Cognitive Models of Program Comprehension

Tim Tiemens

Software Engineering Research Center

December 8, 1989

ABSTRACT

This paper describes some cognitive models in program comprehension. The goal is to use this knowledge about cognitive models to produce a tool (the Cognitive Support Tool, CST) which can reduce the amount of effort needed to understand a program. The models discussed here were derived from both a human perspective and from a source code perspective. After reviewing these models, a synthesis section suggests some implications of the information presented. Finally, a section describing a sample interactive sessions with CST is presented.

1. Introduction

Program comprehension is of vital importance to the process of software maintenance. Before a software system can be correctly maintained, it first must be understood. Yet even though maintenance work consists of 50% of the total effort involved in software development, [12] and understanding consists of approximately 50% of the total maintenance effort, [4] few tools are available that directly support the task of understanding. It is the purpose of this research to develop such a tool.

This tool, which will be part of a set of maintenance support tools, is known as the Comprehension Support Tool (CST). The elements of this tool are derived from models of human program comprehension in order to aid the maintainer in the understanding process.

This paper deals with two basic concepts: models of human program comprehension (by Brooks and by Soloway), and current work being done in the representation of program understanding knowledge (such as Programmer's Apprentice and Cognitive Program Understander.) In this paper, each model or system is first described, then the implications the model or system carries toward the capabilities of the CST are discussed. This paper's summary is a synthesis section which extracts global implications from the combination of the models and systems discussed, deriving requirements not explicit from the viewpoint of a single element.

2. Cognitive Models of Comprehension

The two models discussed in this section both view people as knowledge-based understanders. These two models are Ruven Brooks' model and Elliot Soloway's model. At the highest level, the basic structure for both cognitive models consists of four components:

1) The target system to be comprehended. This consists of all the information sources available to the understander, such as source code and supporting documentation.
2) The knowledge base which encodes the understander's experience and background knowledge used in the comprehension task. This knowledge base is either internal (i.e. in the understander's mind) or external (e.g. on paper, or in a Comprehension Support Tool.)
3) The mental model which encodes the current state of understanding of the target program. It is constantly being updated in the course of comprehension.

4) An assimilation process which interacts with the other three components to update the current state of understanding.

The goal of the CST is to augment the user in the comprehension process. The anticipated method to do this is to automate parts of the recording of knowledge (item 2), to aid in recording the current state of understanding (item 3), and to guide the the assimilation process (item 4).

To achieve any degree of automation, it is first necessary to create a model of what understanding is all about. The following two models of human comprehension shed some light on both the assimilation process and an informal description of the current state of understanding. (Section 3 of this paper, dealing with other models of program comprehension, sheds some light on possible structures of the knowledge base.)

The differences in the two models of human comprehension are in the terminology describing the contents of the knowledge base and (more importantly) in the approach that each model adopts for the assimilation process. While the models contain mechanisms that utilize both top-down and bottom-up approaches to the comprehension process, Brooks' model emphasizes the top-down approach, while Soloway's model is more bottom-up in its approach.

2.1. Brooks' Model of Program Comprehension

Ruven Brooks' model deals with the comprehension of completed programs. [2, 3] It has its basis in area outside of computer science, such as thermodynamics problem solving [1], physics problem solving [6], and chess. [5] The model was initially created to explain four major sources of variation in the act of program comprehension. These four sources are:

1) The functionality of the program being understood. Why do programs that perform different computations vary in comprehensibility?
2) The differences in the program text. Why do programs that are written in different languages differ in comprehensibility, even though the same calculation is performed in each?
3) The motivation the understander has in comprehending the program. Why does the comprehension process vary depending on whether the motivation is to debug the program versus enhancing the program?
4) Individual differences between understanders' abilities in comprehending a program's purpose. Why does one understander find a program easier to comprehend than does another?

To account for these four areas of variation, Brooks created a model based on three main ideas. These three ideas are:

1) The Programming process is the construction of mappings from a *task* domain, through one or more *intermediate* domains, to the *programming* domain.
2) The Comprehension process of that program is the reconstruction of all or part of those mappings.
3) The Reconstruction process is expectation-driven by the creation, confirmation and refinement of hypotheses. These hypotheses describe the various domains, and the relationships between them. ('Expectation driven' is another way to say 'top-down', i.e. an expectation is just a tentative, high level view of what the program actually accomplishes.)

The understander's knowledge about the three domain types listed (task, intermediate and programming) are grouped under the title *domain knowledge*. Knowledge about a specific domain consists of information about the objects and operations within that domain, and well as information between objects and operations of this domain to objects and operations to (conceptually) nearby domains.

The comprehension process is one of (re-)creating the set of mappings which were used to develop the program. This set is best viewed as tree-like, with global relations between domains at the top or root, and with subsidiary, refining relationships forming levels of children. The relationships are known as hypotheses. An example of a high level hypothesis is: "This program produces invoices." This hypothesis maps the task domain (invoicing), to the programming domain (the program itself).

At the very start of the understanding process, the understander forms a *primary hypothesis.* This hypothesis forms the root of the mapping tree, and is a global, gross level description of what the understander thinks the program does. It is formed as soon as the understander obtains any information at all about the purpose or the program. (For example, hearing the program name usually provides enough information to form the primary hypothesis.)

This primary hypothesis then produces a cascade of subsidiary hypotheses. This cascading is carried out based on the understander's domain knowledge, which has been built from experience in the task domain to experience in the programming domain. Also, this cascading is done depth-first, with the decision of which child to follow based on the understander's motivation for comprehending this program, or possibly just on the understander's personal preferences.

This cascading continues until it produces a hypothesis which is specific enough that the understander can verify the hypothesis against the program code and/or supporting documentation. Specifically, the understander begins the verification of a hypothesis when the hypothesis deals with operations which can be associated with visible details which can be found in the program code. The term *Beacons* describes those visible details which show the presence of a particular structure or operation. When the understander feels a hypothesis is at a level for which beacons can be located, the program code is scanned for beacons. As an example, a typical beacon for the hypothesis 'sort' could be a pair of loops, inside which there is a section of code where the values of two elements were compared and interchanged.

