A BASIS FOR THE ACQUISITION OF PROCEDURES FROM PROTOCOLS

Michael A. Bauer*
Department of Artificial Intelligence
University of Edinburgh
Edinburgh, Scotland. EH8 9NW

Abstract

The problem of reconstructing a procedure from traces of its behavior is considered. A representation for procedures, including backtracking procedures, is described and a definition of a trace or protocol is presented. Algorithms for the construction of a procedure from a set of traces and for its correction using additional traces are described.

Introduction

If one is interested in developing systems whose capabilities increase, then it is clear that one must be concerned about the mechanics of how such a system might acquire procedures. This kind of problem is examined in Sussman's HACKER [5], a program whose capabilities increase as it solves problems and acquires procedures. HACKER makes use of problem specific information and general procedural information to write a program and debug it.

Once a solution to a problem is found, one would expect a problem solver to generalize the solution in order to use the knowledge gained in solving that particular problem to solve others. The generalization could involve domain-specific knowledge, like replacing one type of object by a more general type of object. On the other hand, if one views a solution (a sequence of steps) as a trace from some unknown procedure on some input, then the generalization might involve the introduction of variables and the formation of loops.

Hewitt [4l discusses the formation of procedures from protocols - instruction traces. He is concerned with creating variables and generalizing protocols to recursive procedures. Biermann [2D discusses the synthesis of a Turing Machine from traces of its behavior. He illustrates how this technique can be used to form loops.

We build upon and extend the ideas of these authors. We introduce a representation for procedures and an execution rule. Based upon this representation and the rule, we describe the class of protocols we shall consider. Following Biermann's ideas we describe an algorithm for constructing a procedure from a set of protocols. The construction involves the formation of loops, replacement of constants by parameters and the resolution of variable renamings.

We continue by describing techniques for debugging a previously constructed procedure by using new protocols. The debugging process uses the constructed procedure and the new protocols; it does not require the retention of the previous traces. The debugging process cannot always be guaranteed to correct a constructed procedure, however, for many procedures, it works very well.

Representation

We shall restrict our discussion of procedures to function procedures, that is, procedures which, if executed sucessfully, return a single value. The techniques we shall describe are not restricted to this particular form of procedures, but this restriction will simplify many of our descriptions. (A more complete treatment can be found in til.)

We shall also be concerned with backtracking, primarily because of its importance, in some form, in recent work on problem solving [4,5,63. If the solutions we are trying to generalize involve backtracking, then our model of procedures must have this form of control as well. Thus, in addition to normal sequential control, our representation for procedures will involve backtrack control.

Our representation will involve three "kinds" of instructions - assignment statements, tests and return statements. Assignment statements have the form $x \leftarrow t$, where x is a variable and t is a constant, a variable or a function invocation. A function invocation is $f(a_1,\ldots,a_n)$ where f is a function of n arguments and each a_i is a constant or variable. A function returns a single value if executed successfully. Because of the possibility of backtracking, if a function is not executed successfully, but does halt, then we say that it fails.

Tests are instructions of the form $p(t_1,..,t_n)$, where p is a predicate (a function returning True or False) and each t_i is a constant, a variable or a function invocation. Like functions, tests may fail as well.

The remaining kind of instruction is Return(a), where a is a constant, a variable. True or False. The execution of this instruction means a successful execution of the procedure and the return of the value of a.

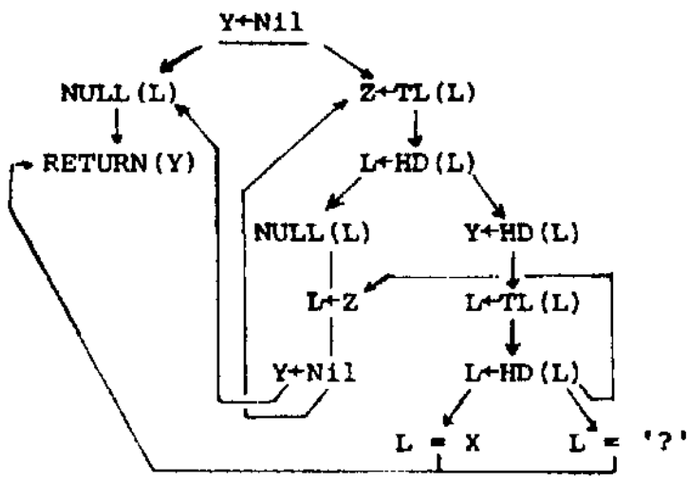
Our basic structure will be a finite, rooted, labelled directed graph. Each node is labelled by an instruction and the successors of each node are ordered. A list of distinct variables from instructions labelling the nodes is designated as the list of parameters. The directed graph and this list form a procedure.

