

RECOGNITION IN A PROGRAM UNDERSTANDING SYSTEM

Stephen Fickas
Information and Computer Science Dept.
University of California Irvine
Irvine, California 92664

Ruven Brooks
Dept. of Psychiatry and
Behavioral Science
University of Texas Galveston
Galveston, Texas 77550

We propose a recognition model for a Program Understanding system based on a top-down control structure that begins from a given high level plan and attempts to refine and decompose the plan until it can be matched against actual code. We introduce the notion of "distinctive features" and "beacons" which help guide the recognition process and are based on a set of functional program "templates". Additional mechanisms are proposed to recover from situations in which the primary top-down approach fails.

1. INTRODUCTION

The goals of our research project are twofold: one, to develop a working definition of Program Understanding which is capable of incorporating all of the diverse pieces of information expert human programmers employ in understanding a program, including efficiency tradeoffs, program comments, knowledge of the task domain, language dependent and independent programming knowledge; two, to define a set of problem solving methods which use such knowledge in constructing an understanding of a computer program. This paper addresses the second of these goals, particularly the types of recognition methods most suited to the program understanding task. The remainder of this section provides the preliminary background for our discussion of recognition in section 2. To test our ideas, we are currently developing an AI system which accepts actual programs from a specific domain, and produces explanations of the programs in the framework we have defined.

1.1. Description Trees

We view a program as the final product of a hierarchical design process similar to Wirth's notion of step-wise-refinement. We define a Description Tree (DT) as a hierarchical plan structure generated from this process. Our DT has much in common with Brown's[3] hierarchical descriptions, de Kleer's[4] plan fragment description's, and Rich and Shrobe's[8] complete program descriptions.

For any given task specification, there can exist a possibly large number of different DTs, one for each unique final program (Barstow[1] defines a "refinement tree" which is analogous to a supertree of all DTs generated from the same specification).

1.2. Program Building Blocks

The basic conceptual programming unit of our system is called a program building block or *pbb*, and is in most ways equivalent to what Rich and Shrobe[8] refer to as a

Segment in their Programmer's Apprentice. Each *pbb* has a set of data inputs, pre-conditions which must hold on entry, a set of resulting outputs, and post-conditions which are true on exit. Other types of information which help guide recognition may also be attached to a *pbb*, eg. possible mnemonic names. Figure 1 shows a *pbb* that copies a file to a linked-list.

```
(pbb file-to-list-copy
  (INPUT: file1)
  (PRE-COND: type(file1,SEQUENTIAL))
  (OUTPUT: linked-list1)
  (POST-COND: type(linked-list1,LINKED-LIST)
    map1to1(linked-list1,file1))
  (MNEMONICS: READ$FILE COPY$FILE])
```

Figure 1: Example *pbb*

1.3. Implementation Plans

Any non-primitive *pbb* can be decomposed into sub-parts (also *pbb*s) linked by dataflow and teleology. We call this decomposition an *Implementation Plan* or *IP* (for more details, see Rich and Shrobe's[8] concept of "deep plan"). In terms of the DT, for any non-terminal node B, all sons of B constitute the sub-parts of an *IP* of B.

2. CONSTRUCTING A DESCRIPTION TREE

Our initial goal is to build a system that constructs a DT given simply the task specification and the program code; however, we are including design features which will enable it to use knowledge from preprocessed mnemonic names and comments. While the final system is not intended as an explicit model of human behavior, a guiding principle in the design is that it should use, wherever feasible, mechanisms derived from the analysis of human protocols.

2.1. Justification of a Top-down Approach

The essential control structure of the system is to start with an initial hypothesized *Implementation Plan*, hypothesizing and confirming further refinements and decompositions, halting when the lowest level *pbb*s (ie. leaves of the DT) match the actual code. There are several reasons why a top-down method is preferred. First, a top-down approach provides a useful high level understanding even when forced to halt before reaching the lower levels of the Description Tree. For example, being able to identify sections of a program which read a file into an array, sort the array, and merge the array with another file is useful even if exact details have not been (or cannot be) worked out for the read, sort or merge *Implementation Plans*.

