

Theory-based Analysis of Cognitive Support in Software Comprehension Tools

Andrew Walenstein

Department of Computer Science,
University of Victoria, Victoria, B.C., Canada
walenste@csr.uvic.ca

Abstract

Past research on software comprehension tools has produced a wealth of lessons in building good tools. However our explanations of these tools tends to be weakly grounded in existing theories of cognition and human–computer interaction. As a result, the interesting rationales underlying their design are poorly articulated, leaving the lessons primarily implicit. This paper describes a way of using existing program comprehension theories to rationalize tool designs. To illustrate the technique, key design rationales underlying a prominent reverse engineering tool (Reflexion Model Tool) are reconstructed. The reconstruction shows that theories of cognitive support can be applied to existing cognitive models of developer behaviour. The method for constructing the rationales is described, and implications are drawn for codifying existing design knowledge, evaluating tools, and improving design reasoning.

1. Introduction

When we speak of “software comprehension” we usually are referring to activities that humans do: understanding, conceptualizing, and reasoning about software. In this regard, a crucial aim of tools for software comprehension is to assist and improve human thinking processes. Simply put, software comprehension tool are considered “good” if they support human cognition. There may be many other reasons for why a tool is considered good (computational efficiency, learnability, etc.), but its ability to support cognition is surely a central one. Ultimately, then, the explanation offered for the design of a software comprehension tool will need to rest on some account of which of its features assist cognition and how.

It is important to be able to clearly articulate explanations for why a tool is believed to support developer cognition. If the “claims” [4] about a tool are not made explicit, it is extremely difficult to test them, to compare tools, or to reuse design knowledge. Clearly, any claim about the *cognitive support* provided by a tool will be at least partly psy-

chological in nature. So it seems prudent to desire that our rationalizations be firmly grounded in well-received theories from cognitive science and HCI.

Unfortunately, software development tools are too rarely analyzed for the psychological rationales underlying their design. This makes it considerably less clear what generalizable lessons can be drawn from the tool. For some of the key lessons will relate to the cognitive benefits of the tools. To grasp these, we must have “deep” psychological descriptions of these benefits—not merely “shallow” explanations at the technological level of visualization techniques, program analysis algorithms, and interface features. The deep psychological explanations make it possible to generalize the lesson beyond the specific implementation context [8].

The problem is not so much that we have no knowledge about how to build good tools—we have plenty of good ideas and promising tools—but that we have lacked facility in the theories and methods needed to articulate this knowledge in a principled manner. It is also not the case that there are no theories to draw upon; there are several existing models of software comprehension [24], and a wealth of cognitive science is available to back any analytic efforts. Instead, the historically vexing problem has been putting this knowledge to good use [3]: to know how to extract design rationalizations from cognitive models. We have the tools and we have the models, but too rarely do the two meet.

This paper proposes steps towards improving the analysis of cognitive support in software comprehension tools. The improvements come from a proposed theory-based method for generating psychological rationales from cognitive models. The method involves applying theories of *cognitive support* to cognitive models. One particular reverse engineering tool, the Reflexion Model Tool [13] (RMTTool) is chosen as a sample for a demonstration analysis. RMTTool is an interesting subject to analyze because it is a currently topical reverse engineering tool, it stands apart from many other tools in the way it supports reverse engineering, and it has a design history that can be consulted [12].

It is worth reiterating that the main aim of the rationale extraction exercise is to illustrate a method of theory-based

cognitive support analysis. The aim is *not* to propose new theories, to argue the validity of any particular theory, or to “prove” the superior qualities of RMTTool. Instead, attention is focused on methods for utilizing our existing theoretical resources to extract the design rationales. Of course, a collection of theories will be used in the paper, but should a more suitable set of theories be chosen (or new, improved ones become available in the future), the techniques for extracting rationales will remain largely the same.

Section 2 introduces the cognitive support theories that will be used to extract the rationales. Section 3 briefly describes RMTTool and covers its terminology, and Section 4 reconstructs core rationales using the theories from Section 2. Section 5 extends this theory-based rationale reconstruction to additional aspects of RMTTool. This illustrates that the theories may be helpful for anticipating desirable tool features. Finally, related work is briefly summarized in Section 6, and conclusions and implications for software engineering research are summarized in Section 7.

2. A simple cognitive support design heuristic

It is widely known that a variety of artifacts can aid thinking and make problem solving easier. For instance, consider the process of performing long division using pencil and paper. Without pencil and paper (or a calculator of some sort), most people cannot divide large numbers because their internal memory is quickly overwhelmed. Yet with a pencil and a scrap of paper, even 100 digit numbers can be readily divided. As Norman aptly states: “it is things that make us smart” [17]. Artifacts can help solve problems that involve hard thinking—and plenty of hard thinking is involved in software development. The key to the long division example is that partial results can be stored externally rather than having to be remembered. Although this is a simple example, it illustrates a generalizable principle for supporting cognition: external memory can augment internal memory.

Over the years a variety of ways of assisting human thinking and problem solving have been studied. A motley potpourri of theories and theoretical frameworks have been advanced to explain how the support works. Rather than attempt to comprehensively gather and reconcile these theories, a limited collection of salient results are extracted below. Specifically, three cognitive support theories are de-

scribed in Section 2.1. With these in hand, a simple design heuristic (theory) called the “greedy cognitive support” heuristic is defined in Section 2.2. These resources are used in subsequent sections analyzing RMTTool.