The execution of a procedure begins when the parameters are given values (that is, bound). The instruction labelling the root node is evaluated first. The execution proceeds depth first until reaching an assignment or test which fails when evaluated or until reaching a test which evaluates to False. The execution then resumes at the last successfully node with an untried successor. The value of all variables are restored to the value they had after the execution of this node. If all the successors of the root node are tried unsuccessfully, then the procedure

fails.

Let us illustrate this rule on a simple procedure. Figure 1 represents a function PAIRS (L,X) of two arguments (the root is underlined; successors are ordered left to right). The first argument, L, is a list, possibly empty, containing null lists or pairs of atoms. The second argument, X, is an atom. PAIRS searches L looking for the first pair whose second element has the same value as X or whose second element is a '?'. PAIRS then returns the first element. If L is Nil or if PAIRS fails to find a pair, it returns Nil.

Figure 1 PAIRS(L,X)



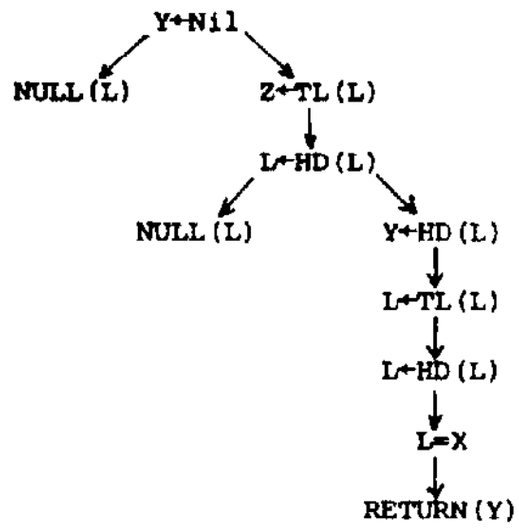
Now consider PAIRS ([[A '?']],'?'). The root is labelled by Y<-Nil; its evaluation makes Y Nil. The first successor, NULL(L),is evaluated - returning False. The execution then resumes at the next successor of Y<-Nil, namely Z<-TL(L) . The evaluation of this instruction makes Z Nil. Its only successor, L«-HD(L), is evaluated making L [A '?']. Its first successor, NULL(L), evaluates to False; the second successor assigns A to Y. The next two instructions assign '?' to L. The first successor of L<-HD(L), namely L = X, evaluates to True. The Return (Y) is evaluated, returning A and signifying the end of the execution of PAIRS.

This backtracking rule is naive, but it does provide a beginning for studying traces from procedures involving more complex backtracking control structures.

Our next task is the description of a protocol Let P(x ,...,x) be a procedure in our representation and let a_1, ...,a be the input we wish to use to illustrate how P works. Imagine that we have "unwound" the directed graph of P to form an infinite tree (or finite if P has no loops). Using our execution rule, if we evaluate P(a_1...,a) and it halts (either returns a value or fails), then the nodes of the infinite tree evaluated during the execution form a finite subtree. Replace each node of this finite subtree by its label (instruction) - this is our trace. We shall call this trace a pure trace.

In our description of PAIRS, the pure trace produced by PAIRS (t[A '?']), '?') would be that in Figure 2.