Second, from our observations, human programmers generally employ top-down methods when attempting to understand a program. If a program understanding system employs the same basic top-down approach, then a user of the system should be able to follow, in a natural way, the system's progress. If the actions of our system can be easily understood by the human user, then it is feasible to consult the user when the system decides it needs help. Both the value of using a human expert as an active partner and the importance of conforming to his problem solving methods has been well documented in the AI literature.

2.2. Templates

In our system, *pbb*s are considered to be independent of the particular implementation language in the sense that, while they may assume the existence of certain data structures or typing conventions, the *pbb*s cannot be compared directly to the string of characters that form the program. The information to make this comparison is represented in our system by a structure we call a template (This term was selected on the basis of a structure proposed to fulfill a similar function in a model of human programmer behavior, Brooks[2]).

A template links together information about the way in which a function or operation is performed in a particular programming language. In many respects, our use of templates resembles the use of axiomatic assertions to describe the effects of pieces of syntax (Hoare[5]); however, for use in our system, we have extended that idea by generalizing the notion of syntactic type from just those structures which are primitive in the programming language to encompass any kind of pattern in the surface structure of the code which has an associated function. For example, the function of "looping through an array" might have associated with it the following pattern:

```
DO <statement*> <index> = 1,<array size>
```

```
... array( <index> ) ...
```

Note that there are places in the pattern in which arbitrary, unmatched pieces of code can occur.

2.3. Distinctive Features and Beacons

To verify a hypothesized refinement or decomposition, code must be found in the program text which helps validate the hypothesis. To do this, a template must be found whose function is the needed one and whose pattern matches some portion of the code. The expected number of templates in our system is large so that matching templates efficiently is a central problem.

Our initial approach in solving this problem was based on an observation about the way human programmers look for particular functions in a program. Programmers generally associate with each function a set of *distinctive features* which enable them to locate the function in the code. For example, an interchange of two elements of an array is a *distinctive feature* of sort routines. We therefore thought that templates could be grouped into sets of *distinctive features* for each *pbb* in our system, and that a hypothesized *pbb* could be bound to the code by searching for just those *distinctive features* associated with its function. A further observation on human programmers caused us to modify that approach somewhat.

While human programmers look for *distinctive features* to help confirm their hypotheses, they also notice particularly salient features of other *pbb*s. We term these unusually salient features *beacons*. For example, a programmer looking for (hypothesizing) a sort routine might also identify the file-to-list-copy *pbb* of figure 1 by noticing a *READ* statement (*beacon*) somewhere else in the program. We felt that such behavior was advantageous in our system as well for nominating new hypotheses, and modifying and refining existing hypotheses; the approach we use for obtaining this behavior is a dynamically adjustable saliency ordering for matching templates.

2.4. Binding

Once a template is matched, the code is bound to the associated *pbb*. This binding is both for operations and data structures so that each surface data structure is bound to its associated input or output slot. For example, suppose that the *pbb* of figure 1 was bound to the piece of code in figure 2.

```
1 = 1
100 READ(20,900,END=200)NEW(1+1)
    NEW(1)=1+2
    1=1+2
    GO TO 100
200 NEW(1)=0
```

Figure 2: Binding code

NEW is now bound to linkedlist1. This allows us to in effect attach the piece of information "holds linked list" to NEW and move the system's understanding of NEW away from the surface and closer to the domain.

If more than one template matches a particular piece of

code, only the more detailed is actually bound. For example, if a section of code was both matched by a Quicksort template, and a more general, abstract sorting template, the code would be bound to the Quicksort *pbb*.