2.1. Cognitive support theories

The essential quality of cognitive support is that it makes human cognition easier or better. We say a “cognitive support theory” is a generalized statement about how and why some abstract class of artifacts (and their uses) manage to make cognition better. The fact that the statement is generalized is important: it is not specific to a fixed set of artifacts or tasks. For instance, although it might be argued that paper and pen support cognition when doing long division, (a) the principle for the cognitive support can apply equally if the pen and paper are replaced with a computerized note pad, and (b) the paper and pen could support many other tasks.

Three cognitive support theories are briefly outlined here: *redistribution*, *perceptual substitution*, and *ends-means structure reification*. These are listed in Table 1.

Redistribution. The key idea behind redistribution is that cognitive resources or cognitive processing that are “in the head” can be moved outside and into the world [17]. The example of long division is an instance of redistributing partial results onto an external memory. Various types of redistribution can be considered by specifying what cognitive resources or processing are being distributed. For instance constraints on problem solving might be externalized [28], previous problem solving states could be offloaded [27], and inferences might be externalized to be performed by electronic or mechanical symbol manipulation [20].

Perceptual Substitution. Human thinking is not uniform in ease and speed: some classes [18] of operations are quick and effortless, while others are slow and laborious. Certain perceptual operations like edge detection are rapid and unnoticeable, whereas deliberate reasoning is comparatively slow and requires conscious effort. Artifacts can support cognition by transforming a task [16] in a way that allows fast operations to substitute for slower operations [10]. For instance, changing a representation can allow for *perceptual substitution* [5] in which fast perceptual operations are utilized in place of more complicated reasoning. A simple example of a bar chart enabling visual search substitution is shown in Figure 1. Answering the question “what cell contains the biggest value?” is generally easier and quicker with the bar chart (especially for large tables) since visual search routines can be employed to search for the tallest bar. The two representations encode “the same” [10] information, but certain questions are answered more easily or quickly with the bar chart representation.

SUPPORT THEORY	SIMPLE EXAMPLES
redistribution	shopping list, theorem prover
perceptual substitution	line chart instead of table
ends-means struct. reification	compile-mode in Emacs

Table 1. Several cognitive support principles.

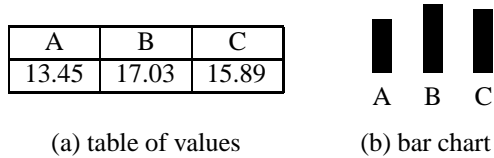


Figure 1. Perceptual substitution example.

Ends-means reification. The classic formulation of problem solving is that it consists of search in a *problem space* [15]. A problem space is a graph of the possible states of the world connected by actions that can be performed to traverse from one state to another. In this view, one way to solve problems is to repeatedly consider the current state, determine what possible actions can be performed, and select an action such that progress is made towards the goal. Such a strategy has been called “means-ends” search.

In classical works on problem solving, the problem space is represented entirely in the head. However it need not be. Parts of the problem space may be *reified* (made concrete) by being embedded in representations and external constraints [28]. In particular, it is often helpful to reify the *mapping of ends to means* [19] at any point in the problem solving process. If the solver has access to the set of means for progressing towards the goal, she can assume an *interaction strategy* [27] of repeatedly examining the display for actions that will progress towards her goals. This general form of problem solving has been called *display-based problem solving* [9]. Being able to perform display-based problem solving normally makes problem solving more facile.

For example, consider the `compile-mode` in the popular editor Emacs.¹ In `compile-mode`, an *error list* is presented in a separate window and whenever the user selects an error to correct, the list is scrolled to show only the selected error message and any following ones. As a result, the error list always shows only the immediately next operations to perform when correcting a program. The user can proceed by display-based selection of error entries. That is, the ends-means mapping remains reified in the error list window.

2.2. Greedy optimization design heuristic

Given the three principles from Table 1, it is easy to formulate a simplistic design heuristic as follows:

1. Maximize redistribution.
2. Substitute perceptual operators wherever possible.
3. Reify the ends-means mapping structures.

¹This functionality varies between different Emacs installations and configurations. The functionality described here is commonly found in later versions (e.g., version 21.1.0).

This greedy heuristic is, of course, simplistic and will fail when the costs of using a support exceeds its benefits. However it suits our use in this paper by generating as many cognitive support ideas as possible.

3. Reflection model tool description

RMTool [13] is a prototype tool for reverse engineering and software comprehension. RMTool was designed to be used in situations where an experienced systems engineer is trying to modify or evaluate a system she is unfamiliar with. Because of her experience, she has a great deal of knowledge that can be applied when understanding the system. The engineer’s particular software development tasks are not that important; it is only required that some understanding is needed of how the system is structured. The main thrust is that she has some knowledge with which to proceed, and so she is not forced to work “bottom-up” [24].

The reader is referred to the RMTool literature [13, 12, 11] for details about the tool. The overall gist of the tool can be conveyed by describing the models it encodes and operates over, and the general process of using the tool.

Low-level model (LLM) and high-level model (HLM).