Figure 2

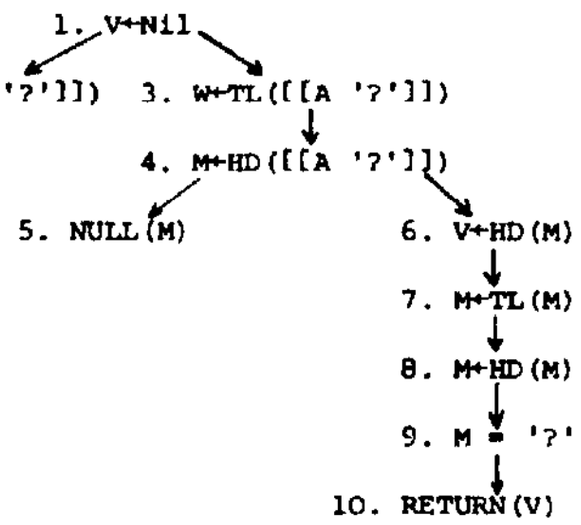


Unfortunately, traces are seldom pure. Most traces are variations of pure traces - for example, using constants in the trace instead of variables. We shall only consider two variations of pure traces. The first involves replacing a parameter that occurs in the trace by the input constant that was bound to it. This can be done in two ways. First, if x is a parameter and c was the constant bound to it, we may add the node x<-c as the new root node of the trace. (One can think of this as our syntactic version of "Let's say x is c", prefacing a verbal description of some operations on x.) Alternatively, we may replace any occurrence of x by c along any number of branches from the root node. In this case however, replacement is permitted in those instructions along a branch occurring before a node of the form x<-... occurs (with replacements of x occurring on the right hand side permitted).

Our second alteration involves renaming variables. We may replace any variable by another as long as we replace all occurrences of a variable by a new one and never introduce a variable which already appears in the trace.

Let us transform our protocol of PAIRS ([[A '?']], '?•) using these rules. Let us first insert constants. Replace L by [[A ' ?'31 and X by '?'. Then let us rename variables as follows: Y becomes V, W becomes Z and L becomes M. The resulting trace is (ignore numbers for the moment) :

Figure 3



We feel that these variations of pure traces permit us to talk about a class of traces which contain important characteristics which we might expect of traces generated by a problem solver or presented by a person - use of constants and use of variables.

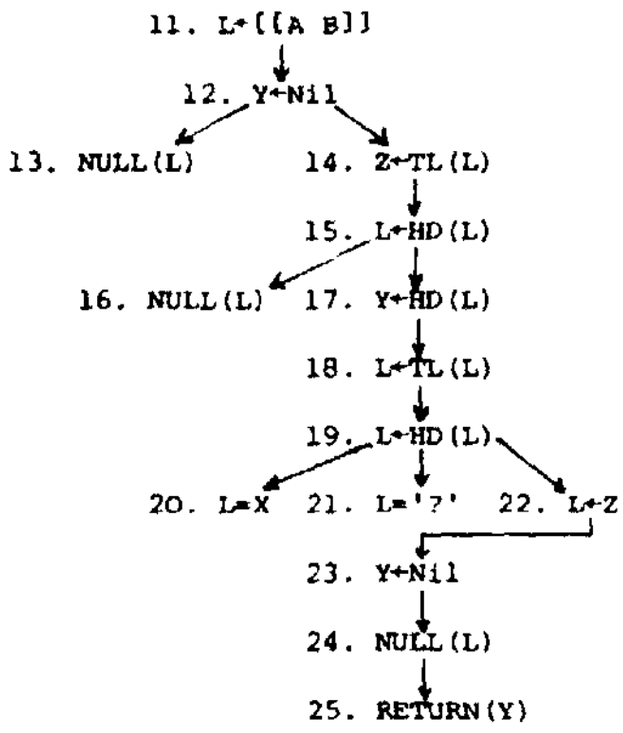
In the next section we shall be concerned with the construction and correction of procedures from pure traces and transformations of pure traces (henceforth we shall refer to both pure traces and transformations of pure traces as just 'traces' or protocols).

Construction

Our first task concerns the construction of a procedure from a given set of protocols. Suppose that in addition to the trace of Figure 3 we also have the trace of Figure 4. Take these two as our set of given traces.

Assume that we have no idea what procedure the traces came from, just that they come from the same procedure and the transformations described earlier may or may not have been used. We shall illustrate the workings of the algorithm on these examples and then briefly summarize the major steps

Figure 4 PAIRS([[A B]], '?')



Before that, however, we need to introduce two preliminary notions.

Let us call a set of nodes <u>similar</u> if for any two nodes from the set either 1) they are both from the same trace and the instructions labelling them are identical except that where one has a constant the other has a variable or 2) they are both from different traces and there is a substitution of variables for the variables and constants of the instructions making the instructions identical. For example, the set {1, 11} is similar and the set {1, 11, 12} is not.