Beacons are important because they are the first link between the top-down hypotheses and the actual program text. Their existence 1has been verified by experiments done by Susan Wiedenbeck. [20] The method used in these experiments was memorization and recall, under the assumption that a high recall of some part of the program, after brief study, indicates that the recalled part is important in the understanding of the program. The tested hypothesis was that experts could locate these parts more efficiently than novices. This effect was confirmed, as well as the effect of experts recalling beacons lines more successfully than non-beacon lines (77% beacon versus 47% non-beacon). This result was opposite for the novices, they recalled non-beacon lines better (13% beacon versus 30% non-beacon). This effect is explained by two factors: first, syntactic markers, such as "begin" and "end" were placed one per line, and were considered non-beacons. Second, since the novices seemed to recall in linear order - i.e. the lines at the beginning of the program had the highest recall, and many of the beacon lines were fairly deeply embedded within the program, the net result was an increase, percentage-wise, of non-beacon lines to beacon lines.

However, even with these explanations, no claim is made that novices use beacons in program comprehension, only that experts do. The statistical analysis performed revealed significance in only two combinations: 1) novice-beacon versus expert-beacon (experts recall beacons better than novices), and 2) expert-beacon versus expert-non-beacons (experts recall beacons better than they recall non-beacons). And when experts do recognize beacons, they sometimes retain only generalized knowledge - for example, experts tend to recognize a class of beacons (a 'sort' beacon), without identifying specifically which kind of beacon they had viewed (a 'shellsort' beacon).

The strategy used in discriminating beacons varies between understanders, but in general it is a broad search. When scanning the program code, the understander is searching for two classes of beacons. The first class is obvious: it is the set of beacons dealing with the current hypothesis. The second class is more interesting: it is the set of unusually 'strong' beacons, even those beacons which do not relate to the current hypothesis. The introduction of 'strong' beacons (as opposed to 'normal' or 'weak' beacons) brings to light a key feature of beacons. A beacon is associated with a structure or operation with some degree of confidence or probability. In other words, the same basic operation for example, "swapping values," can implicate multiple beacons. One possibility is that it indicates a 'sort' beacon. Another possibility is that it is an 'undo' beacon. And so on.

The broad collection strategy helps reduce the re-read time when other beacons are needed for verification of hypotheses process. It also influences the creation of subsidiary hypotheses - e.g. if a 'sort' beacon is located in the code, but is not needed to verify the current hypothesis, the understander has a strong bias to later create a hypothesis which involves a sorting process.

After a hypothesis has been verified to the satisfaction of the understander, then actual *Binding* between the hypothesis and the program code occurs. This binding involves marking individual lines of code as "used", i.e. the purpose behind those lines has been explained. When binding is complete, the understander then returns to some previously created hypothesis which has not yet been verified and/or bound to program code, and the process begins again.

If the understander has created the correct primary hypothesis, as well as all the subsidiary hypotheses, and is able to bind the program code completely and uniquely to these hypotheses, the understander is said to have comprehended the program completely.

If, however, the understander has unbound hypotheses or unbound program text, the understander tries to increase the level of comprehension by either

1) Adopting different hypotheses, or
2) Altering the bindings of the program text.

Since understanders can rarely generate large numbers of alternative hypotheses which have the same behavior, most likely the understander simply repeatedly attempts to interpret and bind program code to existing hypotheses, rather than using known features/beacons of the program to adopt different hypotheses (e.g., if the behavior is "The numbers are printed in ascending order", then a strong hypothesis is "There is a sort taking place." This hypothesis is unlikely to be abandoned, even if no sort beacon is immediately located.)

As stated earlier, this theory attempts to explain four sources of variation in the act of comprehension. They are the role of:

1) The functionality of the program. The intrinsic difficulty of the task domain (e.g. nuclear physics versus invoicing) causes the higher level hypotheses to be more complex and can result in the use of a greater number of intermediate domains between the task domain and the programming domain. Also, it should be noted that in most cases the documentation explaining these intermediate domains is rarer than documentation covering the original program task (which can itself be a rare item.)
2) The program text itself. The code and supporting documentation affect the ease that beacons are located, and the ease of binding the actual source code to hypotheses (e.g. it is easier to bind a hypothesis to contiguous statements than to bind a hypothesis to statements scattered about the code.) Also, features of the language effect the confirmation of beacons. (e.g. the hypothesis that "The variable 'PI' contains 3.1415" is easy to confirm in pascal if PI is defined using a const statement. It is not so easy in COBOL, because even when 3.1415 is put into 'PI' at the beginning of the program, one must confirm that 'PI' is never changed.)
3) The understander's task. The motivation behind the understander affect the strategy used in creating and following subsidiary hypotheses. (e.g. debugging an output format error causes entire subsidiary hypotheses, such as those dealing with input or computation, to remain unexplored because of the lack of relevance to the task at hand.)
4) The understander's individual abilities. The knowledge the understander has about domains affects the process on all levels. Specifically, task domain knowledge affects the quality of the primary and higher-level hypotheses. Programming domain knowledge affects the lower level bindings and beacon location process.

Discussion of Brooks' Model

Since this model emphasizes the mappings between domains, the major implication is that support for creating domains and their interrelations must be provided by CST. It is unclear to what level of formality a domain needs to be defined before it can be of any use (e.g. "English Text" could be a viable level of formalization for the higher domains.)

Also, since all levels of domain eventually meet with the 'actual code' domain level, it is justified that a larger effort be made in supporting this bottom level domain and the relationships it has to the level above it. One way to do this is to make the binding stage more explicit by showing the

relationships between an intermediate hypothesis and the code to which it has been bound. This could also involve some sort of status to indicate if a given section of code has been bound at all, or possibly even that it has been bound twice. Another helpful feature is a viewing capability to focus on the particular lines of code which have been bound to a single beacon or hypothesis.