It is assumed that knowledge about the software system may be decomposed into (at least) two levels of abstraction. The LLM encodes information about the software which is less abstract than the information encoded in the HLM. Mathematically speaking, both models are simply collections of binary relations (represented using directed graphs) [11]. The nodes are normally intended to represent entities and the edges relationships between them. A typical application of RMTool uses the LLM to encode a graph of the calling, data, or type relationships within a program, and the HLM to encode the system’s modules and their relationships. Typically, this type of LLM can be automatically extracted from the source code. The HLM, however, is frequently not explicitly represented, and must be reverse engineered. This, in turn, requires an understanding of the code and its modular decomposition. Loosely speaking, the LLM represents the code of the system and the HLM represents a proposed abstraction or summary of that system.

Mapping between HLM and LLM (MAP). A MAP is a mapping from nodes and edges in the HLM onto nodes and edges in the LLM. A mapping from a HLM onto one or more LLM nodes means that the HLM node is thought to summarize that portion of the LLM. For instance, a HLM node representing a file system module might map down onto LLM nodes representing functions for manipulating files (`open`, `read`, etc.)

Reflexion model (RM). A reflexion model encodes the ways in which an existing system “matches” or conforms to a HLM and MAP. The RM is a graph with the same nodes as

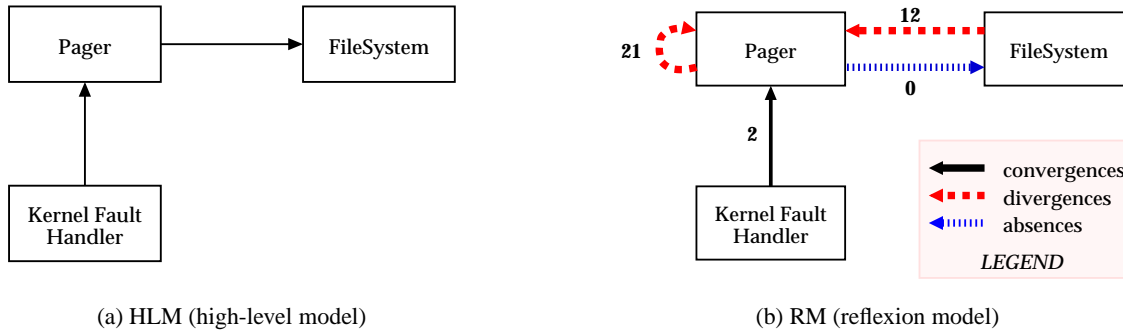


Figure 2. Illustrations of a HLM and a reflexion model.

the HLM, but with edges indicating how well the relations in the HLM are mapped by MAP onto the actual relations in the system. Three different edges encode three different aspects of this conformance. A *convergence* edge indicates that the LLM contains an edge corresponding to the one in the HLM. A *divergence* edge indicates that the HLM does not account for one or more LLM edges. An *absence* means the LLM did not contain an edge corresponding to one of the edges in the HLM. The RM can be computed automatically given the LLM, HLM, and a MAP.

Model Construction Method. The typical aim is for the engineer to incrementally construct a HLM and MAP such that they collectively summarize the relationships in the LLM. In other words, the engineer synthesizes an abstract representation of the system structure and relationships. The procedure is typically (more or less) as follows: (1) A LLM is automatically extracted, (2) an initial, partial HLM is created according to the engineer’s expectations of the system structure, (3) an initial MAP is created which indicates how the engineer expects the HLM to be implemented in the system, (4) a RM is computed to indicate how well the engineer’s HLM and MAP match the actual system, (5) the engineer investigates the RM to determine how accurate her models are, and (6) the engineer iteratively refines the models as needed (i.e., she iterates through steps 2, 3, 4, and 5 as necessary). Investigating the evidence presented in the RM will, in our context, involve examining the source code or other documentation in order to determine the relevance or importance of each arc in the RM.

An illustration of a HLM and a corresponding RM is given in Figure 2 (see the RMTTool [11, 13] literature for a more detailed discussion). In the present example, the relationships are presumed to be calling relationships between modules. For this example, assume the MAP for the HLM in Figure 2(a) maps the `Pager` node to all of the functions in the file `pager.c`, and that it maps the `FileSystem` node to all functions defined in the file `filesall.c`. Figure 2(b) shows two convergences between the `Pager` and

`Kernel Fault Handler`. In this example, this implies that two expected function calls occur between the functions of `filesall.c` and `pager.c`. It is up to the engineer to examine the RM (and the source) to determine whether or not the HLM and MAP are sufficient for her purposes. Generally speaking, the “goal” is to create a RM with only convergence arcs. Thus the engineer might investigate why there are no calls from `pager.c` to `filesall.c` when some were expected, and why there were twelve from `filesall.c` to `pager.c` when none were expected.

The overall process of using RMTTool is one of (hopefully) convergent evolution. An initial HLM and MAP are tentatively defined, and then iteratively refined until they are found to abstract the actual LLM structure to a satisfactory degree. Investigation of the RM drives the refinement. Since the initial models represent a “guess” as to the structure of the system, the length of the iteration cycle is related to how good this initial guess is. In other words, one of the best reasons for using RMTTool is that there are gaps and inaccuracies in the user’s knowledge, or uncertainty in its accuracy. In the end, the engineer gains both an understanding of the system, and an increased level of confidence that her interpretation is valid. So far, reported experience with RMTTool indicates that this general process is relatively simple, quick, and successful [1, 2].