2.5. Searching the Code

There are several possible outcomes when attempting to bind a particular hypothesized *pbb*. First, a template might exactly match the hypothesized *pbb*. In this case, the system has no new information regarding possible refinements to the matched *pbb*, and hence, must choose a refinement to try using its own local information. Currently, a fixed order list of refinements is used, but in the future, a more efficient decision process could be used to take into account the type of knowledge experienced programmers carry around as rules of thumb, such as "use a linked list if data is going to be inserted or deleted frequently", or the inverse "use an array if the data is relatively static" (see also Kant's "implementation selection rules" [6]).

If a template matches a more refined *pbb* than the one hypothesized, then the more abstract *pbb* can be replaced in the corresponding Implementation Plan with the new refined one. The effects of this new, more detailed *pbb* are allowed to ripple throughout the links of the IP, generally causing the refinement of other sub-parts in the IP. An attempt will be made to match each of these new, more refined sub-parts, causing possibly still further refinements and rippling effects.

The depth-first search for *distinctive features* may be preempted by the matching of a *beacon*. If the *beacon's* associated function is a sub-part of the current hypothesized plan, then the system will bind the sub-part and allow any rippling effects to take place. In at least some cases, this will result in the deactivation of the current *distinctive features* being searched for, and the activation of more refined templates. The overall control flow will thus be opportunistic, working on those parts of the *Implementation Plan* for which the most information is available.

2.6. When Things Go Wrong

In the ideal case, a matching piece of code can be found for each hypothesized sub-part in a plan; in practice, of course, this seldom occurs, and a number of things can go wrong. Two of the more common are: 1) no template can be matched for a particular *pbb*, and 2) a *beacon* template is matched for a *pbb* that is not part of the current hypothesized plan. In the first case, if many other *pbb's* in the plan are still unmatched and if an alternative plan is available, then the system will adopt the alternative plan. In most cases, however, an attempt will be made to match the missing *pbb* by assembling it out of smaller functions. In this case the system does behave in a bottom-up manner.

In the second case, the system attempts to revise the hypothesized plan to incorporate the unexpected match. We have not yet defined a satisfactory method for this modification which avoids the problem of re-binding already

successfully bound sub-parts, but we are considering approaches which attempt to minimize the number of changes (see for instance the similarity networks of Minsky[7]).

3. CONCLUSION

The system we have described possesses several unique features. First, it allows a useful partial understanding to be developed whenever time, the task, or lack of knowledge make a full understanding unattainable or unreasonable.

Second, by noticing the behavior of human programmers, the notion of *distinctive features* and *beacons* was incorporated into the system. These features, in combination, allow the system to act in an opportunistic fashion, exploiting whatever looks most promising as recognition proceeds.

Finally, a knowledge representation called templates was defined. Templates link a syntactic pattern with a particular function, and are general enough to be used in both top-down and bottom-up search.

REFERENCES

- [1] BARSTOW, D. (1978) Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Processing Rules PhD Thesis, AIM-308, Stanford AI Lab
- [2] BROOKS, R. (1975) A Model of Human Cognitive Behavior in Writing Code for Computer Programs PhD Thesis, Dept. of Psychology, Carnegie Mellon University 1975
- [3] BROWN, A. (1977) Qualitative Knowledge, Causal Reasoning, and the Localization of Failures PhD Thesis, AI-TR-362, MIT AI Lab
- [4] de KLEER, J. (1977) A Theory of Plans for Electronic Circuits Working Paper 144, MIT AI Lab
- [5] HOARE, C. (1969) An Axiomatic Basis for Computer Programming CACM 12, 1969
- [6] KANT, E. (1979) Efficiency Knowledge for Program Synthesis PhD Thesis (forthcoming), Stanford AI Lab
- [7] MINSKY, M. (1975) A Framework for Representing Knowledge The Psychology of Computer Vision, P. Winston (ed.) McGraw-Hill 1975
- [8] RICH, C. and SHROBE, H. (1976) Initial Report on a LISP Programmer's Apprentice AI-TR-354, MIT AI Lab, Cambridge, Ma.