Still another feature relevant to these low level domains is the idea of beacons. Since it is explicit in the model that beacons are located during any scan of the text, support for the detection, binding and registration of beacons is a need. This involves both a 'current search' registration (so that a needed beacon can be found or not found during a particular scan), and, importantly, a 'noncurrent search' or 'agenda' registration, so that detected but unused beacons can be bound and then forgotten until needed later.

Since Brooks' model takes into account mistakes made on the part of the understander and the actions taken to correct those mistakes (the understander either adopts different hypotheses or alters the bindings of existing hypotheses), a *backtracking* ability should be supported. This requires a flexible relating/binding building method, combined with some version history, so that when the user needs to backtrack, an explicit record of the user's actions is available. Thus it is possible to undo bindings created by those actions.

The automation of degrees of confidence is another area which needs support. If the user is able to assign a confidence mark to a hypothesis, it should be be possible for the tool to backtrack until a hypothesis of high enough confidence is located, or to suggest which hypothesis to explore next, based on the degrees of confidence for all the known hypotheses.

2.2. Soloway's model for Program Comprehension

Elliot Soloway's model is also a theory dealing with the comprehension of completed programs, but divides the knowledge base and the assimilation process differently. [15, 17]

In Soloway's terminology, to understand a program is to have recovered the intention behind the code. *Goals* denote intentions, and *Plans* denote techniques for realizing intentions. Plans work as rewrite rules that covert goals into *Subgoals* and finally into program code. Program comprehension is defined as the process of: recognizing plans in code, combining these plans (by reversing the rewrite rules) to form subgoals, and combining the subgoals into higher level goals. It is clear this process is primarily bottom-up.

The knowledge base used in this model contains many parts. Those which have been identified include:

1) Programming language semantics. This deals with the understander's knowledge of the language in which the target program is written.

2) Goal knowledge. This is the encoding of the understander's set of meanings for computational goals. These goals are encoded independently of the algorithms and the languages which implement them.

3) Plan knowledge. This is the encoding of solutions to problems that the understander has solved or understood in the past. These solutions are low level components and include those universally known to programmers. Plan knowledge also includes plans the understander has acquired of a domain-specific knowledge. An example of plan knowledge is the knowledge an understander gained when a 'sort' computational goal is actually transformed into source code.

4) Efficiency knowledge. This is how understanders detect inefficiencies, and the influence that efficiency issues have on programming code and plans.

5) (Problem) Domain knowledge. This is the understander's knowledge of the world - from the application domain to the computational domain, and those domains in between. (Here, in effect, domain knowledge is merely a specialized type of goal knowledge.)

6) Discourse rules. This is the knowledge of certain programming conventions which allow the understander to attach greater meaning to aspects of the source code and/or documentation than is ordinarily possible. An example of a discourse rule is 'If a variable name forms a word, the

meaning of that word is somehow related to the purpose of that variable,' e.g. it is legal to assume that the variable "UserSelection" concerns input more than the variable "SquareRoot."

The mental model (the current state of understanding) is likewise divided in a different manner. The components Soloway uses are:

1) The specification - the description of the goals of the program
2) The implementation - the description of the actions and structures in the program
3) The annotation - the explanation which relates each goal in the specification to the corresponding parts of the implementation

Soloway's assimilation process is responsible for all the actions taken by the understander, from turning pages of program text to constructing the mental model. However, as noted earlier, Soloway's emphasis is the reverse of Brooks', thus most of the research has been done trying to explain understanding as a bottom-up process.

The supporting experiments which have been completed are concerned with behavior and events which last seconds and minutes. [8] For example, event such as reading a line of code, formulating a question, or stating a hypothesis are all events which have been looked at in detail. It is clear that one of the advantages of this approach is that experimental evidence to support the theory is easier to obtain. Thus, empirical studies have shown the existence of discourse rules and plans, and even the concept of a *de-localized plan,* (a single plan which has been physically implemented in separate sections of source code) to explain varying levels of complexity in comprehension. [10, 16] Also, Stanley Letovsky has conducted experiments explaining and classifying the process that understanders go through in terms of 'Questions', 'Conjectures' and 'Inquiries'. [11] These are the high level classifications of small scale events known as meso-scale events.

Discussion of Soloway's Model

The major implications of this model concern the building, use, and manipulation of the knowledge base. The element of domain knowledge is similar to Brooks' domains, and has already been discussed. The elements of Goal and Subgoals similarly correspond to the task and intermediate domains discussed earlier. The formally defined plans correspond to the relationships between the low level domains that Brooks mentions (e.g the relationship of the 'code' domain to the 'algorithm' domain is well expressed using Soloway's plans.) But the relationships between high level domains (e.g. 'invoicing' to 'functional decomposition') are not addressed by plans. This is a direct consequence of the formal, rigid structure of plans: they have gained expressive power at the low level domains by sacrificing the power needed to express high level domain relationships.

The fact that plan knowledge is the knowledge of solutions to already solved problems implies that a template of such solutions is a beneficial mechanism for the tool to support. These *templates* or patterns of plans are manually or automatically created, from past completed projects. These templates can ease the burden of re-creating a structure similar in nature to the old.

An implication from Soloway's mental model is the idea of annotation. Annotation implies that along with recording the relationships between the domains, some sort of explanation should be attached to signify the importance, or other attributes, of the relationship. This implication is an extension to the idea of code binding. Before, code was merely attached to intermediate domains. Now, that attachment is given annotation explaining why this binding exists.

The idea of consistency and completeness broaches the idea of automated checks, possibly to give an overall 'understanding' rating, or perhaps the ability to suggest which area of the program should be next explored.

The difference in approach (i.e. bottom-up versus top-down) has some implications as well. The bottom-up approach warrants the need for the ability to collect large amounts of small sized, discreet information (the plans), then use abstraction to collect these into more and more generalized packets, with the hope that the top packet correctly summarizes the purpose of the program being studied (i.e. the program is understood). Because of the large quantity of lower level plans, it is

reasonable to expect the CST to identify as many as possible, as well as indicate areas in the code where even a basic plan could not be assigned to the source code.

