns to be used as grouping features, and the third is a list of summaring expressions to apply. An example is given in Table 7.2.15; applying `agg(raw(EyePosition), [eyeY], [(phraseMiddle := avg(eyeX))])`[4] to the table `EyePosition` in Figure 7.3 would output a table with a single row, with `eyeY` containing all the distinct values of the old table's `eyeY` (i.e. just 15) and `phraseMiddle` containing the average of `eyeX` within rows of `EyePosition` for which `eyeY` = 15 (i.e. all rows, in this case).

Justification: This is motivated by I-DISTINCT, described in Section 6.3.2.3. One requirement proposed in Chapter 6 that this operator does *not* meet is [I-ZEROES]. Lea3's aggregation works like SQL: the SQL query `SELECT C, COUNT(C) FROM A GROUP BY C` will only return nonzero numbers in the count column, because it is only summarizing rows that exist in A. One goal for this language was to try to find values of "C" that did *not* exist in the underlying query, because they had been filtered out in subqueries, and include them in the summary anyway, with counts of zero, in order to alleviate modelers' confusion described in Section 6.3.2.3. Unfortunately this turned out to be impractical when applied to real-world model data. Some of the data users filter out are entities of the same kind as what they do not filter out; but other filtered data is filtered precisely because it is irrelevant. So, including filtered out data as zero-count rows in aggregated tables would result in output with many irrelevant rows.

We partially addressed I-ZEROES in a different way in EAST-Env's GUI, by adding indicators at the top of a tabular view of `agg` output that showed what filters were in place on the data underlying the aggregation, and letting users easily remove those filters to see what difference it would make in the output.

7.3 Validation of Lea3 against empirical data

In this section we describe how we validated Lea3 for soundness, coverage, and root viscosity relative to our participants' data in Study 3W (Chapter 6).

[4]This would be equivalent to the SQL query: `SELECT eyeY, average(eyeX) AS phraseMiddle FROM EyePosition GROUP BY eyeY`

For purposes of analysis, we divided Study 3W transcripts into 149 episodes representing participant queries and the 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 with two Lea3 expressions: once as an "as requested" code and once as an "as provided" code. The "as provided" code was what the *experimenter actually provided* during the study (translated from Study 3W's older version of the language into Lea3). The "as requested" code is a subjective coding of what we in retrospect believe the *participant actually asked for*. The double coding is meant to account for the difficulty of the Wizard of Oz role for the experimenter.

7.3.1 Validation of Coverage

Although it is not possible to validate coverage of Lea3 for the universe of modelers' possible evaluation abstractions, we could objectively validate it for our Study 3W participant data: Lea3 was able to represent 125 (94%) of the 133 usable and non-repetitive episodes from Study 3W (Figure 7.11b). 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 an operator that we doubted would be widely used (a co-occurrence matrix).

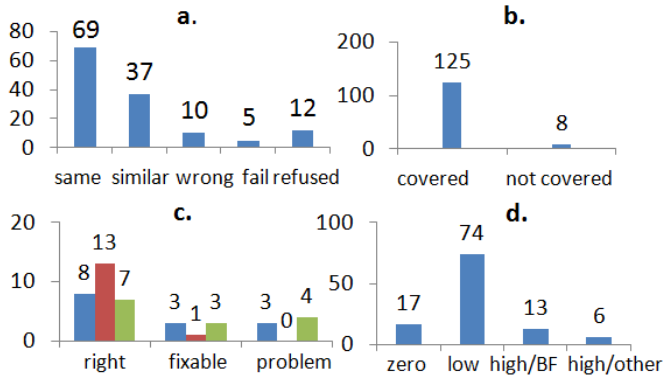
7.3.2 Validation of Soundness

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 Study 3W episodes that Lea3 aims to cover. 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 judge whether the query had been carried out correctly per the original 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

Figure 7.11: Validation against Study 3W: (a) Recoded queries vs wizard's live queries. "Wrong" and "Failed" were experimenter errors (b) Episodes covered by Lea3 (c) Three panel members' ratings for 14 sample episodes. 83% were "right" or "fixable". (d) Move depth: 91 moves (83%) were low or no root viscosity. BF="buried filters" (see text)



it), "some missing", "right assuming..." (panelist did not have enough information about the model to be sure), and "wrong".

After checking panelists' assumptions in the "right, assuming" category and changing the code to "right" or "wrong" if we could determine the truth of the assumption the panelist provided, on average the panelists rated 11.6 (83%) of the 14 queries as either "right" or "fixable", as shown in Figure 7.11c, 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 Study 3W participants had helped refine during the study. As Figure 7.11a 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.

7.3.3 Validation of Low Root Viscosity

In Study 3W, 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 [47].

To measure viscosity, we tightened our focus to *moves* rather than episodes. In the realm of strategy literature, Bates [9] 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 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 defined a measure called *root viscosity* aimed at minimizing the viscosity of the interface that would eventually be built based on the language. Certainly an notation can have low or high viscosity regardless of where in the syntax tree a change is made. But because we were aiming to spare modelers a steep learning curve for our tool, we wanted to let them to see and interact with the *output of queries* (i.e. model content and behavior), not the *syntax of queries* (i.e. some textual Lea3 syntax). Given that assumption, we hypothesized that changes to query output would be easiest to understand when they corresponded to changes to the root of the abstract syntax tree, since the root operation would be the last operation performed, and the one that finally produced the visible output. We also believed that changes to this last operation would be easier for designers to provide recognizable affordances for. Thus we classified moves as *shallow*, low-root-viscosity moves when only the root operation of the abstract syntax tree was modified, and *deep*, high-root-viscosity moves otherwise.

The *root viscosity* of the query language (L_{EA}) provides an estimate of an upper bound on the transformation language's viscosity (i.e. the number of transitions it takes to get from queries to follow-up queries), since a transformation language that spans the topology of possible Lea3 queries must at least include transformations that add and remove operations at the root of the syntax tree[5]. Adding further transformations can only lower viscosity further. Note that an interface with affordances tied to a transformation language will also be subject to this upper bound, if the interface truly provides one navigation for each transform.

As shown in Figure 7.11d, 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 designers tasked with building a usable, fluid debugging interaction language could rely on Lea3 to drive the affordances offered: e.g., menu options to add, remove, or change the "outermost" layers of

[5]If the language did not provide such transformations, there would be no way to introduce operations when building queries from raw trace data. One could in principle describe a transformation language that violated this property; for example by having a single transform that added many operations at once, then supplied a range of transforms for removing operations.

the query could map to the most common ways Study 3W participants sequenced their queries. Our 83% root 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 root 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 `agg` or `addState` 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 additional moves beyond just adding or removing outer layers to the syntax. In the following section we introduce the language Lea3/T, collecting together a set of such moves, which we call "transforms".

7.4 Query transformation language Lea3/T

Lea3/T transforms allow modelers to navigate from one query to the next. The design goals for transforms were to further *minimize viscosity* for the modeler, to make the transformations *understandable*, and to prefer transformations with a *small number of arguments* so that arguments could be either inferred from the context of a modeler's click location or elicited with a few extra navigations in a nested menu.

In the notation for this section, we express a query transform as a function of type $tr : \mathit{Query} \times \mathit{Transform} \rightarrow \mathit{Query}_{\perp}$, where $\mathit{Query}_{\perp} = \mathit{Query} \cup \{\perp\}$. $\perp$ represents the case where a transformation is not defined on a particular query.

If a modeler is inspecting the output of some query q, the set of possible transformations from q are $\{t |\, tr(q,t) = \perp\}$. A tool can use this set of transforms to populate a context menu, or to in some other way decorate the query output with affordances leading to other queries. The tool can further constrain the list to context-relevant transforms by excluding those transforms whose argument values do not match the context of the modeler's action; for example if they clicked on a column, the tool can exclude transforms that happen to fill in a column argument with a column different from what the modeler chose.

7.4.1 Transforms that add operators

The two most frequent kinds of transformation counted in the root viscosity validation in the previous section were additions or removals of outermost operators. So all operators except `agg` have an associated "Add" transform that takes forms like this one for `AddFilter`:

$$tr(q, \texttt{AddFilter}(c, op, v)) = \texttt{filter}(q, (c, op, v))$$

Notice that the transform requires all the same arguments as the operator, except the first; e.g. to add a filter requires a column name, a comparison operator, and a value to match/mismatch. The `Add*` transforms for most operators are defined analogously, except for two special cases, `AddAddState` and `AddUnpack`, described in Subsection 7.4.4.

7.4.2 Transform that removes operations

The inverse of the Add* transforms is the `RemoveOp` transform, defined across all the operators in the same way: it extracts the first argument from the outer (root) operator of the query, and discards the other arguments. In the case of `filter`, for example, it is defined like this:

$$tr(\texttt{filter}(q, (c, op, v)), \texttt{RemoveOp}) = q$$

and more generally:

$$tr(Q(q, \ldots), \texttt{RemoveOp}) = q$$

7.4.3 Transforms that modify `agg`

A set of aggregation-related transformations are each designed to change one part of `agg`'s argument structure at a time. This is because `agg` has complicated arguments: a list of columns to group by, columns to summarize, functions to summarize them by, and new names for their values to take on. Because of this, a simple "AddAgg" transformation would not meet the design goal that transforms have a small number of arguments.

The `Aggregate` transform introduces the `agg` operator, in a way that counts the number of instances of values in just one single column:

$$tr(q, \texttt{Aggregate}(c)) = \begin{cases} \texttt{agg}(q, [c], [(\texttt{numInstances := count(c)})]) & \text{if } c \in \mathit{colnames}(q) \\ \bot & \text{otherwise} \end{cases}$$

The other aggregation-related transforms assume that an `agg` operator is already present, and have the effect of modifying its arguments. In some cases these transforms make sense even if `agg` is not the root of the query's syntax tree: namely when the aggregation operator has been composed with other operators that merely hide rows or columns of its result, or rename columns. To handle this situation, (and a few others later in this subsection) we will define a convenience function that applies a transform to a subtree of the AST instead of the root in some cirumstances:

$$propagate(\texttt{remove}(q, cs),\ T) = \begin{cases} \texttt{remove}(tr(q,\ T)) & \text{if } isColAdder(T) \\ \bot & \text{otherwise} \end{cases}$$

$$propagate(\texttt{filter}(q, (c, op, v)),\ T) = \begin{cases} \texttt{filter}(tr(q,\ T), (c, op, v)) & \text{if } isColAdder(T) \bigvee isAggSet(T) \\ \bot & \text{otherwise} \end{cases}$$

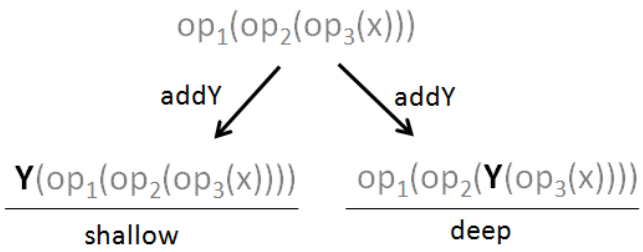
$$propagate(\texttt{rename}(q, m),\ T(c, \ldots)) = \begin{cases} \bot & \text{if } c \notin colnames(q) \\ & \vee \neg isRowChanger(T) \\ \texttt{rename}(propagate(q,\ T(c, \ldots)), m) & \text{if } c \in domain(m) \\ & \wedge isRowChanger(T) \\ \texttt{rename}(propagate(q,\ T(m(c), \ldots)), m) & \text{if } c \notin domain(m) \\ & \wedge isRowChanger(T) \end{cases}$$

$$propagate(Q(q),\ T) = \bot \text{ if } Q \in Query - \{\texttt{remove}, \texttt{filter}, \texttt{rename}\}$$

The *propagate* function depends on predicates that identify transforms with common impact on query rows and columns: *isColAdder* transforms have the effect of adding columns to a query, *isRowChanger* transforms change rows without affecting the query's column structure, and *isAggSet* transforms modify an `agg` operator's arguments.

Calling *propagate* extends the definitions of some transforms by taking their properties into consideration: their effects on the output table's rows, columns, or both. Considering the geometry of each transform and each operator allows us to ensure that the semantics of operations are understandable whether applied to the syntax tree's root or a subtree. That is, in the right side of Figure 7.12, we want to ensure that even if Y is not perfectly commutative with $op_2 \circ op_1$, it will be at least comprehensible when applied to their argument.

Figure 7.12: Two effects a hypothetical transform might have: it could add the new operator to the root of the expression (shallow) or somewhere in the expression tree (deep).



$$isColAdder(T) = T \in \{\texttt{AggSummarize}, \texttt{AddUnpack}, \texttt{AddNext}, \texttt{AddPrevious}, \texttt{AddSimul}, \texttt{AddJoin}, \texttt{AddCollect}, \texttt{AddCalc}\}$$
$$isAggSet(T) = T \in \{\texttt{AggDrilldown}, \texttt{AggCollapse}, \texttt{AggSummarize}\}$$
$$isRowChanger(T) = T \in \{\texttt{AddFilter}, \texttt{UnFilter}\}$$

Using these defintions, we can now describe the other `agg` operators. When an `agg` operator is already present, `AggDrilldown` adds another column to the grouping criteria:

$$tr(\texttt{agg}(q, cs, ss), \texttt{AggDrilldown}(c)) = \begin{cases} \texttt{agg}(q, [c] \texttt{++} cs, ss) & \text{if } c \in colnames(q) \bigwedge c \notin cs \\ \bot & \text{otherwise} \end{cases}$$
$$tr(Q(q, ...), \texttt{AggDrilldown}(c)) = propagate(Q(q, ...), \texttt{AggDrilldown}(c)) \text{ if } Q = \texttt{agg}$$

`AggCollapse` removes a column from the grouping criteria:

$$tr(\texttt{agg}(q, cs, ss), \texttt{AggCollapse}(c)) = \begin{cases} agg(q, cs \texttt{--}[c], ss) & \text{if } c \in cs \\ \bot & \text{otherwise} \end{cases}$$
$$tr(Q(q, ...), \texttt{AggCollapse}(c)) = propagate(Q(q, ...), \texttt{AggCollapse}(c)) \text{ if } Q = \texttt{agg}$$

`AggSummarize` adds a triple to the third argument of `agg`, introducing a function and a column to apply the function over. It invents a name for the column. Note that there is no "unsummarize" transform: the modeler can simply apply `remove` and remove the summary column, achieving the same output. See Section 7.4.6 for further discussion of this.

$$tr(\texttt{agg}(q, cs, ss), \texttt{AggSummarize}(c, f)) = \begin{cases} \texttt{agg}(q, cs, ss \texttt{++}[(c_{new} \texttt{:=} f(c))]) & \text{if } c \in colnames(q) \\ \bot & \text{otherwise} \end{cases}$$
$$where\ c_{new} = unusedColName(\texttt{agg}(q, cs, ss), f, c)$$

This transform refers to a function $unusedColName : Query \rightarrow \ldots \rightarrow String$ (which we will not define here) that outputs a string to be used as a column name. Its arguments include a query and several other arguments. The function *must* output a string which is a legal column name

not already used in the query. The function *may* use some heuristic to choose this string based on the other arguments provided, that a human user might recognize as being related to those arguments. In EAST-Env, for example, this function returned "sum_k" if the extra arguments included an aggregation function `sum` and a column name `k`; but if "sum_k" already existed as a column name in the query, it would add a digit, e.g. "sum_k2".

There are two transforms that remove the `agg` operation: the first, `RemoveOp` was already covered; it transforms the query back to the full set of rows that `agg` summarized. A second transform is `SeeInst`, which returns only a subset of the rows that `RemoveOp` returns: those whose data figured into the calculation of some particular row r in `agg`'s output.

The `SeeInst` transform also can be used by a tool designer to fulfill the I-DETAIL requirement identified in *Study* 3*W* (Section 6.2.1); for example a user's click on a row of `agg`'s output in one window could open the query resulting from `SeeInst` in a second window.

`SeeInst`(r_1) works by replacing the `agg` operator with a sequence of filters matching the group of columns that was summarized by the row that the user had selected in `agg`'s output.

For example consider the sample data in Table 7.4; in this scenario, a modeler starts with a table of people, with name, age, and sex. They apply some transformations to add an `agg` operator, resulting in a 2-row summary with just the oldest person's age of each sex. Finally, they click on the "male" summary row of this `agg` table, labelled r_1 in the figure, and choose a menu option linked to `SeeInst`. The query resulting from this transformation lists just the males from the original table. It also is defined recursively so that further operations composed with `agg` do not prevent the transformation, and are discarded. Q here is a metavariable ranging over the query operations:

$$tr(\texttt{agg}(q, cs, ss), \texttt{SeeInst}(r)) = multiFilter(cs, r, q)$$
$$tr(Q(q, \ldots), \texttt{SeeInst}(r)) = tr(q, \texttt{SeeInst}(r))$$

The `SeeInst` definition depends on *multiFilter*. Note that *multiFilter* is not an operator; it is a function that takes a row and a list of column names, and constructs a nested set of filter operations, using the column names for `filter`'s column arguments, and using the row to look up each `filter`'s value argument.

$$multiFilter(q, [], r_1) = q$$
$$multiFilter(q, [c] \texttt{++} cs, r_1) = \texttt{filter}(multiFilter(q, cs, r_1), (c, =, r_1(c)))$$

Table 7.4: `SeeInst` example: Given a table `People` (left) and an `agg` query over it (middle): `agg(raw(People),[sex],[(eldest := max(age)])`, then `SeeInst`(r_1) yields a query (right), returning rows in `People` which that row was summarizing: `filter(raw(People), (sex, =, "M"))`.

People		
name	age	sex
Alice	34	F
Bob	17	M
Carol	77	F
Nestor	55	M

$r_1 \rightarrow$

agg(People...	
sex	eldest
M	55
F	77

`SeeInst`(r_1)

filter(People...		
name	age	sex
Bob	17	M
Nestor	55	M

7.4.4 Miscellaneous convenience transforms

Finally, a few useful special cases are defined based on observed modeler needs:

First, in addition to `RemoveOp` is another operator for removing a `filter` that is not at the root of the query, but a subexpression inside the `agg` operator. This was an operation observed in *Study* $3W$.

$$tr(\texttt{filter}(q,(c,op,v)),\texttt{UnFilter}(c)) = q$$
$$tr(Q(q,...),\texttt{UnFilter}(c)) = propagate(Q(q,...),\texttt{Unfilter}(c)) \text{ if } Q = \texttt{filter}$$

Next, `AddUnpack`: If there is a packed field that has been partly unpacked, then aggregated, it could be useful to unpack more fields from it and have them automatically added to the summary table produced by `agg`:

$$tr(Q(q,...),\texttt{AddUnpack}(bs)) = \begin{cases} \texttt{unpack}(Q(q,...),bs) & \text{if } bs \subset colnames(q) \\ \bot & \text{otherwise} \end{cases} \text{ if } Q = \texttt{agg}$$

$$tr(\texttt{agg}(q,cs,ss),\texttt{AddUnpack}(bs)) = \begin{cases} \texttt{agg}(\texttt{unpack}(q,bs),cs \texttt{++} bs,ss) & \text{if } bs \subset colnames(q) \\ \bot & \text{otherwise} \end{cases}$$

Finally, the `AddAddState` transform is like the other `Add*` transforms, adding the `addState`(q_1, q_2) operator, (which, recall, adds columns from a state table $[\![q_2]\!]$ to an event table $[\![q_1]\!]$). However if $[\![q_2]\!]$ is not a state but an event table, `AddAddState` has an extra rule that coerces $[\![q_2]\!]$ to a state (using `makeState`).

$$tr(q_1, \texttt{AddAddState}(q_2)) = \begin{cases} \texttt{addState}(q_1, q_2) & \text{if } \texttt{endtime}, \texttt{timems} \in \mathit{colnames}(q_2) \\ \texttt{addState}(q_1, \texttt{makeState}(q_2)) & \text{if } \texttt{timems} \in \mathit{colnames}(q_2) \\ \bot & \mathit{otherwise} \end{cases}$$

7.4.5 A few user interface possibilities for transforms

Lea3/T expressions are abstract transformations, which the designer of an evaluation abstraction tool has the freedom to attach to UI affordances in different ways depending on the visual representation used, the habits and preferences of the modelers, etc.

Two examples from the EAST-Env software used in Chapters 8 and 9 will serve to illustrate a few of the possibilities for transforms:

In EAST-Env, each right-click or drag in any data window triggered the generation of a data structure representing the context: the row, column, data value, and query expression associated with the click or the drag endpoints, and the type of visual representation (tabular, timeline, plot, etc). From this context a heuristic function in the tool generated a large set of possible transforms, along with strings describing the transforms, and the modeler was presented with a nested menu to pick from.

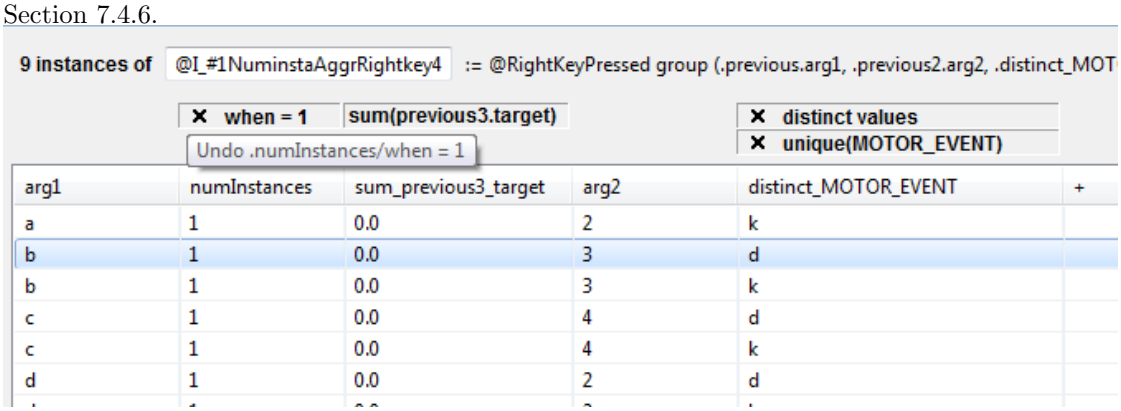
Additionally, Figure 7.13 shows some UI elements attached to transforms that were integrated into the plain grid display in EAST-Env, in a way that served a dual purpose of showing the modeler a little bit about the current query, and allowing selective undo of parts of it.

7.4.6 Implementation note: the inelegance of transformed queries

One design policy we followed in selecting transforms to include in Lea3/T was to keep the set of transformations minimal and understandable at the possible expense of causing Lea3 queries to become more complicated. One example mentioned earlier was that adding a summary column to `agg` with `AggSummarize`, then removing it with `AddRemove` (which simply adds a `remove` operator), resulting in a query like this for example:

$$\texttt{remove}(\texttt{agg}(q, cs, [(c_1 := f(c_2), c_3 := g(c_4)]), [c_1])$$

Figure 7.13: Labels added above columns alert user that something has been filtered out. In this screen capture the label "× when=1" indicates that values other than 1 have been excluded from this column. The tooltip appears when the mouse hovers over "× when=1", and indicates that the user can remove the filter by clicking (applying the transform `UnFilter`($when$), thus expanding the table below to include other values besides 1). The other labels describe other parts of the query, but clicking them simply performed a `AddRemove` transform, as explained in Section 7.4.6.



Note that in this query, the `agg` subexpression produces a table with c_1 among the columns, which the `remove` operation subsequently hides, doing wasted work. It would appear to be more elegant to transform this to:

$$\texttt{agg}(q, cs, [c_3 := g(c_4)])$$

These two are semantically equivalent, but the latter splices the calculation out of the `agg` operation, which is more efficient to calculate.

The disadvantage of this, however, is that there would then be multiple ways to remove the column, whose only difference would be in the query expression, not in the visible representation shown to the user.

A better policy would be to keep the transformations simple, but add a cleanup step that canonicalizes queries after transformations have been applied. This would not be entirely transparent to the modeler, however, since a different set of transforms might be applicable to a canonicalized query than to the original query. Finding a canonical query form and a consistent set of transforms that do not confuse the user is a question for future work.

An easier, if less elegant solution to the potential computational inefficiency of messy queries is the one we used for the EAST-Env environment in Chapters 8 and 9: the environment cached the output of each subquery as a new table in the trace database, and checked for cached subqueries whenever executing a new query. This made the environment quite fast at executing

very complicated queries. Since the modelers built the queries incrementally, cached subqueries were the norm, not the exception. On the downside, this made the database larger than strictly necessary, and some complex queries were time-consuming to refresh when the modeler loaded a new program trace, and that forced the environment to invalidate the old caches.

7.5 Discussion: a comparison of methodologies

A methodological question we wrestled with was how to validate replicability of the complex coding of Study 3W episodes. In coding transcripts of user sessions it is common practice to have two researchers code each episode independently, and use a statistical test of interrater reliability to ensure that the researchers' objectivity in interpreting the data. Because of the complexity of this particular coding scheme, however, it seemed unlikely that an inter-rater reliability scheme would work: two researchers would not likely use Lea3 in precisely the same way to represent modelers' informal programs. Fortunately, the claims we have made for the coding do not depend on replicability; instead they depend merely on that the coding succeed in describing *some* set of information sufficient to meet the user's request, not that the coding be unique or optimal. Thus our choice was to validate soundness, coverage, and root 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 7.5 shows, NP researchers have been solving this by validating other properties of their coding schemes. Ko et al., for example [63], validated the *distribution* of codes from transcripts by having a single researcher code transcripts, then conducting a survey of expert programmers and checking that the overall distribution of codes was the same; this was appropriate because it was only the distribution of codes that Ko was making claims about. Pane et al. [92] also validated distribution of codes 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. Finally, Little et al. [77] checked usability and ease of production in their NP research with a summative validation of whether users with little training could produce Chickenfoot queries and accomplish tasks with the tool. One contribution of this chapter, then, is the identification of these 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. It may be 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, NP+ may help programming language specialists who know how to build an abstract syntax, but who are

Table 7.5: Natural Programming practitioners have validated a variety of properties, using a variety of methods. This chapter is at the bottom of the table.

Language/Tool	Validation method	Property validated
Whyline for Alice [66]	Triangulation; subjective inter-rater coding comparison	Support for cognitive breakdown theory
Contributed to Whyline for Java [63]	Survey of domain experts	Relative importance of information seeking goals
Chickenfoot for end-user web scripting [77]	Usability study	Usability and ease of production
Hands language for children [92]	Experts working independently, ratings averaged together	Reliability of researcher classification
This chapter (debugging for cognitive modelers)	Compare codings, count codings, expert panel, depth-check moves	Coverage, soundness relative to dataset, and root viscosity.

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.

A larger issue is whether user-centered design should influence technical choices at all. A competing approach to language design, "Semantics First" [41], suggests that designers should instead thoroughly analyze the semantic domain, derive a clean set of orthogonal and extensible operators, and only consider user needs when choosing what areas of the domain to support and what extensions to supply.

Although we acknowledge the success of the Semantics First approach in some circumstances, we believe that the success in this area has been partly due to its appropriateness for the particular domains and users it has been applied to. First, semantics-first work has been done in domains where the designers also played the role of domain experts, either because the domain was fairly universal (calendars and scheduling [41]) or because the researchers developed an understanding of the domain and used their own expertise to judge the success of the results (e.g. Choice Calculus [41]). Thus it is hard to be sure user-centered design is not implicitly taking place. Second, the semantics-first approach has been applied to design for users in academia who might be specifically motivated to seek out the most powerful, general, or elegant formalism for their task, and are perhaps willing to endure significant learning curves for that purpose. The users in this thesis, in contrast, are active users, known ([24, 106]) to have little patience for support

tools that have steep learning curves. These users are more likely to use operators in a language that they immediately understand, than to take time away from thinking about their main task to learn a new language. This was demonstrated again and again in Study 1 where modelers were observed to carry out laborious manual processes in preference to making full use of tools like Lisp and R that they already had at their disposal.

The design goals and techniques of Semantics First produce more powerful and elegant languages than Lea3. We would certainly expect Lea3 could be improved by applying some of these techniques, but primarily as a means toward more efficient and less-error-prone implementation, and secondarily as a way of suggesting useful operations that may have been missed by the empirical research. Perhaps such a step should be added to a future iteration of NP+. But we would warn against letting that process too strongly influence the operations of the language that are made available to the user without validating that they are truly usable by that audience.

7.6 Conclusions

This chapter has presented Lea3, a language for representing the evaluation abstractions of cognitive modelers, and Lea3/T, a language of query transformations that a tool designer can use as a guide to what GUI affordances are needed in an evaluation abstraction query tool, and how to carry them out.

We then validated our design specification in several ways, showing that, in the context of the data collected in Study 3W, our design was reasonably complete and sound, and that its design will allow for low viscosity navigation along the query evolution paths that modelers followed.

The work in this chapter along with Chapters 5 and 6 served as initial case study of NP+'s new steps B1-4. 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.

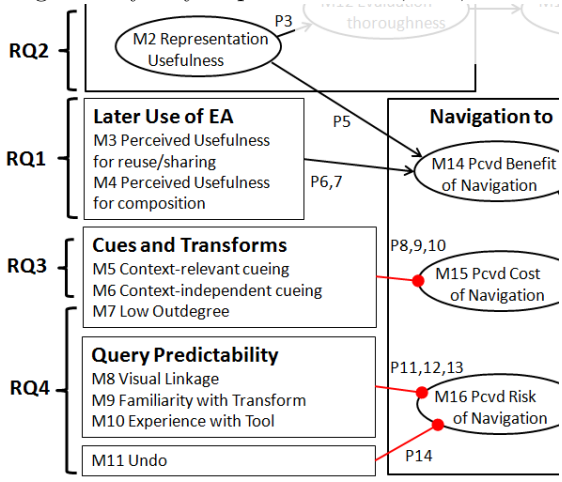
The formative and language design research thus far has focused on the meanings and sequencing of evaluation abstractions that modelers use. In the next chapter we turn to the question of how a tool interface can successfully elicit these evaluation abstractions from modelers. Chapter 8 evaluates EAST's propositions about what properties a language and interface must have to entice modelers to engage in evaluation abstraction building.

Chapter 8 – Eliciting Evaluation Abstractions[1]

8.1 Introduction

In this chapter we describe Study 5, a qualitative observational study in which my coauthors and I ([15]) set out to evaluate some portions of EAST. Would a tool that followed EAST's advice in encouraging navigation help modelers quickly check complex evaluation abstractions without a Wizard of Oz by their sides? Could we attribute the choices they made in doing so to the features that EAST predicts should influence their navigation choices? We found that modelers were indeed able to check a list of expectations, with accelerating speed over multiple runs of a model, and that tool features did contribute to their success in the ways predicted by EAST. The study also gathered evidence about what kinds of context-relevant cueing would be most appropriate for lowering modelers' perceived costs to navigate, and identified factors that discouraged exploratory navigation by increasing *M16 Perceived Risk of Navigation* and *M15 Perceived Cost of Navigation*.

Figure 8.1: Excerpt from Figure 4.2: Study 5 set out to evaluate this set of measure constructs and their influence on navigation by way of perceived cost, risk, and benefit.



[1]This chapter contains material under review for VL/HCC 2013, as "Constructing questions about model behavior: composing, scent-seeking, and learning along the way" [15].

We organize this look at EAST (excerpted in Figure 8.1) around four influences on modelers' choice to navigate: the desire to build something for later use or immediate benefit, the influence of availability and labeling of cues, and factors that encourage or discourage exploratory navigations.

Building for later use EAST's P6 and P7 posit that an evaluation-abstraction supporting environment needs to allow easy composition and reuse of queries that the modelers have previously built (*M3*); we investigated this as RQ1.

Immediate benefit EAST's *M2 Representation Usefulness* refers to usefulness for answering the modeler's question about the model, not merely usefulness for the task of building a query. Thus Proposition P5 implies that the environment should spread the benefit of writing a query throughout the time period spent writing it, through progressive evaluation [46] (i.e. immediate feedback from partial queries), so modelers will be learning immediately useful things about their model's behavior from each successive representation they see as they build the query, helping to motivate them to spend a little more cost to get to the next step. We address this in RQ2.

Cueing *M5 Context-relevant Cueing* suggests that to avoid the high investment of learning a new query language, a designer should keep the learning curve gentle by labeling querying affordances with context-informed cues: i.e. cues that communicate available operations in a way which is relevant to current data and likely to coincide with modelers' next questions (RQ3). This investigation also highlights the need for complementary *M6 Context-independent Cueing*.

Exploratory Navigation Finally, EAST posits (P10-14) five factors that influence their perception that unfamiliar navigations are too risky (M16) or costly (M15) to pursue: *M7 Low outdegree*, *M8 Visual Linkage*, *M9 Familiarity with transform*, *M10 Experience with transform*, and *M11 Undo*. We examine each of these factors in RQ4.

To investigate these hypotheses, we observed 9 modelers using the EAST-Env environment, described below. Our specific research questions were:

- RQ1: Are composability and reuse important to modelers building queries about model behavior? If so, how?
- RQ2: Do modelers value learning along the way while building queries about model behavior?
- RQ3: How or when does context-relevant cueing help by lowering perceived cost?
- RQ4: What risk or cost factors influenced modelers' decisions to make exploratory navigations?

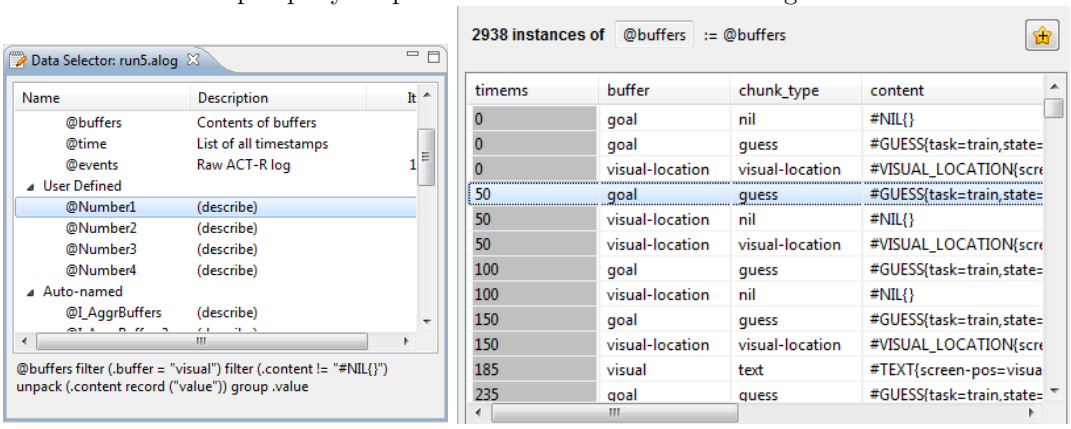
8.2 Method

8.2.1 Study environment EAST-Env

The EAST-Env environment used in this study was an Eclipse plugin that allowed participants to explore trace information from runs of an ACT-R model. The prototype was an instantiation of some portions of the EAST theory, which we built to enable an empirical investigation of these portions of the theory.

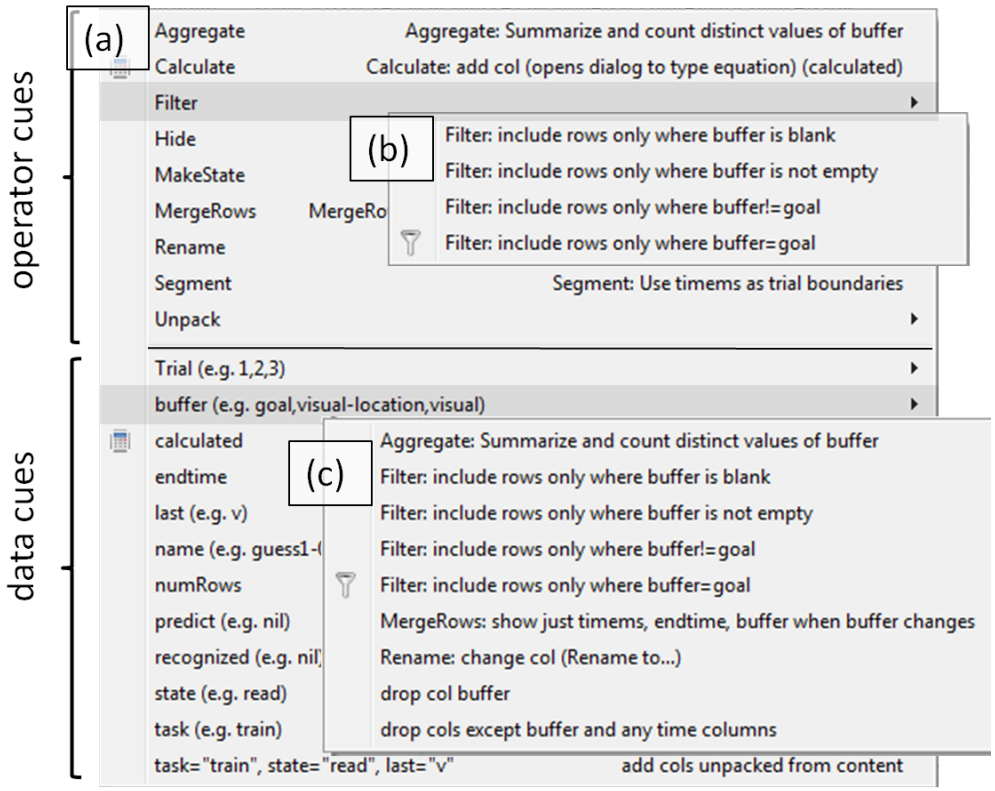
Behind the scenes, EAST-Env incorporated a database-centric implementation of Lea3, allowing a user to load model traces into database tables, then view them "raw" or perform queries on them. The tool generated context-relevant menus whenever a modeler clicked within the results of one query; the menu options were actions drawn from a subset of the Lea3/T transforms. EAST-Env also allowed them to save queries, which they could later reuse via the Data Selector window (Figure 8.2).

Figure 8.2: The "Data Selector" (left window) shows existing queries the user can click on: raw data queries, "User Defined" queries (i.e. named by the user), and "auto-named" queries (i.e. that the user created but did not name), and at the bottom the text of the query named "Number1". An example query output table is shown below in the right window.