Let us call a set of similar nodes <u>identifiable</u> if the sets of all first, second,_____, successors from the nodes in the set are similar. For example, {1, 12} is identifiable because the set of first successors, {2, 13), and the set of second successors (3, 14), are both similar sets. The set {8, 15) is not identifiable since the set of first successors, {9, 16), is not similar.

With these notions presented, we can now describe how the construction algorithm would generalize these protocols.

Firstly, we know that nodes of the form Var.-> Const, might have been added (using the first transformation). We also know that nodes containing function invocations or tests or nodes which have multiple successors could not have been added. Using these rules we proceed along the path from the root node until we reach a node we are sure has not been added. In our examples, these are nodes 1 and 12.

Working backwards from these nodes, we can see if corresponding nodes in both traces are similar. When we reach the root nodes or find non-similar corresponding nodes we stop. We conclude that nodes accepted during this process must have been added - node 11 in our example. We delete these added nodes and shall use them to help determine parameters. We shall call this process Reduction.

We then begin to match the set of protocols with one another. The nodes match if they are similar. The matching process proceeds from the root nodes, along corresponding paths (two paths from two nodes 'correspond' if they begin with the i successor of each node, say n_1 and n , and one of n_1 , n_2 has no successor or the paths from n_1 and n_2 correspond). Matching provides a means of grouping nodes likely to be identifiable and finding substitutions.

The matching in our example proceeds as follows: nodes 1 and 12 match, indicating that v and Y are probably the same variable. Matching 2 and 13 tells us that L has been replaced by [[A '?']]. Matching 3 and 14 we see that Z and W are the same variable and that L has been replaced by [[A ' ?]l) again. This last fact agrees with the previous information. If instead of[tA '?']) we had found [[B '?']], then L would have had to be replaced by two constants. Since this event would be beyond explanation in terms of our transformations (and, in fact, indicate an error) we would terminate the attempt to construct a procedure .

Matching 4 and 15 reveals that L has been replaced by [IA '?')} and that L must represent the same variable as M - an apparent, contradiction. This is not really a problem since variables can be replaced by constants and renamed as well (that is, involved in both transformations). As long as we do not find L replaced by two different constants or variables in the same trace, we are satisfied.

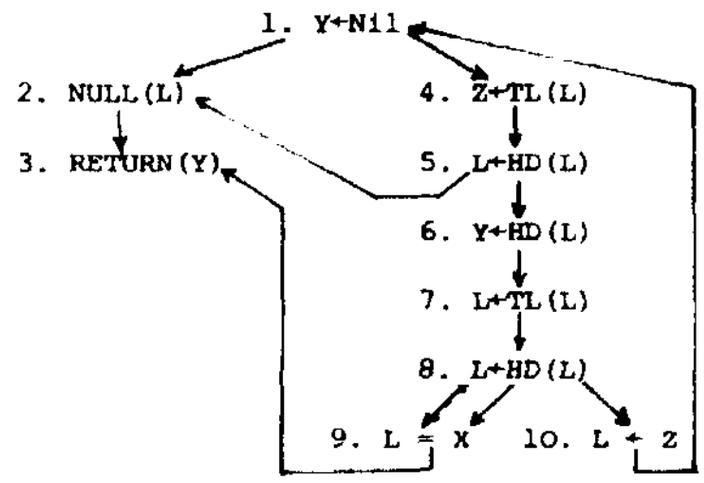
The results of the matching process are the following pairs of nodes: (1,12), (2,13), (3,14), (4,15), (5,16), (6,17), (7,18), (8,19), (9,20). The following 'equivalences' are discovered as well: V and Y, [[A '?']3, M and L and W and Z. Assuming that we use the variables in Figure 4 as a basis, these equivalences suggest that the following substitutions have occurred. V for Y, [[A '?']] and M for L and W for 2.

The next step is to form sets of identifiable nodes and try to collapse the two traces together and form loops in the process. But which sets do we choose? Since we are interested in constructing a general procedure, we choose sets of identifiable nodes which result in the fewest number of sets containing all the nodes. However, we also make sure that nodes paired during the matching process remain together.