4. Theoretical reconstruction of rationales

The RMTTool literature nicely conveys many of the advantages of the RMTTool approach. Nonetheless, improvements to this description may be possible. A particular concern is that essentially no references are made to cognitive assistance principles even though the primary goal of the tool is to aid in software comprehension—a task that is obviously laden with psychological implications. Should it not be the case that a tool for aiding software comprehension will be successful, at least in part, as a result of its effects on cognitive processes? Here one interpretation of the cognitive

support principles underlying RMTool is generated. Specifically, we reconstruct psychological rationales underlying RMTool from the viewpoint of one of the cognitive support theories of Section 2.1.

The reconstruction proceeds by applying the cognitive redistribution theory from Section 2.1 to a model of the cognitive task of software comprehension. In particular, we utilize Brooks' model [2] of software comprehension to consider how RMTool redistributes the comprehension task. In effect, we will be importing an existing high-level cognitive model instead of performing a situation-specific *cognitive task analysis* (see e.g., Chipman *et al.* [6]). Cognitive task analysis is often performed during requirements analysis to determine what functionality and interface features need to be built. In this context, the role of a cognitive task analysis is to determine the knowledge, mental states, and reasoning needed to solve a problem. Once these are modeled, then they can be examined for ways of applying the support theories to them to re-engineer them.

A necessary requirement for analyzing the results of the cognitive task analysis in this fashion is that the resulting model must be *as free from tool-specific issues as is possible*. This is not the norm for typical task analyses in HCI. The reason for requiring tool-independence is simple: the analysis otherwise generates a model of the task in the context of specific tools. This makes it difficult to break the analysis free of the device-specific aspects of the task. For instance, in this analysis we are not interested in the task of comprehending software specifically with the aid of `vi` and `grep`. We therefore wish to avoid interface-specific details (e.g., steps to load a file). It is in a sense fortuitous, therefore, that practically all cognitive models in the field (including Brooks') are effectively models of purely "disembodied" cognition, and so do not consider device-level tasks lest they be rendered tool- or context- specific.

Although a preexisting, tool-independent comprehension model like Brooks' does not lead directly to design ideas, it can be employed as a starting point [26] for further analysis. Specifically, once the cognitive task is modeled, a cognitive redistribution analysis can be applied to determine how cognition can be spread out onto tools. This analysis is done in two steps. First Brooks' model will be used as a generalized cognitive task analysis, and then ways of distributing the cognitive processing onto tools are explored.

4.1. Brooks' "top-down" comprehension model

Brooks [2] proposed a model of expert comprehension of software. The central argument behind this model is that in most circumstances expert software developers will use their extensive knowledge to drive their comprehension processes. Such a knowledge-based process is precisely the context expected for effective use of RMTool. Although

MODEL	MODEL ELEMENTS / DESCRIPTION	
domain task	reconstruct hierarchical mapping	
representation (mental model)	<i>model</i>	hypothesis hierarchy
	<i>bindings</i>	between models
	<i>evidence</i>	for or against hypotheses
process (hypothesis refinement & verification)	<i>retrieve</i>	hypothesized model
	<i>verify</i>	object/relationship and binding
	<i>search</i>	for evidence of binding
	<i>recognize</i>	contradictions to model
	<i>backtrack</i>	to update model

Table 2. Summary of Brooks' [2] model.

Brooks' original works studied modestly sized programs, recent evidence suggests that the basic points generalize to large-scale system comprehension [24].

Brooks' model contains three key features: (1) a domain task analysis, (2) a suggestion as to the mental representations being used during comprehension, and (3) an analysis of comprehension processes. A summary of this analysis is presented in Table 2. There are other significant aspects of Brooks model, but they are not used in the following.

Domain task analysis. Brooks argued that comprehending a program amounts to generating (i.e., *reconstructing*) a hierarchical mapping of models [1, 2]. He called this "domain bridging". The models start at the domain level and proceed through various intermediate levels such as mathematical methods or system structure models. Each model consists of, in part, a set of objects and relations; the mapping between models consists of binding higher-level objects (or relations) to lower-level objects (or relations). There is nothing particularly unusual in this hierarchical way of modeling software systems, as it resembles other previously proposed hierarchical models of software systems. Comprehension of a system is posed as a problem of generating an internal representation of this hierarchical mapping, that is, a *mental model*. Brooks argued that for specific tasks, the required model will be partial, consisting of a partial mapping of relevant aspects.

Mental representation model. At any point in the comprehension process, it is assumed that the mental model of the system is a tentative collection of *hypotheses*. The models at any level are considered to be hypotheses about the system (e.g., "this is a standard Unix virtual memory system"). Bindings to lower level models start out as sub-hypotheses (e.g., "The file system must be implemented in these functions here..."). It is presumed that in order to mentally process the hypothesis refinement, the hypothe-

ses, mappings, and evidence must be mentally represented (at least momentarily).