EAST-Env provided two ways of accessing a context relevant set of query transforms, labeled in ways which we will refer to as "data cueing" and "operator cueing" (Figure 8.3a). Modelers could access a menu both data and operator cues as items in a menu that popped up when they either right clicked on a data item in a result table, or dragged any data item to another one. Each transform was placed in this menu's hierarchy twice, labeled with both a data cue and an operator cue. Both cues contained similar information, but the menu hierarchy was ordered differently.

Figure 8.3: The context menus and submenus shown when clicking a result table cell (e.g. the table in Figure 8.2). Legal transformations of the query are generated, and shown twice in the menu: once above the separator line ordered by operation name, and once below the separator based on the content and name of cell(s) affected or created. Both submenus have the same filtering choices in their submenus, along with other options. (a) the main context menu, shown when right-clicking "buffer". (b) The submenu under "Filter" in the operator menu. (c) The submenu under "buffer" in the context menu.



Operator cues were ordered by names of operators. Content cues were ordered alphabetically. A closer comparison of two filtering cues is shown in Figure 8.3b and 8.3c.

For example: if an operation concatenated a list of letters like [A, B, C] and wrote the result as ABC, an "operator cue" might be a word like "concatenate", and a "data cue" might be "ABC" In Figure 8.3, "filter" was an operator cue because filtering was the operation to perform, and "buffer" was a data cue because it was a relevant column name in the table.

8.2.2 The Study

We recruited nine cognitive modelers, eight male and one female (we refer to all participants as "he" to avoid identifying the female), with experience ranging from a few months to 17 years, from the Air Force Research Laboratory. Their ages ranged from 27 to 59. All participants had ACT-R experience. All the participants had advanced degrees (MS or PhD) in Psychology, Computer Science, or Linguistics. They were all doing cognitive modeling both as pure academic research and in support of the US Air Forces interest in designing more efficient training programs.

The study involved a half-hour training, an hour-long task to answer questions about an ACT-R model's behavior, and a half-hour quiz. Modelers talked aloud as they worked, and we audio-recorded, screen-recorded, and logged their work.

The half-hour training consisted of a hands-on demonstration of 9 Lea3/T transforms (`AddFilter`, `AddUnpack`, `Aggregate`, `AggDrilldown`, `AggSummarize`, `AddUnpack`, `AddCollect`, `AddMakeState`, and `AddMergeRows`). The transforms were grouped into 6 categories by the operators they applied to (`filter`, `unpack`, `agg`, `next`, `collect`, and `mergeRows` grouped with `makeState`). The training for each category was randomized as to whether operator or data cueing was shown to the user. The randomization was done in three pairs: Filter/Unpack, Agg/Merge, and Next/Collect: in each pair one set of operators was shown during the tutorial as data cued, and the other as operator cued.

For the main task, participants worked with a model that was a simplification of just the training phase of a model of intuitive decision making, published on the ACT-R website, from a paper published in 2012 [58]. The model's task was to learn a mini-language of single-letter "words", by seeing example strings on a simulated screen and learning an association between each pair of adjacent letters. The model's authors argue that decision-making partly entails predicting future stimuli based on sequences of past stimuli. In preparation for the experiment we executed this model five times, each time with some small change to its source code, and saved complete traces of the five runs. Each participant examined the same set of five saved traces. They could not modify the model or run it themselves.

Modelers were given four subtasks (Table 8.1), in the form of expectations to verify in each of five model runs, writing "yes" or "no" in the appropriate square of a 5×4 grid of the four

tasks and five runs. They were instructed to check the five runs in order, and not go back, to simulate iterative development of a model. The tasks were designed to be similar to questions asked in Study 2N (understanding properties of trials, and assessing the distributions of values in chunks), and to exercise a variety of the operators in Lea3 (particularly `agg`, `mergeRows` and `collect`). Although modelers found unexpected ways to solve some of the tasks, the intended solutions were as follows:

Task 1 encouraged the user to understand the range of values that could appear in a particular location (a slot in a chunk in one of ACT-R's buffers); this was intended to be solved with `agg`, to list the values that ever occurred and compare them manually with the list of p,s,t,v,x.

Task 2 was intended to be solved by filtering to a particular type of event, temporally segmenting the trace with `segment` or `mergeRows`, then subtracting start and end times and looking for any number over 2000.

Task 3 was intended to be solved by using either Task 2's segmentation, or another possible one, then using `collect` to make a concise summary of a single variable's sequence during each of the time segments, and finally doing visual comparison to check the task's claim.

Task 4 was similar to Task 1, but it asked about the joint distribution of two variables, not just one. It was meant to motivate advanced use of `agg`, specifically the `AggDrilldown` transformation.

Each modeler was told that their goal should be to check the expectations quickly and correctly, and that they should not feel pressure to use any particular features of the software. After the task was complete or an hour was up, modelers were given a paper and pencil quiz in which they were asked to predict the results of 7 sample operations.

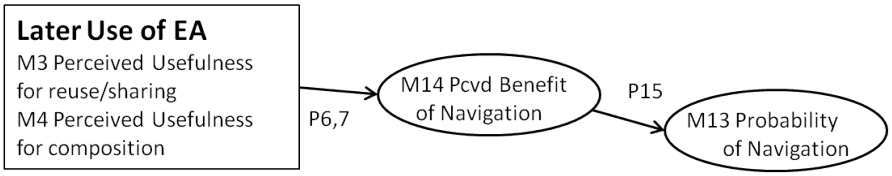
8.3 Results

Three of the nine participants were able to give answers to all tasks for all runs, but only one answered all of them correctly. Three more participants got through all five runs by skipping the most time-consuming task (Task 3) for at least one run. The remaining three ran out of time during Runs 1, 2, and 3, respectively. Figure 8.4 (left) summarizes the time modelers spent in total on each task, and Figure 8.4 (right) shows the time modelers spent on each run.

Table 8.1: Expectations modelers were asked to check for each run

Task 1. No values besides p, s, t, v, x should ever appear in the "value" slot of a "text" chunk in the visual buffer, (except for "start", and that should only occur once)
Task 2. Each visual scan the model makes of the screen (that is, each span of time starting when visual buffer's screen_pos slot holds "visual-location-0-0-0", up until the next time it does so) the sequence of letters that appear in its goal buffer (in the "last" slot of chunk type "guess"), should take less than 2 seconds.
Task 3. In at least three of these visual scans (see above) in each trial, the sequence of letters that appear in its goal buffer (in the "last" slot of chunk type "guess"), should be the same letters, in the same order, as the current trial's stimulus sequence. The stimulus sequence for each trial is shown in @user_events.
Task 4. In chunks of type "guess", some combinations of slot values should never occur: (last=S, predict=P), (last=P, predict=X), (last=X, predict=P)

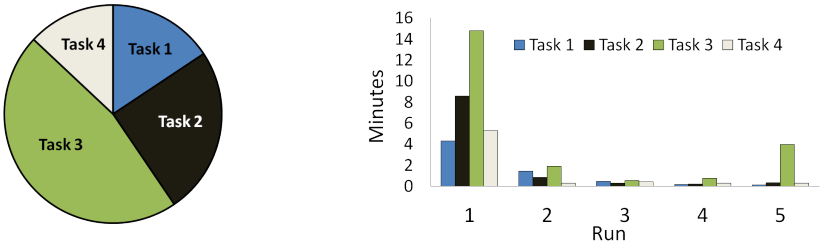
8.3.1 RQ1: Usefulness for Composition and Reuse



In this section we consider influence of propositions P6 and P7 from the EAST theory (Figure 8.1): the effects of a tool feature's *M4 Perceived Usefulness for composition* and *M3 Perceived Usefulness for reuse/sharing* on modelers' perception that navigating to it will benefit them. The theory predicts that one motivation for transforming a query is to create an output that will be useful later, either for composing into a more complex query, or for re-querying in a later run – in other words, that it is part of a longer-term plan. EAST posits in contrast that modelers will be less likely to navigate to tool features that generate separate, uncomposable visualizations whose configuration settings cannot be saved and retrieved.

Modelers had several ways to compose and reuse in the interface. They could compose a query and an operator to create a new query (e.g., apply a filter to a list of events) or compose two queries and an operator to form a third query (e.g., join two lists of events to combine entries that were adjacent in time). Modelers could also reuse queries by naming them and retrieving them later, by finding them using their automatic system-generated names, or by finding them in a window left open. They could take queries developed while exploring one run of the model and apply them to subsequent runs.

Figure 8.4: (Left): Total time modelers spent in each task. Task 3 was the most difficult. (Right): Average time per task in each of the five model runs, averaging only tasks attempted in that run. Modelers who completed runs 3, 4, and 5 did so very quickly. The bump in Task 3 Run 5 was due to a single modeler who gave up on Task 3 early, and returned to work on it in the remainder of the allocated hour.



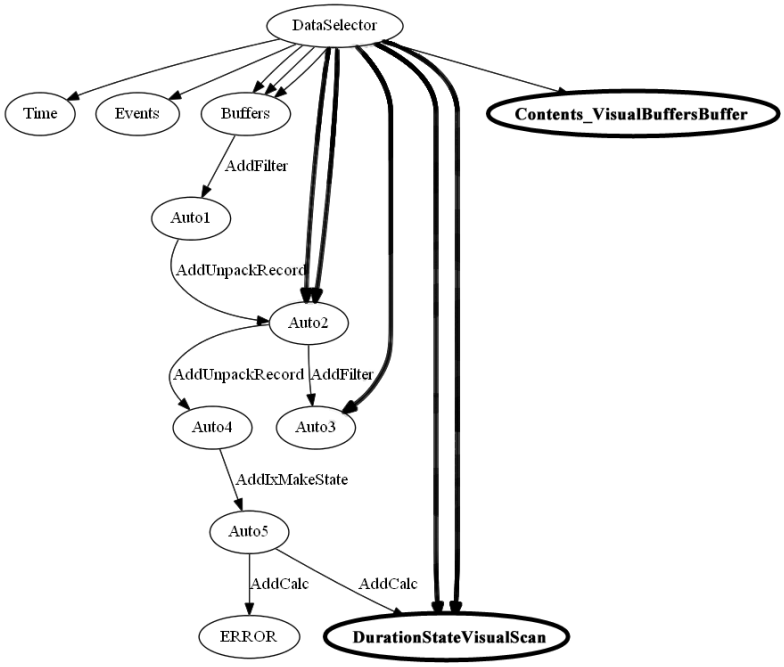
We discuss reuse and composition together because the two activities were highly intertwined: modelers both reused finished queries in new compositions, as well as reused components of existing queries. For example, Figure 8.5 shows reuse of multiple subcomponents of a query. This kind of reuse would not have been possible if queries were not composed from individual subqueries.

Benefits of composition and reuse Composing and reusing evaluation abstractions (*M4 Perceived Usefulness for composition*) provided a clear benefit to modelers. As Figure 8.4 shows, the speedup in task performance in Runs 2-5 over Run 1 was dramatic. In 78 of the 93 answers modelers gave to the tasks in Runs 2 through 5, they answered by simple reuse of a query, either by opening a previously named query, or by using one that they had left open from before.

Modelers used composition both to build up incremental queries from one task to the next within a run, and for composition of previously saved queries that satisfied subgoals of a task. The long vertical chain of nodes in Figure 8.5 represent Modeler EK1 in Task 2 building a complex evaluation abstraction by performing a long composition. The arrows directly from the DataSelector node to various nodes along the way in that composition depict later reuses of these composed subqueries.

Modelers used multiple mechanisms to find queries to reuse and compose. Some re-found the desired queries using spatial or time-based approaches. For example, Modeler EH1 left windows open for Tasks 1, 2, and 4, which he carefully arranged to show enough information to answer all three. Modeler EK1 used the history feature during Run 2 to re-find queries he had not named. When EK1 created a sequence of three "illegal pairs of letters" queries to address Task 4, he created and named the first one, then used "undo" to backtrack to a common unfiltered version, which he used to create the other two.

Figure 8.5: Modeler EK1's navigations for Task 2 over all five runs. Arrows are transformations (GUI actions) and ovals queries. Queries labeled "Auto*" had complicated names chosen automatically by EAST-Env. Bolded arrows show EK1's repeated reuse of already-constructed queries and subqueries. Bolded ovals indicate queries that the participant named.



Motivation to compose/reuse Modelers did not simply benefit from composition and reuse: they knew they would benefit, and they planned for it. EAST proposes that the *perception* of reusabilty and composability actually motivate users, not just that they happen benefit when these properties are present. Reuse often requires some advance preparation, such as naming a query, and we consider evidence of such advance preparation to be evidence of intent/motivation to reuse the query.

In the first few tasks, some modelers did not show any evidence of motivation to reuse, but by Run 2, all eight who got that far were naming and reusing queries, each naming between 3 and 7 queries (mean=5.25). Five modelers named their first query in run 2, three did so in run 1, and one did not name any queries. Modelers got Run 2 done faster than Run 1, but more slowly than Runs 3, 4, and 5 because five of the modelers had not previously named any queries. At that point, those five scrambled to do so, either by searching for or redoing their queries from the earlier run. Runs 3, 4 and 5 went faster because the modelers were reusing named queries at that point.

The names modelers chose for their queries seemed designed for reuse. Some were data-descriptive names that would later provide context for interpreting terse results (such as Modeler EM1's query "more_than_two_seconds", which returned an empty result in runs where Task 2's expectation was met). Others were named after the experimental task (Modeler EH1's "Answer4"), planning ahead for locating the query in each subsequent run. Some names even combined both (Modeler EM2's "Expectation_2_ all_spans_less_ than_2000").

Modelers' query naming was not simply determined by the task structure of the experiment: they did not always name exactly one query for each of the four tasks. One reason for this was that they sometimes used multiple named queries to answer a single question. For example, Modeler EK1 created three queries to answer Task 4, one for each of the three illegal pairs of letters (Table 8.1). He created these by making one query to check for the "last=s, predict=p", another for "last=x, predict = p", and another for "last=p, predict = x". In each run he checked these sequentially until he found something to contradict the expectation.

Some of the named queries were simply ones that modelers originally expected to use, but then abandoned. For example Modeler EK1 named the query for his first answer to Task 1 "Values_Visicon", but then reconsidered his approach and tried a different query, naming the new query "Contents_VisualBufferBuffers". However after looking at that second answer he decided it was not as good as the first answer, so he never returned to it; he used Values_Visicon for later runs.

Finally, some modelers named partial queries in preparation for composing more complex queries that they appeared to think were risky. Modeler EH1 created a query called "LastAndLastNotNil#3" during a failed attempt to answer Task 3 in the first run. He had a query that got him partway to his answer, and he was looking for the "drilldown" operation he had seen in one of the tutorial sheets. He knew that he needed to apply some other operation first to get to it. So he decided on an operation, but he mitigated the operation's risk by naming it first:

EH1: oh, shoot let's [...] save this off as LastAndNotNil Pound 3

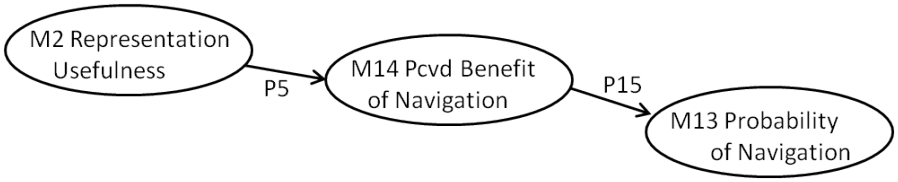
After applying a couple of operations to this query, the drilldown option still did not appear

EH1: What? no! give me drilldown. There is no drilldown... All right, fine, we'll go back to where I was.

He then closed the window and reopened LastAndNotNil#3.

Thus the words, navigation patterns, and naming schemes chosen by the modelers suggest that composition and reusability were indeed motivating factors for modelers.

8.3.2 RQ2: Learning Along the Way



Some program trace query tools, such as PQL [81] and PTQL [45] accept a complex textual query from the user, all at once. But EAST posits that modelers would have more motivation to use such tools if the tools encouraged the kind of incremental work that enables learning along the way. The theory's construct *M2 Representation Usefulness* refers to usefulness for learning about the model itself, not just about the query tool. In this section we present evidence that modelers learned useful things (M7) about the model along the way to building their queries, and that modelers were specifically motivated by this extra learning to do more work with evaluation abstractions.

Three forms of evidence of modelers' learning and how it motivated them emerged. The first was that modelers sometimes continued to work with the queries even after they had obtained the answer they needed to perform the task. This extra work often seemed tied to learning more and resolving confusions about the models' behavior. One example of this extra work was modelers rebuilding queries that they already had in front of them. Two out of nine participants (both from the top 3 modelers in terms of the most correct answers) chose to rebuild at least one query after they already had the correct abstraction open on the screen.

For example, Modeler EM2 expressed confusion when he saw results of his query: when looking at a summary of distinct letters the model "saw" (for Task 1, Run 3), the modeler saw only a single letter ("t") instead of five letters:

EM2: Hm? I don't even know what I'm looking at now. I'm really confused ... it only showed up as just a t.

He then incrementally rebuilt his query from the beginning until he got to the point where he could see that there were going to be a lot of "t"s. At that point he decided that the output he had was correct and stopped. Likewise, for Modeler EO1, only a single "t" was visible in Run 3. However, unlike Modeler EM2, Modeler EO1 interpreted it correctly, but decided to rebuild the query anyway to check his understanding of the trace data:

EO1: Oh, it's ... messed up. Is this really what's happening? [After rebuilding the query from scratch, he then said,] "I see, it ... doesn't seem to be seeing things. Very well."

In another case, Modeler EM2 realized that Run 3 was very different from prior ones with respect to Task 3, and did not know how to interpret the query in this new situation. He expressed surprise when a saved query resulted in a blank table. Pulling up his query, he saw a blank table:

EM2: Uhh, oh no. That's bad... Something's gone horribly wrong. Okay. I'll have to think about that one.

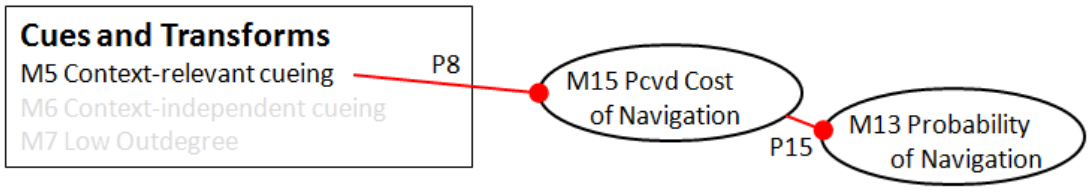
He began rebuilding the model, saw that the data looked very different (the letters did not appear in the same order, and his previous query had assumed they would), and proceeded to spend 8 more minutes answering the question.

The second form of evidence for the importance of immediately useful representations at each step of query building was that some of the modelers' queries referenced elements of the trace that the modelers could not have known about when they started the task: they must have learned these along the way. For example, while building a query for Task 3, Modeler EM2 realized that the "predict" slot can have a special sentinel value, END, when it has completed a scan. This value showed up explicitly in the query he ended up using for Task 3.

The third form of evidence was that one participant, EO1, appeared to plan to learn along the way in future reruns of his queries. For two of the tasks (Tasks 1 and 4), he saved the last two steps of his navigation path: one with a final `agg` operator applied, and one without, and used both in later runs. In later runs he checked both of them, as if reiterating the last construction steps, even though the single final query would have technically been sufficient to answer the question.

Thus, the modelers took advantage of the incremental building of queries not only to build up their final answer, but to learn about how that answer was formed and to understand more about the model trace data. This adds evidence to the notion that modelers perceive a benefit in getting useful information about a model with each navigation, even when they appear to be simply taking steps towards a longer-term goal.

8.3.3 RQ3: Context-relevant cueing



Context-relevant cueing was defined in EAST as the degree to which navigations available to the modeler are contextually appropriate in terms of both what navigations are *offered* and how they are *labeled*. We first discuss the pros and cons of context-relevant cueing (Sections 8.3.3.1 and 8.3.3.2) in terms of navigations *offered*. Then in Section 8.3.3.3. we investigate two styles of contextual *labeling*: *data cuing* and *operator cueing*.

The motivation behind context-relevant cueing was the perspective that cognition is highly situated in the environment; i.e. that modelers would most likely think of operations to apply in response to data they were seeing. For example if a modeler was going to filter out a particular kind of extraneous data out of a table, they would be most likely be prompted/reminded to do so at the moment they saw such a row on the screen. By allowing the modeler to click on the offending data, and by offering the modeler filtering options with arguments pre-filled according to where they clicked, the design of EAST-Env aspired to tightly integrate the affordances of the tool with the modelers' natural cognitive flow.

8.3.3.1 Navigations offered: Context-relevant cueing was helpful

In situations when modelers had gained practice with transforms, the context-relevance of the cues helped them navigate quickly. Filtering rows by the presence or absence of a value in a single column could be done with between one and three clicks, depending on the cues chosen, and the operation took no more than a second to complete. Modelers added and removed filters 161 times in the study. For example in Task 1, EG1 chose to make a list of appropriate values then filter out t, v, x, s, and p leaving behind only values that would prove the Task's expectation untrue. Starting from the whole list and filtering out each of the five letters took him just 45 seconds. EM1 took 93 seconds to do approximately the same thing.

As a comparison, in Study 2N only one modeler tried to filter the ACT-R trace. N727ap tried to use a feature for this in ACT-R's native toolset. The only way he could think of to find the name of the right command was to look in the manual, and he gave up after 44 seconds of searching. The command he was looking for was not very configurable: it only would have let him filter in a predetermined way, by a three-valued system-defined detail level (high, medium, or low detail) rather than by something more task-relevant. Another analog to filtering in the ACT-R environment is a feature of the debugger that lets the user skip ahead to events matching one of a few fixed criteria, for example the firing of a particular rule. (This is akin to filtering because the modeler is jumping ahead to events of a particular kind, seeing sequentially what they would see all at once in a filtered list of events like EAST-Env provides.) Performing this skip in ACT-R's debugger requires typing in the name of the rule, and modelers had to go find it and type it in. In Study 2N, only one of the seven modelers (N702) used this feature, while others chose to just hit the step button multiple times to achieve the same effect. The contrast between EAST-Env's heavily used filtering features and ACT-R's rarely used features serves as a demonstration of EAST's P8: a designer can lower the perceived cost (*M15*) of little-used features, and lift them from obscurity, by adding appropriate context-sensitive cueing (*M5*): that is, by using information readily available in the environment to select and parameterize the transformations that will be offered to the user.

Another way modelers demonstrated a reliance on context-relevant cueing was in the way they went about using the calculation operator (i.e. the AddCalc transform). To enable modelers to build math expressions without traversing menus to do so, EAST-Env offered a "calculate" menu choice when clicking anywhere in a table, which popped up a dialog box, allowing the user to type an equation for a new column to be added to the table. The equation could depend on any of the other fields in the table, which were listed at the top of the dialog box, along with the list of legal expression operators (e.g. +, ×, concatenate). However seven of the modelers at least once dragged the dialog out of the way so they could look at the table to get the column names in the *context* where they had previously been working while they typed an expression.

Recommendations: (1) Designers can boost usage of evaluation abstraction tools by adding affordances for them at times and places they will likely be used, and cueing them appropriately to the situation. (2) Tool feature affordances should be placed on the same screen as both the data items that are likely to trigger a model to use them, and all the values or keywords the modeler will need to know to configure them. Copying those values to a separate tool configuration window is not sufficiently contextual.

8.3.3.2 Navigations offered: Context-relevant cueing was not always sufficient

Context-relevant cueing proved to be a two-edged blade. It was helpful to modelers in completing the experimental tasks quickly after they had gained some practice with it but, contrary to our expectations, it also appeared to have hampered modelers' initial attempts to explore EAST-Env to find out what operations were available and how they worked. When modelers were uncertain of the operation they needed, but instead were trying to get a sense of the possibilities offered, the requirement to click in the right context limited their choices.

Another variant of this problem was that sometimes modelers knew what operation they wanted for reasons less specific to the current context; for example in this experiment they were reading a task sheet and were sometimes cued by the content they saw in the questions rather than the screen. In those cases the tool could not offer relevant cues: EE1 explained the problem well:

EE1: It's really nice it's context senstive where I'm at. It's both nice and frustrating; nice because when, 'BOOM', it's there; and the other times, when, oh, it's not there, I would have liked to have had a pulldown or something like that.

In the quote above, EE1 wanted to filter on a value that he knew from the paper task sheet must be in the table on screen, but was not currently visible.

Recommendation: Context-relevant cueing is crucially important for making sure modelers can find out about appropriate operations, but operations should also be made available by some other mechanism divorced from context, so users can find them when their use is self-motivated or prompted by some cue outside the evaluation abstraction support tool. (P9/M6 in EAST)

8.3.3.3 Navigation Labeling: Operator versus data cueing

EAST-Env had two types of context relevant cue: *operator* and *data cues*. Upon each click or drag operation, EAST-Env generated a list of appropriate transforms given the context (the row, column, table properties, cell content, and the table's original query string), and populated a menu with two cues for each transform, one labeled in an operator-centric way, and one in a data-centric way, sorted into different parts of the menu.[2]

Participants used operator cueing more than data cuing. In fact, even though participants' training was counterbalanced to provide equal training to operator and data cueing, participants overwhelmingly opted to use operator cues instead of data cues. Out of 362 navigations, 270 (75%) were operator-cued, and only 92 (25%) were data-cued.

We at first hypothesized that data cueing was at least useful when people were stuck, but our data did not provide support for this possibility either. Content cueing was no more prevalent among navigations when users were on a tangent that did not yield a solution, than when they were directly on the track to a solution that satisfied them. Clearly, then, the modelers preferred operator cues.

8.3.3.4 Why were operator cues preferred?

Prior experience in information foraging theory led us to expect that modelers would choose data cues over operator cues. Why then did modelers prefer operator cues? Three factors might explain this: (1) The information structure of model traces appears to be different from prior domains of information foraging theory literature, (2) Query transformations are in some ways more like *enrichment* than *navigation*, and (3) When modelers did choose data cues, the cues

[2]The motivation for this experimental design was an earlier version of the EAST theory that had a different version of the context-relevant cueing proposition: modelers would prefer cues organized by strings found in the data itself over cues organized by operation names. However, the evidence suggests that the proposition we present here is more accurate than our original proposition was.

themselves may have taught modelers the names of operator cues, encouraging future operator cue use.

First, as discussed in Section 4.2, prior research on IFT has suggested that strings found in the user's desired destination patch make good navigational cues. (A patch is a collection of information in one location with outgoing links to other patches.) But prior work has focused particularly on domains like Java source code (e.g., [74]) or web pages (e.g., [27]) in which the sets of strings in each patch (i.e., each Java method or web page) are substantially different between patches.

In the domain of cognitive model traces, the patches are not as easily discriminated by simple choice of strings taken from the data, because the traces of the model runs contain fewer unique strings than Java code or web pages. Instead, they tend to contain a small number of strings, repeated many times, and numeric values. Some of the questions modelers asked in earlier studies reflected this property of traces, showing an interest in the arrangement, more than the identity, of data. For example Task 3 was modeled on evaluation abstractions described in Study 1 (the SCANTYPE model described in Chapter 5) that concerned the order that items were perceived on a screen. This interest in the ordering rather than the identity of values may explain why operator cues were more informative: operator cues describe transformations that make such patterns evident, rather than unique data items that might be sought. Thus the clearest way of describing the difference between two summaries of model trace data may be in terms of the operations governing the data's organization, rather than particular strings and numbers that might be found in the data. Both the data itself and some of the modelers' questions about it were in terms of patterns of data, not just the presence of particular strings.

Second, data cues were intended to support navigation to the new "patches" of information that modelers created when they applied operators, but the result of an operation is also a form of enrichment (an IFT concept meaning that the user spends time improving a patch or creating a new one). Most of our operators were enrichment operators since they improved the patch, such as by filtering out the irrelevant rows in tables. Furthermore, filtering does not add different strings to the result table, and instead leaves a subset of the original strings, which differs from the IFT norm of distinct patches. This meant that the word-based cues in some cases perversely highlighted what was common between the old and new patch, instead of what was unique to the new patch.

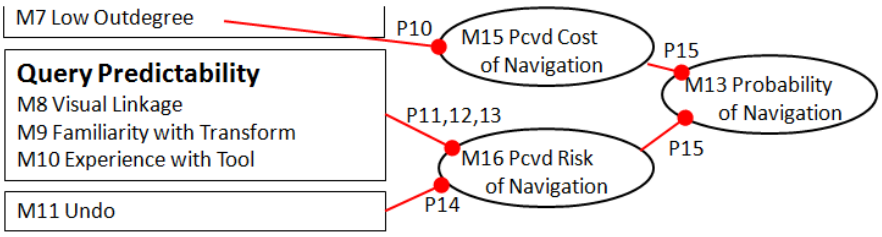
Finally, in some situations, data cueing seems to have actually encouraged modelers to switch to operator cues by teaching modelers the names of the operators. Once modelers knew the names, it could be much faster to find these names on menus than to think of data strings related to the results they were hoping to find. In fact, there were six cases in which a modeler used data cueing first one or two times, then switched to operator cueing for almost all of their subsequent uses of the operator. Three of the cases (Modeler EK1's `AddFilter`, EM1's `makeState`, and

EH1's `mergeRows`) followed the same clear pattern demonstrating learning of the operator name: the modeler mentioned the operator's name for the first time *after* the first data cue use, but then mentioned it again *just before* opening the menu later on when they first used the operator cue. In the three other cases, there was still at least some evidence that they'd taken note of the operator name: (1) In Modeler EO2's use of `unpack`, the modeler also mentioned the operator name before doing the first data cue; (2) in EH1's `unpack`, the modeler read off the operator name while doing the data cue, but not before doing the later operator cue; and (3) in EO1's `AddCollect`, the modeler never mentioned the operator name at all, but went straight to it the second time, despite not having been taught to do so.

Thus, "operator" cues may be more appropriate for enrichment to create new derivative patches, whereas "data" cues are more appropriate for navigation to truly distinct patches. That is, when the modeler is seeking a new patch, navigational cues leading to it ought to relate to what is different about the next patch, but if the desired patches are derivatives of the current patch, then the *way* to enrich them may emanate more scent to the modeler than the *resulting data* would.

> **Recommendation:** Context-relevant cues should emphasize the operations they perform rather than the data they produce, unless the data string itself happens to be part of a clearest description of the operation (e.g. "add column Bob").

8.3.4 RQ4: Perceived Risks and Costs of Exploration



RQ4 considers linkages between evaluation support tool features' design and modeler' perceptions of the cost and risk of exploring unfamiliar features of EAST-Env.

In the context of a user's decision about how to spend their attention, Blackwell defines perceived "cost" as the amount of time a user estimates they will spend towards some end, and "risk" as the perceived "Probability that no pay-off (i.e. reduced future cost in attention units) will result, or even that additional future costs will be incurred from the way the user has chosen to spend attention." [11] As with any complex software tool use, modelers in this study had to

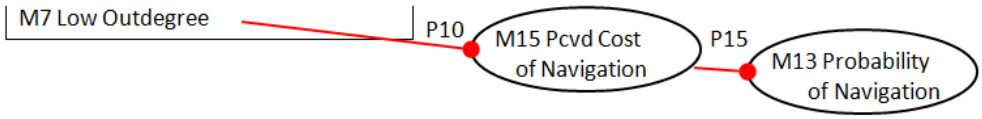
weigh the risk of exploration versus the risk of trying to proceed with the features they already understood.

Exploration was important to success in this study's tasks. Factors influencing modelers' ability to explore were made particularly evident in Task 3, in which modelers' success largely came down to whether they could locate and use a single transform: `AddCollect`. Eight of the nine modelers encountered (and only one overcame) the following problem with Task 3: they got to the point where they needed something like the `AddCollect` transform but they did not know its name or how to accomplish it using EAST-Env's menus. This was a clear example of what Ko [61] called a *selection barrier*, which Ko exemplified with the phrase *I think I know what I want the computer to do, but I don't know what to use....* All eight modelers were slowed down by the barrier, with five of them eventually giving up or running out of time on it, two finding less satisfactory indirect solutions, and one overcoming it. In contrast, as Table 8.2 shows, modelers found quickly, understood, and used repeatedly other operators that had contrasting factors discussed in this section.

Exploration involves both risk and cost. EAST is a theory that describes the choice of individual navigations, and therefore classifies those navigations as bearing perceived risks, but it is also true that modelers can perceive exploration in terms of cost, since the activity of performing repeated explorations takes time. However in EAST we assign most of the factors dissuading modelers' exploration to the category of "perceived risk" in the attention investment subset of the EAST theory (lower right quadrant of Figure 4.2). This is because exploration involves repeated probes of program features, each of which might not pay off.

Evidence in the following four sections will show how four of EAST's influences to perceived risk and cost related to modelers' probability of navigating (*M13* in EAST).

8.3.4.1 Results: Low outdegree of topology (M7) lowers perceived cost of exploration



Modelers' reactions to the large menus in EAST-Env supported EAST's proposition P10 that high outdegree (*M7*) increases the perceived cost of navigation (*M15 Perceived Cost of Navigation*).

Outdegree refers to the number of transformations available to the modeler, from their current query to new queries. Lea3 and Lea3/T together form the nodes and edges of a topology that

Table 8.2: A comparison of operator features and modelers' usage of them. Cells shaded green are features of operators that EAST posits to contribute to low perceived risk or cost. "Good visual linkage" (*M8*), in this study only, means that introducing the operator either adds or deletes either rows or columns from a single table. "Menu Complexity" is one possible operationalization of *M7 Low outdegree*: Low = a fixed number of options in a single submenu; Medium = a non-fixed number of fields to choose from under a single submenu; High = non-fixed number of fields under multiple nested submenus

Operator	Operator Features			Empirical data	
	SQL or Excel has construct (*M9*)	Good visual linkage (*M8*)	Menu Complexity (*M7*)	Num. uses	Explicit statements of confusion
`filter`	yes	yes	low	161	4 (2%)
`unpack`	no	yes	medium	69	0
`agg`	yes	no	high	42	3 (7%)
`calc`	yes	yes	low	26	1 (4%)
`get`	yes	yes	medium	19	1 (5%)
`makeState`	no	yes	low	12	1 (8%)
`mergeRows`	no	no	low	7	1 (14%)
`next/previous`	no	no	low	7	0
`collect`	no	no	high	7	1 (14%)
`segment`	no	no	low	6	4 (67%)
`addState`	no	no	low	5	0

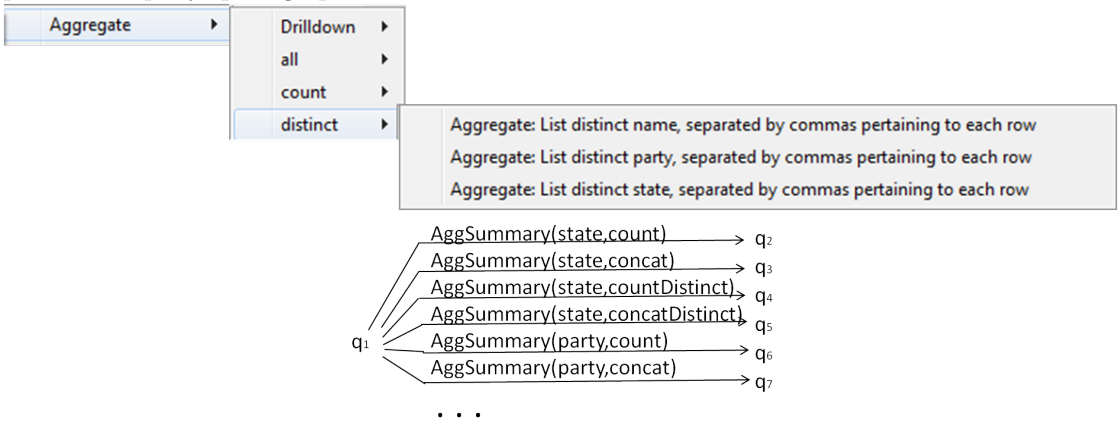
modelers can traverse. The number of transforms that lead from any given Lea3 query will govern how many affordances for navigation that query's output has in an evaluation abstraction support tool.

When a transform takes arguments, and the environment cannot guess at those arguments from context, then that transform contributes more than 1 to the outdegree; it contributes the product of the number of possible values of each argument. For example in the operator-cued part of EAST-Env's context menus, there were many menu items for the `AggSummarize` transform. Recall that `AddSummarize`(c, f) adds a column that summarizes a possibly-hidden column c by applying function f. Since several hidden columns c might be available, and several aggregate functions f could be applied, EAST-Env had to offer a menu item for every possible combination of c and f (which it did via two levels of submenu, shown in Figure 8.6).

In the version of EAST-Env used for this study, each of these combinations was actually added twice, once as an operator cue, and once as a data cue. As Figure 8.3 shows, the resulting menus were sometimes large.

More than one modeler commented on the intimidating appearance of EAST-Env's menus (see a typical menu in Figure 8.3). We hypothesize that this intimidation was due to the menus'

Figure 8.6: How EAST-Env menus related to query space topology. Top: menu items related to `AggSummary` (from a table with three hidden columns; they had non-numeric contents, so EAST-Env did not suggest transforms with functions such as sum or average). Bottom: a small part of the query space graph.



size, and so indirectly due to the query's outdegree. For example when EM2 first opened a menu in Task 1, Run 1 (after navigating a few times without one using the filtering icons), he said:

EM2: Uh oh, I don't want to look at that; it's too confusing.

He closed the menu, but then opened it again shortly after, and began examining the menu items.

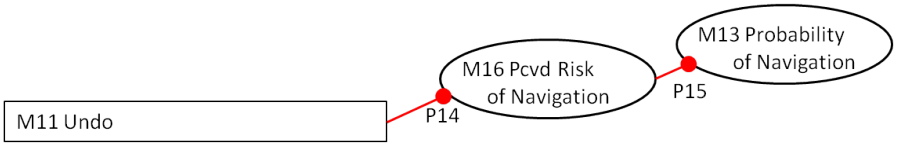
Modelers had particular difficulty with the `AddCollect` transform, because it was in a complex menu alongside other transforms that modelers did not understand well. For example in trying to perform the collection step of Task 3, EH1 got frustrated trying to drag a column from one table to another:

EH1: Um, add col... I just want to add the column. [reads several items off a context menu that pops up after he tries a drag-drop] Please, just add. I don't want to have to click any special options; just add; just do what I want!