Of the other elements, the most intriguing is discourse rules (item 6 of Soloway's knowledge base.) It appears that discourse rules are an intermediate step between the code and the formation of beacons; in other words, looking at raw source code and applying discourse rules to individual segments, the understander recognizes beacons. (This brings up a deficiency in Brooks' theory: he explains that a beacon is a visible detail which shows the presence of a particular structure or operation. He does not explain in any manner the process of recognizing beacons in the code. Thus, if these discourse rules do describe the process, then formalizing these rules is a step toward the automation of beacon detection. Other discourse rules could encode accepted methods for combining beacons, thereby encoding knowledge of acceptable/expected programming methods.)

3. Representing Program Understanding Knowledge

The two previous models approached the problem from the human perspective: What do humans do when trying to understand a program? From that perspective, various theories and experiments are then created and tested.

The following systems approach the problem from a concrete perspective: Given the source code and little or nothing else, what techniques and representations can be developed to do the understanding in a more automated fashion? From this perspective, the model is, at least initially, driven by the source code. The goal is to convert actual lines of source into a higher level representation (a specification of the important behavior) without the need to deal with the unimportant clutter.

3.1. Programmer's Apprentice

The Programmer's Apprentice (PA) is billed as one possible solution to overcoming the crisis in software engineering. [14] Its intent is to be a knowledge-based tool for program development, with the PA assisting the human programmer wherever possible. This assistance can only be of value if the PA has some underlying understanding of what it is the programmer is attempting to do. How this knowledge is encoded is of major interest because of its possible application to CST, which has similar requirements.

The major data structure is what the PA developers call a *Plan.* (It has the same flavor as Soloway's 'plan', but because the PA plan is even more formal, it is also more constrained than Soloway's. [13]) A plan can represent both actual program text and knowledge about programs. Plans dealing with actual program text are termed Surface Plans (or intrinsic plans); those plans dealing with knowledge about programs are termed Deep Plans (or extrinsic plans).

Plans are intended to be abstract and do not encode efficiency knowledge. They are important because they abstract out non-essential features of a program, eliminating the "How it must be expressed" portion. Plans make things as local as possible, thus making it possible to add or combine plans without causing disturbances in expected behavior.

The plan calculus used by PA is a synthesis and formalization of ideas from flowchart schemas, data flow schemas, program transformation, and abstract data types. The primary representation of a plan is graphical, however, a systematic method has been defined to translate the plan diagram into its axiomatic representation. A sample axiomatic representation for the plan "Adding to a Set" is:

```
(DEFIOSPEC SET-ADD
  ((OLD SET) INPUT)              :inputs (two)
  ((NEW SET))                    :outputs (one)
  ()                             :pre conditions (none)
  ((MEMBER INPUT (NEW))              :post conditions (three)
   (FORALL X (IMPLIES (MEMBER X (OLD)) (MEMBER X (NEW))))
   (FORALL X (IMPLIES (AND (MEMBER X (NEW)) (NOT (EQUALS X INPUT)))
                      (MEMBER X (OLD)))) )
)
```

The features of plan calculus are: language independent expression of an algorithm; straightforward combination (additivity) of two plans into a plan which satisfies the constraints of both the original plans; multiple and overlapping points of view (overlays) which express the relationships between levels in an explicit manner. [19]

Currently, the plan library (a library of surface plans) doesn't understand about data structures or specifications. It is oriented towards HOW a programs works, not WHY, (e.g. plans need to have efficiency knowledge added to begin to understand why code was written one way and not another, or perhaps, as in Soloway, annotations need to be added). It also needs the ability to contain knowledge from domains other than just general knowledge of programming.

Also available is the Plan Editor/Viewer. This tool allows programmers to modify the program by modifying its plan. The key feature is that the vocabulary used is part that of the programmer, and part restricted by the PA (e.g., the use of the word "it", "input", "implement", etc., have pre-defined meanings.) This makes it possible for the programmer to identify parts of the program by their role in an algorithm, rather than by their position in the source code or by their position in the abstract syntax tree. The primary benefit to making changes to the plan is that it is easier to avoid creating unwanted side effects. It also allows plans to be viewed from varying levels of abstraction or hierarchy, depending on the needs of the user.

The process that PA uses to analyze code deals only with the information available from the source code (i.e. no task domain information is incorporated, only the concrete code is examined.) These three steps are:

1) The surface analysis (locating surface plans in the code),

2) The grouping of the surface plans, followed by the recognition of deep plans,

3) Verification that the code meets the deep plan.

If all the actual program text cannot be matched to the deep plan (i.e. the program is still incompletely understood), then an attempt is made to regroup the surface plans and again try to recognize the deep plans. If all the actual text cannot be matched and it is not possible to regroup, the analysis section reports a "coding" bug. (It must be noted that this last statement strongly indicates the approach that PA has taken. It is assumed the deep plan exists, i.e. is already known and documented, and the only task left to do is to match it to the code. This matching is done as a bottom-up reshuffling process. The assumption that the deep plan is known does not hold when the understander is confronted with a new program. In other cases, such as a program which has undergone extensive and faulty maintenance, the deep plan may no longer exist.)

Discussion of the Programmer's Apprentice

Probably the single largest limitation on the applicability of the PA is its evolutionary approach: it starts with plans, then it produces code. From that point on, the very strong relationship between the plan and code is maintained, even if the code is changed directly. Thus, even though there exists (in theory) the ability to develop plans from code, the PA effort is directed in the wrong direction to be applied to understanding existing code. (Remember, the PA's original intent is as a developer's

aid, not an understander's aid.)

The explicit hierarchical structuring used in PA is a very useful concept. The fact that plans can be constructed and composed of other plans supports the concept of template discussed above. It is reasonable to enforce PA's rigidity in order to gain the benefits of composition of discourse rules. Thus the total number of atomic rules is kept at a minimum.