Processing model. Brooks’ model is termed “top-down” because the hierarchical mapping is built by starting at the high-level domain models and working “downwards” to low-level code models. This aligns with so-called “top-down” development methods, which propose that programs are to be hierarchically refined in an analogous manner. Brooks argues that comprehenders will develop high level hypotheses about the meaning and structure of the system being studied (e.g., by considering the program name). These set up the gross hypotheses which are hierarchically refined until bindings are considered verified. Verification of a hypothesis is performed by searching for confirmations or disconfirmations. Sometimes this search fails, or encounters contradictory evidence. This causes backtracking to occur, resulting in refinements to the hypothesized bindings, or to higher level hypotheses. Processing occurs until the full (or partial) hierarchical model is constructed and confirmed to the degree required. Cognitive tasks involved therefore include: *retrieving* relevant structures from expert memory, *verifying* a binding, *searching* for evidence of a binding, *recognizing* conditions that contradict the current hypotheses, and *backtracking* by refining the model. In Brooks’ model, backtracking is initiated when a search for evidence fails or happens to turn up contradictory evidence.

4.2. Redistributing Brooks’ model

Brooks’ model is essentially a disembodied, unassisted model of comprehension. The model is presented as being applicable regardless of the tools available to the comprehender. Even so, if suitable care is taken when interpreting the model, it is highly compatible with the projected applications of RMTTool, and can be utilized to understand the cognitive support that RMTTool provides. First, the terminological differences between Brooks’ account and the RMTTool account must be reconciled. This is done by reinterpreting RMTTool concepts in cognitivist terms matching Brooks’ concepts. Then RMTTool can be analyzed to determine how RMTTool serves to redistribute cognitive resources and processing.

The first step is relatively straightforward. The HLM and MAP are effectively two-level *hypotheses* from Brooks’ model: the HLM is intended to represent the (hypothesized) objects and relationships at a higher abstraction level, and the MAP is intended to represent the (hypothesized) way that these higher level abstractions are bound to the code level. Thus the construction of an acceptable HLM and MAP is RMTTool’s version of Brooks’ concept of domain bridging. The RM corresponds to known evidence about the three conditions for refining the hypotheses. Specifi-

TERMINOLOGY	
RMTTool	BROOKS’ MODEL
HLM	models (hypotheses)
MAP	bindings (hypothesized mapping)
RM	evidence (hypothesis verification outcomes) absences — failed search convergences — evidence for found divergences — contradiction / evidence against
reflexion	search for evidence for/against hypotheses

Table 3. Mapping of RMTTool terminology.

cally, a convergence corresponds to a case where a hypothesis binding would seem to succeed, an absence corresponds to a case where a search for evidence would fail, and a divergence corresponds to a case where contradictory evidence is encountered in the verification search. As a result, the processing performed to construct the RM corresponds to the processing to search for verification (reading through code, following relationships, etc.). This mapping of terminology is shown in Table 3.

Knowing this mapping, RMTTool can be viewed as a redistributor of cognition. This is accomplished by examining how cognitive resources and processing identified in Brooks’ unaided model (Table 2) are externalized. The main insights are (1) that RMTTool redistributes the hypotheses so that they are externalized and need not be kept internally, and (2) that RMTTool redistributes part of the hypothesis verification search so that it need not be performed by the engineer.

Notice that the cognitive processing is only partially externalized, and still relies upon the engineer to perform part of it. In particular, the evidence for verifying a hypothesis cannot be fully evaluated externally, and the engineer must still go through the RM to determine whether the evidence is relevant, and how it should impact on the HLM and MAP.

Thus with RMTTool it can be seen that the engineer and tool form a *joint cognitive system* [7] in which processors coordinate to incrementally refine a shared model. This can be visualized as if two processors (human and computer) share a memory, and then take turns updating it. Such an architecture is reminiscent of agent or blackboard architectures [14]. An illustration is shown in Figure 3. In the figure, the mental model is externalized onto a *shared blackboard*. Agents which update the model are shown as labelled ovals. These implement the process model of Table 2.

Given this analysis, the redistributions enabled by RMTTool are as follows:

- 1. Redistribution: hypotheses.** An unsupported engi-

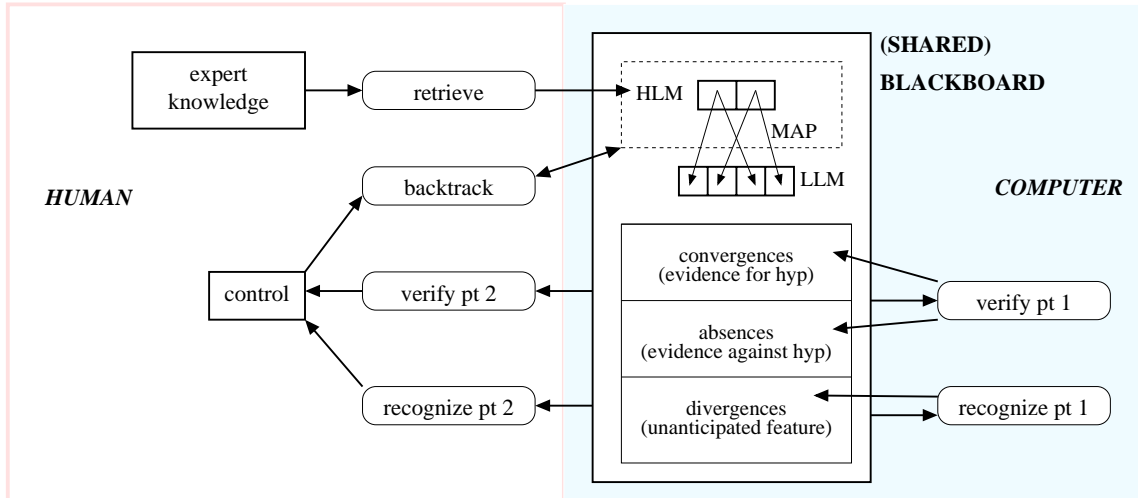


Figure 3. Shared blackboard visualization of RMTTool processing.

neer would need to maintain their current understanding of the system in their head. That is, they would need to maintain the abstract model and its binding to code features. The HLM and MAP are seen to offload these onto an external memory.