All the "special options" he saw were transforms that were consistent with the context of his dragging action: the menu contained not only `AddCollect` with a submenu of aggregate functions it could apply, but also transforms for `next`, `previous`, and `addState`. This issue is an example of what Green and Petre [47] called "imposed guess-ahead" or "premature commitment": the interface forces users to specify things in an order other than how they cognitively occur. One way to resolve this premature commitment problem would be to redesign Lea3/T in a way that separated "adding a column" and choosing the exact semantics of the column's addition into separate transforms.

Recommendation: Designers can partly reduce perceived navigation cost by designing a L_{EA}/*Transform* with smaller outdegree from each query. One possible strategy is to introduce intermediate nodes in the topology to break up the introduction of transforms with multiple arguments, for example as we did with the transforms associated with `agg` in Section 7.4.3.

8.3.5 Results: *M11 Undo*



Providing "Undo" is well-established usability advice for reducing perceived risk, (e.g. [24]) but many modeling tools do not have "undo" for features used in evaluation. Stepping backwards in the debugger or changing to a different program visualization may require typing commands at a prompt to restart model execution from the beginning; the state of the interface before taking these actions is not retained.

EAST posits (P14) that "undo" decreases the perceived risk of an action, making navigation more likely. In this study all nine modelers used undo, between 1 and 9 times; they also sometimes implicitly undid by closing new queries and returning to a previous query by name.

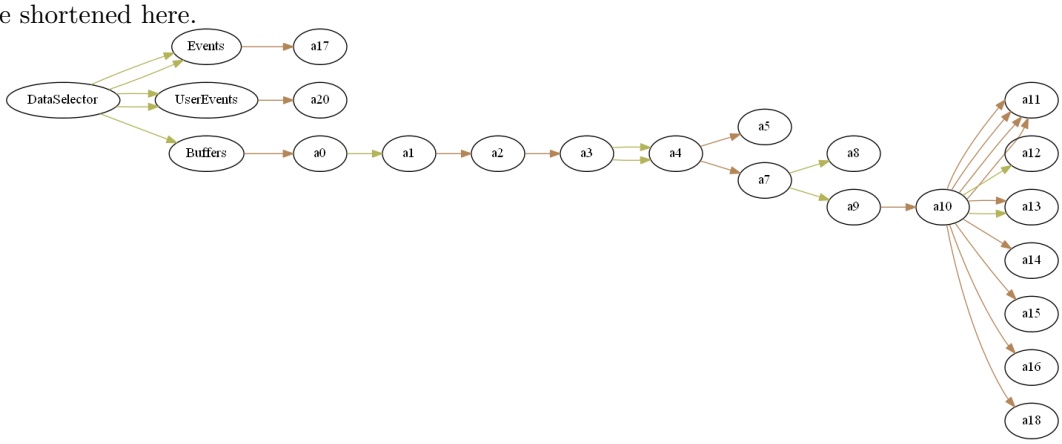
Modeler EO2 articulated the perception of lower risk that comes from an "undo" feature:

EO2: Now if it'll give me the list, and that I'm not sure of, ... I'm not confident in this, but I feel like I can undo it. So I'm just going to select it.

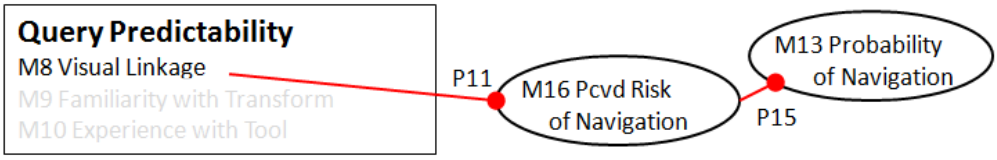
Modeler EG1 used undo heavily in his search for an appropriate operation in Task 3; as Figure 8.7 shows, he tried seven different operations, some of them multiple times, using the undo feature. He did not mention risk or cost during this sequence, but the number of navigations here suggests that the lower risk encouraged exploratory navigations.

Recommendation: Consider tool infrastructure that allows for undo in evaluation tools.

Figure 8.7: EG1's navigations for Task 3. Ovals are queries and arrows are navigations. The wide fanout at the right shows EG1's experimentation with menu items available when right-clicking from the query at the second-from-right column. Automatically generated query names are shortened here.



8.3.5.1 Results: Linkage between representations (*M8*) reduces perceived risk



EAST posits (P11) that when a modeler performs an unfamiliar transformation, good visual linkage will help them predict the effect of similar navigations in the future, lowering their perception that the navigation is risky, and thus increasing the probability that they will perform the navigation next time.

For the purposes of this analysis, we measure *M8*'s visual linkage geometrically: we consider a *high* visual linkage to be one in which *few* distinct types of the following geometric operations would be necessary in a hypothetical animation that transformed output of one query into the next query's output: horizontal translation, vertical translation, insertion, deletion, and substitution. For example an operation that deletes some rows from a grid and shifts the remaining rows together to be contiguous would have just two kinds of operation: horizontal deletion and vertical translation. By our measure, this is a higher (and thus better) visual linkage than an operation that deletes both rows and columns and both vertically and horizontally translates the remaining cells to consolidate them.[3]

[3]This measure neglects techniques that can help perceptually connect two visualizations, such as animations

Although we discuss alternate ways of visualizing abstract *Table* objects more in Chapter 9, in this study, EAST-Env simply depicted a *Table* visually as a spreadsheet-like grid. In grid form, the geometric operations describe above correspond in obvious ways to differences between *Table* objects: Lea3 operators that change the *Table* data by adding or removing rows or columns caused the exactly analogous change to the grid representation: deletion, addition, or shifting to fill in gaps left by missing rows.

Transforms with high (good) visual linkage within the grid representation used in this study are shown shaded in the "Good Visual Linkage" column in Table 8.2. For example animating a filter operation would involve deleting rows and translating them to close up the gaps; unpacking would simply add a new column.

Transforms relating to `agg`, `makeState`, and `segment` had fairly low (poor) visual linkage within the grid representation. Transforms introducing these operators all removed both rows *and* columns, then added at least one new data column. Transforms introducing operators that involved two tables also had poor visual linkage: `addState`, `collect`, `next`, `previous` and `simul` operators. EAST-Env displayed the operators' two input tables in separate windows, and the operators combined them based on temporal column values. As Figure 8.8 illustrates, a hypothetical animation building the query result from the two prior tables would have to translate rows both horizontally and vertically, delete rows that were not joined, delete columns that were not part of the join, and translate horizontally to consolidate cells.

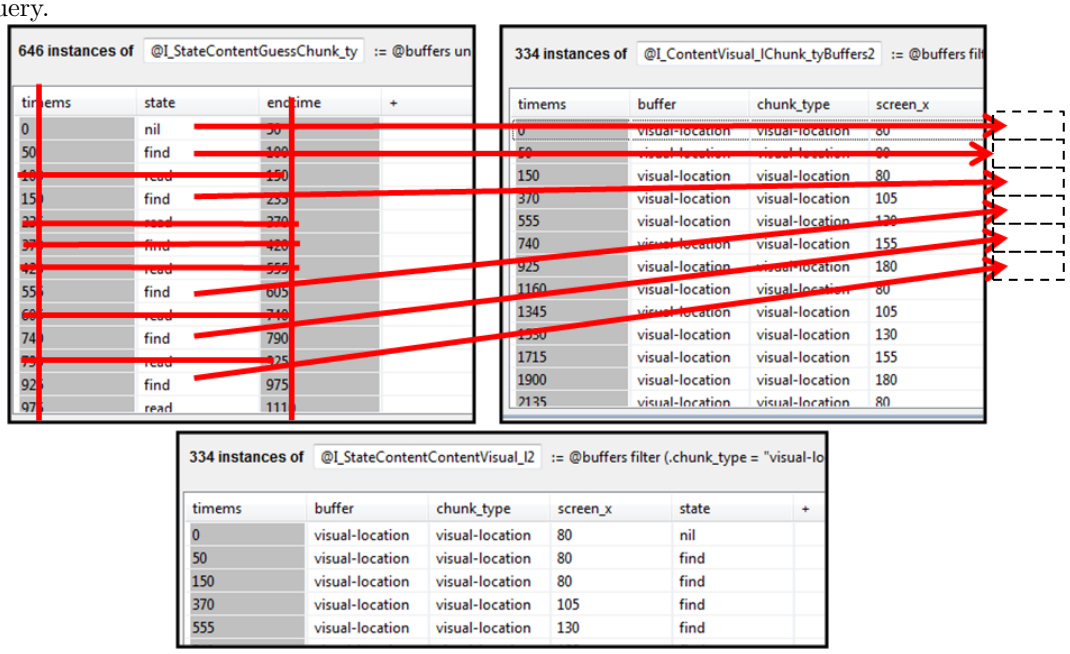
As Table 8.2 shows, the operators that turned out to have the highest probability of navigation had good visual linkage as defined above. The lone exception is the `agg` operation; but as Section 8.3.5.2 speculates, this might be attributed to the operator's familiarity as an SQL construct.

Although we did not have a direct way of estimating perceived risk, modelers' reactions to low-visual linkage output after navigations suggest that the geometrical complexity of the transformations may have increased their perception that the navigations were risky. There were 16 times in the experiment when modelers performed an operation, explicitly said that they did not understand the output, and then immediately either closed or undid the operation. Despite the fact that these poorly-linked operations were relatively rare, they accounted for 9 (56%) of all of such expressions of confusion by the modelers. In other words 12% of the poorly-linked operations led to explicit statements of confusion, compared to 2% of the well-linked operations.

---

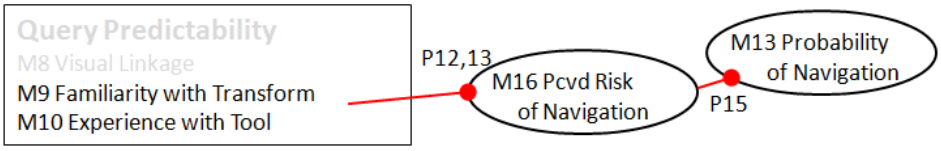
and highlighting of differences. It is common for visualization frameworks to offer functions that support visual linkage; for example both Eclipse and Protege have API support for highlighting or animating changes between displays. However this measure is sufficient to distinguish among the visualizations discussed in this chapter, since EAST-Env did not employ these techniques.

Figure 8.8: A hypothetical animation of a transformation adding the `addState` operator. Rows and columns in one table would need to be deleted, and both horizontal and vertical translations would be needed. Top: two tables before the transformation. Bottom: the output of the new query.



Recommendations: (1) Design representations and transforms together so that every operator has at least one representation for which its effect has high (good) visual linkage (*M8*). Adjust the context-sensitive cueing so that it takes into account the current visual representation, preferentially offering to the user transformations that have high visual linkage *in that representation*. (2) Consider depictions of transformations that overlay or animate between the new and old representations, to make the relationship between them clearer. There is a body of research about how to show the relationship between views during transitions to show changes to data (e.g. Chang and Unger's user interface for SELF [26]) or different visual representations of code (e.g. Dessart et al. [36]).

8.3.5.2 *M9* and *M10*: Experience with the operations



A final pair of factors that EAST posits (P12, P13) to encourage navigation are *M9* and *M10*: familiarity with the transform from other contexts and from within the EA tool itself, respectively. In this study, modelers drew on their experience with other tools, and well as their accumulating experience with EAST-Env, to inform their predictions of what navigations in EAST-Env would do, thereby lowering their perception of those navigations as risky. For example, modelers often tried navigations motivated by some connection to what they knew from other languages, or related to their process of performing the task by hand. As Table 8.2 shows, four of the five most commonly used operators also exist as constructs in either Excel or SQL. Modelers' reliance on familiarity with other tools (*M9*) was evident in the strategies they verbalized when searching for navigations that would accomplish tasks in the interface.

Modeler EE1 drew several times on SQL as an analogy for thinking about EAST-Env's capabilities. Although EAST-Env's implementation depended in part on a SQL database, the same words were not much used in the menu cues for navigations, so this analogy was not very helpful. For example near the beginning of the experiment when he was first trying to figure out how to filter, he wished for a command line where he could type SQL:

EE1: I'd like to actually type in my select statement: "where ... is equal to this, this, this sort by".

Differences between EAST-Env and SQL's familiar set of operations threw him off the trail later, when he wanted to create a calculated column, but looked for it under the Filter menu:

EE1: Now we just want to subtract next time from [the other time] ... OK, this would be a weird select clause. So, select seemed to go like filters [looks around under Filters in the operator-cued menu] No, no, no, this minus that. Hm, I don't want to filter, I want ... select this minus this from this. OK, how would we do that? [spots Calculate in the menu] Calculate! Well calculate sounds like something that could do something magical.

When EH1 wanted to do a subtraction, he thought of Excel:

EH1: So if this were Excel I would just subtract, but it's not Excel. How about a delta time... [spots Calculation in the menu] Calculation! Add column.

EH1 also wished for other familiar tools when looking for a way to add an end time column matching the previous row's time, and wished he could do it in a stateful or imperative way:

EH1: I just want to add time milliseconds. ... Where are my finite state machines? Where is Python?

In a related process of reasoning from experience, the "familiar transform" of *M9* was modelers' own manual accomplishment of the task: Modelers drew strategies from self-observation of their own cognitive process, which drove their search for appropriate navigations. In a few cases, after giving up on an operator and deciding to some step by hand, modelers would see what they themselves were doing and look for an operator to do the task for them.

EK1, for example, at the point in Task 3 where `collect` would be useful, at first simply decided to do the task by hand from that point. He opened up a list of slot values in one window, and the list of trials in another, and resolved to start:

EK1: I'm just going to manually check now and compare user events... [Looks at the first few values and compares them to the other table] Uhh, is there a way for me to aggregate over a range of times?

In attempting part of Task 4 by hand, another modeler noticed that he was scrolling past "nil" values in the table:

I'm just going to check it manually again. If I find one example that contradicts it then I can say that it's false. ... Last state s... uhh... I guess I can get rid of the nils. Getting rid of the nils from the last and the predicted. [right clicks and filters out nil]

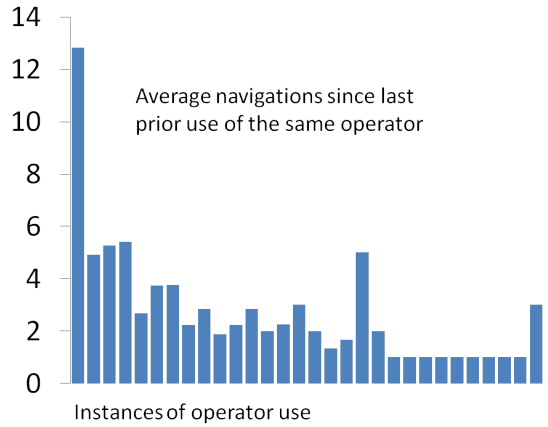
Recommendation: Designers should favor transformations that users are familiar with (*M9*), either from other tools or from their own manual process. This allows the designer to use familiar words and interface techniques the modelers are already familiar with, instead of having to invent new ones, and bear the associated cost of scaffolding new users into fluency with the novel operators. Note that the NP+ design process ensures that the outputs produced by a Lea3's queries should be close to what users will want to see, although the designer still has some flexibility in how to design the steps of those transformations.

As proposition P12 posits, experience with the tool's own operators (*M10*) also helped modelers predict what the operators would do. Figure 8.9 shows that modelers were more likely to use an operator again the more times they had used it.

8.4 Conclusion

This chapter has described a qualitative, summative evaluation of our Evaluation Abstraction Support Theory. We found evidence to support EAST's predictions about how modelers will choose to navigate. These claims imply that EA tools should provide trace querying features that are reusable, composable, and that reward modelers with answers to their questions along the way as they are building their queries. They also imply that affordances for transforming

Figure 8.9: The number of navigations separating subsequent uses of an operator shrink as the operator is used repeatedly. The k^{th} bar in the graph represents the k^{th} use of an operator, and the height of the bar indicates the average number of navigations since the previous use (or since the start of the experiment, in the case of the first bar).



queries should provide a small number of familiar, contextually appropriate tools, whose effects are visually evident, and undoable.

We also found that:

- Not only did immediately useful feedback at each step help modelers build queries, but in fact that incremental data was so useful that modelers sometimes repeated the building process even after their query was complete.
- Context-sensitive cues helped modelers perform common tasks very quickly, although they became a hindrance when the right context for an operation was not easy for modelers to find. Context-sensitive cueing can be helpful, but needs to be backed up with a non-contextual facility for building queries.
- We gained new insight into how model traces differ from other domains of information foraging theory, in that cues describing operations, rather than cues taken from the data itself, seem to provide better scent. We hypothesize that was because cognitive modelers often looked for patterns in the data as opposed to individual data values, and because the data values were too similar to help to differentiate the patches modelers derived from other patches. Thus, cues inspired by enrichment operations rather than patch-to-patch navigation may be more effective for this kind of environment.

The ultimate intent of EAST is to provide guidance to developers of evaluation abstraction support tools, such as debuggers, testing frameworks, and IDEs. The research literature is littered

with powerful evaluation abstraction tools that have little uptake in professional practice. It is our hope that the measure constructs that EAST posits to influence modelers' navigations, and hence modelers' use of features, provide the designer with a checklist of features to include in their designs, that have both empirical and theoretical support.

Chapter 9 – Generality: evaluation abstraction support beyond ACT-R

Most of the research described before this chapter has focused on supporting modelers who use ACT-R as a modeling language and paradigm. How far does a particular evaluation abstraction language developed for one paradigm, using NP+'s methods, generalize to other cognitive modeling paradigms and languages? Is the design of Lea3 applicable beyond the paradigm used by the modelers we studied (in this case, ACT-R), or specific to the ACT-R paradigm, or somewhere in between?

Our hypothesis was that despite being designed around empirical results from ACT-R modeling, Lea3 would nonetheless be powerful enough capture a useful subset of modelers' questions about models in other paradigms, and that the ACT-R-specific design choices we had made would turn out to be relatively unimportant.

Supporting this premise is the fact that every cognitive modeling project we have seen has simulated external, visible human behavior in some way, and in these projects modelers have attempted to reproduce some aspect of that external behavior as faithfully as possible. So one might expect at least some of modelers' evaluation abstractions – the ones over the domain of simulated behavior events – to relate to human behavior, and be relatively independent of the particular implementation method and paradigm that the modeler chose.

On the other hand, modelers' evaluation abstractions also refer to *internal*, explanatory model events, such as buffer activity or rule firings, which the modeler may have no ability or desire to directly map to phenomena observable in real-world experiments. Since such hidden constructs are less closely anchored to observable behaviors, the space of possible constructs is less constrained; the constructs might differ between paradigms in more pronounced ways. These different structures could influence the kinds of evaluation abstractions modelers make about them. For example modelers might form different evaluation abstractions if they have implemented their model using discrete inferential logic in a language like Prolog (e.g. [7]), than they would if they had built a model of the same behaviors using a neural network.

To investigate this further, we applied an expanded version of the EAST-Env Eclipse plugin described in Chapter 8 to support five different cognitive modeling paradigms in use at the Air Force Research Laboratory. We studied the detailed log files of models in these five different paradigms, talked to modelers about their evaluation practices and tools, and went through the exercise of creating EAST-Env import software for each of the five.

In doing this we found that the Lea3 language provided enough general-purpose support that a variety of useful questions could be answered in each of the paradigms, only transforming the trace files structure enough to import them into the timestamped database format that EAST-Env's implementation of Lea3 required.

9.1 Procedure

In the summer of 2012, we helped eight users at the Air Force Research Laboratory install EAST-Env, adapting it to three other cognitive modeling paradigms beyond the two (ACT-R and RML) it already supported, for a total of five modeling paradigms (Table 9.1).

Table 9.1: Participants in Study 6 and the modeling paradigms they were using.

Modeler	Paradigm	Special issues
GM1	ACT-R	GM1 was interested in rules' changing "utilities" (i.e. competitiveness with each other) over time
GM2	ACT-R	Model had rules that could fire instantaneously, not typical of ACT-R
GM3	ILM	Python-based model, used Python data structures
GM4	ACT-R	Interested in using EAST-Env for regression testing; collaborating with GM2
GM5	ACT-R	Wanted to combine information from multiple model runs under different conditions
GM6	RML	Clock not always available; strict sequentiality was not mandatory
GM7	NERML	Esper-based language; import required use of JSON
GM9	EPIC	Many events can happen simultaneously

EAST-Env was a different version of the Eclipse plugin described in Chapter 8; in addition to supporting importation of trace files and menu-based construction and persistence of Lea3 queries over those traces, it also added support for four different ways of visualizing Lea3 query outputs. Sample visualizations are shown in Figure 9.1; they are accessible through the same menus as Lea3/T transformations.

For each of the five paradigms, we asked modelers for log files at the greatest amount of detail that their paradigm provided, for some model they were currently working on. Using the files, the paradigm's documentation or code, and asking modelers for information to inform our

design, we built import software for each paradigm. The process of mapping these files to EAST-Env's world and testing the results provided real-world data that we analyzed to investigate commonalities and differences between platforms. We also interviewed modelers about the kinds of questions they most wanted to explore with their models, and the current techniques they were using to abstract model runs into usable output, and when possible, we used their own evaluation abstractions as examples to assess the success of each paradigm's adaptation to EAST-Env's way of structuring trace data.

In describing the results we consider not only the questions that can be answered by applying Lea3 queries and examining the output tables in a grid format as previously shown in Figure 8.2, but also how useful the query outputs are when visualized in other ways (Figure 9.1).

There were five factors that influenced how well Lea3 could support the paradigms: time representation, telicity, recursion, object persistence, and dynamic typing.

9.2 Factor 1: Time representation

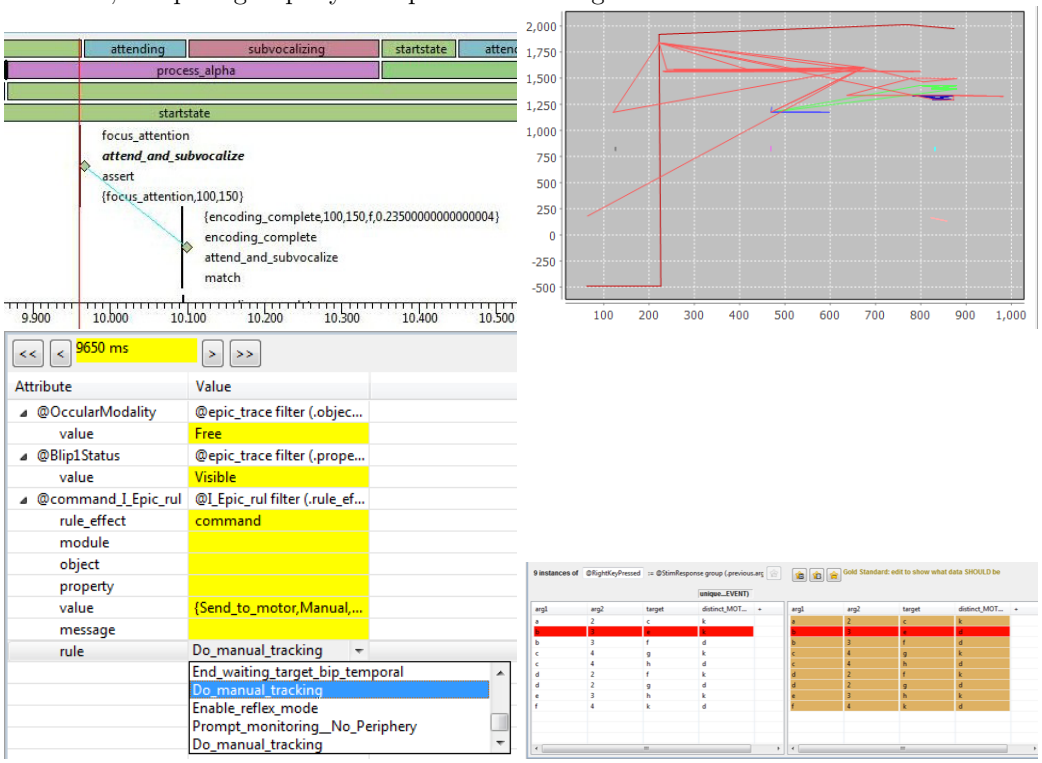
Sequence and Simultaneity All five paradigms allowed for some forms of parallelism in models, but different forms of parallelism seem to have different implications for the semantics of Lea3 operators, and different implications for how visualizations are produced from them to ensure valid and comprehensible visual representations.

First, considering the Lea3 operations' semantics, recall that the three sequence operators, `next`, `previous`, and `simul`, all have time-based criteria for linking rows between two tables (Section 7.2.3). When several simultaneous candidate rows match the time criterion, the first relevant row is chosen. With ACT-R data, it takes only minimal filtering to get a list of events down to a point where there are no simultaneous events to trigger this condition; but this is not the case with all paradigms. As Figure 9.2 exemplifies, there far fewer rows in the ACT-R trace that have the same timestamp than there are in the EPIC trace. This means that in ACT-R when using `next` to combine two tables, an ACT-R modeler would necessarily have to do less filtering in advance than an EPIC modeler to ensure a specific kind of row was chosen by the `next` operation.

The `makeState` operation, similarly, which adds an end time column to each row copied from the start time column of the next row (Section 7.2.9), segments the trace into time segments of greater than zero length only when each row starts at a later time than the previous; again, an EPIC modeler would in general have to do more filtering if they wanted a query whose output rows provided a set of unique timestamps.

Even in cases where a modeler was content with Lea3's "anchoring" property of choosing just the first matching row, EPIC traces pose another challenge that ACT-R traces do not: they were written to two files which must be interleaved on import into EAST-Env, and the programmer

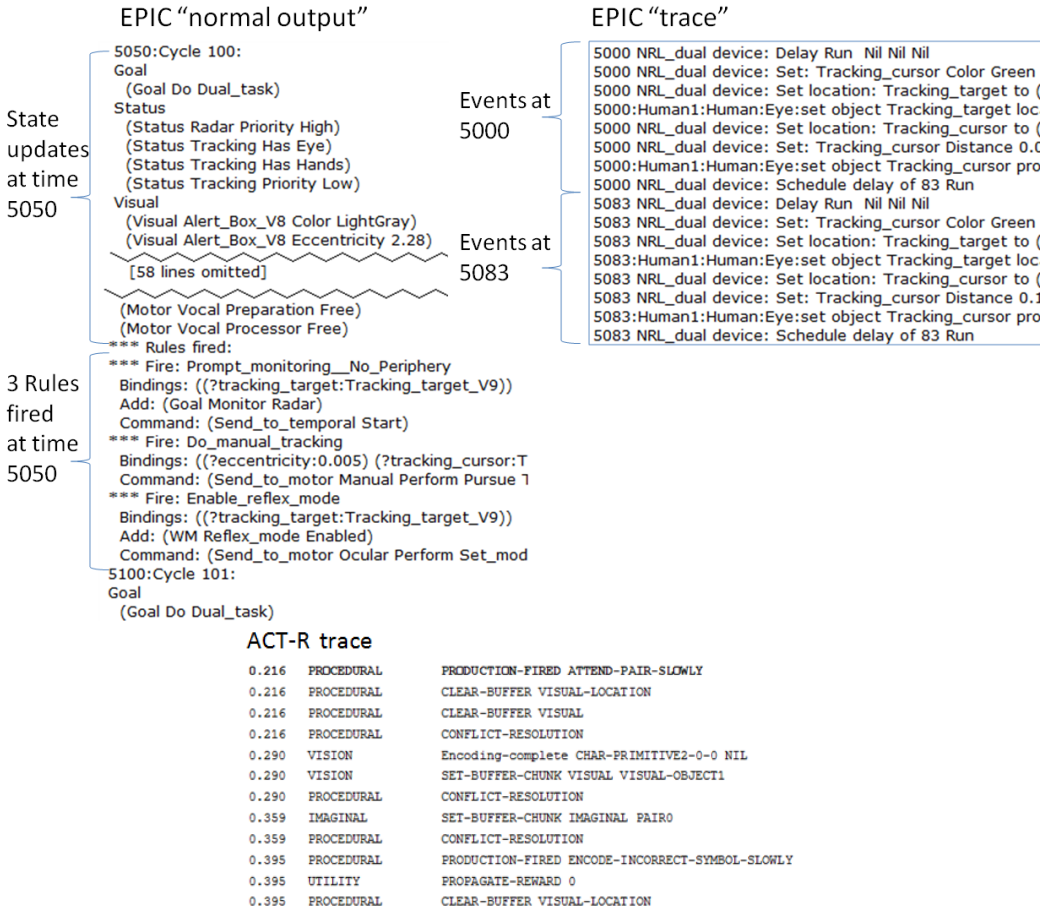
Figure 9.1: EAST-Env allows modelers to visualize the output of Lea3 queries in several ways besides the tabular view depicted in Figure 8.2: (a) Timeline, showing states as bars, events as diamond glyphs, and `next`, `previous`, `simul` relationships as connecting lines; (b) Plot, in which modelers can draw line graphs of Lea3 query outputs, specifying which columns to interpret as x, y, and color values (c) Stepper, which shows a more compact form of a single row of a query, emulating the time snapshots typically shown by debugging tools, and (d) A regression testing visualization, comparing a query's output table to a "gold standard" the modeler can edit.



of the import algorithm must hardcode some arbitrary choice about which of the log files' events come first in the EAST-Env trace, when they have the same timestamps. Figure 9.2 shows timestamps in two files, although in the figure the same timestamp does not happen to appear in both snippets of log.

Thus in both paradigms, modelers can make use of time-related Lea3 operators, but there is somewhat more burden on an EPIC modeler in the preliminary step of filtering the tables down to unique times.

Figure 9.2: EPIC traces show more simultaneous events than ACT-R traces. The EPIC log file called "normal output" (top left) shows 67 lines of state at time 5050 (any of which may have changed since the last update), and three simultaneous rule firings. The EPIC "trace" output (top right) shows eight events at times 5000 and 5083. In contrast the ACT-R trace (bottom) shows fewer simultaneous events.



So in most cases, multiple simultaneous events in a query output table do not cause special problems, and when they do, modelers can choose to first filter on other criteria until only truly sequential rows remain; they will have to do this more often with EPIC than ACT-R, but at least there is a viable strategy.

Beyond the question of the appropriateness of the Lea3 operators to nonsequential event streams is the matter of how a user interface such as EAST-Env can usefully present visualizations of the results.

In ACT-R, for the most part, a visualization generated from a table of events can usually be laid out in a one-dimensional visualization, as can rule firings, without any parallelism distorting the semantics of the visualization. This is because in ACT-R, activities can take place simultaneously, but when they do, they often involve different buffers or modules, representing different kinds of activity within different parts of the brain. Within each module, though, activity is mostly sequential. For example, the visual system can encode things it "sees", while, concurrently, rules are busy firing in the central "procedural" module; but only one such rule can fire at a time, and the visual system only encodes one symbol at a time. Productions, motor actions, memory actions all happen one at a time. Buffer changes can be simultaneous but are distinguished by buffer name and can be filtered.

In EPIC, in contrast, multiple rules can fire at once. So in any visualization of rule firings, it is not possible to lay out rule firings along a one-dimensional timeline without overstriking simultaneous events (and therefore making a visualization of it difficult for the modeler to use).

RML fell somewhere in between EPIC and ACT-R in this regard, since multiple transitions (RML's equivalent of rules) could fire at the same time, but only in separate, named "behavior models". Because there was no parallelism within a single behavior models, the modeler can simply graph or depict the events separately.

Another related distinction between paradigms is that ACT-R rarely deals with undifferentiated collections of things: e.g. a buffer can hold only one chunk at a time. In some other paradigms, (and even in some variant uses of ACT-R), a *collection* of things can be stored in a location, and act or be acted on concurrently, thus leading to multiple simultaneous events of the same type. This is another reason some other paradigms may require visualizations designed to represent multiple simultaneous events. In these paradigms Lea3's time-based operators will require more pre-filtering from the modeler.

Without empirical work with EPIC modelers (or other paradigms where multiple rules can fire at once), it is not clear how, or whether, the issue of simultaneity should be extended. When, as in ACT-R, rules can only fire one at a time, then it makes sense to ask what the "next" rule that fired was. When multiple rules can fire at once, however, it is an open question how modelers might interpret the "next" relationship: would it be the next one that meets some criterion of relevance? The one at the next timestamp that happens to be listed in the trace? Or

perhaps modelers would simply not talk about "next" rule firings in such a scenario. Although it is possible for a modeler to filter a set of EPIC events down to the point where only non-simultaneous events remain, and apply the next operator at that time, we do not know whether that is the most natural way for them to construct EPIC evaluation abstractions.

Clock Discipline Modeling paradigms also differed in how thoroughly and reliably they represented simulated time. ACT-R, for example, aims to precisely model the durations of cognitive events [4]. RML, on the other hand, was built for integration with other simulations, potentially with their own time disciplines [37], so the modeler is responsible for keeping track of time. In RML models, timestamps were sometimes inconsistent, absent, or nonmonotonic, since modelers only bothered with time when it met a modeling need. For example one of the sample models distributed along with RML simulates a vending machine, and does not have any variable representing time at all. In ILM, the modeler was only concerned with relative time proportions, and told us to just assume each new record in the model's log file was 50ms after the last.

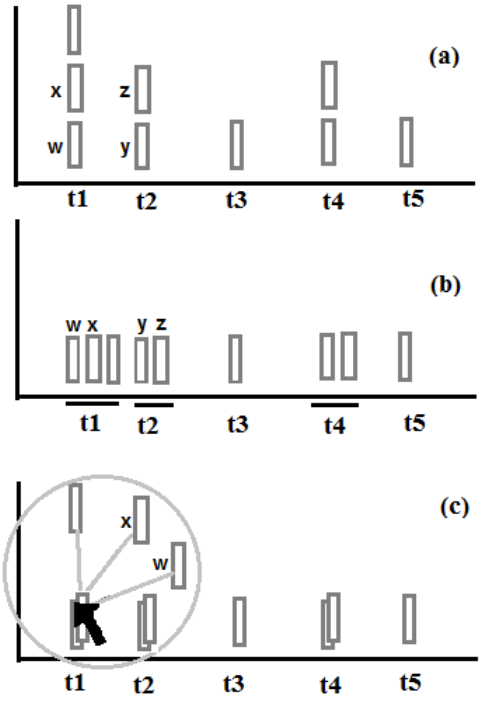
Implications for Visualization Some of the visualizations we designed with ACT-R in mind relied on the assumption that the concrete instances of an abstract description could be laid out unambiguously on a timeline. For domains in which multiple events can happen simultaneously, some design options are:

- Use one axis for time and another to make room for simultaneous occurrences. Several modelers explicitly suggested this solution, so it has the advantage of being expected. One danger is that the user can wrongly interpret that other axis to have some particular meaning, i.e. if a and b are simultaneous, then later c and d are simultaneous, care should be taken to distinguish the cases where a and c have some special relationship (Figure 9.3a).

- Another option is to impose an arbitrary ordering and stretch the time axis to fit simultaneous events. EAST-Env can also be configured to import traces using character position in the input file as time, or to make use of an additional "event id" field in the file format. This, like character position, also allows for sequentiality independent of time, but allows more control over what is sequential and what is simultaneous. One advantage of this option is that it avoids the spurious correlations that can be implied by the parallel portrayal, or obscuring items by overwriting them. Another advantage is that it can cause the viewer to infer causal dependencies between simultaneous events, which, surprisingly, was the case for some models. Of course, this is a disadvantage if there is no such causal dependency.

  For example participant GM4's theory about language parsing involved rules that fired every 50ms, as in ACT-R; but GM4's rules were more generic than ACT-R allows, so he

Figure 9.3: Three ways of distinguishing simultaneous events on a timeline. In all three, time is the horizontal axis, and the vertical axis has no special meaning, but is only used to make room for multiple events. Style (a) gives the impression that there is some relationship between events w and y, and between x and z. Style (b) gives the impression that event x is "after" w. Style (c) visually indicates that there are simultaneous events, then shows them on hover, distributed in a way designed not to imply an ordering.

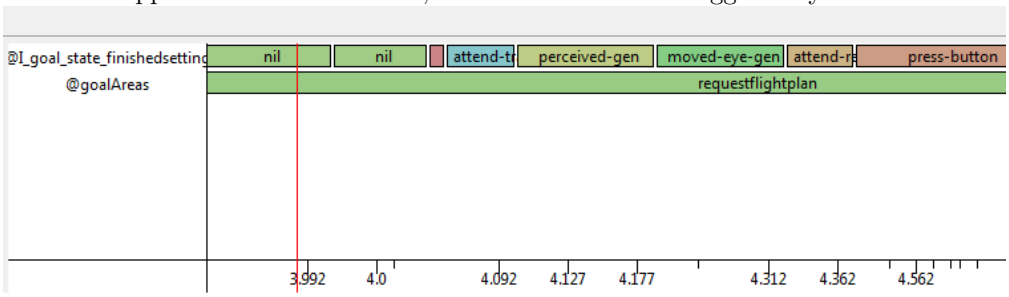


represented each of his theory's rules as a small collection of ACT-R rules, fired in sequence with no time elapsed in between them.

We handled this case in EAST-Env through a feature that gave the modeler the option, on importing a trace, to treat character position in a log file as "time", but then explicitly tag records with the true simulation time, as an extra attribute. The timeline display then shows events spaced out with a rhythm proportionate to actual model activity, telescoping simultaneous events out, and shrinking long idle periods down. This is diagrammed in Figure 9.3b, and a screenshot is show in Figure 9.4.

- A third option would be to use an approach in which simultaneous events are marked with a special symbol that would be expanded on mouse hover to show all the matching events, as in Figure 9.3c.

Figure 9.4: Fluid time visualization: a model worked on by GM2 and GM4, in which multiple events could happen in the same time step. The width of bars in the chart corresponds to their extent in the input file, and the time legend at the bottom is labeled with simulation time only at points where the time is known for certain. The short unlabeled red bar and the adjacent blue bar labeled "attend-t" start at the same simulation time, but GM4 told us that he considers the blue event to happen "after" the red one, since the blue one is triggered by the red one.



Implications for evaluation abstraction language design Lea3's principle of "anchoring" in operations such as `next` and `collect` depends on the assumption of an implicit ordering of simultaneous events. This provides graceful degradation in the case of pathological situations, for ACT-R models in which such simultaneous events are rare; but when simultaneous actions are common and fundamental to the paradigm, as in EPIC, the arbitrariness of this choice is more likely to become unpredictable and thus not useful to the modeler.