Some useful ideas come from the implementation of the plan editor/viewer. The first is the relative success that the interface (restricted natural language) has enjoyed. The CST could do worse than to have an interface that allowed commands like "show the code that does the sort on the address array." This is especially useful in the hypothesis verification/refutation stage, when it is not yet known if such an operation exists. Also, this query/response model fits in well with the low level actions that Letovsky have observed of experts trying to understand programs. [8]

3.2. Plan Recognition Systems

The next two systems are similar because they both approach understanding from the bottom-up, by scanning from the source code level and matching the code to a higher level of specification. Unfortunately, a given program behavior can be achieved by multiple *syntactically* different programs. This variability makes matching the code to its higher level specification difficult (in fact, it is impossible to do in general.) The two approaches to this problem are:

1) Table lookup. By doing exhaustive searches and parses, all possible variant forms are recognized.
2) Canonical form. First the program is converted into its canonical form. The idea is to make all programs that DO the same, LOOK the same. Thus, matching the code to a higher level specification becomes a trivial operation.

3.2.1. The Recognizer

The Recognizer is an implemented attempt by Wills to perform the analysis procedure referred to by the PA developers. [21] This reverse engineering tool, which locates plans in existing code, is of great relevance to the CST.

The Recognizer is one possible assimilation process which can create the data structures needed to understand a program. The Recognizer uses plan calculus as its knowledge-representations notation, retaining only net data and control flow arcs (as plan calculus is intended to do). The Recognizer uses exhaustive parsing of the source code, trying to match segments of code to its known base of plans (the *Cliches),* which are stored in its knowledge base (the *Plan Library).* These cliches are then combined to create its working model. As stated by Wills, "Program understanding comes from recognizing familiar parts and hierarchically building an understanding of the whole based on the parts." Thus the Recognizer starts with what any fully automatic understanding process must start with - the source code alone. The Recognizer locates surface plans and exhaustively combines them to form deep plans.

The Recognizer is presented here from the point of view of the limitations the developers have noted. The first general restriction on explaining human understanding is stated by Wills: "No claim is being made ... that formal parsing is a psychologically valid model of how programmers understand existing programs." [21] Two other main obstacles to automatic program recognition using plan calculus are:

1) Recognizing some structures requires that the plan be seen in more than one way. (And since it is not obvious when this is needed, the only way to guarantee success it by exhaustive search.)
2) Programs are rarely constructed entirely out of familiar structures; recognition is therefore always be partial.

What the Recognizer can handle correctly are nested expressions, conditions, and loop exits. What the Recognizer cannot handle are: side effects, recursion, arbitrary data abstraction, functional arguments. Nor can it explain what is the intent of a piece of code (e.g. hashing versus list

implementations). It also cannot extract information from variable names.

The more applicable improvements the author feels are necessary are: first, an automatic highlighting of a recognized cliche in the code. This corresponds nicely to Brooks' idea of code binding. Second, the ability to put *Bug-Cliches* into the plan library. These intentional faulty cliches catch coding errors such as: a missing operation, performing the wrong operation on the right data, out of order arguments, conditional branch reversal, or even the locating of superfluous operations. Thus, instead of reporting a segment of code as unrecognizable and halting, the tool continues to operate, alerting the user at some later time of the defective code.

Discussion of the Recognizer

The CST is being designed with the thought of overcoming the restrictions in current tools such as the Recognizer - it must deal with existing code; thus completely automated understanding is out of our reach for now. Instead, the CST must be considered more of an aid to comprehension, not a complete solution.

The most exciting idea to come from the Recognizer is the concept of locating bug cliches, as well as correct cliches. It has applicability to the entire comprehension process: from locating faulty surface plans, to locating incorrect beacons, to locating incorrect algorithms.

3.2.2. CPU

The CPU program (Cognitive Program Understander) is an analysis tool that converts programs into formal specifications. [7, 9] The CPU was targeted specifically for the maintenance task, but adopted a more restrictive view of the tasks that maintainers do when understanding a program. CPU is used to analyze a program to:

1) Generate summaries of the code. This is a direct consequence of the bottom-up approach the analysis takes. Lower levels are abstracted by the use of *transformational analysis* in which recognized plans in the code are replaced by descriptions of their goals.

2) Answer questions about the code. There are three classes of questions: 'why', 'how', and 'data description'. All three result from the data structures used in representing the analysis of the code. A 'why' question deals with a given section of code and traces upwards in the analysis to find the goal(s) it supports. A 'how' question is the opposite: given a goal, it traces downward to find the code that implements it. A 'data description' question extracts information about a data type that was recognized in the code, including the operations on that type.

CPU produces the (near) canonical form for a given program and then matches this form to a data base of such forms (this data base is called the knowledge base.) This representation is "near" canonical because of the undecidability issue of proving programs equivalent. Because it is impossible to have complete canonical form (which eliminates variability entirely), the goal of the CPU is to reduce variability between representations of programs. To reduce the variability in a set of equivalent expressions, the CPU chooses a single expression from that set and considers it the canonical form. Then CPU maps all the other expressions to the chosen canonical form by using auxiliary transformations.

A representation language used for the canonical form must be highly expressive if it is to cover both the programming level and all the summation levels above the programming level. CPU uses lambda calculus as its internal representation language. Mechanical translation of source programs into lambda calculus is a simple matter, which makes CPU less language dependent.

The analysis consists of applying different transformations to the lambda calculus representation. The three classes of transformations are:

1) Basic transformations to perform arithmetic simplification, dataflow simplification and parallelizing assignment operations.

2) Simple rewrite rules, which consist of a pair of lambda calculus expressions. When the pattern on the left hand side matches a given expression, that expression is replaced by the right hand side.

3) Complex transformations, which perform loop recognition, data structure recognition, and conditionals recognition. Loop recognition is performed by a process called *recursion elimination.* [18] There are four primitive looping plans that CPU recognizes: enumeration, accumulation, maps and filters. Data structure recognition involves recognizing an abstract type, such as a "sequence", which has been implemented in a variety of ways: with arrays, linked lists, files, etc. The data structure transformations require simultaneous, coordinated transformations of all the expressions in the code that affect a given data structure.

Discussion of the CPU

Transformational analysis in general appears a viable method for recovering high level information from existing code. The unanswered question that remains is whether or not lambda calculus is the best representation of that high level information where humans are expected to grasp its meaning (especially when a non-trivial program is involved.)