The actual procedure used to find such a set is similar to the refinement procedure used to minimize the states in finite state machines t3).

Now using the nodes omitted during the reduction and the substitutions discovered, it is easy to deduce that L was a parameter. Since '?' occurred in the parameter list, we create a new variable, say X, and replace occurrences of '?' by X. The resulting procedure appears in Figure 5.

Figure 5 PAIRS (L,X)



Summary of Construction

The following descriptions of the steps undertaken in the previous example provides a concise overview of the construction process. A detailed description can be found in [1].

1. Reduce the protocols, if more than one, to find similar start nodes. Save nodes omitted.

2. Match the protocols, beginning at the start nodes, recording successfully matched sets of nodes and any substitutions that are discovered.

3. Find classes of identifiable sets such that for each class each node of each protocol is contained in one of the identifiable sets in the class, any nodes paired during the matching are in the same set and for any identifiable set, the set of first successors of nodes in that set are all contained in some set, the set of second successors are in some set, etc.

4. Choose the class containing the fewest identifiable sets; resolve ties arbitrarily.

5. Given a chosen class of identifiable sets, find the substitutions for the class which make the instructions in each set identical. If any of the substitutions produced contradict those found in matching, partition the class by grouping nodes to avoid the contradictory substitutions and continue with step 4. (see example in the next section.)

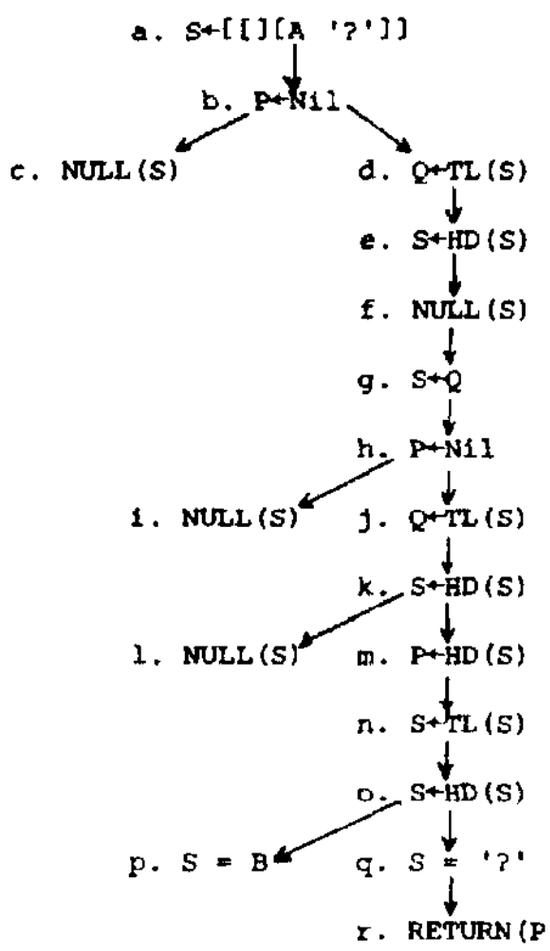
6. Replace constants in each parameter list by the variables replacing them in the substitutions. Replace the constants remaining in the lists by new variables. The variables in the same position of the argument lists of the protocols must be the same. This becomes the parameter list of the procedure. Finally, replace constants in the constructed procedure by variables in the parameter list which replaced these constants.

Modification

The procedure constructed is not the one we intended. Of course, in the way we are building procedures, one could not expect a correct procedure to be constructed from these two limited protocols.

Our task, now, is to correct this procedure, if possible, given new protocols. In this case, such a correction is possible. Let Figure 6 be the new protocol.

Figure 6 PAIRS([[][A '?']],B)



We would like to proceed as we did in the construction process - first reducing the protocol and then matching. Again, we do this by imagining that we have unwound our diagraph and have represented it by an infinite tree. We then begin as we did in construction, by reducing the protocols (treating our infinite tree as a protocol). We conclude that the start nodes are 1 and b; the node a has been added.

We then match our old procedure with our new protocol. We find that 1 and b, 2 and c, 4 and d, 5 and e, 2 and f all match, but that 3 and g do not! This indicates that something is wrong.

Consider the consequences of identifying two nodes to form a loop - we create a node which has more than two predecessors or a root node with one or more predecessors. We call such nodes critical nodes. They will form the basis of our correction process. The critical nodes in Figure 5 are: 1 - Y<-Nil, 2 - NULL(L), 3 - RETURN(Y), 9 - L = X.

Now, continuing at the point of our match where we found that 3 and g do not match, we proceed back along the matched path. We proceed until we reach a critical node; in this case, node 2. If we reach the start node without encountering a

critical node, then we fail.

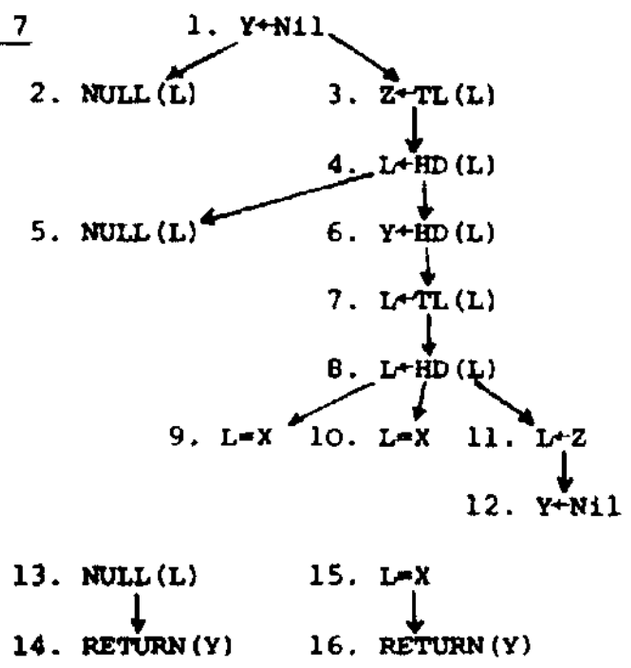
We terminate our successful matches at the predecessor of the critical node and its corresponding node in the trace. We note which node in the trace and which node of the procedure cannot be paired. In our example, these are nodes 2 and f.

Pseudo-traces

If we had kept the previous traces, then we could just use the construction procedure with the new protocol and the previous ones. In practice we would not want to retain all protocols, so this approach is not possible. However, we still have most of the information available - in the constructed procedure. All we need to do is extract it in some form to use with the construction process. This is the purpose of the pseudo-traces.

A pseudo-trace is a tree beginning at a critical node or the start node and contains all the paths from that node to a node which either has no successors or is a critical node. (This notion is somewhat simplified, but for our discussion and example it is adequate. A more complete discussion can be found in [1].) Figure 7 contains the pseudo-traces of our procedure. Again, we number the nodes for discussion purposes

Figure 7



We renumber our matched sets in terms of the numbers assigned to the nodes of the pseudo-traces. This gives us the sets {1,b), {2,c), {3,d}, {4,e} and we know that 13 and f cannot be identified. (We use 13 instead of 2 or 5 because 13 is the root of a pseudo-trace). Using the construction process on these pseudo-traces and the protocol (excluding the reduction, matching and creation of parameters) we find the following identifiable sets of nodes: {1,12,b,h}, {2,13,c,i}, {3,d,j}, {4,e,k}, {5,f,l}, {6,m}, {7,n}, {8,o}, {9,10,15, p,q}, {14,16,r}, {11,g}.

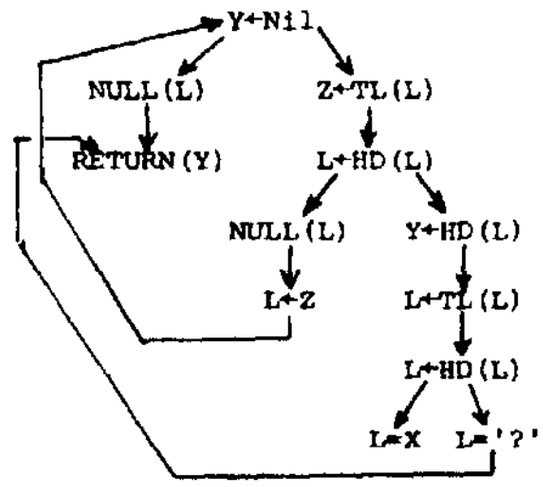
Taking each set in turn, we compute the substitutions to make the instructions labelling nodes in the same set identical. This proceeds smoothly until we try the set {9,10,15,p,q} which is the set of instructions {L=X, L=X, L=X, S=B, S = '?'}. We know that L is S by previous substitutions. This means that X must have the value of two different constants.

This implies that we have made an error in replacing a constant by a variable (which, of course, we did). We know that X is a parameter in our constructed procedure, so this analysis is reasonable. We note that B is also a parameter, and so '?' must be a constant. As a result, node 10 becomes L='?'.

We refine this set into two others, (9,15,p) and (10,q}, replace it by these two and add this new class to our set of identifiable sets, deleting the old set (this is an example of "finding the substitutions" in step 5 of the construction algorithm as well). Of course, this new set of sets will eventually be chosen (since all other sets must also refine {9,10,15,p,q} in a like manner).

Making our substitutions and identifying parameters results in the procedure of Figure 8.

PAIRS(L,X)



This process of "fixing" parameters by constants is also incorporated into the matching process of the correction procedure. In that case, once the constant has been replaced, matching continues as usual.

Summary of Correction

We briefly summarize the basic steps in the correction process. Further details and explicit descriptions can be found in [1].

1. Imagining the procedure as an infinite tree, reduce the procedure and the new protocols.

2. Match the procedure and the protocols. During the matching check variables in the parameter list of the procedure, seeing if they agree, when they occur, in usage with the constants in the same position of the argument list of each protocol. If a disagreement occurs, then replace that variable in the instruction by the constant in the same position of the protocol where the disagreement was discovered. Also replace the variable by the same constant in any substitutions formed.

If we match without finding any contradictions and each node of each protocol is paired with a node in the procedure, then halt - our traces agree with our procedure. If we find no contradictions, but some of the nodes of the traces have not been paired, then continue at step 4. Otherwise, we find a contradiction, so we continue at step 3.

3. Using the critical nodes and critical predecessors, back up the matched tree until we are sure our match is successful. Record the set of nodes we are sure match and the substitutions they imply.

4. Form the pseudo-traces of our procedure.

5. Carry out steps 3 through 6 of the construction procedure, taking care to correct any parameters if necessary.

Conclusion

The protocol abstraction and correction process described are domain independent. The form of the protocols has been rather rigid. These characteristics of the protocols are the results of the transformations we considered - the first being a result of considering only syntactic or structural transformations and the second resulting from the simplicity of the transformations.

The transformations may not seem a 'part' of the protocol abstraction problem and may seem an additional burden. However, we feel that they help to define the problem, that is, they help specify what we mean by a protocol or trace and tell us what we mean by 'abstraction' - inversion of the transformations.

This approach can be extended to other syntactic transformations (see [1]) and we feel that it can be used to include semantic transformations as well - for example, omitting "obvious" steps in a certain domain. These permit more variety among protocols and allow a larger class of protocols. Of course, the construction and correction processes must become more complex and use more information about the domain - for example, use hypotheses about 'likely' steps in the particular domain. Protocols in program synthesis, aay from a dialogue with a person, may be different from those using traces formed from solutions by a problem solver. These differences may suggest different transformations and therefore different abstraction processes. Some of these problems, as well as work on seeking additional transformations yielding broader classes of protocols and finding more general construction and correction algorithms, are currently under investigation.

References

1. M.A. Bauer, "A Basis for the Acquisition of Procedures from Protocols", Ph.D. Thesis, University of Toronto, 1975,

2. A.W. Biermann, "On the Inference of Turing Machines from Sample Computations", Computer Science Department, Stanford University, STAN-AI-152, October, 1971.

3. J. Hartmanis and R.E. Stearns, Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Inc., Englewood Cliffs,N.J., 1966.

4. C. Hewitt, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", AI-TR-258, Artificial Intelligence Laboratory, MIT, April, 1972.

5. G. J. Sussman, "A Computational Model of Skill Acquisition", AI-TR-297, Artificial Intelligence Laboratory, MIT, August, 1973.

6. p. H. Winston, "Learning Structural Descriptions from Examples", MAC-TR-76, Artificial Intelligence Laboratory, MIT, September, 1970.

*Work done while author was a student at the University of Toronto.