Depending on the characteristics of the paradigm and the needs of the paradigm's users, then, a language designer looking to adapt Lea3 has several choices:

- Choose an interpretation based on an absolute fixed ordering, as Lea3 has done. When a trace is generated from a single log file, the position in the raw log file can be used as an artificial "time" for this purpose. We tried to assign a fixed ordering for representing EPIC, which used a pair of log files (e.g. Figure 9.2), and although it was sufficient to represent and work with the traces, this solution still required troublingly arbitrary ordering choices between simultaneous events in the separate files, in which the tool had to generate arbitrary orderings that the modeler would not be able to guess at.

- Have time-based operations fail if ambiguity is found in the data, and prompt modelers with a choice of filters to apply that would result in an unambiguous sequence.

- Include all the possible interpretations, and drop the "anchoring" requirement, sacrificing the *M8 Visual Linkage* motivated requirement that operators should minimize the amount of change to the underlying table (or, satisfy *M8 Visual Linkage* in some other way).

- Choose one interpretation arbitrarily, but also include a field that characterizes the set of interpretations that were not chosen (e.g. a count of them). Add a GUI affordance to a query that enumerates the hidden interpretations. This would be especially compatible with the visualization solution in Figure 9.3c.
- Design entirely new time-based operators by doing new empirical work to find out how modelers in a simultaneous-rules paradigm talk about "sequence".

9.3 Factor 2: Telicity

As discussed in Section 7.2.9, atelic data is data for which identical values repeated over time can be safely merged without change in semantics; and telic data is data for which repeated presentations of the same value have some distinct significance.

When a variable's assigned value is indicated in a log file, (e.g. "At time 3.200 the value of x is: 4.02") there are several possible interpretations: the notification could be telic or atelic, and if it is telic, it may mean that x changed to 4.02 at exactly time 3.200, or it could merely be a sample of a value that is changing more quickly than the log is reporting.

With unfamiliar modeling paradigms logging a great number of variables, we could not track down the telicity semantics of everything reported, and in fact it is not clear (see Section 7.2.9) that telicity can be determined outside the context of a modeler's task. So EAST-Env's import facility represented them all as simple atelic events in most cases.

Implications Outside of very specialized circumstances, EA support will need ways for the modeler to impose their understanding of events' telicity. It follows that the designer of an import utility can be less concerned about the telic semantics of log files to be imported.

On the other hand, reasonable defaults for frequently-used values in default visualizations might provide an easier learning curve for new users. At the request of participant GM6, we built in a default evaluation abstraction and an associated visualization that would appear immediately on import of an RML log file, that relied on a telic interpretation of automaton state (i.e. it merged long sequences of self-transitions in a finite state machine into a representation like Figure 9.4 that showed the machine's state as having a start and end time).

9.4 Factor 3: Recursion

None of the five modeling paradigms allows for recursive function calls. If they had, this would have been a challenge because a naive import of logged local variables in a recursively called function would result in a table that would be difficult to manipulate in a sensible way using

Lea3 operators. This is because information is lost if variable names and their changing values are simply printed: without some mechanism for representing the depth of recursion, there is no way to distinguish trace lines of a local variable's instances in different call stack frames. Figure 9.5 gives an example of this problem.

Figure 9.5: A recursive C function and a log of a local variable. The trace is not sufficient to reconstruct the lives of the different instances of `a` on the stack since the stack frames are not distinguishable in the trace. For example `a` in the top level holds the value 3 from time 1 to 9, when it is decremented; but this is not evident in the log since other recursive instances of `a` are interleaved, and indistinguishable.

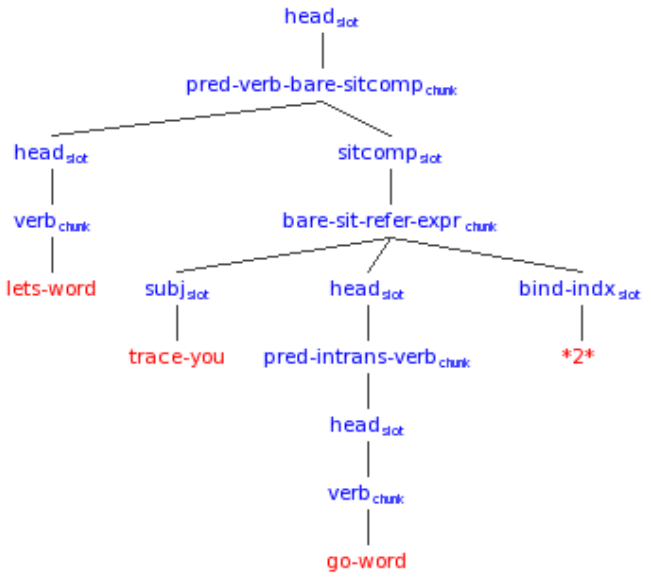
```
int recur(int a) {
    log("a", a);
    if (a>0) {
       recur(a-1);
       a--;
       log("a",a);
       recur(a-1);
    }
}

int time = 0;
void log(char *var, int val) {
    time++;
    printf("%d,\t%s,\t%d\n", time, var, val);
}

int main(int argc, char **argv) {
   printf("time,variable,value\n");
   recur(3);
}
```

```
time,variable,value
1,a,3
2,a,2
3,a,1
4,a,0
5,a,0
6,a,-1
7,a,1
8,a,0
9,a,2
10,a,1
11,a,0
12,a,0
13,a,-1
```

One ACT-R model, however, did use recursive data structures, representing English sentences comprehended as a parse tree of chunks in declarative memory. The modeler already had a custom visualization of these trees (see Figure 9.6), omitting and depicting different chunks from the diagram based on their chunk types. EAST-Env contained the raw data necessary to reconstruct such a tree, but to produce a useful diagram like this would have required a tree visualization tool, and also some added facilities in Lea3 for defining a set of objects recursively, by following all their references. No other cognitive modeling project we studied had tree-like visualizations.

Implications Lea3's generalization to languages with recursive behavior or data structures is an interesting open research question. A researcher who wished to apply NP+ to design of a language for examining recursive program traces should pay special attention to how users think

Figure 9.6: A syntax tree produced as part of the regression test output of the LANGCOMP project of Study 1, in Chapter 5. Branched nodes in the tree connect a chunk name to its attributes; nodes with single children connect an attribute with the type of chunk it points to. The software that produces this tree has been configured by modelers on this project to omit or include certain node types and attribute names, so that the resulting tree would match the linguistic aspects of the chunks, while hiding some features they consider to be merely part of the mechanism.



about recursion, and more thought would be needed about how to infer recursion behavior from trace logs, perhaps by adding more information to the trace, analyzing source code, or exploiting the modeler/programmer's knowledge of intended recursive behavior.

9.5 Factor 4: Object persistence

In ACT-R, chunks are created, never destroyed. Thus when no information about a chunk or its slots appears in the trace for a while, ACT-R modelers can safely assume it still exists. Applying Lea3's `mergeRows` operator to occasional update events about a chunk's contents would correctly turn these events into correct representations of their changing state.

In the other paradigms, however, this assumption did not hold. In later versions of RML, for example, modelers could program a higher-level control mechanism to dynamically bring entire automata into or out of the model. EPIC traces also appeared to describe objects going out of scope, but there was no particular indication in the log file that the object had been destroyed

in the interim time, so applying the `makeState` operation to such entries falsely made it appear that the objects were merely quiescent, instead of nonexistent.

Implications For a paradigm that includes non-persistent objects, an evaluation abstraction language should designate a special "does not exist" value. This may require adjustments to the semantics of some operators. It also may require affordances for modelers to communicate their domain knowledge that, in certain cases, lack of data about an object's state should be interpreted as evidence that the object no longer exists.

9.6 Factor 5: Dynamic Typing

Lea3's semantic domain includes tables with fixed column structure. But all five paradigms, in one way or another, allow dynamic typing: a variable can hold values of a generic type which may hold a different collection of named fields at different times in a model run. We needed some way of representing this variability in a fixed initial trace that modelers could easily understand, and from which they could apply Lea3 operators to navigate to evaluation abstractions that interested them.

In ACT-R, for example, although chunks are never destroyed, they are swapped in and out of buffers. Since chunks have a slot structure that depends on the chunk's type, and a buffer can hold any type of chunk, then a different set of slot values may be associated with a particular buffer from moment to moment.

One unsuitable solution for this domain, used in some other database-based applications, would be to represent each distinct record type that ever occurs in the trace (distinguished by the set of named fields in the record) as a distinct table, and use indices to connect the tables. Fowler [43] called this "Concrete Table Inheritance".

Concrete table inheritance is unsuitable for representing arbitrary program traces, because there may be a large number of such records possible, and the whole list and their list of slot values may not be known in advance. So to represent "what the imaginal buffer was holding" in an ACT-R model, would require an unintuitive use of several tables with indices between them. Instead we used what Fowler calls a "Serialized LOB" (i.e., Serialized Large Object): the contents of the buffer are simply represented as a structure containing both the names and values of each slot, and "packed" (serialized) into a string field. (The types of slots did not need to be represented since, except for timestamps, Lea3 only has string values. An evaluation abstraction language with value type distinctions would need type annotations in the SLOBs as well.)

Section 7.2.11 describes how the modeler can access these packed fields.

Implications Designers of EA-supporting tools might consider SLOB as a cross-platform choice of representation for record data whose fields are not known to the tool designer, or which may vary over time or between models. An advantage is that it gives users a way of working with a heterogeneous collection of objects and seeing why they must be filtered before attempting to access fields that are not universal to the objects in the collection. Disadvantages include: they are difficult for users to read if presented naively as string data in a table, and require an extra manipulation step for users (i.e. the application of an unpacking operator).

Chapter 10 – Conclusion

When modelers talk to each other about models, they speak in a rich language of spatial, temporal, and statistical abstraction that allow them to explain model behaviors quickly, at a high level, with a broad brush. But sit them in front of a debugging tool, and they will talk aloud using a more impoverished set of abstractions. Observe what their hands are doing with the tool itself, and see that their abstraction vocabulary becomes poorer still again, narrowing down to just low-level strings and numbers structured by a handful of prefabricated layouts chosen by tool designers. Because of this impoverished human-computer communication channel, they find themselves sometimes spending dozens of minutes carrying out long repetitive manual tasks to answer questions they could ask another human in seconds, and that the computer could carry out in milliseconds, if only it were programmed to do so. Lack of abstraction in their communication channel is a bottleneck.

The research in this dissertation was motivated by the desire to remove this bottleneck, and enrich the conversation between a modeler and her tools with this power of abstraction. So we used a methodology we developed, NP+, to approach this problem: first we listened to modelers talking, to identify the elements of abstraction within the way they talk to each other when computers aren't listening. We learned more about these abstractions by coaxing modelers into using the fuller set of abstractions to interact with computers on an evaluation task, by having an experimenter pose as an intermediary Wizard of Oz, capturing the content and sequence of modelers' abstractions while scrambling to provide them with real interstitial responses from the computer. We codified and validated those findings into a domain-specific language, Lea3.

To bring these findings back to modelers in the form of a tool they could use, we developed a theory, EAST, describing the desired properties of tool features supporting evaluation abstractions; and we evaluated its claims for the case of cognitive modelers and Lea3. EAST recommends letting modelers build their abstractions up piece by piece, helping them learn something about the model at each step, and helping them express each abstraction by reusing or composing earlier abstractions. It also posits factors that can make abstraction-building cheap and predictable for users. EAST's concreteness serves a prescriptive function as well, allowing it to serve as a checklist of criteria that a tool designer can use to design new tool features, or understand why existing ones are not being used.

Finally we showed that the NP+ and EAST process can produce a result that generalizes beyond the particular situation of the formative work behind it, by demonstrating that Lea3 could generalize beyond the modeling paradigm, ACT-R, that we had developed it for. We also identified ways it could be better optimized for use with a range of other paradigms.

In conclusion, this research has demonstrated a feasible path towards better support for human-computer cooperation at a higher level of abstraction in model evaluation tasks, as well as a new methodology for doing this work along the way. It is our hope that researchers and tool designers in other domains will be able to use EAST and NP+ to improve the features of their evaluation tools.

10.1 Future Work

This research has uncovered several open questions that we would like to pursue further:

Integrating with debugging workflow We have shown that modelers can use EAST-Env with Lea3 to answer questions about model behavior when they are explicitly tasked with doing so, but it remains to be seen how this technique would fit into modelers' workflow when embedded in a larger task of debugging, testing, or validating their models. In particular we have experimented with, but not evaluated, ways to offer querying cues from within non-query views such as source code.

Further applications of evaluation abstractions We set out on this line of research with the intuition that there would be a great number of practical benefits flowing from tools having access to a database of abstractions that a modeler cared about: an awareness of "relevance" could be used to enhance error reports, generate tests, explain the effects of code changes, and many other things. This research has brought us to the point where we have a platform for exploring some of these applications.

Improvement of Lea3 We would like to formalize its semantics, selectively applying "Semantics First" [41] techniques retrospectively to look for a more concise set of operators that could represent the larger set that we identified, as well as applying lessons learned from Study 5 about better ways for the operators and transforms to interact. We saw examples of modelers looking for contextually inappropriate transforms that they felt "should have" been there, and these misconceptions about Lea3 suggest ways that Lea3 could be made more consistent.

Bibliography

[1] SQLite Home Page. `http://www.sqlite.org`, 2013. Retrieved May 21, 2013.

[2] Robin Abraham and Martin Erwig. GoalDebug: A spreadsheet debugger for end users. In *Proceedings of the 29th international conference on Software Engineering*, pages 251–260. IEEE Computer Society, 2007.

[3] Abstraction. Oxford Dictionaries. `http://www.oxforddictionaries.com/view/entry/m_en_us1219441?rskey=43Bhu8&result=2`, 2010. Retrieved November 30, 2010.

[4] John R. Anderson. *The Adaptive Character of Thought*. Lawrence Erlbaum Associates, 1990.

[5] John R. Anderson, Dan Bothell, M. D. Byrne, Scott Douglass, Christian Lebiere, and Y. Qin. An Integrated Theory of the Mind. *Psychological Review*, 111(4):1036–1060, 2004.

[6] John R. Anderson, C. Franklin Boyle, Albert T. Corbett, and Matthew W. Lewis. Cognitive Modeling and Intelligent Tutoring. *Artificial Intelligence*, 42:7–49, 1990.

[7] Jerry T Ball. Advantages of ACT-R over Prolog for Natural Language Analysis. `http://www.doublertheory.com/act-r-vs-prolog.pdf`, 2013. Retrieved January 9, 2013.

[8] Howard Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *AIAA Journal of Aerospace Computing, Information and Communications*, pages 1–43, 2010.

[9] M. J. Bates. Where should the person stop and the information search interface start? *Information Processing & Management*, 26(5):575–591, 1990.

[10] A. Bauer, R. Goré, and A. Tiu. A first-order policy language for history-based transaction monitoring. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, pages 111–128. Springer, 2009.

[11] Alan Blackwell. First steps in programming: A rationale for attention investment models. *IEEE Symp. Human-Centric Computer Langs. Envs*, pages 2–10, 2002.

[12] Alan F Blackwell. Metaphors we Program By: Space, Action and Society in Java. *18th Workshop of the Psychology of Programming Interest Group*, pages 7 – 21, 2006.

[13] Christopher Bogart, Margaret Burnett, Scott Douglass, David Piorkowski, and Amber Shinsel. Does my model work? Evaluation abstractions of cognitive modelers. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 49–56, 2010.

[14] Christopher Bogart, Margaret Burnett, Scott Douglass, Rachel White, and Hannah Adams. Designing a Debugging Interaction Language: An Initial Case Study in Natural Programming Plus. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2469–2478, 2012.

[15] Christopher Bogart, David Piorkowski, Margaret Burnett, and Scott Dougalss. Constructing questions about model behavior: composing, scent-seeking, and learning along the way. In *IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC)*, page (under review), 2013.

[16] Dan Bothell. ACT-R Tutorial, distributed with ACT-R 6.0 version 1.33 r766. `http://act-r.psy.cmu.edu/actr6/`. Retrieved August, 2009.

[17] Dan Bothell. ACT-R 6.0 Environment Manual. `http://act-r.psy.cmu.edu/actr6/EnvironmentManual.pdf`, 2009.

[18] Dan Bothell. ACT-R 6.0 Reference Manual. `http://act-r.psy.cmu.edu/actr6/reference-manual.pdf`, 2009. Retrieved August, 2009.

[19] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J Laviola Jr. Code Bubbles : A Working Set-based Interface for Code Understanding and Code Bubbles : A Working Set-based Interface for Code Understanding and Maintenance. *Human Factors*, pages 2503–2512, 2010.

[20] B. Bruegge and P. Hibbard. Generalized path expressions: A high level debugging mechanism. *ACM SIGSOFT Software Engineering Notes*, 8(4):34–44, 1983.

[21] M. Burnett, A. Sheretov, and G. Rothermel. Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. *IEEE Transactions on Software Engineering*, 28(6):576–594, June 2002.

[22] Julius Caesar. *Comentarii de Bello Gallico*. Eldredge & Brother, Philadelphia, 1882 edition.

[23] Jill Cao, Irwin Kwan, Rachel White, Scott D. Fleming, Margaret Burnett, and Christopher Scaffidi. From barriers to learning in the idea garden: An empirical study. *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 59–66, September 2012.

[24] J. M. Carroll and M. B. Rosson. Paradox of the active user. *Interfacing thought: cognitive aspects of human-computer interaction table of contents*, pages 80–111, 1987.

[25] John M. Carroll. *Minimalism beyond the Nurnberg Funnel*. MIT Press, Cambridge, MA, 1998.

[26] Bay-Wei Chang and David Ungar. Animation: from cartoons to the user interface. *Proceedings of the 6th annual ACM symposium on User interface software and technology - UIST '93*, pages 45–55, 1993.

[27] E H Chi, P Pirolli, K Chen, and J Pitkow. Using information scent to model user information needs and actions on the Web. *Proc. of the ACM Conference on Human Factors in Computing Systems, CHI*, pages 490–497, 2001.

[28] R.L. Cobleigh, G.S. Avrunin, and L.A. Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 208–218, Portland, Oregon, USA, 2006. ACM.

[29] S Colin and L Mariani. Run-Time Verification. In M Broy, B Jonsson, JP Katoen, M Leucker, and A Pretschner, editors, *Model-Based Testing of Reactive Systems, no. 3472 in LNCS*, chapter 18, pages 525–555. Springer-Verlag, 2005.

[30] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[31] Jacob Crossman, Robert Wray, Paul Nielsen, Randolph M. Jones, Al Wallace, and Christian Lebiere. A high-level symbolic representation for intelligent agents across multiple architectures. Air Force Research Laboratories, Tech report: AFRL-HE-WP-TR-2005-0006). Technical report, Air Force Research Laboratories, 2005.

[32] Fred D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13(3):319, September 1989.

[33] Wim De Pauw, Sophia Krasikov, and J.F. Morar. Execution patterns for visualizing web services. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, pages 37–45, Brighton, United Kingdom, 2006. ACM.

[34] Paul Dekkers. *Complex Event Processing.* Ms thesis, Radboud University Nijmegen, 2007.

[35] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger Canvas: Industrial experience with the code bubbles paradigm. *2012 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073, June 2012.

[36] Charles-Eric Dessart, Vivian Genaro Motti, and Jean Vanderdonckt. Animated transitions between user interface views. In *Proc. Advanced Visual Interfaces (AVI)*, pages 341–348, 2012.

[37] Scott A Douglass and Saurabh Mittal. Using Domain-Specific Languages to Improve the Scale and Integration of Cognitive Models. In *Behavior Representation in Modeling and Simulation (BRIMS)*, pages 17–24, 2011.

[38] Mireille Ducassé and Benjamin Sigonneau. Building efficient tools to query execution traces. In *INRIA Tech Report No. 5280.* 2004.

[39] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, Clearwater Beach, Florida, United States, 1998. ACM.

[40] M Eisenstadt. Tales of Debugging from the Front Lines. In *Empirical Studies of Programmers, 5th Workshop*, pages 86–112, Palo Alto, CA, 1993, 1993.

[41] Martin Erwig and Eric Walkingshaw. Semantics First ! Rethinking the Language Design Process. *International Conference on Software Language Engineering*, (LNCS 6940):243–262, 2011.

[42] EsperTech. Esper. `http://esper.codehaus.org/`, 2013. Retrieved May 14, 2013.

[43] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[44] W.T. Fu and P. Pirolli. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. *Human-Computer Interaction*, 22(4):355–412, 2007.

[45] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. *ACM SIGPLAN Notices*, 40(10):385–402, October 2005.

[46] T R G Green. When Visual Programs are Harder to Read than Textual Programs. In G. C. van der Veer, M. J. Tauber, S. Bagnarola, and M. Antavolits, editors, *Human-Computer Interaction: Tasks and Organisation, Proc. ECCE-6 (6th European Conference on Cognitive Ergonomics)*, number 1987. Rome, 1992.

[47] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[48] Volker Grimm. Visual Debugging: a way of analyzing, understanding and communicating bottom-up simulation models in ecology. *Natural Resource Modeling*, 15(1):23 –38, 2002.

[49] Philip Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th annual ACM symposium on User interface software and technology (UIST '11)*, pages 65–74, 2011.

[50] Abdelwahab Hamou-Lhadj, T.C. Lethbridge, and Lianjiang Fu. SEAT: A usable trace analysis tool. *Proceedings of the 13th international workshop on Program Comprehension - IWPC'05*, pages 157–160, 2005.

[51] S Haynes, M Cohen, and F Ritter. Designs for explaining intelligent agents. *International Journal of Human-Computer Studies*, 67(1):90–110, January 2009.

[52] M. Heinath, J. Dzaack, A. Wiesner, and L. Urbas. Simplifying the Development and the Analysis of Cognitive Models. *proceedings of EuroCogSci07*, 2007.

[53] Jing Jin and Pedro Szekely. QueryMarvel: A visual query language for temporal patterns using comic strips. In *Visual Languages and Human-Centric Computing, 2009.*, pages 207–214. IEEE, 2009.

[54] B E John and D E Kieras. Using GOMS for user interface design and evaluation: which technique. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(4):287–319, 1996.

[55] B E John, D D Salvucci, P Centgraf, and K Prevas. Integrating models and tools in the context of driving and in-vehicle devices. *In Proceedings of the Sixth International Conference on Cognitive Modeling*, pages 130–135, 2004.

[56] P. N. Johnson-Laird and Monica Bucciarelli. Strategies in syllogistic reasoning. *Cognitive Science: A Multidisciplinary Journal*, 23(3):247, 1999.

[57] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler : Interactive Visual Specification of Data Transformation Scripts DataWrangler. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, pages 3363–3372, 2011.

[58] William G Kennedy and Robert E Patterson. Modeling Intuitive Decision Making in ACT-R. In *Proc. Intl. Conf. Cognitive Modeling (ICCM)*, number April, pages 1–6, 2012.

[59] David Kieras. A guide to GOMS model usability evaluation using NGOMSL. `ftp://ftp.eecs.umich.edu/people/kieras/GOMS96guide.pdf`. Retrieved Sept 3, 2009.

[60] C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Beckwith, S. Yang, and M.B. Rosson. Supporting end-user debugging: what do users want to know? In *Proceedings of the working conference on Advanced visual interfaces*, pages 135–142. ACM, 2006.

[61] A Ko, B Myers, and H Aung. Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 199–206, 2004.

[62] A Ko, B Myers, and H Aung. Six learning barriers in end-user programming systems, IEEE Symp. *Vis. Lang. Human-Centric Comp*, pages 199–206, 2004.

[63] A.J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.

[64] A.J. Ko and B.A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.

[65] Andrew Ko and Brad A Myers. Finding Causes of Program Output with the Java Whyline. Technical report, Human-Computer Interaction Institute, Paper 164., 2009.

[66] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16:41–84, 2005.

[67] A. J. Kok. A formal approach to user models in data retrieval. *International Journal of Man-Machine Studies*, 35(5):675–693, 1991.

[68] Sascha Konrad and B.H.C. Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381., St. Louis, MO, USA, 2005.

[69] J. Koskinen, M. Kettunen, and T. Systa. Profile-Based Approach to Support Comprehension of Software Behavior. *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 212–224, 2006.

[70] Todd Kulesza, Simone Stumpf, Margaret Burnett, Weng-Keen Wong, Yann Riche, Travis Moore, Ian Oberst, Amber Shinsel, and Kevin McIntosh. Explanatory Debugging: Supporting End-User Debugging of Machine-Learned Programs. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 41–48. IEEE, September 2010.

[71] David Kurlander and Steven Feiner. A history-based macro by example system. In *Proceedings of the 5th annual ACM symposium on User interface software and technology (UIST)*, pages 99–106, Monterey, California, United States, 1992.

[72] John E Laird and Clare Bates Congdon. The Soar Users Manual Version 9.1. `http://ai.eecs.umich.edu/soar/sitemaker/docs/manuals/SoarManual.pdf`, 2009. Retrieved December 1, 2010.

[73] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott Fleming. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[74] Joseph Lawrance, Margaret Burnett, Rachel Bellamy, Christopher Bogart, and Calvin Swart. Reactive information foraging for evolving goals. *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, pages 25–34, 2010.

[75] Bongshin Lee, Catherine Plaisant, and CS Parr. Task taxonomy for graph visualization. In *BELIV'06: Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization*, pages 1–5. ACM, 2006.

[76] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, 10(1):39–74, 2003.

[77] Greg Little and Robert C. Miller. Translating keyword commands into executable code. *Proceedings of the 19th annual ACM symposium on User interface software and technology - UIST '06*, page 135, 2006.

[78] S. Ljungblad and L.E. Holmquist. Transfer scenarios: grounding innovation with marginal practices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 737–746. ACM, 2007.

[79] S. Maoz, A. Kleinbort, and D. Harel. Towards trace visualization and exploration for reactive systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 153–156. IEEE, 2007.

[80] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering*, 14(1):59–86, 2006.

[81] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, 2005.

[82] R.G. McDaniel and B.A. Myers. Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 449. ACM, 1999.

[83] Silvio Meira, Regina Motz, and Fernando Tepedino. A formal semantics for SQL. *International Journal of Computer Mathematics*, 34:43–63, 1990.

[84] GA Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63:81–97, 1956.

[85] Megan Monroe, Rongjian Lan, Juan Morales del Olmo, Ben Shneiderman, Catherine Plaisant, and Jeff Millstein. The Challenges of Specifying Intervals and Absences in Temporal Queries: A Graphical Language Approach. *Conference on Human Factors in Computing Systems (CHI)*, pages 2349–2358, 2013.

[86] P. Mulholland and S. Watt. Learning by building: A visual modelling language for psychology students. *Journal of Visual Languages & Computing*, 11(5):481–504, October 2000.

[87] Brad Myers, David A Weitzman, Andrew J Ko, and Duen Horng Chau. Answering Why and Why Not Questions in User Interfaces. In *Proceedings of the SIGCHI conference on Human Factors in computing systems CHI 06*, pages 397–406, 2006.

[88] Christoph Neumann, Ronald A Metoyer, and Margaret Burnett. End-user strategy programming. *Computing*, 20(1):16–29, 2009.

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

[90] D. A. Norman. Some observations on mental models. In Dedre Gentner and Albert Stevens, editors, *Mental Models*, pages 7–14. Erlbaum Associates, Hillsdale, N.J., 1983.

[91] Donald A. Norman. *The design of everyday things*. Basic Books, September 2002.

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

[93] John Pane and Brad A. Myers. More Natural Programming Languages and Environments. In Henry Liberman, Fabio Paterno, and Volker Wulf, editors, *End User Development, vol. 9 of the Human-Computer Interaction Series*, chapter 3, pages 31–50. Springer, Dordrecht, The Netherlands, 2006.

[94] John F. Pane. *A Programming System for Children that is Designed for Usability*. PhD thesis, Carnegie Mellon University, May 2002.

[95] Wim De Pauw and Stephen Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *SIGOPS Oper. Syst. Rev.*, 44(1):97–102, 2010.

[96] P Pirolli and S K Card. Information foraging. *Psychological Review*, 106(4):643–675, 1999.

[97] Matthew D Plumlee and Colin Ware. Zooming Versus Multiple Window Interfaces : Cognitive Costs of Visual Comparisons. *ACM Transactions on Computer-Human Interaction*, 13(2):1–31, 2006.

[98] Andrew R Post, Ana N Sovarel, and James H Harrison. Abstraction-based temporal data retrieval for a Clinical Data Repository. *AMIA ... Annual Symposium proceedings / AMIA Symposium. AMIA Symposium*, pages 603–7, January 2007.

[99] Ben Potter, Jane Sinclair, and David Till. *Introduction to Formal Specification And Z*. Prentice Hall, 2 edition, July 1996.

[100] Orna Raz. Helping Everyday Users Find Anomalies in Data Feeds. *PhD Thesis. Carnegie Mellon University.*, (May), 2004.

[101] S P Reiss. Visualizing program execution using user abstractions. In *in Proc. ACM 2006 Symposium on Software Visualization*, volume pp, pages 125–134, 2006.

[102] S.P. Reiss and Suman Karumuri. Visualizing threads, transactions and tasks. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–16, Toronto, Ontario, Canada, 2010. ACM.

[103] F.E. Ritter and J.H. Larkin. Developing process models as summaries of HCI action sequences. *Human-Computer Interaction*, 9(4):345–383, 1994.

[104] Frank E Ritter and Steven R Haynes. High-level Behavior Representation Languages Revisited. In *Proc. 7th International Conference on Cognitive Modeling (ICCM)*, pages 404–407, 2006.

[105] Michael Sedlmair, Petra Isenberg, Dominikus Baur, Michael Mauerer, Christian Pigorsch, and Andreas Butz. Cardiogram: visual analytics for automotive engineers. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1727–1736, 2011.

[106] J. Segal. Some problems of professional end user developers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 111–118, 2007.

[107] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: beyond human-computer interaction*. John Wiley & Sons, Inc., New York, 2nd. edition, 2007.

[108] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, 1990.

[109] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[110] R. S. Siegler and J. Shrager. Strategy choices in addition and subtraction: How do children know what to do? In C. Sophian, editor, *Origins of Cognitive Skills*, pages 229–293. Erlbaum, Hillsdale, N.J., 1984.

[111] D I K Sjø berg, T Dybå, B C D Anda, and J E Hannay. Building Theories in Software Engineering. In F Shull, J Singer, and D I K Sjø berg, editors, *Guide to Advanced Empirical Software Engineering*, pages 312–336. Spring, 2007.

[112] Richard T Snodgrass, Curtis E Dyreson, Ramez Elmasri, Fabio Grandi, Christian S Jensen, Wolfgang Kafer, Nick Kline, Krishna Kulkarni, T Y Cli Leung, Nikos Lorentzos, John F Roddick, Arie Segev, Michael D Soo, and Suryanarayana M Sripada. TSQL2 Language Specification. *ACM SIGMOD Record*, 23(1):65–86, 1994.

[113] Robert St Amant, Andrew R. Freed, and Frank E. Ritter. Specifying ACT-R models of user interaction with a GOMS language. *Cognitive Systems Research*, 6(1):71–88, March 2005.

[114] Simone Stumpf, Vidya Rajaram, Lida Li, Weng-Keen Wong, Margaret Burnett, Thomas Dietterich, Erin Sullivan, and Jonathan Herlocker. Interacting meaningfully with machine learning systems: Three experiments. *International Journal of Human-Computer Studies*, 67(8):639–662, August 2009.

[115] Glenn Taylor, Randolph M. Jones, Michael Goldstein, Richard Frederiksen, and Robert E. III Wray. VISTA : A Generic Toolkit for Visualizing Agent Behavior. In *Proceedings of the 11th Conference on Computer Generated Forces and Behavioral Representation*, pages 157–167, 2002.

[116] Paolo Terenziani and RT Snodgrass. Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Transactions on Knowledge and Data Engineering*, 16(4):1–13, 2004.

[117] JJ Thomas and KA Cook. A visual analytics agenda. *IEEE Computer Graphics and Applications*, (January/February):10–13, 2006.

[118] K. Tor, F.E. Ritter, S.R. Haynes, and M.A. Cohen. CaDaDis: A tool for displaying the behavior of cognitive models and agents. In *Proceedings of the 13th Conference on Behavior Representation in Modeling and Simulation*, pages 192–200, 2004.

[119] E Wagner and H Lieberman. Supporting user hypotheses in problem diagnosis on the web and elsewhere. *Proc. IUI*, pages 30–37, 2004.

[120] Q Wang, W. Wang, R. Brown, and K. Driesen. EVolve: an open extensible software visualization framework. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 37–46. ACM, 2003.

[121] Jeffrey Wong and Jason I Hong. Making Mashups with Marmite : Towards End-User Programming for the Web. In *Proceedings of the SIGCHI conference on Human Factors in computing systems CHI 06*, pages 1435–1444, 2007.

[122] Krist Wongsuphasawat, John Alexis Guerra Gómez, Catherine Plaisant, Taowei David Wang, Ben Shneiderman, and Meirav Taieb-Maimon. LifeFlow: visualizing an overview of event sequences. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1747–1756, 2011.

[123] R. Yaremko, H. Harari, R. Harrison, and E. Lynn. *Reference Handbook of Research and Statistical Methods in Psychology for Students and Professionals*. Harper and Row, New York, 1982.

[124] Robert K Yin. *Case Study Research*. Sage Publications, Los Angeles, CA, 4th edition, 2009.

[125] N.J. Zbrodoff. Why is 9+7 harder than 2+3? Strength and interference as explanations of the problem-size effect. *Memory & Cognition*, 23(6):689–700, 1995.

[126] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., 2000.

[127] Shikun Zhou, Hussein Zedan, and Antonio Cau. Run-time Analysis of Time-critical Systems. *Journal of System Architecture*, 51(5):331–345, 2005.

APPENDICES

Appendix A – EAST-Env user documentation

This document was written for users of RML, one of the languages EAST-Env (called "CMAV" at that time) was adapted to support in Chapter 9.

CMAV Tutorial

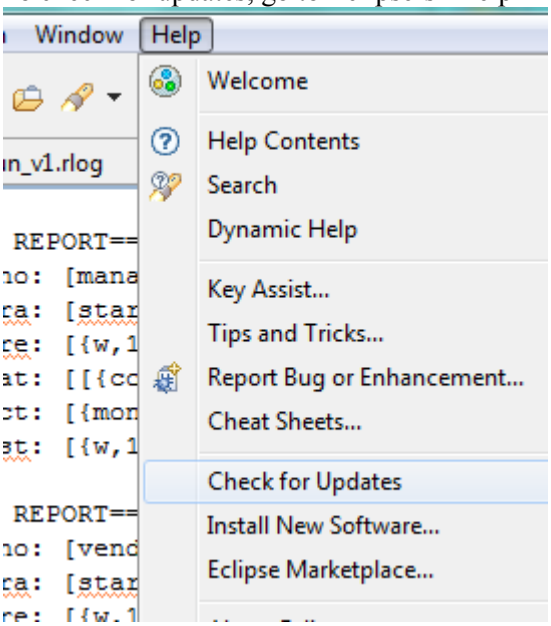
What is CMAV?

CMAV (Cognitive Model Abstraction Viewer) software lets you explore the trace of an RML run. It starts off by importing just about everything in RML's rlog file into a database. It gives you a handful of very verbose listings of events and internal variable changes over time. You can filter, combine, and summarize these listings in various ways to create more useful summaries of what happened during the run, then apply these summaries in the future to later runs of your model.

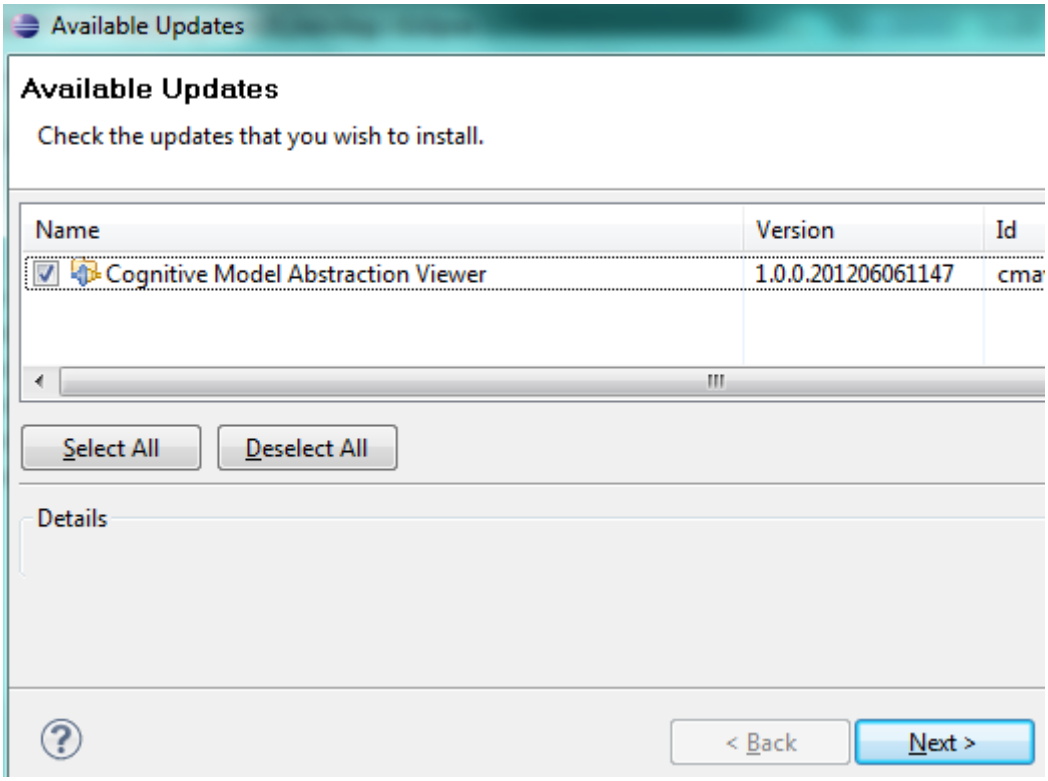
When you work with the software, what you are doing is creating new ways of summarizing the data. These are called "patterns". So when you "filter" or "limit" a table, for example, you are not erasing anything; you're just creating a more selective pattern. The raw data is always kept the same (unless you choose to run the model again and reimport from a new .rlog file); and you can get back to it by using Undo (control-z or File>Undo) or by finding your old pattern in the Data Selector view or the Command Line view

Updating the CMAV Plugin (assuming it's already installed)

The CMAV plugin is being actively maintained, and there will be changes to it based on your feedback. To check for updates, go to Eclipse's "Help" menu and choose "Check for Updates":



Eclipse will check your plugins' distribution sites for updates, and give you a dialog box like this if there are any new versions available:



Make sure the new version of CMAV is checked, "Next>" to continue, and follow the rest of the prompts.