2. **Redistribution: hypothesis verification.** Hypothesis verification is partially redistributed. This involves redistributing the *search* and *verify* cognitive tasks. Computing the reflexion model uses the HLM and MAP to effectively *search* for possible evidence that may confirm or disconfirm the hypotheses. The arcs in the resulting RM constitute potential sources of evidence for the accuracy of the sub-hypotheses. This results in only a partial redistribution of *verification* because these arcs must be investigated by the engineer to determine what they truly imply. However even this partial redistribution saves a great deal of work because otherwise these would have to be individually navigated to and examined. The processing transforms the tasks the engineer must perform [16] and in doing so offloads cognitive processing.

Using the above analysis, the core rationale for RMTTool is the redistribution of the hypotheses and their processing. Externalizing the hypotheses can reduce memory load and increase the complexity of the hypotheses explored. Although these could have been externalized onto, say, scrap paper, externalizing them onto a computer makes it possible to electronically process them. The external processing means that internal cognitive loads are reduced. Because the systematicity of hypothesis exploration depends upon diligence and a capacity to remember pending goals, the wholesale processing of the hypotheses by RMTTool suggests

that RMTTool has the potential to make hypothesis evaluation more systematic and thorough.

Two important aspects of the above account deserve to be highlighted. First, the analysis tracks and accounts for the cognitive work done by RMTTool. The tool reduces the cognitive burdens of the user, but these do not disappear. Instead, the tool picks them up. Thus cognitive work is viewed as being conserved when using a tool, much as physical work is conserved when using a lever. Second, by accounting for the cognitive work that was offloaded, it is possible to properly perform a tradeoff analysis. New task burdens are introduced, of course, by using the tool: externalizing the interpretation, invoking reflexion analysis, etc. These are *overheads* in the form of device tasks and human-computer cooperation efforts. It is important to recognize that these overheads are necessary if the support is to be provided, and that they can be tolerated only if the value of the cognitive support they provide exceeds their cost.

This overall explanation of cognitive support in RMTTool aligns nicely with the RMTTool literature. The main differences between the two accounts are their vocabulary and the knowledge used to construct them. Prior accounts are steeped in the particulars of the tool, which arguably hinders the appreciation of generalizable design principles. In contrast, the analysis here is framed in cognitivist terms, and the generalizable principle for design (cognitive redistribution) was separated from the particulars of the tool or task (cognitive task analysis). The prior accounts were made possible through practical experience with using the tool. The analysis here stems instead entirely from an application of a pre-existing cognitive task analysis and theory cognitive support. The two evaluations are substantially different in character and spring from entirely independent sources, yet

they are extremely compatible.

Thus it appears possible to more systematically articulate psychological rationales for explaining the value of a tool's features. The rationales can be made in terms that relate to general cognitive support principles, and can be grounded in existing models of cognition.

5. Theory-based design analysis

One of the grand promises for studying developer cognition is that an improved understanding of how developers think would lead to improved designs for tools. Unfortunately such models and theories are rather infamously difficult to convert to usable design knowledge [3]. This difficulty led Singer *et al.* [21] to ask

... how does knowing that programmers will sometimes use a top-down strategy to understand code ... inform tool design? It doesn't tell us what kind of tool to build... ([21], pg. 210)

We wholeheartedly concur. What is needed *in addition* is a theory of cognitive support. Without such a support theory, the comprehension model itself says little to the tool designer. As the previous section hints, such theories can be invoked to generate arguments saying what parts cognition might be beneficially redistributed. As it happens, this was shown using a top-down comprehension model—exactly the type singled out by Singer *et al.* as being relatively unhelpful during design.

Since cognitive support theories appear to be able to generate tool design suggestions from cognitive task analyses, they may be extremely useful during formative design. This section further explores this possibility by expanding the analysis of RMTTool and comparing it to the design iteration experienced published for RMTTool. Early prototypes of RMTTool lacked some of the features that were added to later versions in response to user feedback [12, 11]. The question we ask here is: can design theories be used to help anticipate some of the requirements for tools so that the necessary features do not have to be discovered after the tools are delivered to the users? It is impossible to fully answer this question with a retrospective analysis of prior design histories: hindsight, as they say, is 20:20. But the results can be suggestive. RMTTool's published design history provides a good case in point.

Using the “greedy” cognitive support design heuristic from Section 2.2, we can expand our current analysis of how Brooks' model may be reengineered. The key is to focus on the partial redistribution of the evidence search and evaluation process. Even though a great deal of the evidence evaluation is automated by RMTTool, the evidence is only partially checked: the user needs to sort through the RM and determine how to refine the HLM and MAP appropriately.