The canonical form approach used by CPU is a workable alternative to search-based approaches (such as by the Recognizer.) It is especially noteworthy that the emphasis taken is to reduce variability, not try to eliminate it entirely. This explicit acknowledgement - that 100% success/recognition/representation is not possible - is critical when dealing in the area of program comprehension. A tool that can reduce the nonessential details of a program is sufficient to allow the understander to focus on the important aspects.

4. Synthesis and Summary

This section combines concepts taken from all five approaches studied. It begins with some requirements that the CST must meet in the cognitive model areas: the knowledge base, mental model, and assimilation process. It then discusses the differences and implications of the top-down approach versus the bottom-up approach. Finally, interface considerations for the CST are addressed.

4.1. Components of the Cognitive Model

The three components of the cognition - the knowledge base, the mental model and the assimilation process - form the frame for the CST.

4.1.1. Knowledge Base

The first component of the cognitive model discussed is the knowledge base. This section encompasses the idea of domain knowledge and the ideas of beacons, plans, and cliches. Complete domain representation (from the task domain all the way through to the source code domain) is the root of the CST. It is clear that programming consists of transforming the task domain through intermediate domains until the programming domain is reached. One possible (partial) ordering of this transformation is as follows: (where items enclosed in '[ ]' indicate the name of the domain, and '<-->' indicates that there exists an immediate relationship between two domains.)

[task domain]
<--> [.. intermediate domains ..]
<--> [algorithms]
<--> [plans]
<--> [beacons]
<--> [programming language]
<--> [source code]

This brings up the question of formality in the definition of domains, and also whether the same degree of formality is carried through from the task domain to code (or even, if it CAN be carried through). A higher degree of formality is a detriment at the lower levels; as stated before, programs are rarely constructed entirely from formal methods and structures. Thus, a higher formality causes higher degrees of non-representable code.

However, a high degree of formality is a great benefit if automatic domain transformations at the lower level is to be achieved. Also, this higher degree makes composition an easier task. Thus, it makes it easier to achieve another requirement for the CST: the use of templates and libraries. Given the large quantity of information that must be sifted through at the lower domains, the ability to abstract away as much detail as possible must be supported.

4.1.2. Mental Model and Assimilation Process

The next two components of the cognitive model are the mental model and the assimilation process. The mental model is the encoding of the actual relationships between the domains. The assimilation process is the set of all actions that modify the mental model. This discussion focuses on the actions, with the implicit assumption that the underlying mental model structure is available.

The first requirement of the CST is a relating mechanism which varies from highly structured to uninterpreted. Once these relations are created, it is necessary to present them in a variety of manners. From the lowly beacon which has been bound to a highlighted line of source code, to the complex plan which encompasses a several beacons, to the higher domains like "write invoices" which encompasses a large number of plans. The user must be given: 1) an easy way to create the relationship, 2) an easy way of displaying the relationship, both in abstracted (e.g. showing overall structure) and in concrete mode (e.g. showing lines of code).

Modifiers to a given relation should include annotation and degree of confidence assignment. Annotation is used mainly by the human, to document non-obvious or non-structured information about a given relationship. The degree of confidence assignment records both the human's and CST's generated assurance measurements.

These measurements are used as an aid for yet another operation that the relation mechanism must have: backtracking. This backtracking must be at a level higher than the simple "undo the last 6 relations entered." It must instead be context oriented, guided by the confidence markers.

Yet another requirement is the ability for the CST to produce consistency and completeness checks. This serves as a global monitoring of confidence levels, and can also to point to areas in need of further attention.

4.2. Approach Differences

This next synthesis section deals with the differences between the bottom-up and top-down approach, as they affect the development and use of the CST.

A bottom-up oriented approach is more intuitive to use. Maintainers and understanders are familiar with the idea of physically grabbing the listing and reading it. If the top-down approach is taken, it forces them to make explicit their higher level hypothesis - which seems an unnatural act to someone unaware of its happening. A bottom-up orientation also is more likely to have a better code scanner: the creation of beacons, plans and representations is much more thorough because of the increase in importance the bottom-up orientation gives it.

However, most likely the bottom-up orientation is bound to fail: it simply creates much more data than a human can handle. The top-down orientation ensures that human limitations are incorporated at every step of the understanding process; at no time should the human feel overwhelmed.

Keep in mind this is only a discussion of the *major* emphasis of the CST. The CST most assuredly has facilities to deal with both the top-down and bottom-up approaches; the end product is a synthesis of the two ideas.

4.3. User Interface Considerations

The last synthesis section deals with user interface considerations. The main inference drawn here is that a command line or short query interface is suitable, based on the studies done by Letovsky. The understanders are in fact building a huge knowledge tree in their minds, but it is clear that actual steps taken to create it are very small, and fragmented. The interface needs the ability to context shift very rapidly, so that users can reach a relevant section in the mental model and record their thoughts there. It must not be so slow and ungainly that users make remarks or hypothesis but do not take the time to record them. Also, since making mistakes is a part of the understanding process, it must not be difficult to back out of various tracks. Again, if it is too unwieldy, the user simply keeps the hypothesis internalized. That must not occur, because recording the implicit information about a program is exactly what the CST is being designed to do.

5. Introduction the Sample Session

If understanding a program consists of "recovering the intention behind the code" or "binding the program code completely to hypotheses", then in either case there has been an *increase* in information about the program. True, this information is totally "created" from the information in the program and from the understander's knowledge, but in the end it is separate information.

If this information is recorded explicitly then future understanders can take advantage of it without the need to recreate it themselves. In CST the term *Map* encompasses the total increase in information gained through the understanding process. The Map records hypothesis information, binding information, intention information, annotation information, etc. All the information in a Map is specific to an individual program. The structure of a Map (how a Map records all this information) is as yet informal - we cannot yet predict all the information of value to the understanding process. But it is possible to define a major requirement of the Map: it must represent relations. Such as relations between intentions and the code which realizes those intentions. Or, saying the same thing, relations between hypotheses and the code to which they are bound. Thus the Map contains two major elements: relations and the information they relate (the arcs and the nodes).