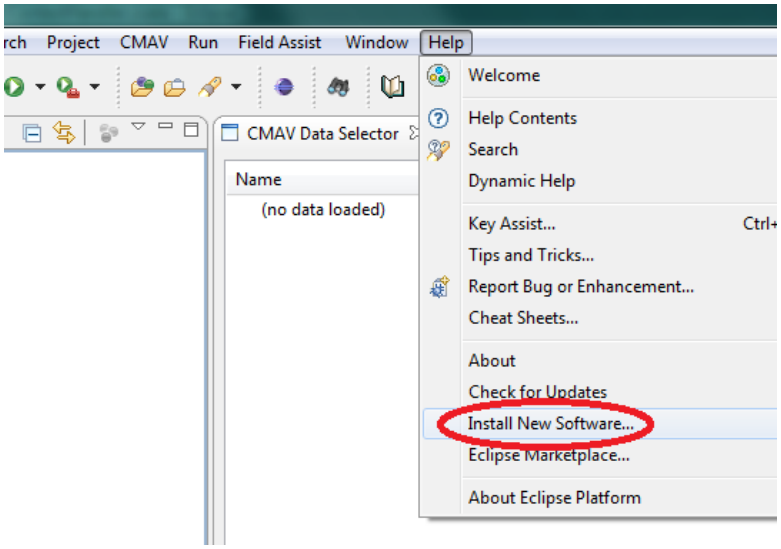
Note that you can also go through the regular install process (described below), and Eclipse will correctly upgrade the plugin; you do not need to uninstall it first to use that process. The regular install lets you install from a local drive, which is convenient if you don't have an internet connection to get to the regular update site.

Installation (assuming it's NOT installed yet)

(This section assumes you have Eclipse installed, and may or may not have the CMAV plugin installed.)

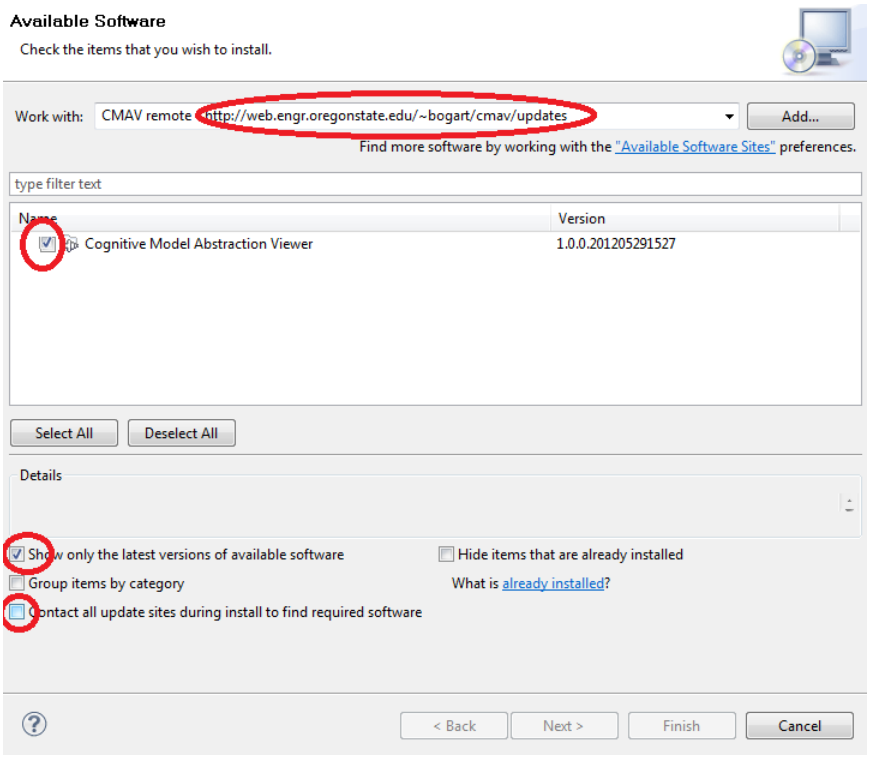
If you have an internet connection…

You can install or upgrade CMAV by going to the Help menu and choosing "Install New Software…"



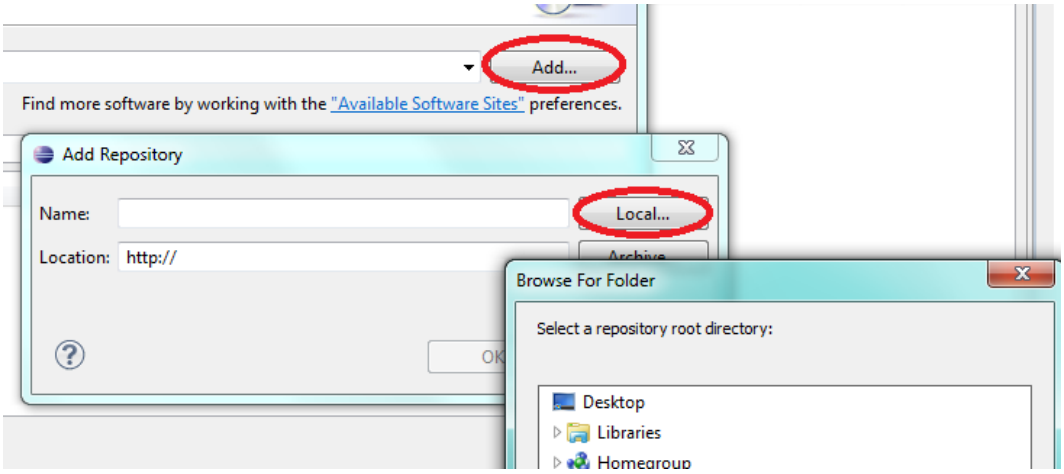
In the big "Available software" dialog box that comes up, put in **http://web.engr.oregonstate.edu/~bogart/cmav/updates** in the "Work with" box. (top red circle in the diagram below). It may take a moment to check the site and list some options. Set the two options at the bottom of the dialog to

"Show only the latest versions of available software", and not to "contact all update sites". After you do this, the main white pane in the middle should show you just the latest version of CMAV. Check it, and hit the Finish button at the bottom.
The process has several more steps, but they are self-explanatory.



If you do not have an internet connection…

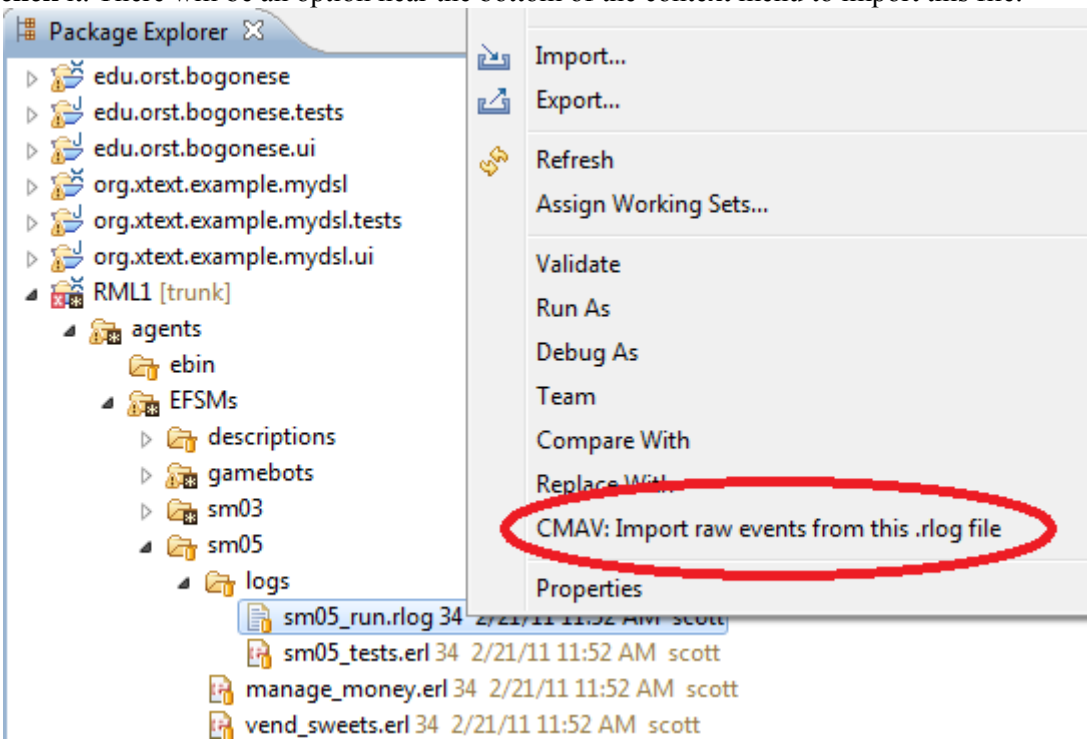
then you can do the installation locally, if you have a location you can reach from your computer (i.e. on a shared network drive or external hard drive). Click the Add button, then choose Local, and navigate to the right directory. After you pick the local installation directory, the steps are the same as for an internet install.



Opening an RML rlog file

Make sure logging is on when you run RML. RML provides some options for controlling the amount of information in the log; that's not documented here. If your model is complex or you are running the model for a long time, it might be helpful to limit the amount of stuff you output to the log.

After the .rlog file is completed, fire up Eclipse, find the .rlog file in the Package Explorer, and **right-click** it. There will be an option near the bottom of the context menu to import this file.



That menu choice gives you a dialog with several options:

- You can import just a portion of the file. If the .rlog file is very large, it may be too much for CMAV, in which case it will let you pick a part of the log to import.
- You can specify whether to use the model's notion of time, or to ignore the time in the .rlog file and just assign an arbitrary time scale. You will need the latter if the RML model does not have monotonically increasing time; for example if time resets to zero between trials, or if the model does not track time at all (for example with the Vending machine sample)

If you already had an RML log open in CMAV when you clicked on .rlog, you'll have the additional option to import the log, but keep your current patterns/visualizations and display the new data in that format. If you do that, it will overwrite the raw trace data in your currently open file. But this can be a useful option when you fix a bug in your RML model and rerun it and want to replace the data from the old run. You can always manually back up .rlog and .db files if you are concerned about going back to old data

Rerunning RML and seeing the new results

After you rerun your RML model, you can open the .rlog file again the same way, and it will prompt you to read the new data, but keep the old patterns (the displays and tables you've configured). If you don't want to lose the data from the first run, just make a backup copy of the .rlog.db file; you can always open .db files directly.

Using CMAV with ACT-R

There is a file included with the distribution called statelog.lisp. To use CMAV with ACT-R:

- Drop **statelog.lisp** into the actr6/tools directory
- Start lisp and load your model
- Optionally, set any CMAV parameters (see table below)
- Type (cmav-start-log) or (cmav-start-log "filename.alog")
- Run your model. It will run more slowly than usual
- Find the "log.alog" file that is created, and put it under your Eclipse workspace folder
- Hit F5/Refresh in Eclipse
- Right click log.alog and choose CMAV: Import from the menu

The basic set of patterns you start with will be different, and there is an ACT-R specific source code (i.e. production rule) view that you can get to by clicking on User Defined and then @RuleViewer.

You can control what information (cmav-start-log) collects by setting some variables to either true **'t** or false **'()**. New rules and rule parameters are turned off by default because they slow down both the model and CMAV, especially for models with hundreds of productions.

`(setf *cmav-collect-audicon* 't)`	Collect audicon information ('t by default)
`(setf *cmav-collect-visicon* 't)`	Collect visicon contents ('t by default)
`(setf *cmav-collect-newrules* 't)`	Collect new production rules that are created by production compilation ('() by default)
`(setf *cmav-collect-rule-params* 't)`	Collect rule utility ("U") and "whynot" information for each rule at each time step
`(setf *cmav-collect-dm* 't)`	Collect activation values of ALL chunks in declarative memory. This option can completely overwhelm CMAV if a model has a lot of chunks in DM.
`(setf *cmav-collect-params* '(:fpc :xyz))`	Collects other parameters using (sgp).

Customizing ACT-R data collection

There are two ways you can customize collection of ACT-R traces:

1. User events: From your Lisp device functions that interact with the model, you can explicitly send your own events into the CMAV log, by calling (cmav-register-event). This function takes a list of key-value pairs. CMAV will timestamp these and insert them into a "user-events" table with column names determined from the keys you use the first time this function is called. Example: (cmav-register-event '((height 110) (width 40) (depth ,*current-depth*) (name ,*name*)))
2. Collecting other parameters from ACT-R: If you're interested in parameters or values in ACT-R that aren't currently captured by statelog.lisp, let me know; I can easily add things.

Recording your activity

CMAV gives you the ability to keep a log of the queries you make. There is a red circle  on the lower border of the Eclipse window; if it is not recording, it will be crossed out.  You can turn recording on and off by clicking on that circle, or through the CMAV part of the main menu, or on the Feedback pane.

The Feedback pane is just a blank where you can type comments that go in that log. The intent is that you can make comments about the software and send the log to me, and I'll be able to use the log to get some context about what you were doing at the time. But after the experiment, or if you don't intend to send the log, you can also just use it to add annotations to the log file.

From the CMAV menu you can also change the directory where the recordings are stored, or examine the logs themselves. Note that these logs are stored on your hard drive as html files, one for each calendar date.

If you choose to participate in a formal user study of CMAV later in the summer, I will be asking you to save your logs, examine each one to make sure it does not contain any confidential or classified information, then email it to me.

Concepts you need to know

Patterns, Instances, and Attributes

CMAV collects data from a program run into tables. "Patterns" are queries that extract data from tables to create other tables. Tables and patterns are given names starting with @: for example @BLACKBOARD is a table of all messages sent between RML automata, and @BBD_respond_1 is a query that shows just "stop" messages with a single argument. Patterns have names, and their data is cached just like a table. Patterns are defined by query strings like "@BLACKBOARD filter (.tupletype = "respond")" that describe how the pattern is derived from other tables or patterns.

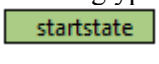
The rows of the tables/patterns are called "instances" and the columns are "attributes" (since "row" and "column" don't make sense anymore when looking at non-tabular visualizations of data). Within a pattern's query syntax, attribute names always begin with a period (e.g. **.tupletype** or **.next.timems**)

States vs. Events vs. other data

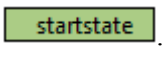
Each pattern can have one of three kinds of instances, depending on how they are timestamped. "Events" have an attribute called **timems** (time in milliseconds), representing when the event occurred. "States" have both **timems** and **endtime** attributes, and represent some status that extends over time. Some tables have instances that are neither events nor states, and have no **timems** or **endtime** attributes.

Some tables may have several time attributes, with names like **.next.timems** or **.previous2.endtime**; but what makes them states or events is just the presence or absence of plain **.timems** and **.endtime** attributes.

Timelines, Tables, and Syncing

Tables and timelines depict the same data in different form, and it's important to understand how they relate. A thin vertical bar with a diamond glyph on it  shown in the timeline is the same as a row in a table with a **.timems**, but no .**endtime**, attribute. The glyph's horizontal position represents the .timems value. Horizontal colored bars in the timeline 

represent rows in a table with both .**timems** and .**endtime** attributes.

A table, when dragged to the timeline, shows up as a horizontal stripe of information with either events or states 

. Multiple tables or patterns can be shown on the timeline simultaneously.

If you highlight a specific time on a timeline and hit SYNC, or a row in a table, and hit SYNC, then all displayed data in any open Eclipse pane will jump to show that particular time, or as close to it as possible.

Making new patterns

In order to define a new pattern, open an existing one (by double-clicking or right-clicking in the Data Selector pane), and define your new pattern in terms of the old one. This can be done in three ways:

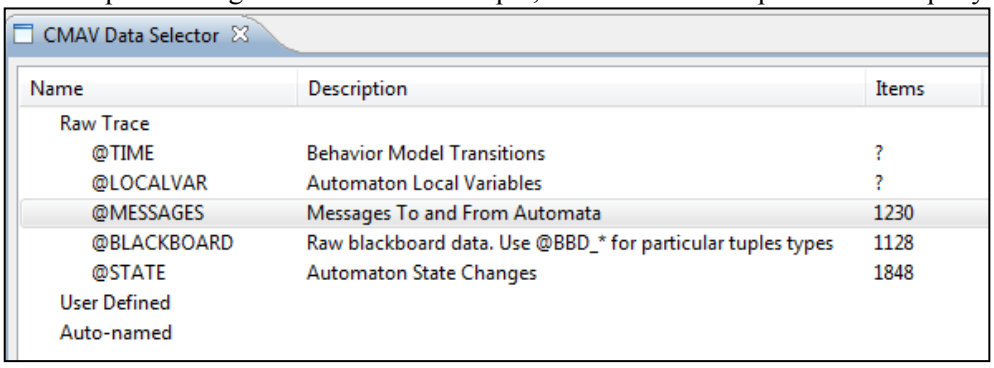
- Right click inside a table or on an instance in a timeline, and choose some operation from the menu. Different options are offered relevant to the place you click, so if you choose a cell that is relevant to your query, the options will be more specific.
- Drag from one instance to another, or from the Data Selector to some instance, to get a list of options for merging two patterns: for example patterns that show when instance of one pattern happen before, after, or during another pattern.
- Use the Command Line pane to type out a query. A little about the syntax is described in the Command Line section below. You can learn the query syntax either from the syntax part of this document, or by performing queries by clicking and dragging, then going to the command line to see what the syntax was. You can also see query syntax by selecting items in the Data Selector: the syntax is shown at the bottom of the pane.

Examples that illustrate some common tasks

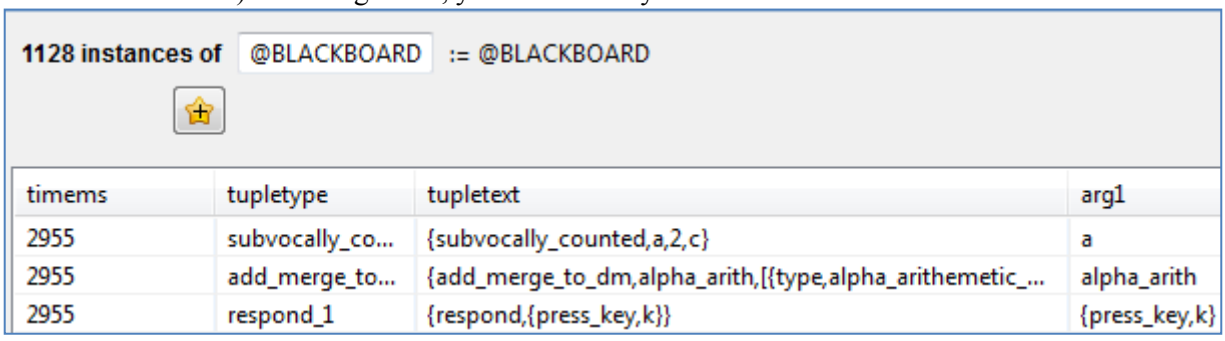
These examples all use the sample RML model zbrodoff_run_v2.rlog

Example 1: Listing a particular kind of event of interest

Suppose you wanted to find all the times the model pressed the "k" key.
If you had no idea where to start looking, you could just poke around in the Raw Trace section of the Data Selector pane. The goal is to find an example, then use the example to build a query.



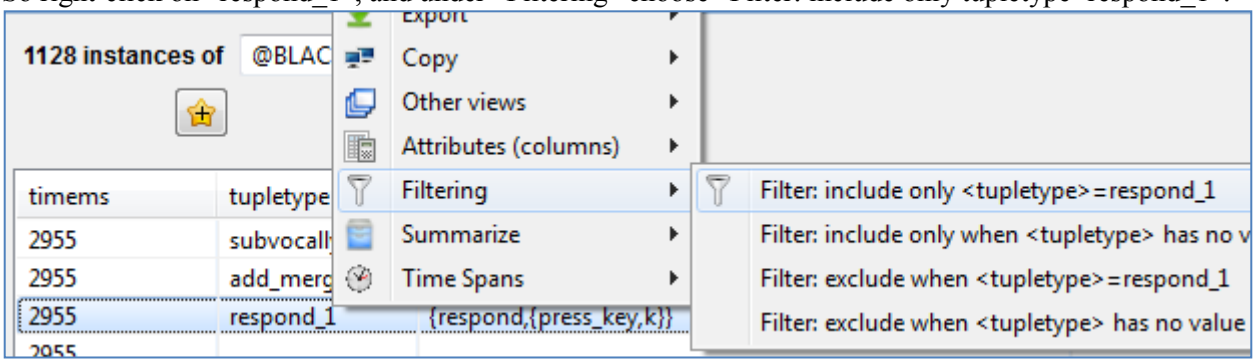
Each of these represents some aspect of the model's run. Open @BLACKBOARD, for example, to see messages placed on the blackboard. (Note: these are currently just blackboard items that eventually are consumed by automata: for efficiency reasons CMAV currently does not capture blackboard items that are never consumed). Scrolling down, you'll eventually see this:



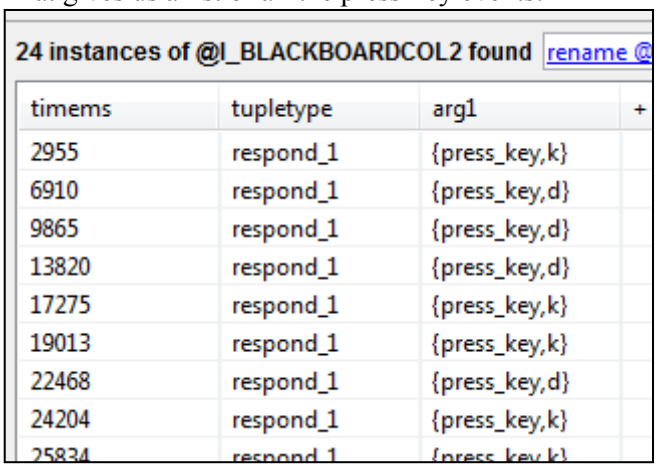
The third line shows a "press key" action, which seems to be the argument to a "respond" message. Given this example, we can now build a query to isolate all these "k" presses and identify them to CMAV.

First, let's reduce the amount of extraneous stuff in the table by filtering down to just the "respond_1" events. Filtering in works by finding an example value in a column, and telling CMAV that you want just the rows with exactly that value.

So right-click on "respond_1", and under "Filtering" choose "Filter: include only tupletype=respond_1":

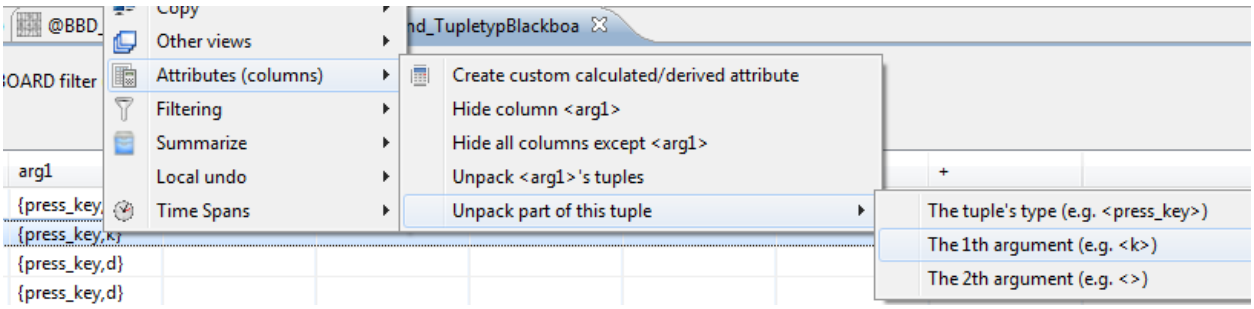


That gives us a list of all the press-key events:

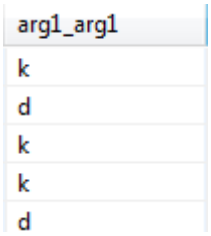


Next, we'd like to filter down to just the "k" events. We could do it similarly by filtering for all cases where arg1="{press_key,k}", but what if there are anomalous cases where it says something like "{press_key,k,24}"? We'd still want lines like that even though the whole string doesn't exactly match. So we'd like to be able to "unpack" these kinds of columns, and talk about a particular part of the expression, namely the "k" itself.

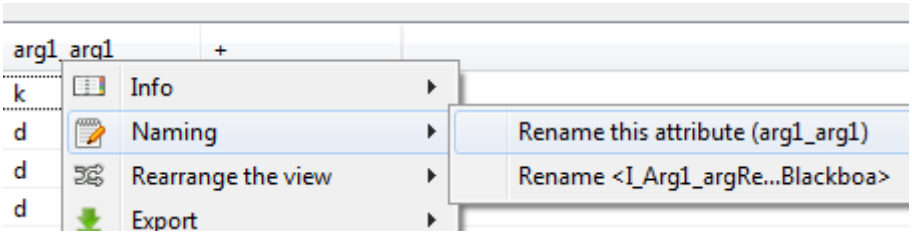
Right-click any cell that contains "{press-key,k}", and choose "Attributes", then "Unpack part of this tuple" then "the 1th argument (e.g. <k>)"



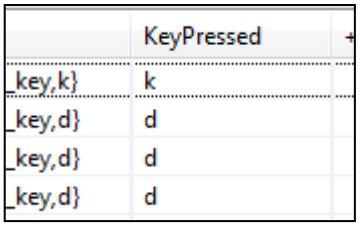
Now the first argument of all these tuples appears as a new column.



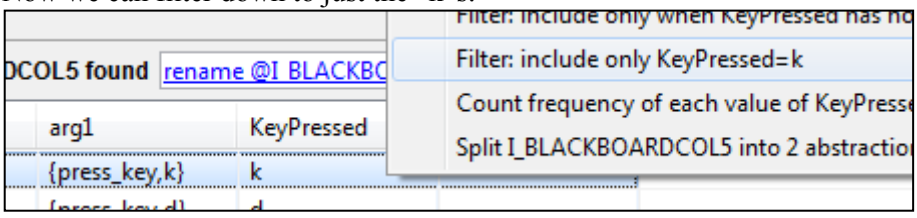
CMAV has generated this name because it is was the first argument of a field called “arg1” (which had already been similarly unpacked, by default). You can rename it to something that conveys the meaning of the data by right-clicking anywhere in this column, and choosing “Rename this attribute”.



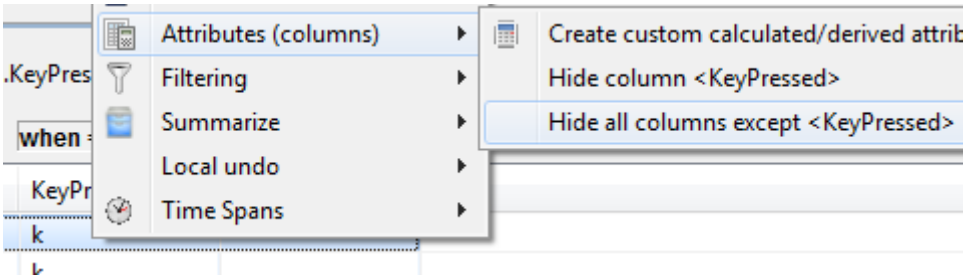
A dialog box lets you enter a new name, for example “KeyPressed”, and it will look something like this afterwards:



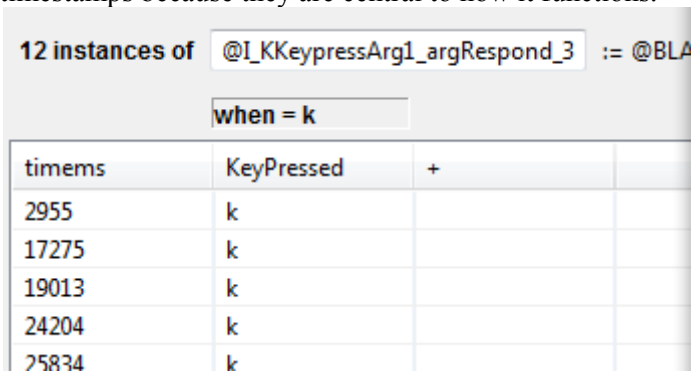
Now we can filter down to just the “k”s:



At this point we have several columns of data, and not all of them may be interesting. So as a further optional cleanup step we could eliminate all the surplus columns, by right clicking KeyPressed and choosing the option under “Attributes” that says “Hide all columns except KeyPressed”

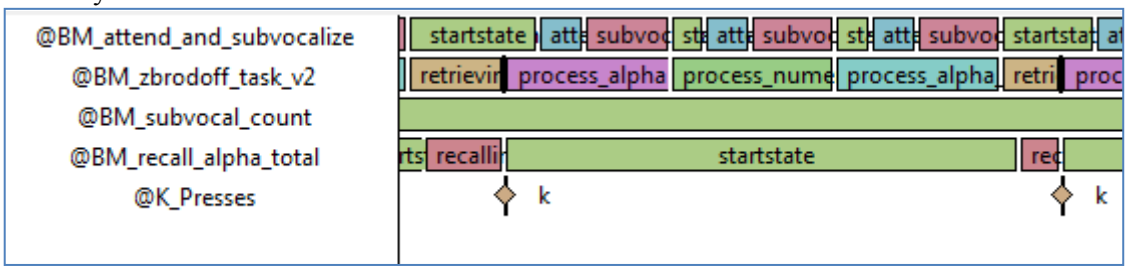


Notice that the resulting table still has the time in milliseconds. CMAV discourages eliminating timestamps because they are central to how it functions.



Another nice cleanup step is to give this abstraction a more meaningful name than the name CMAV chose: "@I_KKeypressArg1_argRespond_3".
Click on the blank with "@I_KKeypressArg1_argRespond_3", and choose a new name (for example "K_Presses").

You might also want to view this new abstraction as a timeline: Right-click anywhere in the table, find "Other Views", and choose "show K_Presses in timeline view", and CMAV will add it to whatever you currently have in the timeline.



(If the timeline is blank, or looks different from this, you may need to scroll back and forth and/or zoom to see things at a nice position and scale. You can zoom with the mouse's scroll wheel, or use the extra scrollbar way off to the right side of the timeline.)

Example 2: Getting latency data between two different events

In this example suppose we want to find out how long it takes between when a behavior model sends a message and receives a reply back. The model is "attend_and_subvocalize", and the messages we're interested in are 'focus_attention" and "encoding_complete".
The strategy for doing this is:

- Find both kinds of messages in the @MESSAGES table
- Filter down to just "focus_attention", and show it in the timeline
- Undo that filter, then filter to just "encoding_complete", and show it in the timeline

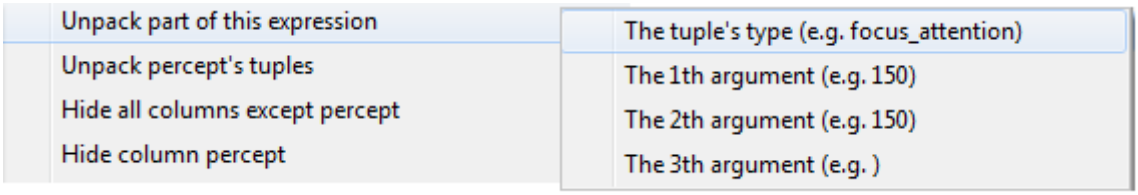
- Connect the two in the timeline using drag and drop
- Add a calculated field showing the time difference

This takes a little work, but the resulting tables and timelines are all available for future use, even if you change the model and rerun it.

First, open the @MESSAGES table, and find an example of the first message type ("focus_attention"):

timems	bmodel	percept	direction	+
0	zbrodoff_task_...	{start_trial}	match	
0	zbrodoff_task_...	{get_visual_location,alpha_...	assert	
50	zbrodoff_task_...	{visual_location,alpha_base,...	match	
50	zbrodoff_task_...	{assert_intention,{attend_an...	assert	
100	attend_and_su...	{attend_and_subvocalize,al...	match	
100	attend_and_su...	{focus_attention,100,150}	assert	
235	attend_and_su...	{encoding_complete,100,15...	match	
235	attend_and_su...	{subvocalize,a}	assert	
485	attend_and_su...	{subvocalization_complete,a}	match	

Next we want to filter down to just the "focus_attention" items. But filtering only works on whole fields, and the "focus_attention" text is just a part of the field shown here, so we need to "unpack" that field, extracting the relevant value. Right click on "{focus_attention,100,150}", and under "Attributes" pick "Unpack part of this tuple", then find the term "focus_attention" in the submenu:

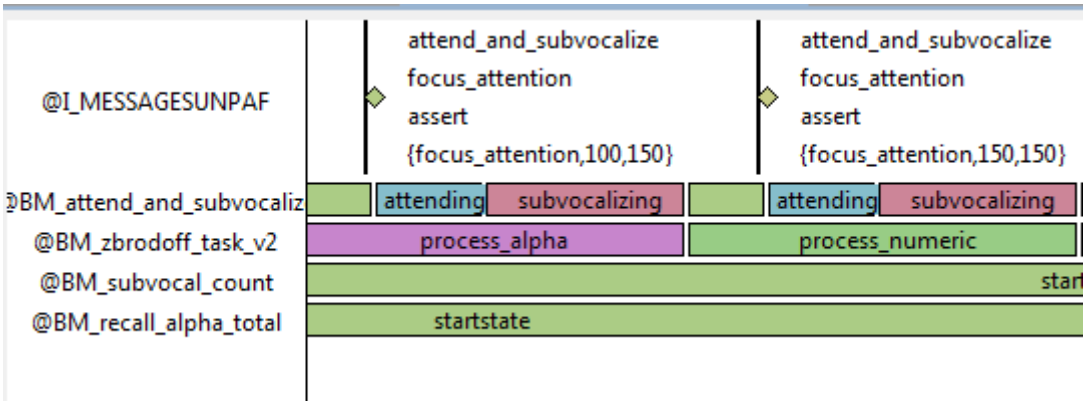


A new column gets added to the far right side of the table, with just the first element (the type) of each tuple in the "percept" column:

timems	bmodel	percept	direction	percept_type	+
0	zbrodoff_task_...	{start_trial}	match	start_trial	
0	zbrodoff_task_...	{get_visual_loc...	assert	get_visual_loc...	
50	zbrodoff_task_...	{visual_locatio...	match	visual_location	
50	zbrodoff_task_...	{assert_intenti...	assert	assert_intention	
100	attend_and_su...	{attend_and_s...	match	attend_and_su...	
100	attend_and_su...	{focus_attenti...	assert	focus_attention	
235	attend_and_su...	{encoding_co...	match	encoding_co...	

Now filter on focus_attention: (right-click the word "focus_attention" in the table and choose "Filter" then "Filter: include only percept_type=focus_attention")

Finally, put this pattern in the Timeline: right-click and choose "Show <table name> in Timeline" (or, if the table and the timeline are both visible on the screen, you can drag from any cell in the table, drop into the timeline, and choose "add to timeline" from the menu that comes up). It should look like this:



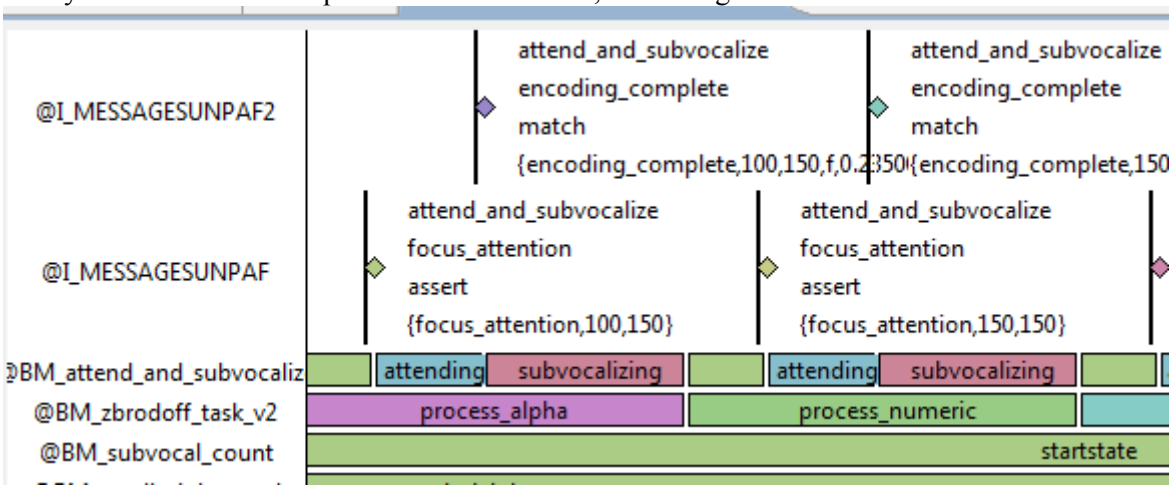
(If the timeline is blank, or looks different from this, you may need to scroll back and forth and/or zoom to see things at a nice position and scale.)

Now go back to the table view: it should still be available as a tab.

Next we'll undo the filter on focus_attention, and instead filter on attend_and_subvocalize.

Right-click on the column that says "focus_attention", and choose "Remove filter (.percept_type=focus_attention)". Then find an example of encoding_complete, right-click, and choose "Filter: Include only percept_type=encoding_complete". Right-click or drag to place this pattern in the timeline.