The greedy heuristic can thus be applied to the evidence evaluation task (“verify pt 2” in Figure 3). This will generate suggestions for creating further redistributions, perceptual substitutions, or ends-means reifications.

To evaluate the evidence, the engineer makes a series of decisions that the computer cannot. The computer relies on the human to be able to distinguish irrelevant and important indicators of evidence, and to determine if and how the HLM and MAP should be updated. The task presented to the user is to go through each piece of evidence, determine its relevance, and update the HLM and MAP as necessary. At various points the user may backtrack by updating the HLM and MAP. Updates would necessitate re-calculation of the RM. After recalculation, some of it might change, while other parts might not. The engineer must take these changes into account when processing the evidence.

Invoking the greedy cognitive support design heuristic yields at least four possibilities, as follows:

1. **Redistribution: progress state.** It will be a cognitive burden for the engineer to mentally keep track of her decisions about the salience of evidence. In other words, the engineer must track *progress*. It should often be helpful to offload this information. One type of data that might be beneficially offloaded are the decisions to ignore particular LLM features. For instance, the engineer may realize that the LLM contains false or unimportant dependencies [13]. Another type of data that might be offloaded are decisions to remove from consideration those features that have already been understood as being important and accounted for in the HLM and MAP. For instance, the edges from the `Kernel Fault Handler` to the `Pager` might be, at some point, investigated to the engineer's satisfaction. To avoid unnecessarily revisiting these edges, she will have to remember this fact, and then take it into account when progressing. This progress information may also be offloaded.
2. **Ends-means mapping reification: evidence selection.** The engineer is responsible for going through the evidence. This is the engineer's “ends”. At any point in the process there exists a pool of unvisited evidence to examine: the pool defined by the RM minus the edges already visited (i.e., those already ignored or accounted for). To progress through the task, the engineer must iteratively select the next bit of evidence to examine. This is the engineer's “means”. If the external display can be made to show the unvisited evidence pool, then the engineer can resort to display-based methods to progress through the task. For instance, the pool of unvisited nodes might be represented in an on-screen priority queue (to-do list) or by annotating the display of the RM in a way that indi-

cates which evidence remains unvisited.

3. **Perceptual substitution: visual search.** If the RM display is annotated as suggested above, then the engineer can use it as a resource from which to select actions. However she may still need to search the display for unvisited edges. If the appropriate visual cues are used, efficient perceptual search can replace effortful, conscious search. For instance, unvisited edges can be marked by colour that can be perceptually searched for (much as unvisited links in hypertext browsers are).
4. **Redistribution: dependent decision rollback.** Verification of evidence will normally be interleaved with refinements to the HLM and MAP. Each refinement may effectively invalidate prior decisions, especially decisions to ignore bits of evidence. Determining the decisions to unroll requires cognitive processing; this processing can be offloaded if rules are formalized for determining invalidated decisions.

The updated [12, 11] RMTTool contains some features that implement the above suggestions. In particular, the updated version added *tagging* and *annotation* features. Tagging features allow the user to “tag” specific arcs in the RM to indicate that they are to be considered temporarily irrelevant. The visualization engine uses these tags to elide the ignored evidence arcs. If the HLM and MAP are changed such that the relevance of that evidence might change, these tags are undone. These features implement the supports numbered 1 and 4. The annotation mechanism allows the user to externalize whether and how an arc is resolved. The visualization engine subsequently indicates this resolution status visually (by displaying the fraction of evidence resolved for any given arc on the diagram). This way of representing resolution status is unlikely to enable visual search for the next goal to examine (the engineer must interpret [17] the numbers). Nonetheless, it still enables display-based processing because the user can search for unfinished arcs when considering what to do next. Collectively these features implement the support numbered 2 above.

In sum, the tagging and annotation facilities effectively implement all supports suggested above except the substitution with visual search (#3). The experiences reported indicate that these features are significant aspects of the overall RMTTool approach. The important point to note is that a theory-based analysis could predict several tool improvements which became apparent after user studies.

6. Related work

Attempts are occasionally made to rationalize or justify the designs of comprehension tools using comprehension models or other theories. One common use of such models is to argue the design is consistent with the model. For

instance, Storey *et al.* [22] argued that tools must facilitate switching behaviour noted in cognitive models. In a different vein, von Mayrhauser and Lang [23] performed a detailed evaluation of a tool based on an *information needs* analysis constructed from a model of software comprehension. Tool feature rationalizations in this tradition typically rely on a (tacit) design heuristic of automating functionality or reducing task complexity. Our analysis complements these prior approaches by explicitly using theories to rationalize the value of the tool in terms of the cognitive support provided.

Our approach to linking tool features to theory-based rationalizations was heavily influenced by the *psychological claims analysis* work of Carroll *et al.* [4]. A key suggestion in claims analysis is that the psychological claims of tools need to be made explicit if design is to be well grounded in theory. Another similarity is that the claims analysis method of Carroll *et al.* [4] uses a high-level model of HCI to structure the generation of possible claims. Our application of the greedy design heuristic to cognitive task analyses follows a similar route. A key difference between their claims work and our work is that we concentrate on theory-driven rationalization generation whereas in the claims analysis, claims are first created and only afterwards are suitable theories sought out for justifying the claims.