Note that much more information than Maps are placed in this database. There is also program knowledge, encoded in a form such that the computer can use it to recognize low level plans and structures in a program. There are also discourse rules, templates, bug-cliches, etc.

In the following scenario, the specific information generated in understanding the program is stored in the Map. This includes the elements mentioned above, plus all the session-derived information, such as the placement of windows, the list of agenda items. etc.

5.1. A Sample Session with CST

The User (*TU*) is faced with a common problem: TU is given a listing of program which TU has never seen before, and which must be modified in some manner. The following context information is available to aid in understanding: TU works for an accounting firm; that firm develops general ledger accounting packages; and the name of the program is "GLAcctsSubOne". From this information, plus any information available from the listing, TU must figure out the behavior of the program so TU can complete the modification.

Following standard procedure, TU runs the Cognitive Support Tool (*CST*). CST prompts for the name of the component; in this case TU enters "GLAccts.c" and indicates the entry is a source code file name. Unfortunately, in this case, CST responds by indicating that "This file has not been Mapped." TU is partially irked, but understands that CST is new, and approximately half of the company's 250 programs still need to be Mapped. ("Give it another week," TU muses.) TU will create the Map for this program this afternoon. Because TU has used Maps created by other workers, which has saved TU an immense amount of time, TU is not annoyed at the company policy that dictates that as unmapped programs are "discovered", the discoverer must completely create the Map, and save the Map in the company data base.

While TU has been musing about the world, CST has been busy. After recognizing that the file has not been Mapped, CST does multiple activities to make TU's job even easier. First, after a scanning the file, CST decides it really is a 'c' file, then loads the appropriate knowledge base of programming language/plans. This includes templates which enable CST to recognize various arrangements of source code (such as loops, conditionals and data structures), as well as "buggy" templates designed to catch common 'c' semantic errors (like doing sizeof() operations on the pointer type instead of the structure type). Next, CST pre-builds as much of the lower levels of the Map as possible. This includes creating nodes which represent functions, then relating that function node to the lines of source code that implement that function, and setting the confidence rating to 100%. It also includes recognizing and creating plan nodes, and mapping those nodes to the functions which implement them (or possibly only to the lines of source code, in the case of distributed plans.) These low level nodes are the ones which TU must "explain away" - i.e. define the purpose of in the context of some higher concepts.

CST has also scanned the code looking for data plans. It has created nodes and arcs relating all the variables to their types and location in the source, and it has created nodes and arcs relating those types to the lines of source code which define them. In the case of global variables, it has related them to the functions and/or lines of code which reference or modify them. In the case of local variables, it has done the same thing, but the relationships are context sensitive. And in the case of types, it has related those nodes to the nodes which reference or modify variables of that type.

All of these nodes and relations are added automatically to the major data structure: the Map. CST displays this structure in a graphical manner in a tall window, nodes being circles, lines being relations. The nodes at the top of the window represent the highest level specification, the nodes at the bottom represent functions and chunks of source code.

The first thing TU sees is a dialog requesting "Describe The Overall Function Of This Program." TU doesn't really have a clue yet, but that doesn't matter. TU simply enters "does some accounting junk." Later, as TU fills in the Map, this description (which resides at the topmost node of the Map) will be further refined. With this information, CST displays the current Map, with a solitary node at the top of the window, not connected to anything, and with a multitude of nodes at the bottom of the window, some with connections, others isolated. In between, for the moment, is blank space - representing the fact that the high level specification nodes have yet to be related and bound to the low level source nodes

This last step is the only part of the methodology which CST enforces. The root node (the highest level description) must exist, but CST doesn't force TU to build strictly top-down from that point. TU is free to combine top-down and bottom-up approaches as desired.

TU first decides to record the overall goal of this session by selecting "Enter The Primary Agenda Item," - either with the mouse or from the command line - to which TU responds "understand this entire program." If not for the company policy, TU could have entered a more specific task, such as "find the format bug in the check printing routine."

TU then decides on a more specific course of action and records this by selecting "Enter An Agenda Item". At the prompt, TU enters "scan the function names for clues." TU then directs CST to bring up two windows - one with an alphabetic list of functions names, another window which is blank. TU begins by grouping the functions that seem to have similar names, such as "EnterChecks", "PrintChecks", "StoreChecks", etc., and storing these unsubstantiated relations in the second window.

Then TU finds a routine of special interest, "main", and decides, on a whim, to explore that routine further now.

Before beginning on a new track, TU selects "Save Context To Current Agenda". Then TU enters a new agenda item as "Explore the main() routine". This has the visual effect of closing the two recently created windows and clearing the display for the next task. By clicking on the node which represents the function "main()", TU brings up a window with the source code for that routine; unfortunately, the entire main() routine consists of a mere three function calls: init(), process(), and stop(). Unflustered, TU clicks on the 'init()' text, and a new window with the source for that routine appears. Here, TU makes an interesting discovery: a function called 'Zinitchks()', which should be grouped with "EnterChecks", "PrintChecks", etc., from the previous task. Before TU forgets this, TU selects "Save Context To Current Agenda", then TU activates the agenda item named "scan the function names for clues." This second action has the visual effect of closing the two source code windows, and opening the previous two windows which contained the function names and the current grouping. TU quickly associates 'Zinitchks()', then selects "Restore Previous Context" so that TU can continue scanning source code from the last point. This action closes the two open windows, and re-opens the windows displaying the source code.

Finally, after examining enough functions, TU hypothesizes that the classic 'inputchecks', 'processchecks', and 'outputchecks' breakdown is appropriate here, and creates these new nodes in the Map window, and also relates them to their parent node, which TU now changes to "process corporation checks." Next, TU starts an agenda item to explore the 'inputchecks' hypothesis node. During this time, TU is locating pieces of code that relate to the 'inputchecks' process (working bottom-up) while at the same time TU is creating new hypotheses (working top-down). TU quickly discovers the existence of routines that input some header information, and then enters the next breakdown under 'inputchecks' as 'input header' and 'input blocks'. This process of working from the top and the bottom continues until the 'inputchecks' hypothesis node is completely bound to function or source code nodes. TU checks this completeness by selecting "Completeness Rating" and clicking on the 'inputchecks' node. CST responds with "100% - All 20 Hypotheses Nodes Are Bound."