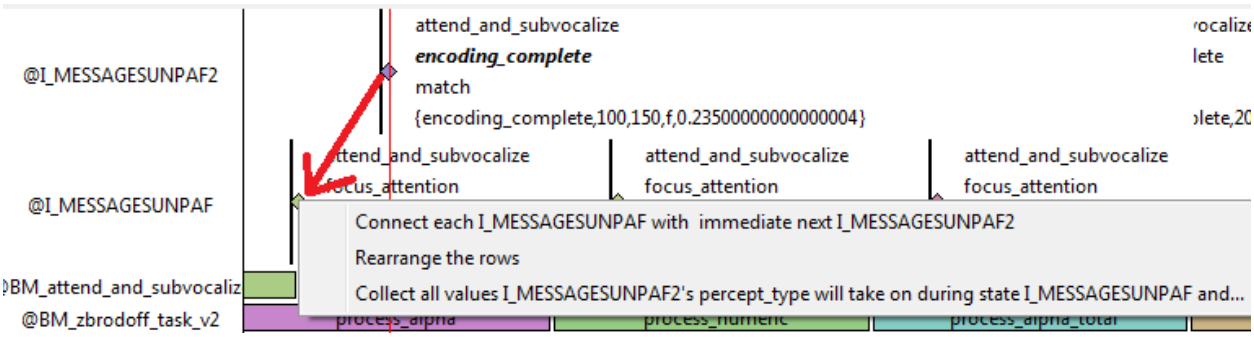
Now you should have both patterns in the timeline, something like this:



Now you can see how the two items relate to each other and to whatever else is in the timeline. You can use the mouse wheel, or the slider on the far right of the screen, to zoom in and out.

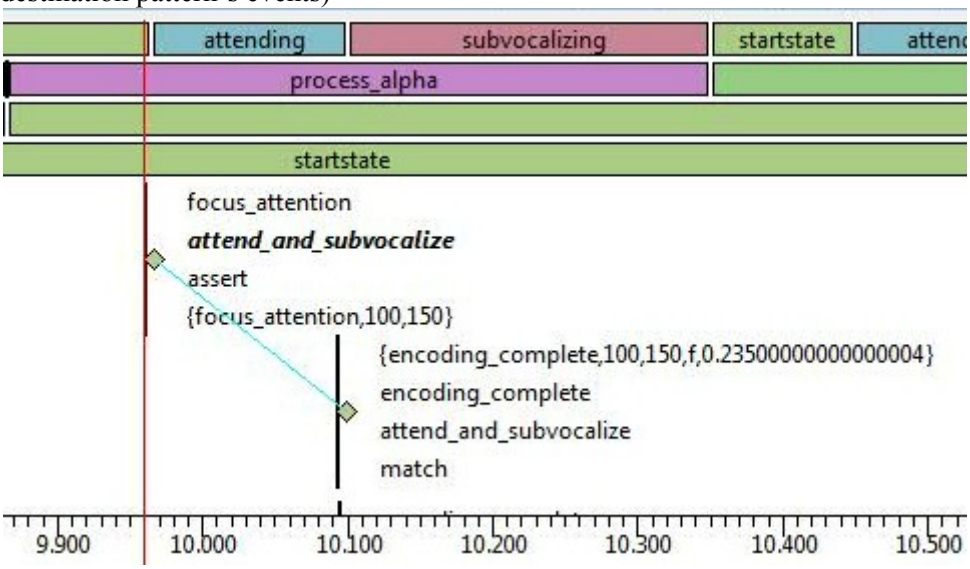
To make the relationship between these events explicit, drag an instance of the "encoding complete" pattern to the preceding instance of "focus_attention". (When dragging things in CMAV, think of it as adding new information to the destination of the drag. So when you drag from thing A to thing B, you're saying "Add new information to each instance of B about nearby or related A's".)

A dragging action usually has more than one possible interpretation, so CMAV will offer several options as a menu you have to choose from before the drag will complete:

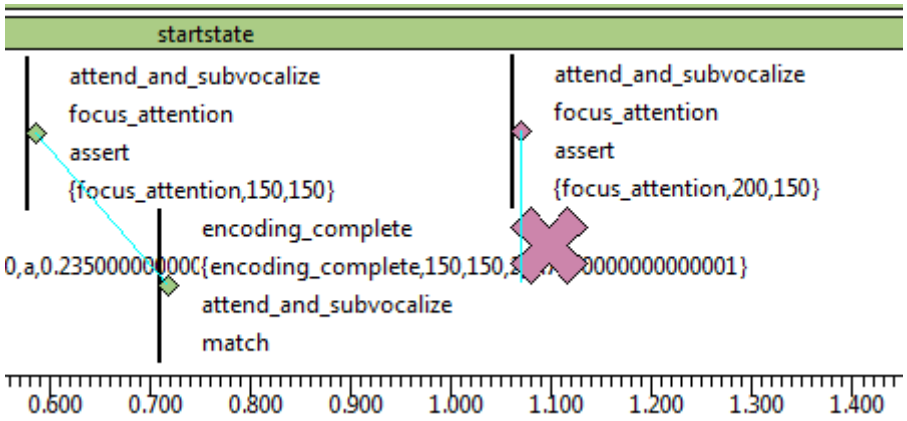


Use the option "Connect each <table> with immediate next <table>". This means that CMAV will create a new pattern that consists of **all** the instances of focus_attention, augmented by **only** the instances of encoding_complete that come **immediately afterwards** in time.

(As for the other options: "Rearrange the rows" just shuffles the rows of the timeline. "Collect all values…" is useful when you're dragging a *more frequent* pattern to a *less frequent* pattern: it creates a new attribute in the destination pattern that summarizes all the dragged events that happened *between* the destination pattern's events)



In this example, the two events are in a one-to-one relationship. If there had been any "extra" encoding_completes, they would be omitted. If there were any "extra" focus_attentions, they would be shown with a line connecting to an X-mark showing that no encoding complete happened:

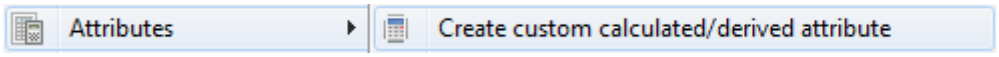


For the converse behavior, showing all the encoding_completes but only the previous focus_attention events, you'd just drag in the other direction.

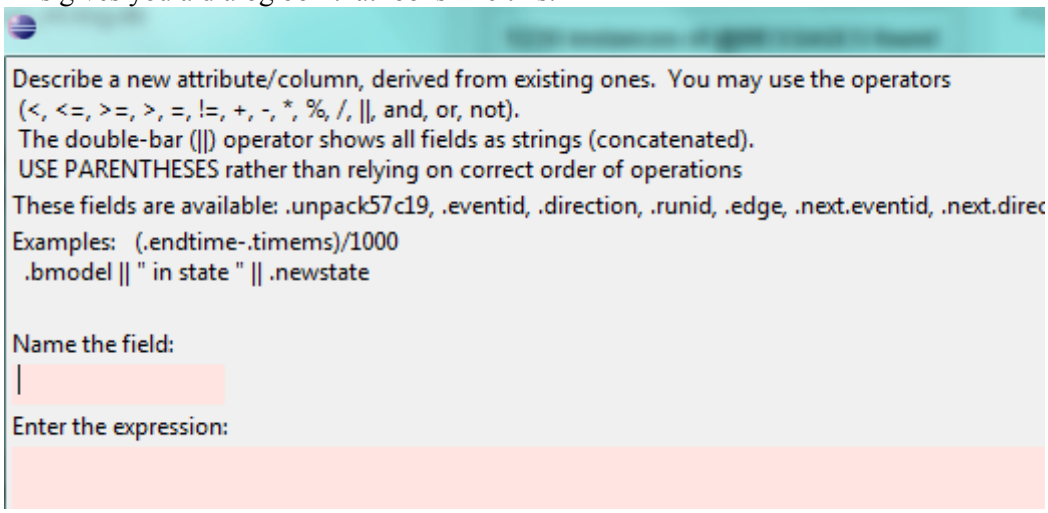
Now let's look at the new pattern we've created, and calculate the time span between these events.

To show it as a table, (right-click somewhere on it and choose "Show <name> as table"), you'll see that it's got columns for attributes in both parts of the pattern, with the new ones being prefixed with "next." if they duplicate existing attributes.

The "encoding_complete" message seems to contain a floating point number that looks like it might represent latency, (i.e. the fourth argument: {encoding_complete,100,150,f,0.23500000000000004}) but we can recalculate the latency in CMAV and see if it matches. Click on any row in this table and choose "Attributes" then "Create custom calculated/derived attribute":

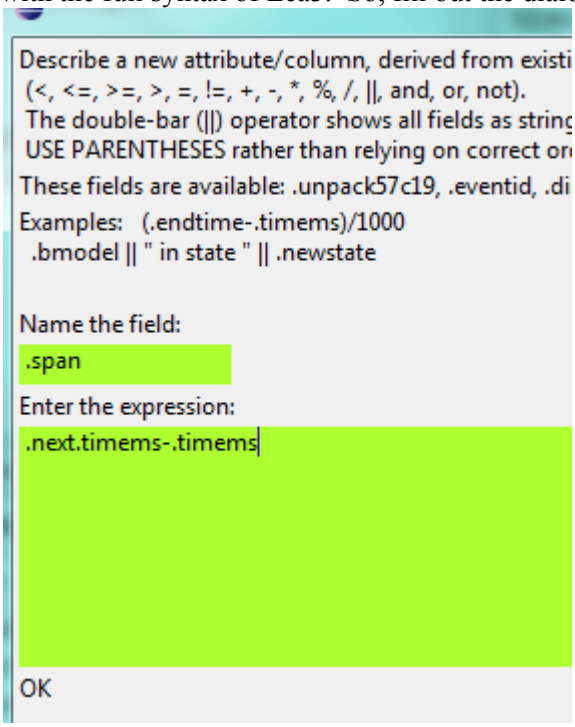


This gives you a dialog box that looks like this:



The dialog lets you enter a mathematical expression describing a new attribute (table column). The legal operators and fields you have available are listed at the top. To calculate the span of time between the focus event and the encoding complete event, the fields you want to subtract are **.timems** from **.next.timems** For this example we'll call the new attribute "span", but you could name it anything as

long as the name isn't already in use in this pattern. Note that attributes begin with a dot, to be consistent with the full syntax of Lea3. So, fill out the dialog like this:



The boxes will turn from pink to green when their contents are legal: i.e. the expression is parseable, the attributes used in the expression are all present in this pattern, and the name you have chosen is legal and not already in use. When the boxes are green, then you can press the OK button.

The new column appears at the far right. The numbers do not appear to match the fourth argument of "encoding_complete"! This was a good check then; it showed that we may have misunderstood the semantics of that fourth argument of the encoding_complete tuple.

next.percept	next.direction	next.percept_t...	span
{encoding_complete,100,150,a,0.235...	match	encoding_co...	135
{encoding_complete,150,150,2,0.720...	match	encoding_co...	135
{encoding_complete,200,150,c,1.205...	match	encoding_co...	135
{encoding_complete,100,150,c,0.235...	match	encoding_co...	135
{encoding_complete,150,150,4,0.720...	match	encoding_co...	135
{encoding_complete,200,150,h,1.205...	match	encoding_co...	135

Summarizing

Let's explore that discrepancy further, by separating out that fourth argument of encoding_complete and looking for an overall pattern to the numbers.

First, let's separate out that number into a new column: Click on the "next.percept column" and choose "Attributes", then "Unpack part of this expression" then "4th argument (0.23500000)":

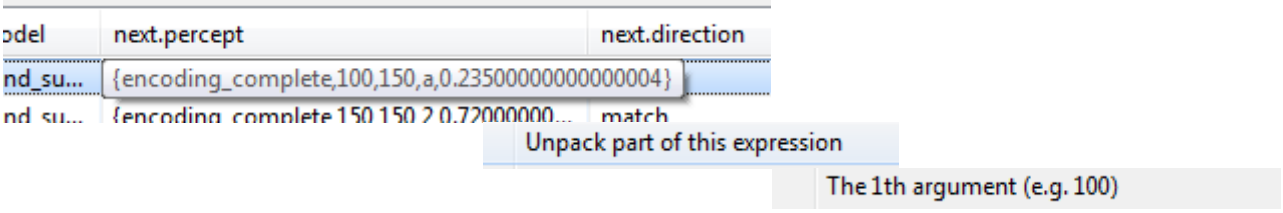
next.percept	next.direction	next.percept_t...	span	next_percept_arg4
{encoding_co...	match	encoding_co...	135	0.23500000000000004
{encoding_co...	match	encoding_co...	135	0.7200000000000001
{encoding_co...	match	encoding_co...	135	1.2050000000000003
{encoding_co...	match	encoding_co...	135	0.23500000000000004
{encoding_co...	match	encoding_co...	135	0.7200000000000001

The three values look like they repeat, but they're a little hard to scan visually. So, right click the column and choose "Summarize" then "Summarize and count next_percept_arg4":

next_percept_arg4	numInstances	+
0.23500000000000004	24	
0.7200000000000001	24	
1.2050000000000003	24	

This is a tally of how often each value in a column appears. The left column is just a list of unique values from the next_percept_arg4 column from before; and the "numInstances" column is how many times each one appeared there. This confirms what we thought we saw: there are only three distinct values, and each one appears 24 times, suggesting that they're probably interleaved all the way down the list.

Let's also investigate one of the other arguments of encoding_complete that seems relevant: its first argument. Go back to the full list (use control-Z to undo, or if you've done other stuff in between, right click and choose "Ungroup").
Right-click the "{encoding_complete,100,…" column and unpack the 1st argument:



That gives us a new column, which appears to be correlated pretty strongly with next_percept_arg4:

next.percept_type	span	next_percept_arg4	next_percept_arg1
encoding_complete	135	0.235000000000000...	100
encoding_complete	135	0.7200000000000001	150
encoding_complete	135	1.2050000000000003	200
encoding_complete	135	0.235000000000000...	100

We can use the summary table to verify this (and to quickly check this again in the future, if we change something in the Zbrodoff model and rerun it)
As before, right click on next_percept_arg4, and create the summary again ("Summarize" then "Summarize and Count…")

next_percept_arg4	numInstances	+
0.23500000000000004	24	
0.7200000000000001	24	
1.2050000000000003	24	

Now right click this summary table, go to the "Summarize" submenu, and look at the options available.

Pick "Disaggregate further by", and in the submenu choose "next_percept_arg1". "Disaggregate" here refers to breaking up those 24 instances into separate lines, depending on their having different values of "next_percept_arg4", so you can see how the different values are distributed. However, this is the result of the operation:

next_percept_a...	next_percept_a...	numInstances	+
0.235000000000...	100	24	
0.720000000000...	150	24	
1.205000000000...	200	24	

This proves the two values are exactly correlated all the way down the file: if they hadn't been, for example if sometimes next_percept_arg1 was 150 when next_percept_arg4 was .235000, then there would have been an extra row in this table showing that combination.

The "Summarize" options in let you go beyond just counting those 24 instances in each of the three cases. For example if you picked "Summarize next.timems" and chose "minimum", it would show you the lowest value of next.timems within each of the three categories. "Concat" and "Unique" will show all the values, or all the distinct values, that that attribute ever has within those 24 instances. "unique" ("distinct_direction") and "concat" ("all_direction") are shown below, for a field that happens to have the same value in every row.

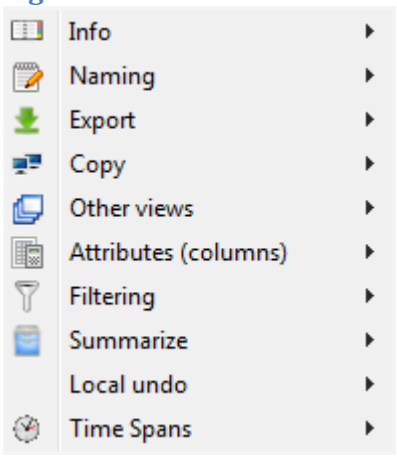
next_percept_arg4	next_percept_arg1	numInstances	distinct_direction	all_direction
0.23500000000000...	100	24	assert	assert,assert,assert,assert,assert,assert,assert,a...
0.72000000000000...	150	24	assert	assert,assert,assert,assert,assert,assert,assert,a...
1.20500000000000...	200	24	assert	assert,assert,assert,assert,assert,assert,assert,a...

From here you can also drill down, from the summary back to a subset of the data it represents, by clicking on a row and choosing "See these 24 instances".

Summary of Menu Operations that Modify Patterns

There are two ways to get this menu: one is by right clicking, another is by dragging items from one table, timeline, or detail item into another pane.

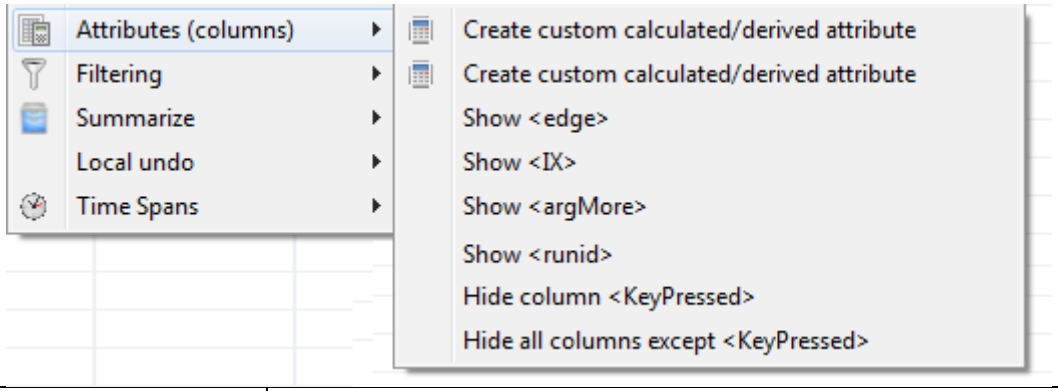
Right click menu:



Info	A non-functional text menu that explains how the current window was calculated
Naming	Rename columns or patterns
Export	Export CSV of data, or PATT of the pattern syntax
Copy	Copy/paste options
Other Views	Timeline, Detail, Scatter plot, Table
Attributes	Various column-oriented things (see below)
Filtering	Filter rows in or out of the view
Summarize	Average over time
Local Undo	Undo the most recent operation on this window (use ctrl-Z for general undo)
Time Spans	Connect up events to describe spans of time, like trials or program/variable states

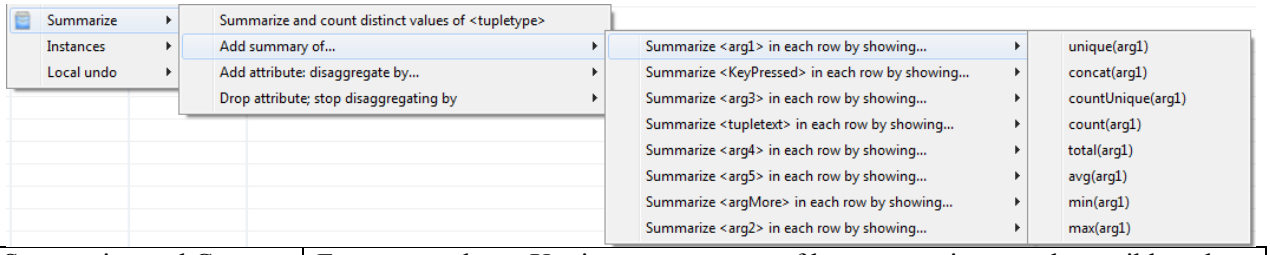
Attributes

Options relating to adding or removing columns



Create custom attribute	Calculate one column based on other columns in the same row: this brings up a dialog box where you can type a formula
Show <X>	Show a column that you've hidden
Hide column <X>	Hide a column
Hide except <X>	Hide all columns except one (does not hide the time column)

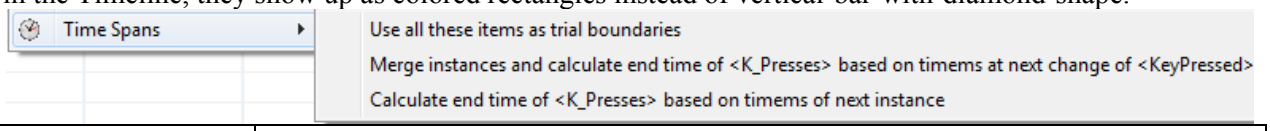
Summarize



Summarize and Count <X>	For some column X, gives you a count of how many times each possible value occurs.
Add Summary of <Y>	After you've created a summary/count, this just adds a new column that summarizes some other field, not necessarily the one you grouped by. E.g. If you had a table of facts about people, and you've summarized by gender and now have just two rows, showing 102 males and 107 females, then "add summary of <age> by showing avg(age)" will add an "avg_age" column, averaging males and females separately.
Add attribute: disaggregate by <Y>	After you've created a summary/count, this adds a new criterion for breaking down your summary further. E.g. In the example above, if you instead disaggregate by <age>, then many new rows will be created, so you have a unique row for every age that occurred in the table, for each gender.
Drop attribute: stop disaggregating by <Y>	Opposite of Add Attribute/disaggregate

Time Spans

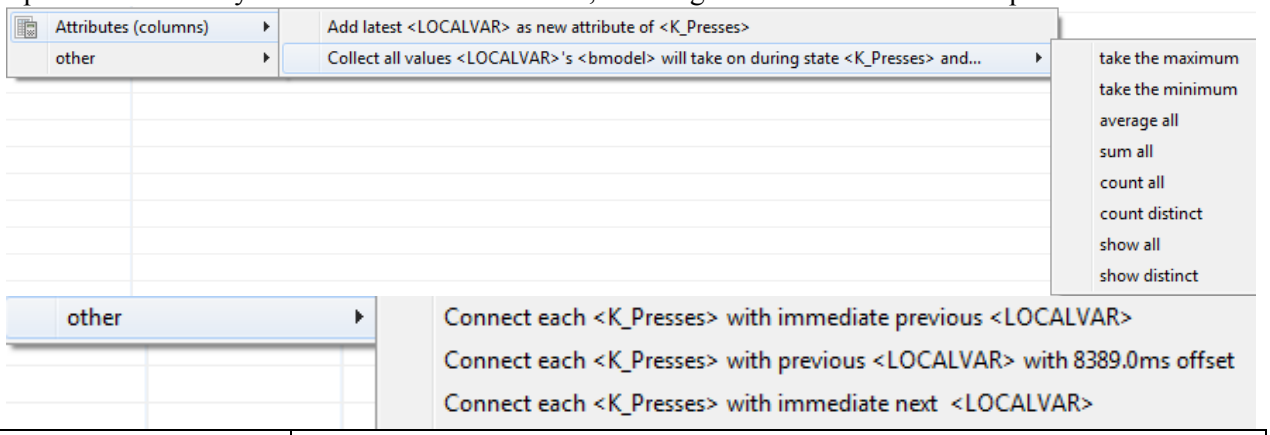
These options turn "events" into "states": In a table view, they add an "endtime" column. When shown in the Timeline, they show up as colored rectangles instead of vertical-bar-with-diamond-shape.



Use as Trial boundaries	Define "trials" by using this list of events as the boundaries between them. This also numbers them.
Merge instances	Treats some value that repeats down a column as a "state": gives you a simpler table with just a column for that value, and a start and end time.
Calc end time	Leaves the rows as they are, but just adds an end time to each row which is the start time of the next row.

Drag-drop menus:

Open two views so you can see them both at once, and drag one to the other. These options are available:

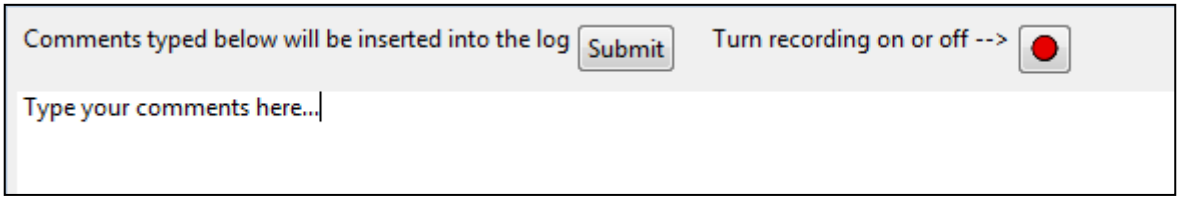


'Add latest <X> as new member of <Y>'	If you drag a **state** to an **event**, this annotates each row in the event table with the value that that state took on. E.g. drag a list of trial numbers (with start and end times) to a list of keypresses, to add a trial number column to your list of keypresses.
'Collect all <X> during <Y>'	If you drag **event** to **state**, then you can summarize all the events that took place during that time. E.g. drag a list of keypresses to a list of trials, and choose 'count', to add a count of how many keypresses; or "show distinct" to add a cell with a comma-separated list of keys pressed.
'Connect each <X> with immediate previous <Y>'	Drag an **event** to an **event**, and this will annotate each **destination** rows with information about the most recent row from the **drag source.** If either table has a **state**, it'll just work with the onset of that state. E.g. Drag a "rule fired" table to a "key pressed" table, and pick "connect previous", and the result will show ALL the keys pressed, and whatever rule fired most recently (ignoring rules that did not fire right before a keypress)
'Connect each <X> with immediate previous <Y>'	Drag an **event** to an **event**, and this will annotate each **destination** rows with information about the subsequent row from the **drag source.** E.g. Drag a "key pressed" table to a "rule fired" table and pick "connect next", and the result will show ALL the rules fired, and whatever key was pressed after that (ignoring keypresses that did not follow a rule firing, if that is even possible)

What the different panes/views do

Feedback View

You can type comments here, and they will go into the log file mentioned above. If you are using the recordings for your own purposes this can be a way of annotating the log for later reference. If you plan on giving me the logs to help with development of the CMAV software, please jot down any comments on things that seem to be good or bad about the software at the moment you notice them; that way I have your comments in context of what you were doing at the time.



Using the Data Selector

The Data Selector view shows you all the data from the run. Under Raw Data it has tables summarizing what was found literally in the .rlog file.

Under User-defined abstractions, it has some convenient tables filtered out by automaton. Any pattern you decide to name will show up here alongside these.

Under Auto-named abstractions will appear any pattern you create incidentally as you experiment with the system. They have auto-generated names, which may or may not be very helpful. If you name one of these, it will move to the User-defined abstractions subtree.

Double click on any of these to see a tabular view of the data. Other views are timeline view, and the detail view.

You can also drag items from the data selector onto some other view, and you will be offered a menu of possible actions. This is one way of combining data from different sources: for example if you have some events listed out in a table, and you drag in a pattern representing the values some variable took on over time, you'll be offered the option to annotate each event with the variable's current value at that time.

The table view

The Table view shows just one pattern at a time. Right clicking or dragging from this table gives you menu options for creating a new pattern based on the old one. The new pattern will show up in the same window, but notice that it has a new name, because it is a distinct query.

At the top of the table view is the name of the table: if you right click it you can change its name. That name will show up in the Data Selector under User-defined patterns. As soon as you modify the table, the name will change again, but the old pattern will still be available in the Data Selector by the name you gave it.

If you highlight a row in this table and hit the SYNC button, CMAV will attempt to move other views to the corresponding point in time. The time may not be perfectly synced, though, since different views handle time differently.

Timeline view

In this view, time runs horizontally, which is always tied to the "timems" and "endtime" attributes of patterns. Any patterns you choose to depict are laid out vertically, one below the next. You can zoom in and out in time. Note that overlapping labels can make the timeline hard to interpret, so when you need to verify what's really happening, examine the same pattern in a table view.

If you highlight an item in this view or a point in time, and hit the SYNC button, CMAV will attempt to move other views to the corresponding point in time. The time may not be perfectly synced, though, since different views handle time differently.

Detail view: (experimental)

This shows all the attributes of any patterns you choose to drag there, at a single point in time. There is a next/previous time control at the top that steps forward *to the next time any of those particular patterns change*. If you pick some patterns that don't change often, this control will jump you through the trace in just a few steps.

In some cases the detail view will show a message that says “Items overlap!” in the right column where the values of attributes should be. This means that there is more than one item that should be shown at that time: either there are two instances with different start and end times that happen to overlap at the time being shown, or two event instances with identical start times that match the time being shown.

Another feature of the detail view: if you click a value in right column, it will display a dropdown list showing how that value changed over the whole model run. If you pick a different value from that list, you can jump to the appropriate point in time.

If you hit the SYNC button, CMAV will attempt to move other views to the corresponding point in time. The time may not be perfectly synced, though, since different views handle time differently.

This view is still a little buggy, so be patient if you try to use it, and let me know about any problems you encounter

Command-Line View

When you modify patterns using the GUI, there is an underlying query language ("Lea3") being used by CMAV, and you can see its syntax in the Command-Line View. It is not necessary to learn Lea3, and you can mostly ignore this window, although if something goes wrong there may be useful error messages here.

If you do prefer to use the command line, you can type those queries directly in this window. Using the GUI and watching what Lea3 syntax appears is a way to learn the language.

The syntax of Lea3 is described in detail below, but to get started, here is a useful subset of the language:

- You can just type the name of a pattern here, and a table view will pop open.
- Patterns start with @, and column/attribute names start with a period.
- Pattern and column names are case sensitive: @ba is different from @bA or @BA
- Filter a pattern like this:

    @automata filter (.bmodel = "Main")

  This will show just rows of @automata for which the word Main is the value in the bmodel column
- You can add a calculated column like this:

    @automata calc (.doubletime := .timems + .timems)

  This silly example will create a new column called doubletime. For a list of operators, right click in the + column of the table view, and chose "custom calculation". The dialog box that takes you to is just a convenient way of using this calc operator.
  These operators can be stacked up on the right: so for example the previous two examples could be combined as a single pattern:

    @automata filter (.bmodel = "Main") calc (.doubletime := .timems + .timems)

- You can name a pattern like this:

    @mainmodel := @automata filter (.bmodel = "Main")

- If you don't name a pattern, the system will choose a name.

To learn more operators, explore the options offered in the right-click menu or when dragging a pattern onto another pattern: the command-line window will show the equivalent Lea3 syntax.

Operators in CMAV's Command Language

Selecting relevant events

Operator	Syntax Example	Function
filter	@A filter (.x = 4) @A filter (.x != 4)	Creates a new abstraction that includes only those rows/instances that meet the criterion. The criterion in parentheses is NOT any arbitrary expression: it must be an attribute (from @A), then a comparison operator (=, <, >, >=, <=, !=), and then a string or integer value. To simulate more elaborate filtering expressions, use the "calc" operator to define a new attribute, then filter on that attribute. For example instead of "@A filter (.x = .y)", use "@A calc (.z := .x = .y) filter (.z = 1)"

Adding and removing attributes

Operator	Syntax Example	Function
get	@A get .x @A get (.x, .y)	Eliminates all but the named attributes from a pattern. If there are more than one attribute, they need to be enclosed in parentheses.
rename	@A rename (.x := .y) @A rename (.x := .y, .z := .w)	Renames attributes
remove	@A remove .x @A remove (.x, .y)	Removes attributes. If there are more than one, they need to be enclosed in parentheses
calc	@A calc (.v := .x * .y)	Adds a new calculated attribute. The menu option "Create custom/calculated attribute" is helpful for this, since it lists out the names of all attributes and operators that are available. Currently supported operators: +, -, *, /, =, <, >, >=, <=, !=, ||, round() (The two vertical bars are string concatenation. Round converts floats to integers)

unpack	@A unpack .f @A unpack (.x := .f(3)) @A unpack (.x := .f(tupletype)) @A unpack (.f sep ",") @A unpack (.x := .f(3) sep ",") @A unpack (.f record "x") @A unpack (.f record ("x", "y"))	The unpack operator creates a new attribute by pulling apart the contents of some attribute .f that has more than one value encoded in it. When no new attribute name is specified (i.e. the .x := is missing from the examples here), CMAV will choose attribute names, and unpack just the first 5 items of each field. With no "sep" or "record" argument, unpack assumes the item is an Erlang tuple, list, or atom format. Erlang tuples use curly brackets and commas: {t,2,3,zz}, where the first argument is the tuple type. Erlang lists use square brackets: [a,3,4], and there is no tuple type. Atoms are just plain values. @A unpack (.x := .f(3)) means to treat .f like an Erlang tuple, and take the 3rd argument, and assign its value to a new attribute .x. The third argument is really the fourth item in the curly-bracketed list, since the first item is the tuple type. When "sep" is specified, this means that instead of Erlang format, the field will just be a list separated by the specified separator. So (.x = .f(3) sep " ") applied to an instance where .f contained "To be or not to be" would create a new instance .x holding the value "or". When "record" is specified, the field will be in Erlang record format: #a{b=c,d=e,f=g} In this format, "a" is a record type, "b", "d", and "e" are the names of record attributes, and "c", "e", and "g" are the corresponding values. Specify record keys exactly as they appear, and CMAV will convert them to legal attribute names when unpacking them.

Combining two patterns

Operator	Syntax Example	Function
addstate	@A addstate @B	Adds the "current value" of @B to @A. Assumes @A is an event, @B is a state, and that instances of @B are NOT overlapping time segments. If @A.timems falls between @B.timems and @B.endtime, then add @B's attributes to @A.
next	@A next @B @A next [3,200] @B	For each instance of @A, find the very next instance of @B (if there is one, before the next @A comes around), and add its attributes to @A. If two numbers in brackets are specified, then @B must occur in that range of milliseconds after @A

previous	@A previous @B @A previous [3, 200] @B	For each instance of @A, find the very most recent instance of @B (if there is one, after the previous @A), and add its attributes to @A. If two numbers in brackets are specified, then @B must occur in that range of milliseconds after @A
simul	@A simul @B	For each instance of @A, find the instance of @B that has the exact same .timems value, and add its attributes to @A. Assumes there is either 1 or 0 such instances of @B.
collect	@A collect (.x := fn(.b in @B)) @A collect (fn(.b in @B))	Assuming @A is a state (i.e. has a .timems and an .endtime), find all instances of @B whose .timems is within that period. Take the attribute .b from those @B instances, and apply the function "fn" to them. "fn" can be: sum, avg, min, max, count, concat, unique. "concat" puts all values of .b together interspersed with commas, in time order. "unique" is the same but omits duplicate values of .b If no name for the new attribute is specified, CMAV will choose a name.
join	@A join (@B where .a = .b)	Non-temporal join: augment each instance of @A by adding values of @B for which @A's .a attribute matches @B's .b attribute. Assumes there are 1 or 0 such matches for each @A; if there are extra @B's, one @B will be chosen arbitrarily. In lines of @A for which there are no matches, the new attributes will be left blank. Join isn't currently available from the menus.

Clustering nearby events

Operator	Syntax Example	Function
segment	@A segment @A segment (upto @B) @A segment (num .t) @A segment (num .t upto @B)	Defines a "trial": a series of .timems and .endtime instances and a trial number, with no other attributes, where the boundaries are defined by @A's events. If "upto @B" is specified, then the trial ends with the next @B event. If "num .t" is specified, then the attribute name .t will be used as the trial number (otherwise, CMAV will choose a name)

state	@A state @A state (.x)	Assumes @A has a .timems attribute, but no .endtime attribute. "@A state" assigns an .endtime attribute based on the next row's .timems attribute: in other words it turns a sequence of events into a sequence of states that butt up against each other. If an attribute is specified, then the behavior is very different: all attributes except .x are removed, an .endtime value is chosen from the next instance of @A in which the value of .x is *different* from the current one, and intervening rows are discarded. In other words, it defines states which represent spans of time where .x maintained the same value.

Statistical summary operations

Operator	Syntax Example	Function
group	@A group () @A group .x @A group (.x,.y) (etc)	@A group () creates a very small output table, with just one row and one column, "numInstances", which says how many instances @A had. @A group .x creates a 2-column table, showing each distinct value of .x found in @A, and the number of times that value appeared. @A group (.x,.y) creates a 3-column table, showing each distinct combination of .x and .y that appeared in @A, and the number of times that combination appeared.
aggregate	@A group (.x,.y) aggregate (.z := sum(.w), .z2 := avg(.w)) @trials group (.stimulusType) aggregate (.latency := avg(.latency), .response_seq := concat(.response))	Aggregate must always come after group. For each set of instances in @A that had a particular .x,.y combination, it adds new attributes summarizing them in other ways. The first example gives the total and the average of @A's ".w" column for each combination of .x and .y The second (hypothetical) example might summarize experimental results, showing the response latency and a comma-separated list of responses for all trials, in chronological order, but separated out by .stimulusType.

Visualizations

Patterns that omit any of these operators will simply be presented as a table. These operators instruct CMAV to present the information in a different form

Operator	Syntax Example	Function
plot	@A plot (.x, .y)	Draw a plot of .x, .y values. The CMAV menus currently only offer the option of plotting (.timems, .x), (as a "time series plot"), but other variables can be plotted on the horizontal axis by using the command line version of plot.
actrRule	@events filter (.RULE_FIRED != "nil") join ((@rulesource) where .RULE_FIRED = .NAME) get (.RULE_FIRED, .SOURCE) actrRule	Shows ONLY the .SOURCE attribute of each instance, along with a control for moving forward and backward in time. This view is specific to the query shown here, and only makes sense currently for ACT-R models. It's provided as @RuleViewer whenever an ACT-R model is imported, so there shouldn't be a need to use this operator
tc	tc { @A, @B, @C }	TC stands for "timeline canvas" – this operator shows many patterns alongside each other on a timeline.
dc	dc { @A, @B, @C }	DC stands for "detail canvas" – shows a view of several patterns that is meant to imitate the feel of a stepper debugger. It has a time control with a forward and back button, and shows only instances of @A, @B, @C, etc. that match or span that point in time.

Appendix B – Study materials for Study 5

The following pages were the tutorial materials for Study 5, decribed in Chapter 8. The first two pages were given to each participant, on a double-sided sheet, and they were stepped through it. Then a randomized subset of the remaining pages (the procedure is described in the chapter) were given to them, one at a time, as they were asked to carry out the task shown.

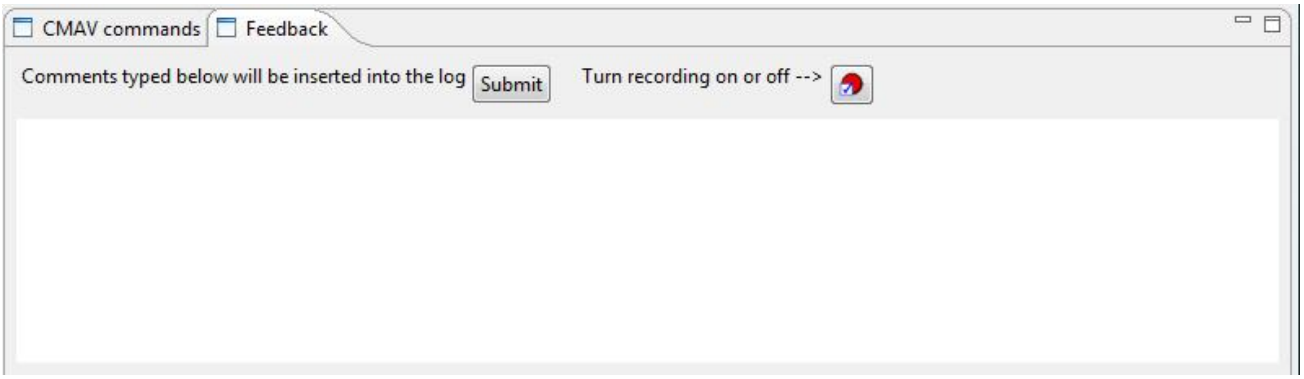
CMAV Tutorial

pane mgmt First I want to show you how window management works in Eclipse. The window panes within the workbench are called "views". They are tabbed, so there can be several views stacked up within one pane.