7. Conclusions and implications

This paper outlines a general method for constructing psychological rationales for complicated software comprehension tools. These rationalize the value of the tool in terms of the cognitive support they provide. We showed that key design ideas for a tool of current research interest (RMTTool) can be reconstructed using pre-existing theories. To perform this analysis, no new theories or models needed to be created. This leads us to suspect that the currently available theories have been insufficiently plumbed for use in tools research. Obviously, it would be helpful to have a more complete and well-organized exposition of different theories of cognitive support. We have made a start on that project [25], but that is outside the scope of this paper.

It is important to note that, as a scientific explanation of cognitive support, the preceding analysis may rightly be viewed with suspicion. The skeptic may wonder, for instance, about the validity of Brooks’ model, or of the cognitive support theories applied. Proper validation of the rationalizations might require demanding experimentation. However the point is that such experimentation is possible only after one has articulated rationales to validate. This paper describes a method for building cognitive support-related ones.

Furthermore, as a design analysis, veridicality plays second fiddle to utility. The preceding analysis could be valu-

able to designers merely by providing an explicit explanation to reflect upon [4]. Moreover, the essential hope of using an existing theory is that there is a good chance that the theory-based analysis will provide a better explanation than raw intuition alone would generate. This is especially important for relatively novice designers, or designers not trained in cognitive science.

Ultimately, the analysis technique presented here links the world of cognitive theory to the world of tool design. Currently, these two worlds are tenuously connected. This need not be so. Existing cognitive theories can be leveraged to build cognitive support explanations. Such explanations can be tested; they may be used to guide design. In short, the ability to systematically generate design rationales from theories is an important step towards grounding tools research in the existing science base.

References

- [1] R. E. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 196–201, 1978.
- [2] R. E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.
- [3] J. M. Carroll, editor. *Designing Interaction: Psychology at the Human-Computer Interface*. Cambridge University Press, 1991.
- [4] J. M. Carroll and M. B. Rosson. Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Transactions on Information Systems*, 10(2):181–212, 1992.
- [5] S. M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, 1991.
- [6] S. F. Chipman, J. M. Schraagen, and V. L. Shalin. Introduction to cognitive task analysis. In J. M. Schraagen, S. F. Chipman, and V. L. Shalin, editors, *Cognitive Task Analysis*, chapter 1, pages 3–23. Lawrence Erlbaum, 2000.
- [7] E. L. Hutchins. *Cognition in the Wild*. MIT Press, 1995.
- [8] A. Kirlik. Requirements for psychological models to support design: Toward ecological task analysis. In J. Flach, P. Hancock, J. Caird, and K. J. Vicente, editors, *Global Perspectives on the Ecology of Human-Machine Systems*, chapter 4, pages 68–120. Lawrence Erlbaum Associates, 1995.
- [9] J. H. Larkin. Display-based problem solving. In D. Klahr and K. Kotovsky, editors, *Complex Information Processing: The Impact of Herbert A. Simon*, chapter 12, pages 319–341. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
- [10] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–99, 1987.
- [11] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, Dept. of Computer Science and Engineering, University of Washington, 1996.
- [12] G. C. Murphy, D. Notkin, and K. J. Sullivan. Extending and managing software reflexion models. Technical Report TR-97-15, University of British Columbia, Department of Computer Science, Sept. 1997.
- [13] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.
- [14] A. Newell. Some problems of the basic organization in problem-solving programs. In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Proceedings of the Second Conference on Self-Organizing Systems*, pages 393–423, New York, 1962. Spartan Books.
- [15] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice-Hall, Inc., 1972.
- [16] D. A. Norman. Cognitive artifacts. In Carroll [3], chapter 2, pages 17–38.
- [17] D. A. Norman. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Addison-Wesley, Reading, Massachusetts, 1993.
- [18] J. Rasmussen. Skills, rules, knowledge: Signals, signs, and symbols and other distinctions in human performance models. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):257–267, 1983.
- [19] J. Rasmussen, A. M. Pejtersen, and L. P. Goodstein. *Cognitive Systems Engineering*. John Wiley & Sons, Inc., New York, NY, 1994.
- [20] M. Scaife and Y. Rogers. External cognition: How do graphical representations work? *International Journal of Human-Computer Studies*, 45(2):185–213, 1996.
- [21] J. A. Singer, T. C. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the Seventh Centre for Advanced Studies Conference (CASCON'97)*, pages 209–223, Nov. 1997.
- [22] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2):183–207, Mar. 2000.
- [23] A. von Mayrhauser and S. Lang. Evaluating software maintenance support tools for their support of program comprehension. In *Proceedings of the 1998 IEEE Aerospace Conference*, pages 173–187, 1998.
- [24] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug. 1995.
- [25] A. Walenstein. *Cognitive Support in Software Engineering Environments: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, May 2002 (to appear).
- [26] A. E. Walenstein. Developing the designer's toolkit with software comprehension models. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 310–313. IEEE, 1998.
- [27] P. C. Wright, R. E. Fields, and M. D. Harrison. Analyzing human-computer interaction as distributed cognition: The resources model. *Human Computer Interaction*, 15(1):1–41, Mar. 2000.
- [28] J. Zhang and D. A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122, 1994.