TU then starts another agenda item to explore the 'outputchecks' process, skipping the 'processchecks' process for now. After that is complete, TU finishes up the last sub-hypothesis, and saves the Map to the data base.

TU can now start to work on the real problem. TU begins by entering another agenda into the now-empty agenda list ("understand this entire program" has been completed.) The task is to "find the format bug in the printing the header of a check." Using the existing map, TU quickly focuses on those parts of the Map which concern output, locates the specification node of "print one check" and clicks on it to bring up the source code lines that implement it. TU then condenses the source code lines which don't deal with "printing the header", traces the variable in which the header text is stored and determines that it is being printed properly. The error must be in the set up, so TU requests that the variable's assignment statement(s) be shown, CST displays three lines of source that modify the variable, (and unknown to TU, one of those statements has been bound under the 'processchecks' hypothesis, but CST doesn't artificially restrict the search space, and displays all the statements.) Since TU can view all three of the assignment statements simultaneously, TU quickly notices the format inconsistency in one of the statements, and changes it. (That statement, by coincidence, is the one bound under 'processchecks', so in retrospect TU surmises that the previous maintainer had changed the format in the statements that were located relatively close together in the source code, but had missed the third, because the maintainer had incorrectly hypothesized that those routines dealt only with processing the individual check data.)

CST has given TU a big productivity gain the initial time a program is examined, and CST gives even bigger gains for future maintainers who can access this information without the need to re-create it for themselves.

References

1. R. Bhaskar and H. A. Simon, ‘‘Problem Solving in Semantically Rich Domains: An Example from Engineeering Thermodynamics,’’ *Cognitive Science*, vol. 1, no. 2, pp. 270-283, 1977.
2. Ruven Brooks, ‘‘Towards a Theory of the Cognitive Processes in Computer Programming,’’ *International Journal of Man-Machine Studies*, vol. 9, no. 6, pp. 737-742, 1977.
3. Ruven Brooks, ‘‘Towards a Theory of the Comprehension of Computer Programs,’’ *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.
4. Linore Cleveland, ‘‘A User Interface for an Environment to Support Program Understanding,’’ *Proceedings Conference on Software Maintenance-1988*, pp. 86-91, IEEE Computer Society, Phoenix, Arizona, October 24-27, 1988.
5. A. D. DeGroot, *Thought and Choice in Chess*, pp. 1335-1342, Mouton, The Hague, 1965.
6. J. Larkin, J. McDermott, D. P. Simon, and H. A. Simon, ‘‘Expert and Novice Performance in Solving Physics Problems,’’ *Science*, vol. 208, pp. 1335-1342, 1980.
7. Stanley Letovsky, *How Abstraction Can Reduce Ambiguity in Explanation Problems*, Cognition and Programming Project, Department of Computer Science, Yale University.
8. Stanley Letovsky, ‘‘Cognitive Processes in Program Comprehension,’’ in *Empirical Studies of Programmers*, ed. E. Soloway and S. Iyengar, Ablex Publishing Company, Norwood, New Jersey, 1986.
9. Stanley Letovsky, ‘‘A Program Anti-Compiler,’’ Draft Technical Report, Department of Computer Science, Yale University, July 8, 1988.
10. S. Letovsky and E. Soloway, ‘‘Delocalized Plans and Program Comprehension,’’ *IEEE Software*, vol. 3, no. 3, pp. 41-49, May 1986.
11. S. Letovsky, J. Pinto, R. Lampert, and E. Soloway, ‘‘A Cognitive Analysis of a Code Inspection,’’ in *Empirical Studies of Programmers, Second Workshop*, ed. G. Olson, S. Sheppard, and E. Soloway, Ablex Publishing Company, Norwood, New Jersey, June 1987.
12. B. Lientz, E. Swanson, and G. E. Tompkins, ‘‘Characteristics of Application Software Maintenance,’’ *Communications of the ACM*, vol. 21, no. 6, June 1978.
13. Charles Rich, ‘‘A Formal Representation for Plans in the Programmer’s Apprentice,’’ *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1981.
14. Charles Rich and Howard E. Shrobe, ‘‘Initial Report on a Lisp Programmer’s Apprentice,’’ *IEEE Transactions on Software Engineering*, vol. SE-4, no. 6, November 1988.
15. Eliot Soloway, Jeffrey Bonar, and Kate Ehrlich, ‘‘Cognitive Strategies and Looping Constructs: An Empirical Study,’’ *Communications of the ACM*, vol. 26, no. 11, November 1983.
16. Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert, ‘‘Designing Documentation to Compensate for Delocalized Plans,’’ *Communications of the ACM*, vol. 31, no. 11, pp. 1259-1267, November 1988.
17. E. Soloway and K. Ehrlich, ‘‘Empirical Studies of Programming Knowledge,’’ *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 595-609, September, 1984. C. Rich and R.C. Waters, editors, Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, 1986
18. Richard C. Waters, ‘‘A Method for Analyzing Loop Programs,’’ *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 237-247, 1979.
19. Richard C. Waters, ‘‘The Programmer’s Apprentice: Knowledge Based Program Editing,’’ *IEEE Transactions on Software Engineering*, vol. SE-8, no. 1, pp. 1-12, 1982.
20. Susan Wiedenbeck, ‘‘Processes in Computer Program Comprehension,’’ in *Empirical Studies of Programmers*, ed. Eliot Soloway and Sitharama Iyengar, pp. 48-57, Ablex Publishing, Norwood, New Jersey, 1986.

21. Linda M. Wills, ‘‘Automated Program Recognition,’’ AI-TR-904 (Masters Thesis), Massachusetts Institute of Technology, January 1987.