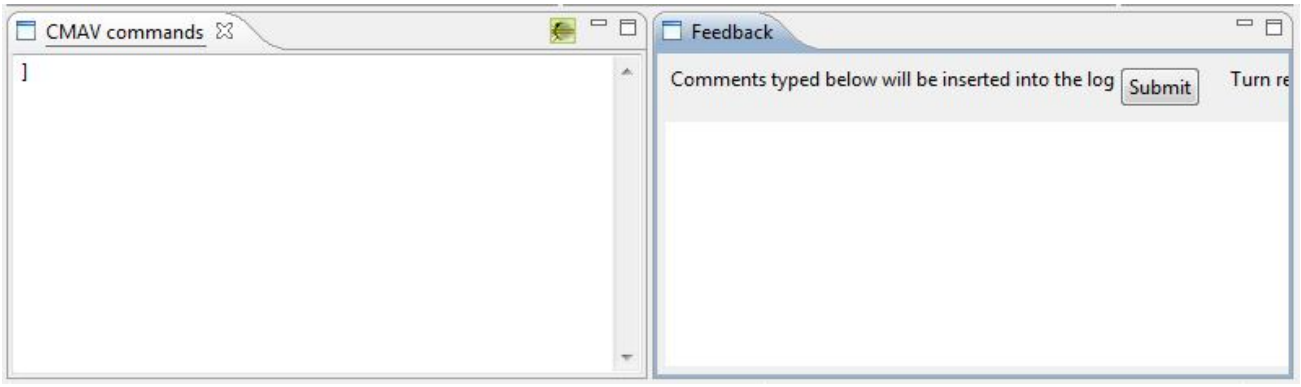
You can change the size of the panes by dragging the edges between them. Try making a pane bigger.

You can also move views from one pane to another by dragging the tab. Try moving a pane.

Finally, you can make new panes, by dragging a view to the edge of a pane, until you see a border showing you its new location. Try making this:



Look like this:



CMAV files and databases You can double-click on an ".alog" file to open that run. You can see which is open by the name at the top of the file. Opening a different file replaces the raw data, but in terms of the queries you've already got open.

You can go back and forth and open different runs; however for the purposes of this study, once you've answered the questions for a run, you can't go back and change your answers.

Data Selector This pane shows the set of queries that are available. The "raw" ones are just plain trace data that has been imported from a model. The "User" ones are queries and views derived from those that you have explicitly named.

You can double-click to open one of these as a table and see its data. The tables provided by default are:

@buffers The contents of all the buffers, timestamped with a new entry every time a chunk or slot in a buffer changes

@buffer_status The status flags of all the buffers, over time

@events This is equivalent to the ACT-R trace, with full detail turned on

@user_event For this particular model, @user_event lists the strings of stimuli that were presented to the model. These were placed in the visicon at the time of each entry of @user_event, and the model visually scans the stimuli to learn the string.

@visicon All changes to the visicon chunks over the lifetime of the model

@time Just a list of timestamps at which anything anywhere changed within the model state (but the details of these changes are found in the other tables.)

So, for example, click on @buffers. You can close it again by clicking the X. But don't do this yet.

right-clicking to do things In CMAV, you create queries by starting with raw data, and gradually rearranging it into a more useful form.

One way to do this is to use these icons. The green circle filters just rows that match that cell and column. The X filters excludes items that match. The Sigma summarizes the column. And the dots give you more options. You can undo the effect with control Z. So give that a try.

Another way to do this is by right-clicking some item in the table. If it's not clear where to click, then just click on any data item. The menu gives you a variety of ways to do the same thing: at the top the actions you can take are organized by the operation they perform, and at the bottom they're organized by the data that they affect.

Naming and patterns When you manipulate a table in some way, like filtering it, the software comes up with a new name for it. In this case, here is the name it made up for the filtered @buffer table.

You can save this query by typing a new name here. So go ahead and give it a new name.

Notice that the new name shows up here in the Data Selector under User. So, if you close this window, you can get back to it by clicking here. You can also see how many rows it had.

If you haven't named something, you can still get to it by looking at the automatically generated name for it under AutoNamed.

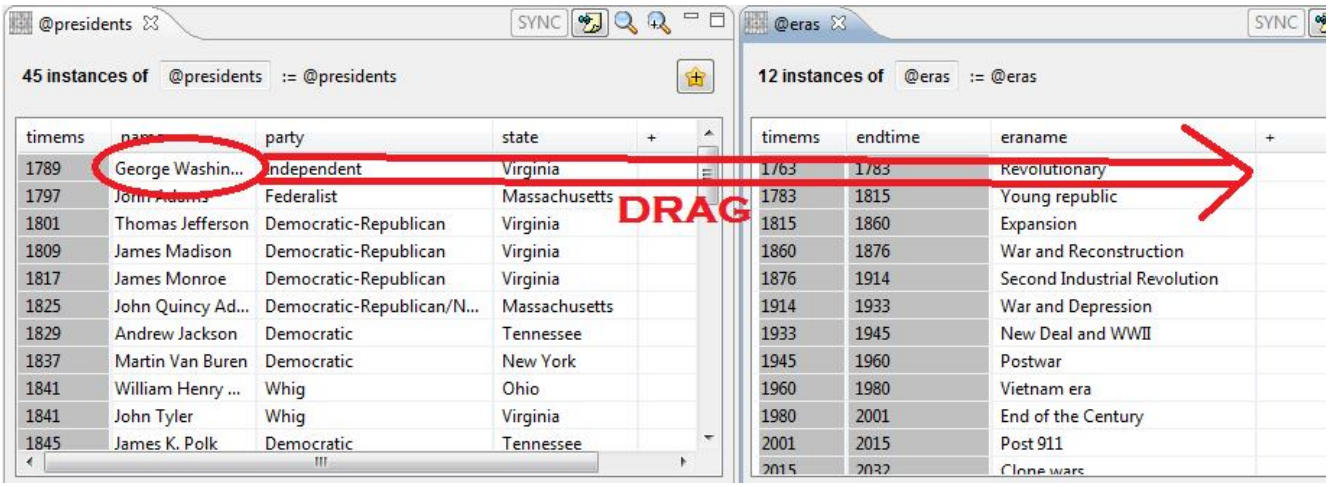
Your named queries, and your window arrangement, will stay in place when you load each subsequent model trace, but reflect the data from the new trace.

Undo Global "undo" for the last action you performed is ctrl-Z. It's also up here in the main menu.

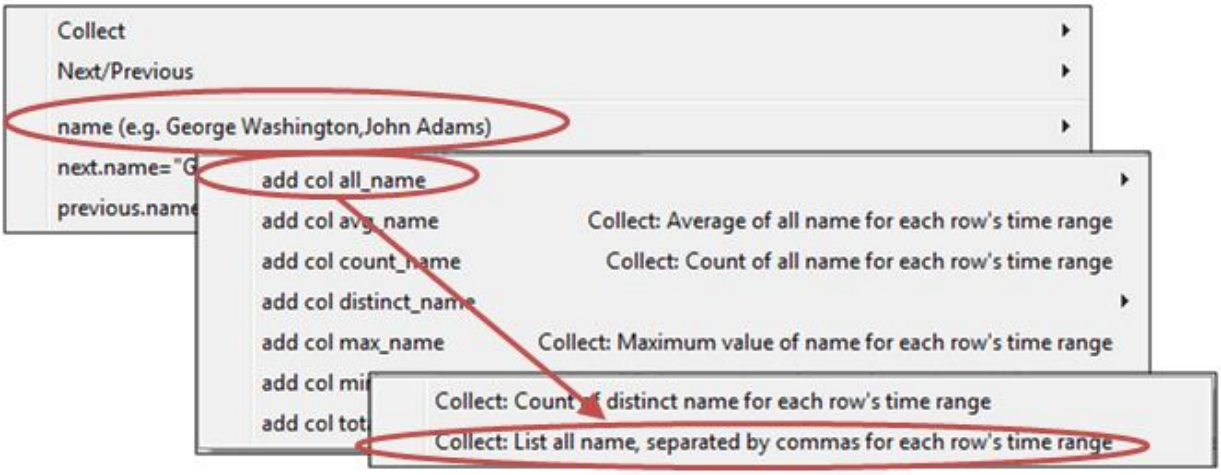
context buttons In a tabular view of data, you can perform some functions with some icons on the screen. Hover over a cell to get clickable icons for filtering *in* or *out* a particular value in a column, or for summarizing that value. Clicking the dots brings up the same menu as right-clicking does.

At the top of the column are some labels representing restrictions on your query; you can eliminate that restriction by clicking on the label.

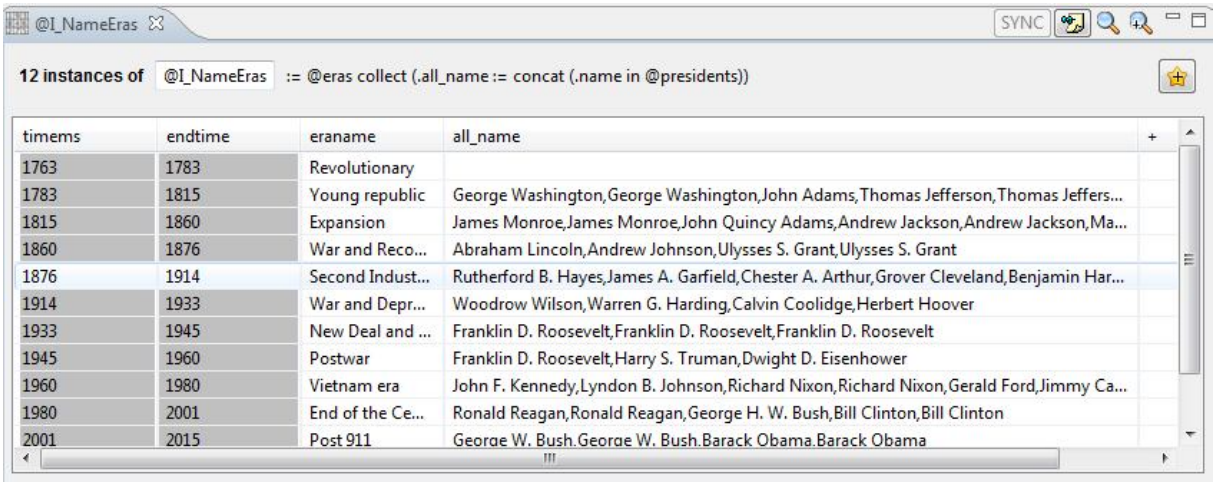
(C-CC) What presidents were inaugurated in each era? Suppose you have a list of eras of US history, and you'd like a concise list of presidents in each era. Here is the raw data you might have available:



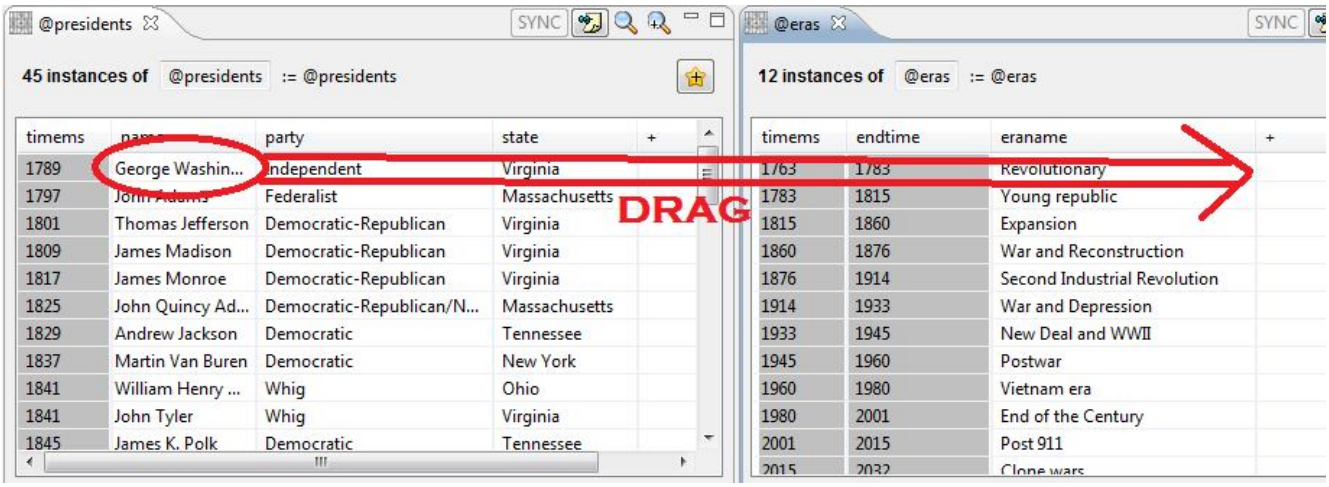
To combine these, drag an example data item (in this case, a president's name) over to the table you want to add information to. There is more than one way to combine these tables, so look for content related to your query in the menu, and pick the option that best describes the result you want:



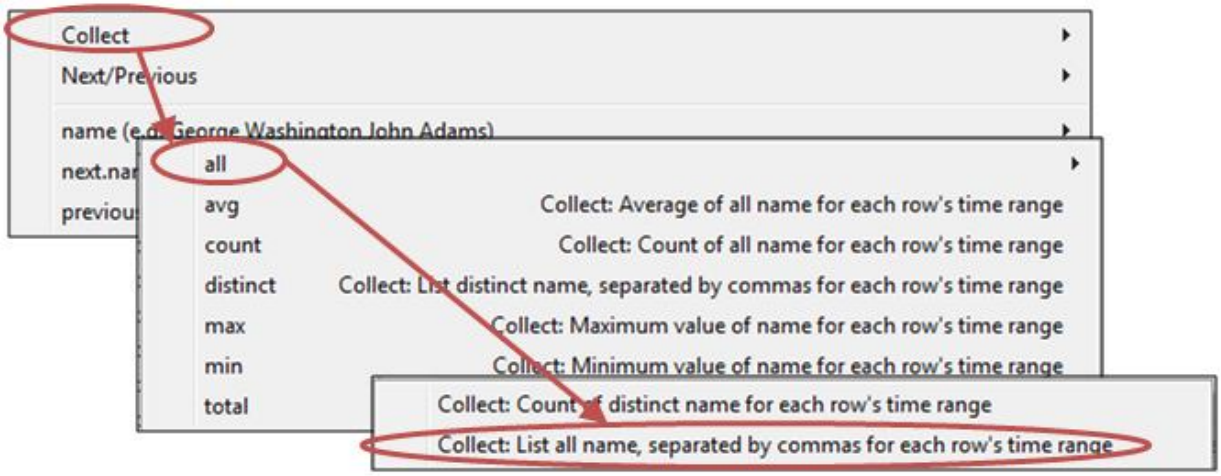
This is the result:



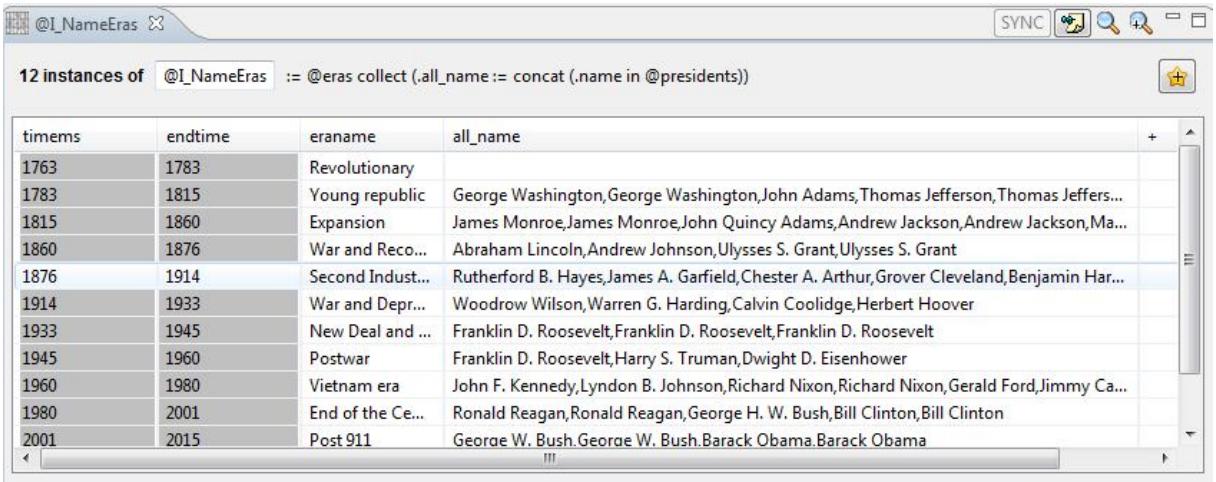
(C-OC) What presidents were inaugurated in each era? Suppose you have a list of eras of US history, and you'd like a concise list of presidents in each era. Here is the raw data you might have available:



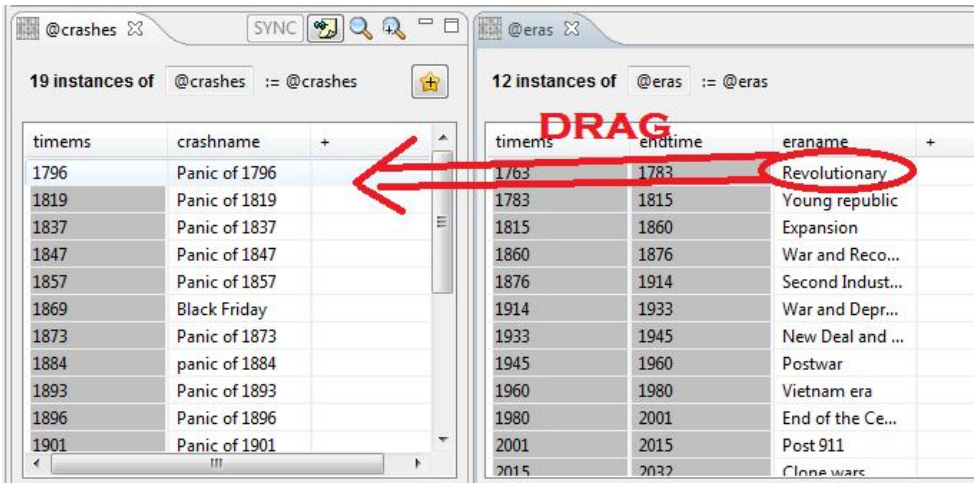
To combine these, drag an example data item (in this case, a president's name) over to the table you want to add information to. There is more than one way to combine these tables, so look for the "Collect" operation, and pick the option that best describes the result you want:



This is the result:

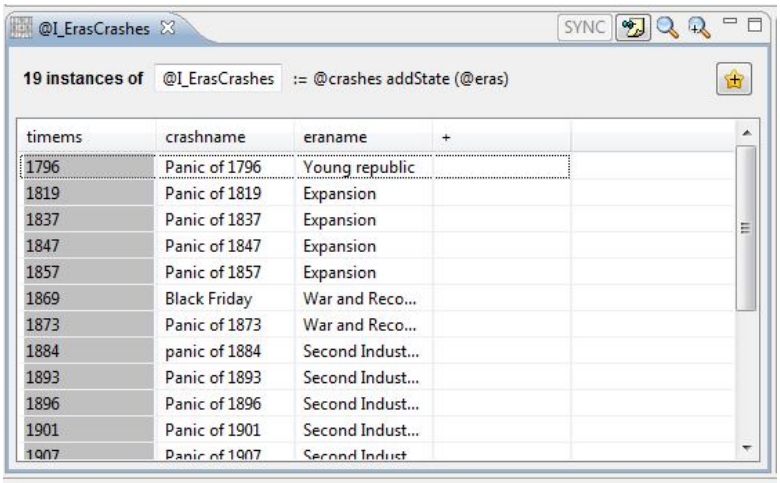


(A-CC) In what era was each stock market crash? Another example: for each of a list of stock market crashes, in which era of history did it occur?

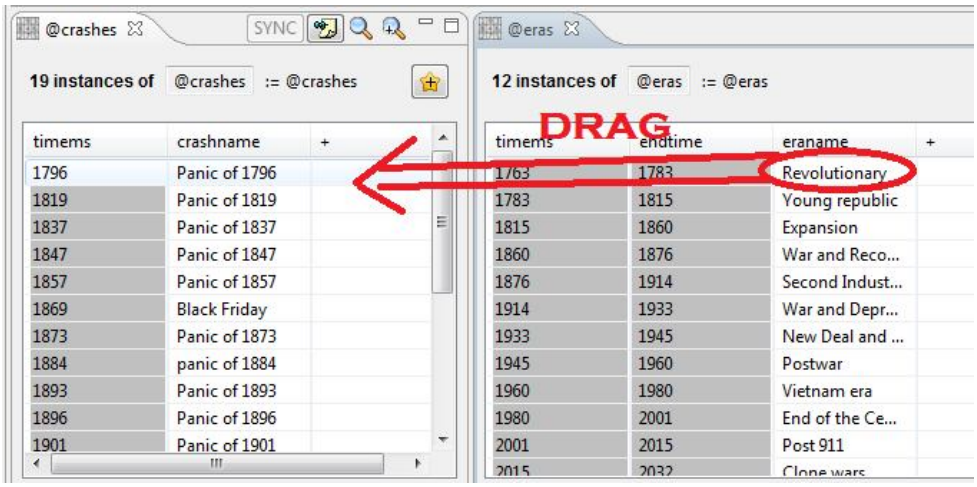


To combine these, drag an example data item over to the table you want to add information to. There is more than one way to combine these tables, so look for content related to your query in the menu, and pick the option that best describes the result you want:

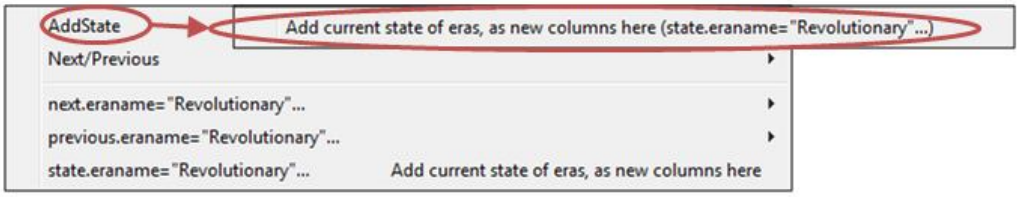
This is the result:



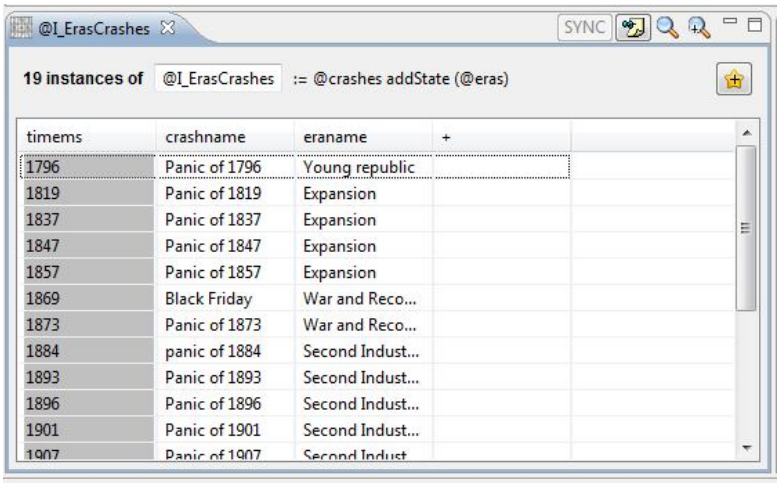
(A-OC) In what era was each stock market crash? Another example: for each of a list of stock market crashes, in which era of history did it occur?



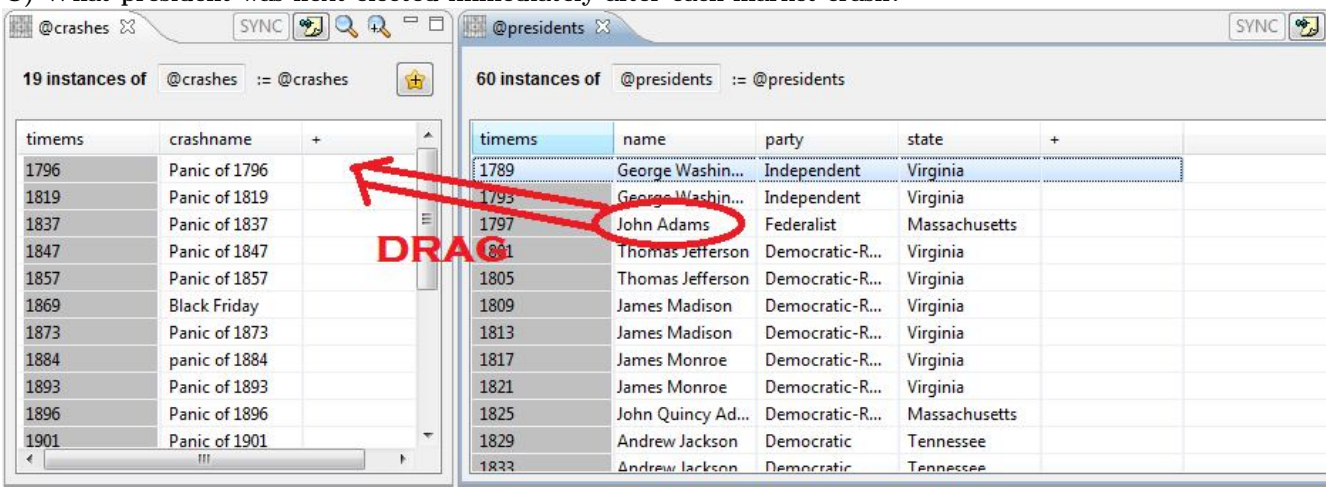
To combine these, drag an example data item over to the table you want to add information to. There is more than one way to combine these tables, so look for the "AddState" operation, and pick the option that best describes the result you want:



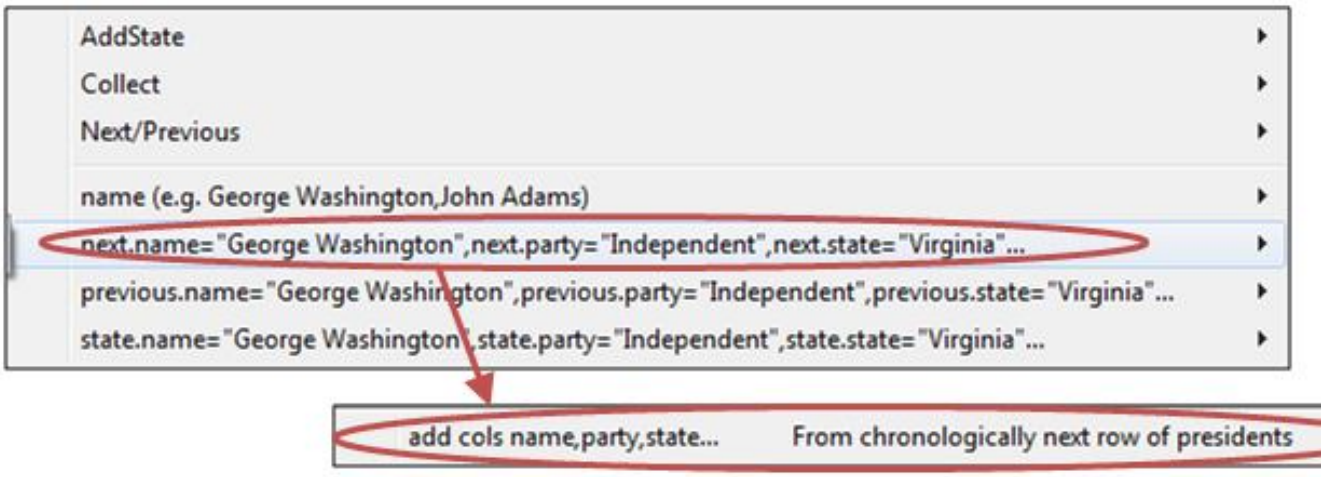
This is the result:



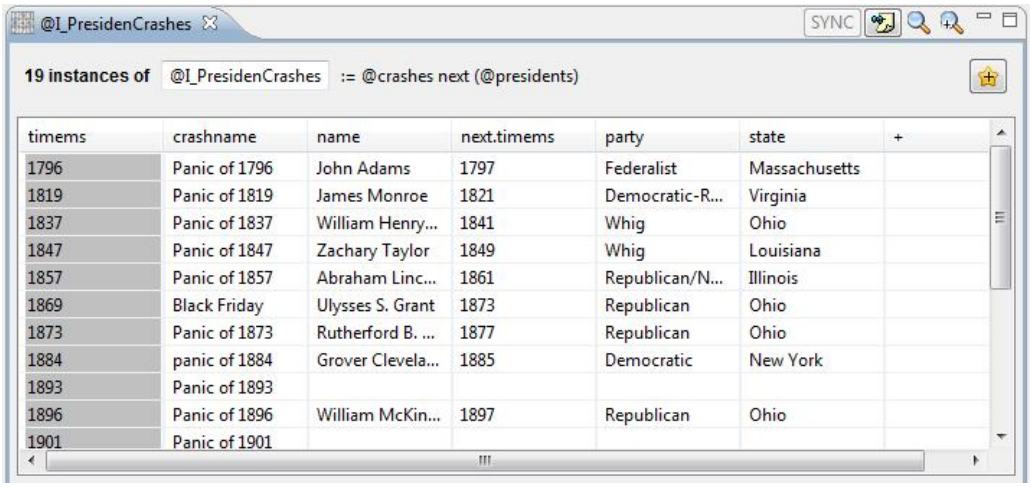
(N-CC) What president was next elected immediately after each market crash?



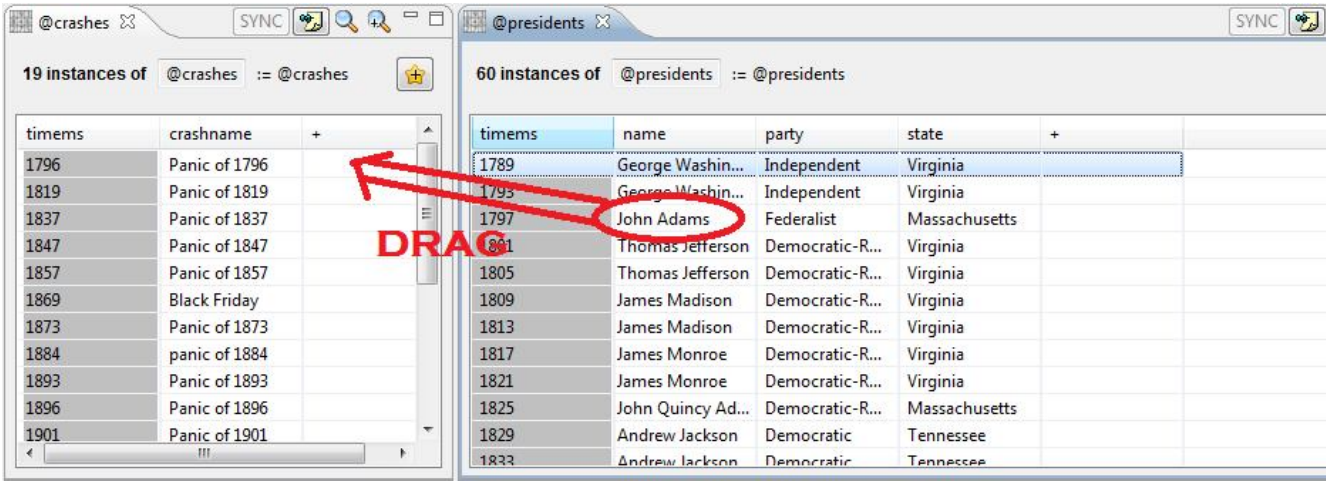
To combine these, drag an example data item over to the table you want to add information to. There is more than one way to combine these tables, so look for content related to your query in the menu, and pick the option that best describes the result you want:



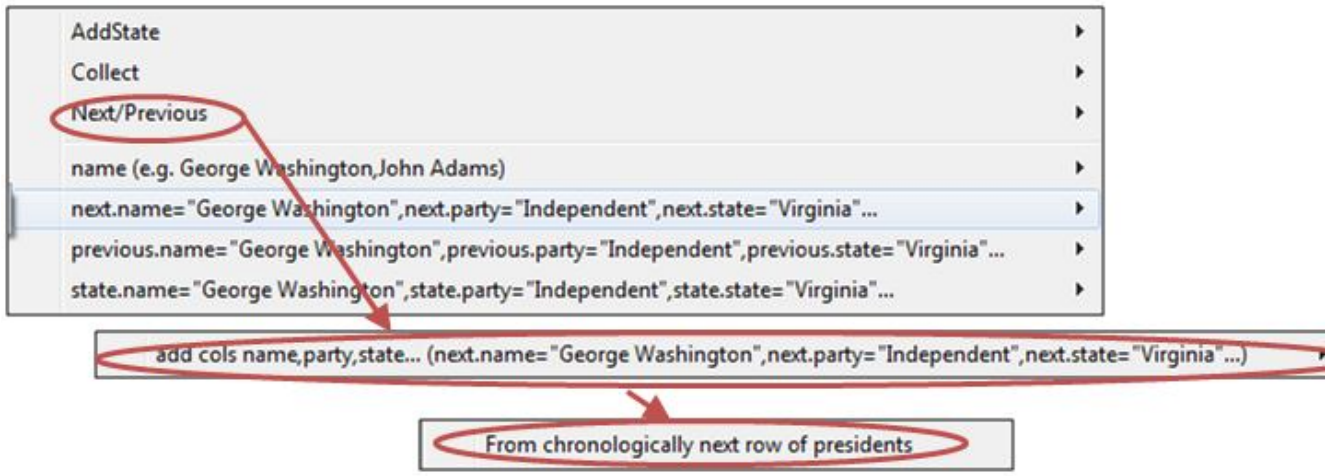
This is the result:



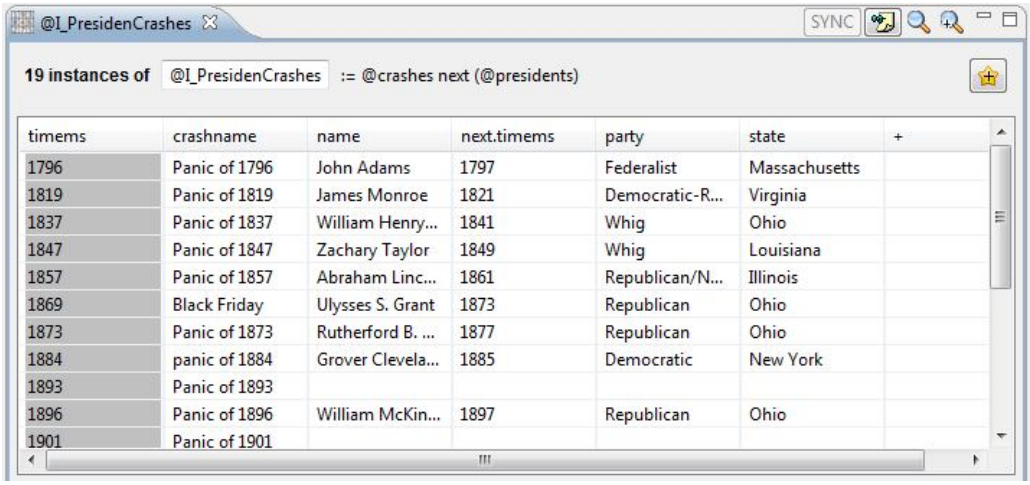
(N-OC) What president was next elected immediately after each market crash?



To combine these, drag an example data item over to the table you want to add information to. There is more than one way to combine these tables, so look for the "Sequence" operation, and pick the option that best describes the result you want:



This is the result:

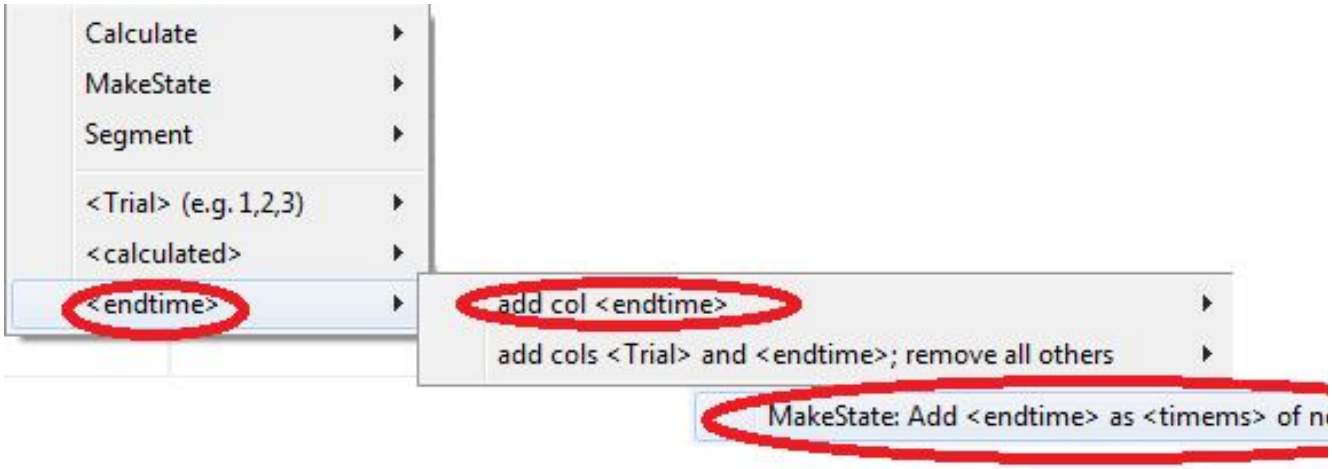


(M-CC) When did each presidential term of office end?

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To turn a list of events into a list of time periods, you need to add an end time. Right click in any cell, and choose "endtime", and pick the appropriate menu option.



This is the result:

@I_StatePresiden

60 instances of @I_StatePresiden := @presidents state

timems	name	party	state	endtime	+
1789	George Washin...	Independent	Virginia	1793	
1793	George Washin...	Independent	Virginia	1797	
1797	John Adams	Federalist	Massachusetts	1801	
1801	Thomas Jefferson	Democratic-R...	Virginia	1805	
1805	Thomas Jefferson	Democratic-R...	Virginia	1809	
1809	James Madison	Democratic-R...	Virginia	1813	
1813	James Madison	Democratic-R...	Virginia	1817	
1817	James Monroe	Democratic-R...	Virginia	1821	
1821	James Monroe	Democratic-R...	Virginia	1825	
1825	John Quincy Ad...	Democratic-R...	Massachusetts	1829	
1829	Andrew Jackson	Democratic	Tennessee	1833	
1833	Andrew Jackson	Democratic	Tennessee	1837	

(M-OC) When did each presidential term of office end? (Counting reelections as **separate** terms)

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To turn a list of events into a list of time periods, you can use the "makeState" operation. Right click anywhere in the table, find "makeState", and pick the appropriate menu option.

This is the result:

@I_StatePresiden

60 instances of @I_StatePresiden := @presidents state

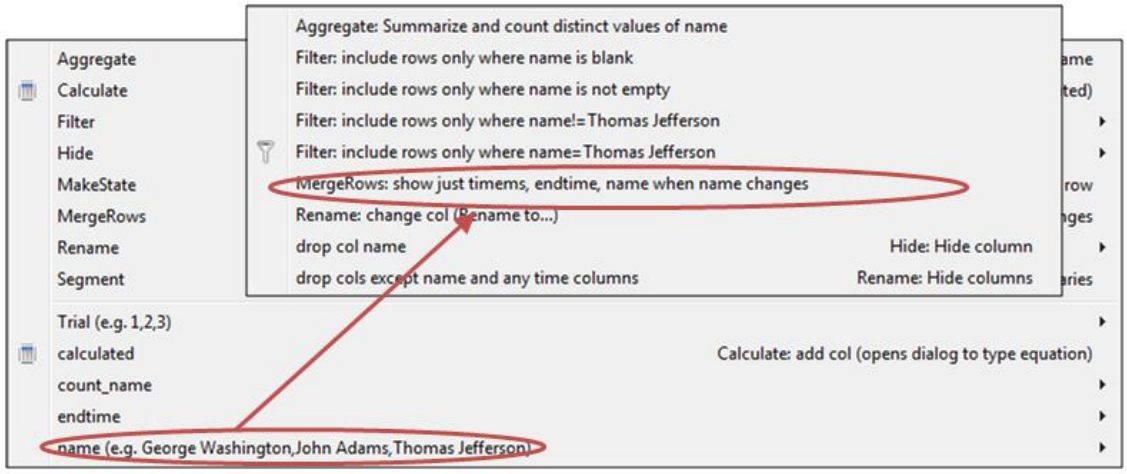
timems	name	party	state	endtime	+
1789	George Washin...	Independent	Virginia	1793	
1793	George Washin...	Independent	Virginia	1797	
1797	John Adams	Federalist	Massachusetts	1801	
1801	Thomas Jefferson	Democratic-R...	Virginia	1805	
1805	Thomas Jefferson	Democratic-R...	Virginia	1809	
1809	James Madison	Democratic-R...	Virginia	1813	
1813	James Madison	Democratic-R...	Virginia	1817	
1817	James Monroe	Democratic-R...	Virginia	1821	
1821	James Monroe	Democratic-R...	Virginia	1825	
1825	John Quincy Ad...	Democratic-R...	Massachusetts	1829	
1829	Andrew Jackson	Democratic	Tennessee	1833	
1833	Andrew Jackson	Democratic	Tennessee	1837	

(M2-CC) When did each president finally leave office (Counting reelections all **together** as a single unit)

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To turn a list of events with unneeded repetition in it into a list of time periods, you need to add an end time whenever the column in question changes, and throw out the intervening rows. Right click on an example cell in a column whose duplicates you want to merge together (in this case, any cell in column "name"), and choose "endtime", and pick the appropriate menu option.



This is the result:

@I_StatePresiden2

44 instances of @I_StatePresiden2 := @presidents state (.name)

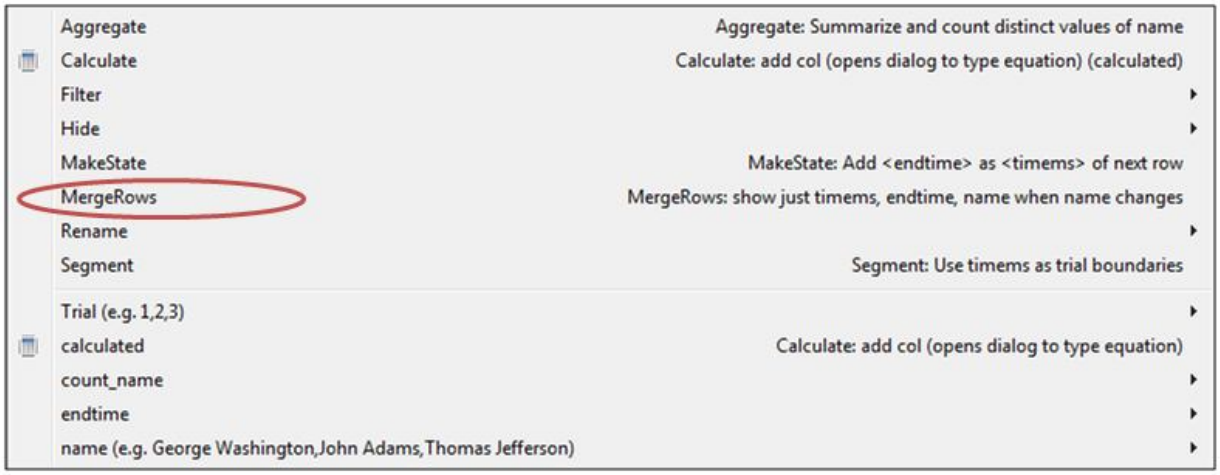
timems	name	endtime	+
1789	George Washington	1797	
1797	John Adams	1801	
1801	Thomas Jefferson	1809	
1809	James Madison	1817	
1817	James Monroe	1825	
1825	John Quincy Adams	1829	
1829	Andrew Jackson	1837	
1837	Martin Van Buren	1841	
1841	William Henry Harrison	1841	
1841	John Tyler	1845	
1845	James K. Polk	1849	
1849	Zachary Taylor	1850	

(M2-OC) When did each president finally leave office? (Counting reelections all **together** as a single unit)

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To turn a list of events with unneeded repetition in it into a list of time periods, you need to merge rows that repeat the value in question. Right click on an example cell in a column whose duplicates you want to merge together (in this case, any cell in column "name"), and choose "Merge", and pick the appropriate menu option.



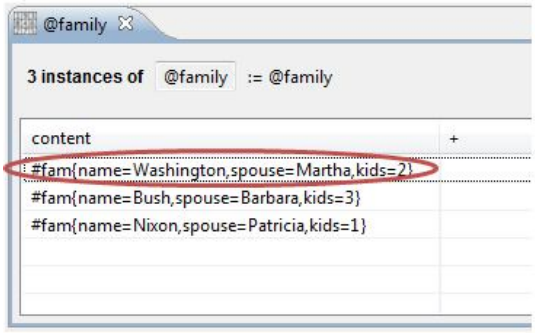
This is the result:

@I_StatePresiden2

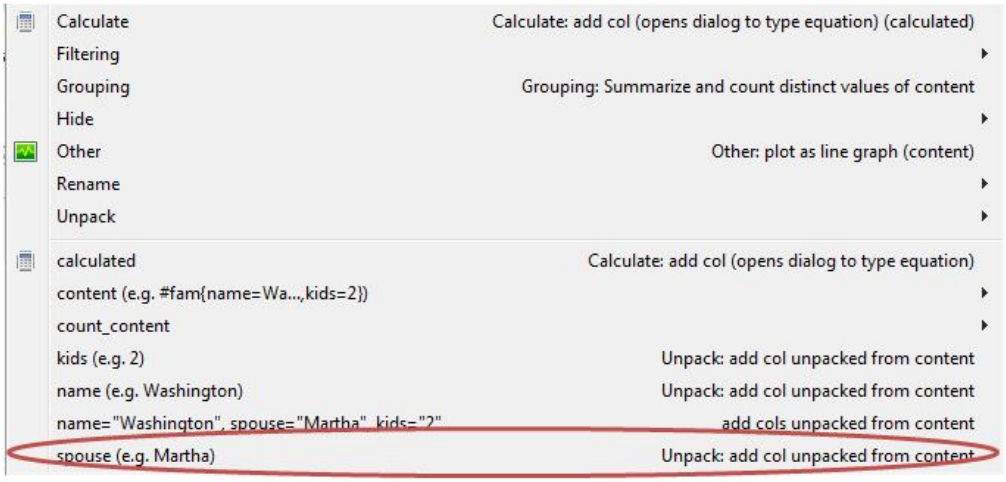
44 instances of @I_StatePresiden2 := @presidents state (.name)

timems	name	endtime	+
1789	George Washington	1797	
1797	John Adams	1801	
1801	Thomas Jefferson	1809	
1809	James Madison	1817	
1817	James Monroe	1825	
1825	John Quincy Adams	1829	
1829	Andrew Jackson	1837	
1837	Martin Van Buren	1841	
1841	William Henry Harrison	1841	
1841	John Tyler	1845	
1845	James K. Polk	1849	
1849	Zachary Taylor	1850	

(U-CC) Who was each president's spouse?

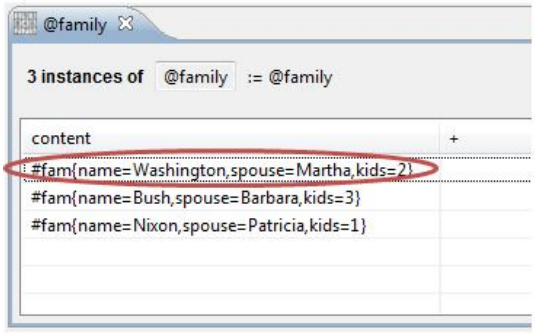


To extract a column out of a "packed" cell like the one in the @family table, right click on one of the packed cells that contains that field, and choose from the menu the name of the field you'd like to extract.

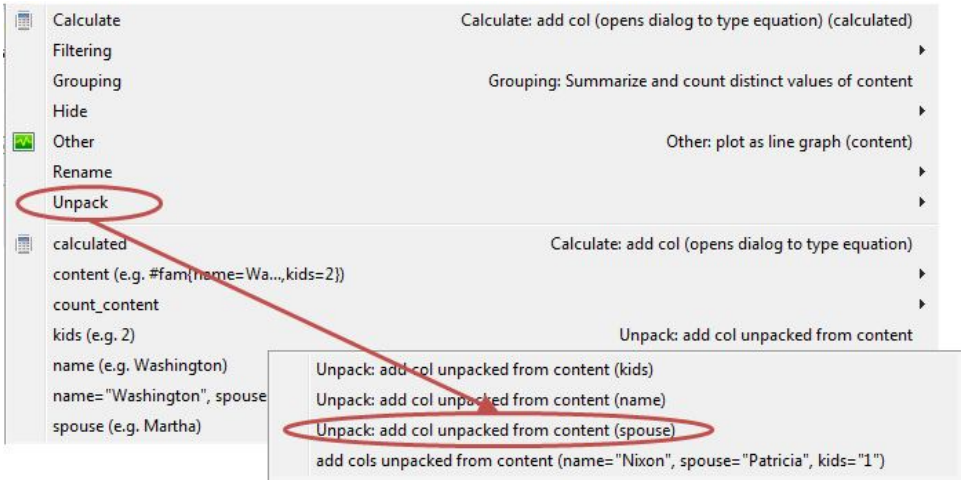


This is the result:

(U-OC) Who was each president's spouse?



To extract a column out of a "packed" cell like the one in the @family table, right click on one of the packed cells that contains that field, choose the "Unpack" submenu, and find the appropriate menu item underneath it.



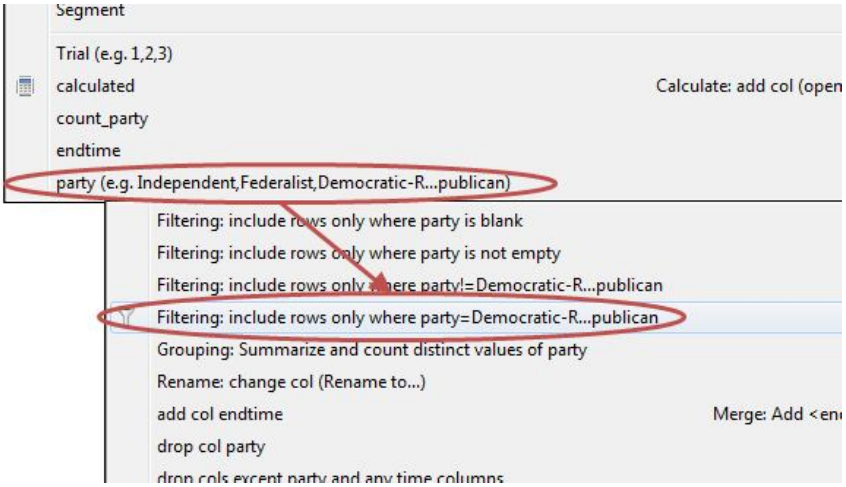
This is the result:

(F-CC) Which presidents were in the Democratic-Republican party?

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To reduce a list down to just rows that match a particular value in some column, right click on an example value in the column you'd like to filter on, choose the name of the field you are filtering on, and pick the appropriate menu option.



This is the result:

6 instances of @Democratic_Republicans := @presidents filter (.party = "Democratic-Republican")

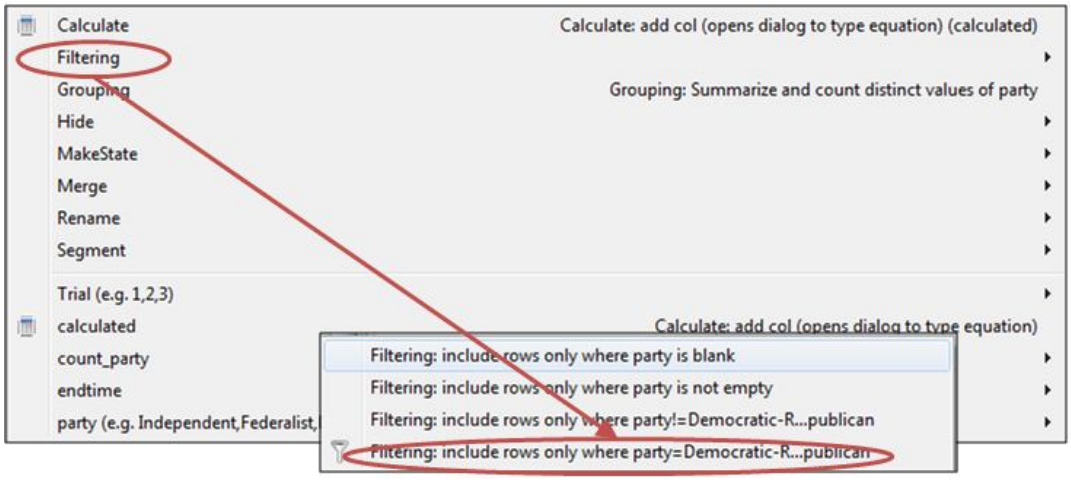
timems	name	party	state	+
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	

(F-OC) Which presidents were in the Democratic-Republican party

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To reduce a list down to just rows that match a particular value in some column, right click anywhere in the table, choose the "Filter" submenu, and pick the appropriate menu option.



This is the result:

6 instances of @Democratic_Republicans := @presidents filter (.party = "Democratic-Republican")

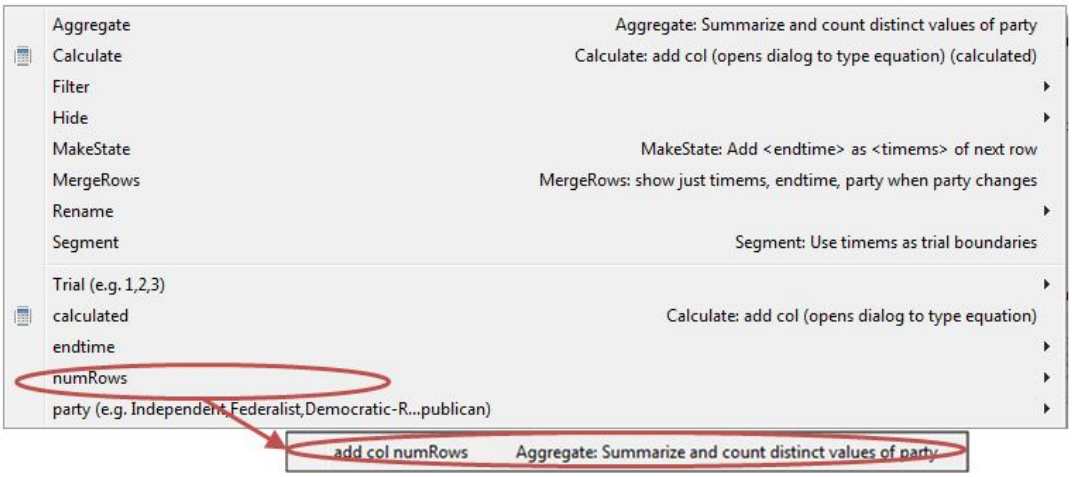
timems	name	party	state	+
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	

(S-CC) How many presidents were in each party?

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To count the number of rows in the list that contain each party, right-click the state column, and find "numRows" in the menu, since that's a column you might expect to appear in the resulting table. Under its submenu, pick the item that best describes the way you want the information displayed.



This is the result:

9 instances of @I_AggrPresiden := @presidents group .party

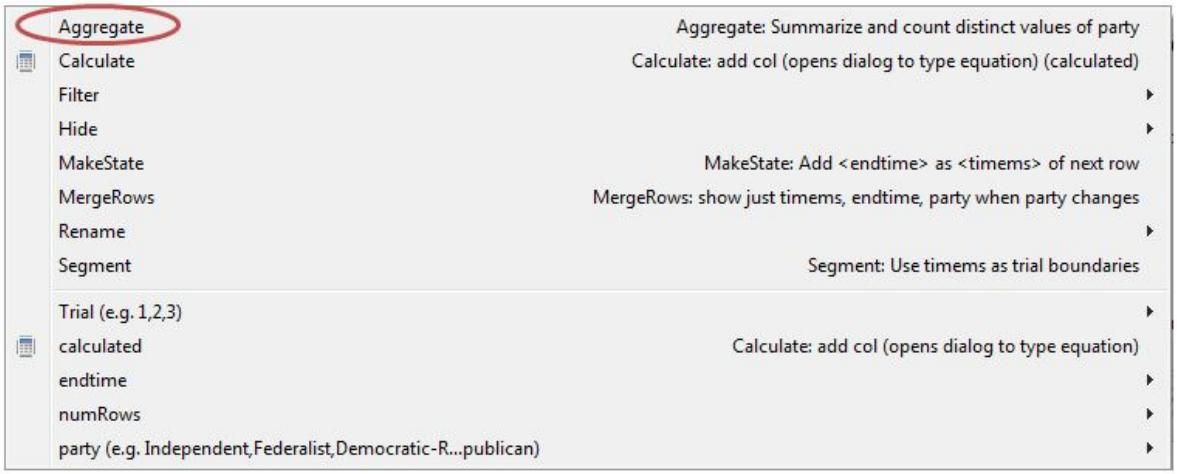
party	numRows	+
Democratic	22	
Democratic-Republican	6	
Democratic-Republican/N...	1	
Democratic/National Union	1	
Federalist	1	
Independent	2	
Republican	22	
Republican/National Union	1	
Whig	4	

(S-OC) How many presidents were in each party?

60 instances of @presidents := @presidents

timems	name	party	state	+
1789	George Washin...	Independent	Virginia	
1793	George Washin...	Independent	Virginia	
1797	John Adams	Federalist	Massachusetts	
1801	Thomas Jefferson	Democratic-Republican	Virginia	
1805	Thomas Jefferson	Democratic-Republican	Virginia	
1809	James Madison	Democratic-Republican	Virginia	
1813	James Madison	Democratic-Republican	Virginia	
1817	James Monroe	Democratic-Republican	Virginia	
1821	James Monroe	Democratic-Republican	Virginia	
1825	John Quincy Ad...	Democratic-Republican/N...	Massachusetts	
1829	Andrew Jackson	Democratic	Tennessee	

To count the number of times each party appears in this list, right-click the party column, choose "summarize and count" option under the "Aggregate" category:



This is the result:

9 instances of @I_AggrPresiden := @presidents group .party

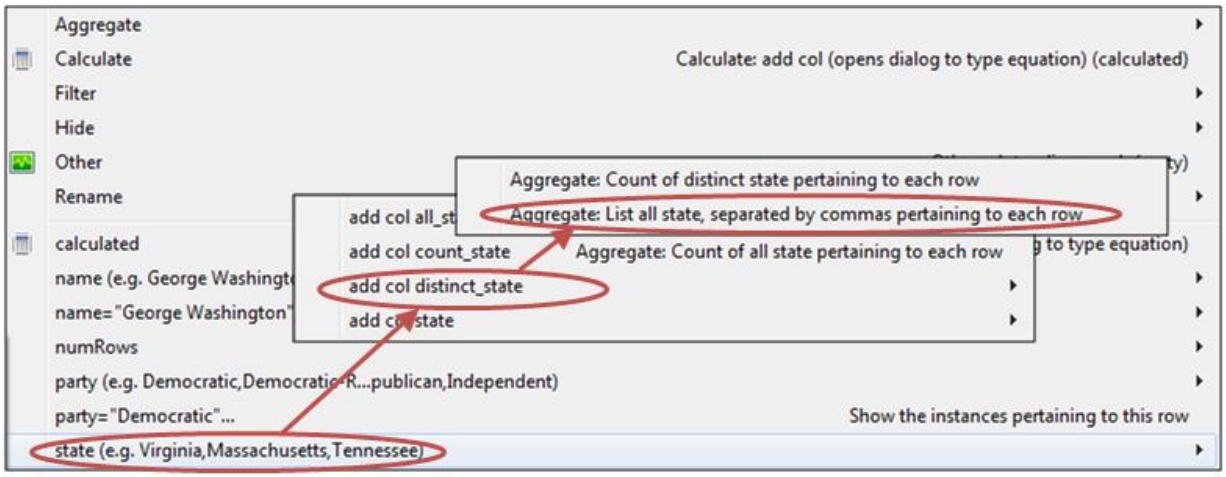
party	numRows	+
Democratic	22	
Democratic-Republican	6	
Democratic-Republican/N...	1	
Democratic/National Union	1	
Federalist	1	
Independent	2	
Republican	22	
Republican/National Union	1	
Whig	4	

(S2-CC) For each party, which states did it draw presidents from?

9 instances of @I_AggrPresiden := @presidents group .party

party	numRows	+
Democratic	22	
Democratic-Republican	6	
Democratic-Republican/N...	1	
Democratic/National Union	1	
Federalist	1	
Independent	2	
Republican	22	
Republican/National Union	1	
Whig	4	

To grab more information out of a summary listing like this, right click in any cell, and find the field you are interested in in the menu. Find the submenu that best describes the way you want to display the information.



This is the result:

9 instances of @I_AggrPresiden2 := @presidents group .party aggregate (.distinct_state := unique

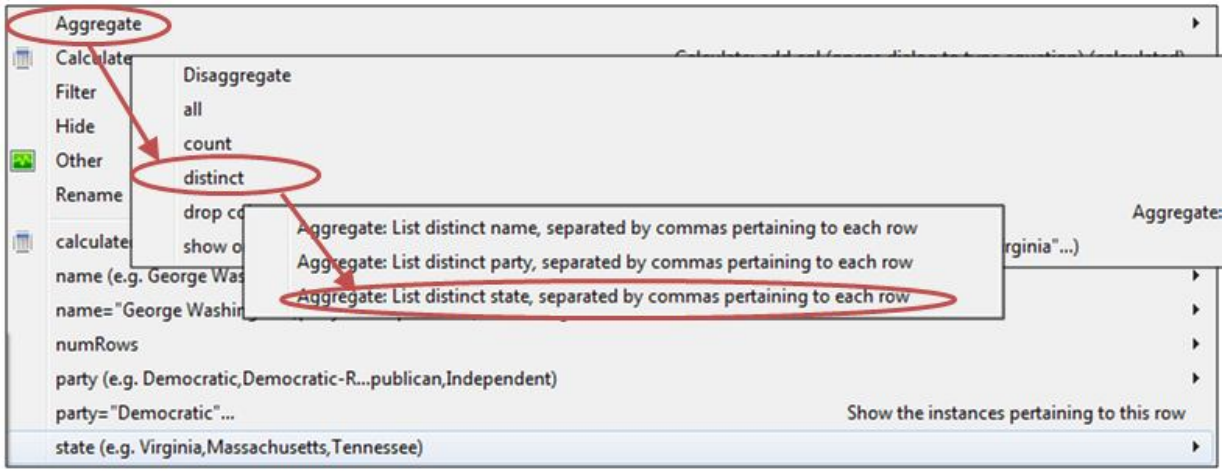
party	numRows	distinct_state	+
Democratic	22	Tennessee,New York,New Hampshire,...	
Democratic-Republican	6	Virginia	
Democratic-Republican/N...	1	Massachusetts	
Democratic/National Union	1	Tennessee	
Federalist	1	Massachusetts	
Independent	2	Virginia	
Republican	22	Ohio,New York,Indiana,Massachusetts...	
Republican/National Union	1	Illinois	
Whig	4	Ohio,Virginia,Louisiana,New York	

(S2-OC)For each party, which states did it draw presidents from?

9 instances of @I_AggrPresiden := @presidents group .party

party	numRows	+
Democratic	22	
Democratic-Republican	6	
Democratic-Republican/N...	1	
Democratic/National Union	1	
Federalist	1	
Independent	2	
Republican	22	
Republican/National Union	1	
Whig	4	

To add more summary information to this table about each of the states, like what parties fielded presidents from those states, right click in any cell, and choose "summarize" under the "Grouping" submenu, and find the submenu that best describes the way you want the information displayed.



This is the result:

9 instances of @I_AggrPresiden2 := @presidents group .party aggregate (.distinct_state := unique

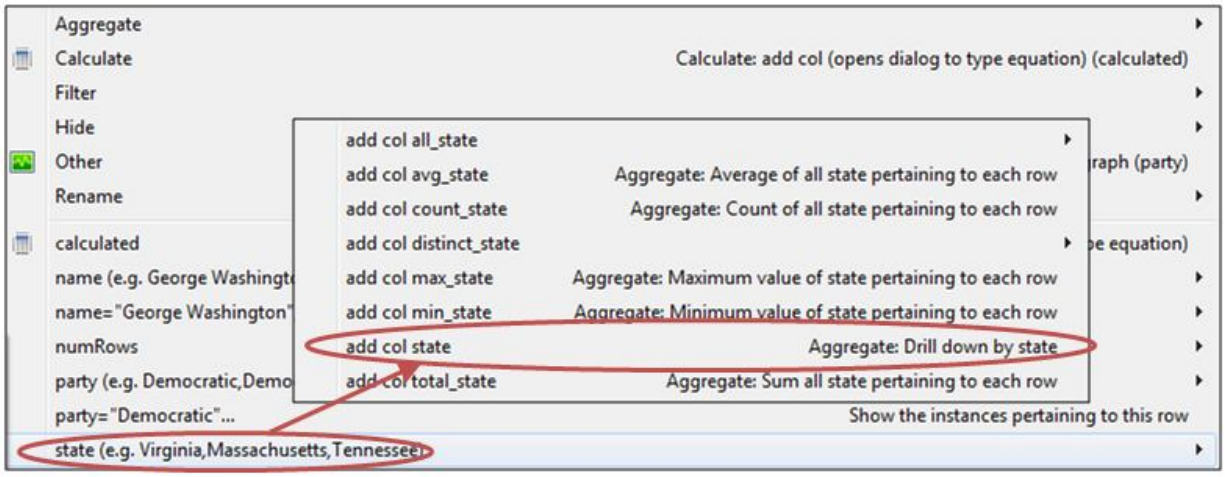
party	numRows	distinct_state	+
Democratic	22	Tennessee,New York,New Hampshire,...	
Democratic-Republican	6	Virginia	
Democratic-Republican/N...	1	Massachusetts	
Democratic/National Union	1	Tennessee	
Federalist	1	Massachusetts	
Independent	2	Virginia	
Republican	22	Ohio,New York,Indiana,Massachusetts...	
Republican/National Union	1	Illinois	
Whig	4	Ohio,Virginia,Louisiana,New York	

(S3-CC) Summarize presidents by state of origin and also party

9 instances of @I_AggrPresiden := @presidents group .party

party	numRows	+
Democratic	22	
Democratic-Republican	6	
Democratic-Republican/N...	1	
Democratic/National Union	1	
Federalist	1	
Independent	2	
Republican	22	
Republican/National Union	1	
Whig	4	

To grab more information out of a summary listing like this, right click in any cell, and find the field you are interested in in the menu. Find the submenu that best describes the way you want to display the information.



This is the result:

29 instances of @I_AggrPresiden4 := @presidents group (.party, .state)

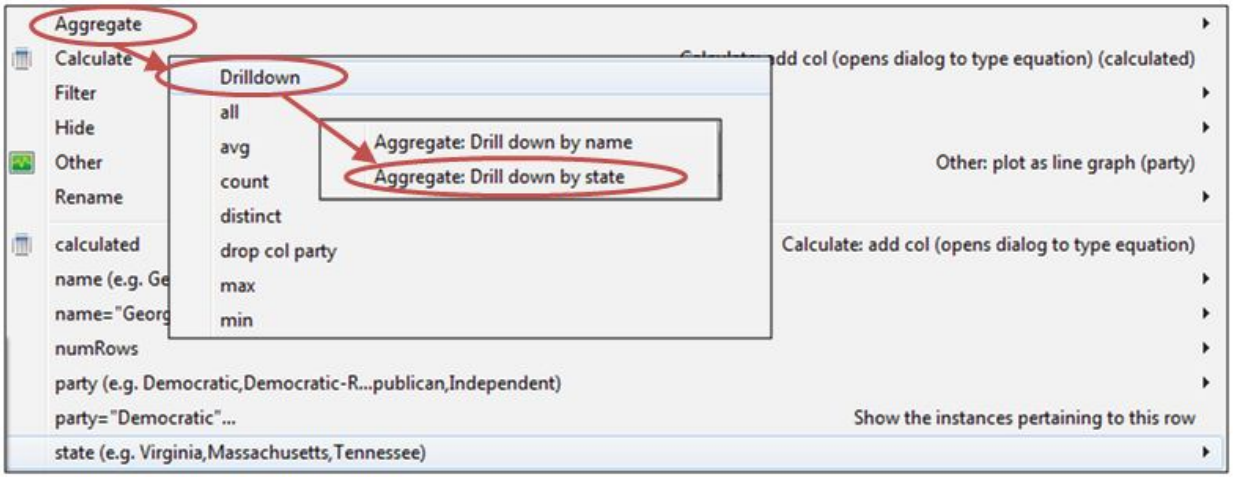
party	state	numRows	+
Democratic	Illinois	2	
Democratic	Massachusetts	1	
Democratic	Missouri	1	
Democratic	New Hampshire	1	
Democratic	New Jersey	2	
Democratic	New York	7	
Democratic	Pennsylvania	1	
Democratic	Tennessee	3	
Democratic	Texas	1	
Democratic-Republican	Virginia	6	
Democratic-Republican/N...	Massachusetts	1	

(S3-OC) Summarize presidents by state of origin and also party

9 instances of @I_AggrPresiden := @presidents group .party

party	numRows	+
Democratic	22	
Democratic-Republican	6	
Democratic-Republican/N...	1	
Democratic/National Union	1	
Federalist	1	
Independent	2	
Republican	22	
Republican/National Union	1	
Whig	4	

To break a listing like this down further into more specific categories, right-click anywhere in the table, and find the "Drilldown" option under the "Aggregate" menu. Find the submenu that best describes the way you want to display the information.



This is the result:

29 instances of @I_AggrPresiden4 := @presidents group (.party, .state)

party	state	numRows	+
Democratic	Illinois	2	
Democratic	Massachusetts	1	
Democratic	Missouri	1	
Democratic	New Hampshire	1	
Democratic	New Jersey	2	
Democratic	New York	7	
Democratic	Pennsylvania	1	
Democratic	Tennessee	3	
Democratic	Texas	1	
Democratic-Republican	Virginia	6	
Democratic-Republican/N...	Massachusetts	1	

The following page is the task sheet that modelers in Study 5 worked from for their main task.

Table 1: Write "yes" or "no" in each box to indicate whether the run meets this expectation or not

Expectation	Run 1	Run 2	Run 3	Run 4	Run 5
1. No values besides p, s, t, v, x should ever appear in the "value" slot of a "text" chunk in the visual buffer					
2. Each visual scan the model makes of the screen (that is, each span of time starting when visual buffer's screen_pos slot holds "visual-location0-0-0", up until the next time it does so), should take less than 2 seconds.					
3. In each trial, there should be at least three "full" visual scans, that is, the sequence of letters that appear in the goal buffer (in the "last" slot of chunk type "guess"), should be the same letters, in the same order, as the current trial's stimulus sequence. The stimulus sequence for each trial is shown in @user_events.					
4. In chunks of type "guess", some combinations of slot values should never occur: (last=s, predict=p), (last=p, predict=x), (last=x, predict=p)					

Lab study:Experimental tasks

Here (Table 1) is a list of expectations a modeler has about this model. Your task is to check these as quickly and correctly as possible. I've shown you some features of the software that might help with the task, but you're not obligated to use any particular feature. If you judge that some feature is intended to help, but you can get through the experiment faster without it, then don't use it.

You have the source code of the model to look at here. You're welcome to examine this model, but I want to caution you against trying to draw firm conclusions about runtime behavior just from reading the code: the task is about actual behavior, not intended behavior. There may be subtle bugs in the code, or experimental modules installed that cause it to behave in unexpected ways.

You'll be asked to test each of five expectations against each of five runs of the model, one model at a time. You can work in any order you want on the five expectations for one run, but when you are done with a run and load the next one, you may not go back to a previous run.

Although I'm asking you to do the tasks quickly and correctly, the point of the study is not to judge your ability; the point is to assess in what ways the software ends up providing good or poor support for you as you go about this task.

About the Model

The model whose trace you'll be examining attempts to learn a simple grammar: in each trial a string of letters is presented in the visicon, and the model visually scans them several times. It learns associations between one letter and the next in the sequence by placing the letters in a "last" and "predicted" slot, in a chunk of type "guess", in the goal buffer, then flushing that chunk to declarative memory.

This model is a simplification of just the training phase of a model of intuitive decision making, from a paper published by William Kennedy and Robert Patterson in 2012 [1]. (The five versions of the model you will look at are altered in various ways, so the model behavior shouldn't be considered reflective of Kennedy and Patterson's work).

[1] Kennedy, W. G. and Patterson, R.E. (2012) Modeling Intuitive Decision Making in ACT-R. In Proceedings of the 11th International Conference on Cognitive Modeling (ICCM), Berlin, Germany. 12-15 April 2012.

Appendix C – Example screen from Validation study

The validation study Study 4 described in Chapter 7 asked a panel of experienced modelers to review our retrospective *corrections* to what the Wizard of Oz should have provided to the modeler in Study 3W. This was administered through web forms. The content of a sample question is shown below. The choice of RIGHT, FIXABLE, ... was implemented as a set of radio buttons, so the modelers could only choose one.

The modeler and experimenter had this conversation:

Modeler: OK, so I want to find a regular "ED" and how do you see that regular ED being categorized as incorrect? when does that happen? Who does that?
Experimenter: OK, so this... so the model presents
Modeler: in the imaginal, right
Experimenter: correct examples, and then but when it produces something, the correctness assessment is done in the experimental frame.
Modeler: ok
Experimenter: the child is never informed whether it's correct or not.
Modeler: so, I mean, how do I see...
Experimenter: We do have that information from the experimental frame in this trace though
Modeler: OK, so give me a list of when the incorrect was produced,

Would the following display have been an appropriate response to the request?

49 instances of @176 found

WORD	STEM	SUFFIX	VERBTYPE	correct
DO	DO	BLANK	I	0
GET	GET	BLANK	I	0
HAVE	HAVE	BLANK	I	0
LOSE	LOSE	BLANK	I	0
MAKE	MAKE	BLANK	I	0
STAND	STAND	BLANK	I	0
TELL	TELL	BLANK	I	0
DO	DO	ED	I	0
GET	GET	ED	I	0
HAVE	HAVE	ED	I	0
MAKE	MAKE	ED	I	0
DO	NIL	NIL	I	0
DO	UNKNOWN	UNKNOWN	I	0
GET	UNKNOWN	UNKNOWN	I	0
HAVE	UNKNOWN	UNKNOWN	I	0
LOSE	UNKNOWN	UNKNOWN	I	0
MAKE	UNKNOWN	UNKNOWN	I	0
STAND	UNKNOWN	UNKNOWN	I	0
TELL	UNKNOWN	UNKNOWN	I	0
ASK	ASK	BLANK	R	0
CALL	CALL	BLANK	R	0
FOLLOW	FOLLOW	BLANK	R	0
LIVE	LIVE	BLANK	R	0
LOOK	LOOK	BLANK	R	0

Given the participant's request, the final screen they were shown:

RIGHT!	Had exactly the information they asked for
FIXABLE	I could fix it rearranging, removing, or doing simple calculations with what's visible (explain below)
SOME MISSING	Was missing some critical information (explain below)
RIGHT, ASSUMING...	It could be right, but I would need to check on the following to be sure: (write in assumptions)
WRONG!	Was completely wrong (explain below)

Explain your answer: