HOW SAIL RELATES TO A.I.

by: John Franco

DCS-TR-70

3/78

Department of Computer Sci.
The Hill Center for Mathematical Sciences
Rutgers, The State University of New Jersey
New Brunswick, New Jersey

08903

March 1978
HOW SAIL RELATES TO AI

by John Franco
CS 593
What should an AI language be able to do? In my estimation the following AI techniques should be covered by an AI language: first, general searching procedures allowing full control over the kind of search (depth first, breadthfirst, etc.), backtracking, smart backtracking, interruption of search to allow continuance of another search path; Second, problem reduction approaches hint that it may be desirable to have some easy means of creating end/or search nodes; Third, it is sometimes desirable to include a theorem prover in an AI system; Fourth, the popular notion of frames should be expressible in our language; Fifth, and most importantly, we should have some means by which we can carry out multiple concurrent processes. This last point is especially important to robotics. Consider a very simple robot scenario in which a robot turns a valve causing water to fill a bucket and then proceeds to look in his toolbox for a sponge. We see here that the two processes of filling the bucket and finding the sponge are continuing together. If the bucket overflows before the robot has found his sponge we need to know about it so we can change the state of the world (perhaps a flood results or the robot is alerted by the high level of water in the bucket and drops looking for his sponge). Alternatively, if the robot finds a sponge, he can proceed to the valve to turn off the water or he can do something else if there is enough time before the bucket overflows. These possible situations require us to monitor both processes at the same time. In general we need to monitor many processes at the same time - the number of processes not necessarily being fixed. This feature, unfortunately, is lacking in many too many AI languages, particularly the Lisp based ones.
In this paper we discuss an ALGOL based language called SAIL which attempts to satisfy our requirements. We do this by using a number of examples to illustrate the concepts and some details. The examples are simple so it should be understood that solutions to more difficult problems will be more complex because minor fixes will generally become necessary. It should be understood also that since the solutions presented are intended for illustration they are not necessarily the best for the problems they are applied to.

SAIL constructs that will be useful to us are the foreach statement, matching procedures, associational nets, lists, sets, process statements including sprout interrogate, cause and join, recursive procedures, records and contexts. We will not cover all of these but some mention will be made of each.

The foreach construct attempts to find assignments for all variables in its variable list which satisfy the conditions expressed by its element list. We may view such a statement as a theorem; if the theorem (expressed by the element list) is true then we can find an assignment of variables that makes it true. Which assignments variables may take on is decided in the element list. We may pick the variables from sets, lists or assertions that have been added to the associational data base by means of a retrieval triple. By using matching procedures in the element list we may further pick variables based on the outcomes of other theorems. Once variables have been picked they may be checked for satisfying certain conditions such as causing a function having these picked variable assignments as arguments to become true. This checking is done in one or more
matching procedures in the foreach element list. If the theorem is found to be true SAIL allows us to perform some action like making an assertion, applying another theorem or acknowledging the present theorem is true by printing something.

To illustrate the above consider the four queens problem in which we try to place four queens on a 4x4 chess board so that any one queen does not attack any other. We may represent this problem as a theorem in which we attempt to find four variables, X1, X2, X3, X4, which satisfy the condition that Xi ≠ Xj for i ≠ j, Xi + i ≠ Xj + j for i ≠ j, Xi - i ≠ Xj - j for i ≠ j, and that 1 ≤ Xi ≤ 4 for all i. If the theorem is true for a particular assignment that assignment should be printed.

Figure 1 contains a programmed version of the above statement. Notice that in line 2710 NUMBERSET is defined as the set of integers ranging from 1 to 4 (in words due to a quirk of the language). In the element list each Xi is in turn picked from NUMBERSET and tested by matching procedure fndrow (used here as a function) according to the conditions stated above. If it is accepted (fndrow succeeds) Xi+1 is picked. If it is rejected (fndrow fails for all Xi in NUMBERSET) the program backtracks to find a new value for Xi-1 from NUMBERSET. We could have chosen to pick four variables and test them all against each other at one time but the method of figure 1 is more efficient as it eliminates testing a good many combinations of assignments. Notice that fndrow not only checks the conditions of the theorem but records the succeeding placement as well in line 1300. If all queens are placed without conflict the program manages
to arrive at line 3300 to print the recorded results. Unless told to stop at this point the program attempts to find another solution to the theorem.

The same problem may be solved by using ALGOL constructs only as in figure 2. We have here a recursive procedure which prints the recorded rows when the fourth level is successfully reached. Although the recursive procedure maintains a stack it may also be thought of as maintaining a set of context frames. In this latter sense recursive procedures in SAIL are very important to search processes. For example, if alternative recursive procedures performing search processes are coroutines (it will be mentioned later how this could be) one search may be suspended with its context frame intact so that another may proceed from where it had been suspended. In this way search techniques which continue along the most favorable path can be implemented. Another example will be illustrated later.

Figure 2 brings up an important point: the ALGOL portion of SAIL may also be exploited for its efficiency and easy number, array, boolean, etc. handling advantages. The figure 2 solution took .18 cpu seconds vs. .41 cpu seconds for the figure 1 solution. The generation of all eight queen solutions took just 3.4 cpu seconds with the figure 2 solution.

Just to emphasize the efficiency of ALGOL consider the figure 3 solution to the four queens problem which is identical to the strictly ALGOL solution except that queen placement is recorded in a list, L, instead of in the array positionincolumn. Time for this
solution was .33 cpu seconds and for the eight queens solution was 17.41 cpu seconds. We are talking here about an improvement in time complexity not by just a constant multiplier. Incidentally, notice the simple list operations of put and remove; this easy readability is omnipresent in SAIL.

The most important concept in SAIL is that of concurrent processes or simply processes. We may think of the creation of a process as the incipience of the need to accomplish the task defined by the process itself. The sprout statement creates new processes. Since SAIL is implemented only on sequential machines only one process is really active at any one time. Therefore, when a process is created some rescheduling of processes is in order. This is controlled by the third field of the sprout statement. The user may activate the new process, continue the old process, assign priority to the new process and much more.

When the process has accomplished its goal it needs to report its result. On the opposite side of the coin if a result is needed by a process but the result may be achieved by any of a number of processes, the wanting process must put out a call to all concerned about that need. The last two sentences represent the concept of interprocess communication as employed in SAIL. The two constructs which carry out these transactions are the cause statement which is used to report a result (not necessarily to a calling process, however, since some results may be by-products of the result and therefore may be required by other processes either at the time the report is made or later) and the interrogate statement which is used
to request some result or results. When a process has finished its task it can be terminated (and should be terminated) by the terminate or return (in which case execution is passed to the calling process) statements.

Suppose we wish to solve the four queens problem using a breadth first search. Such a solution appears in figure 4. At the start of execution we may sprout new processes which are assigned the task of finding all solutions for one specific choice of row in the first column. There are four such choices so four processes will be sprouted. Each of these processes will sprout four new processes which have the task of finding all solutions for one specific choice of row in the second column given the first column choice. The sprouting continues until the fourth level is reached whereupon all valid assignments are printed and the associated processes terminated (line 2800). On the way to the fourth level, to keep efficiency high, a process may be terminated if it fails the consistency conditions stated earlier by using a terminate(myproc) statement (line 2300). The new processes are sprouted in lines 3100 to 3400. Since we are operating breadth first we want to sprout all four processes before we consider completing the first one. This is what RUNME in the third field of the sprout statements does. The join statement (line 3500) tells the sprouting process to wait until all the sprouted processes are terminated (in this case either by consistency failure or successful assignment). When all the fourth level processes are completed (and terminated) the third level processes are caused to be terminated and so on until the main process, the one that started the problem, terminates and execution stops. Note that this is another
example of a recursive procedure in which backtracking does not occur but in which a set of context frames was maintained.

In the above example we used RUNME to control the scheduler. To appreciate the ease with which the scheduler may be controlled see figure 5 - the depth first solution using sprouted processes. Note the solution is almost the same as in figure 4. The third field of the sprout statement is missing. This means run the sprouted process first. The join statements are not necessary but are included to emphasize that the sprouting process waits for the "leftmost" process to finish before proceeding. Incidentally, this solution took .44 cpu seconds to complete the four queens problem which isn't bad relative to other SAIL and ALGOL contracts. The breadth first solution, with all the extra memory it requires, took .51 cpu seconds.

We now consider some other AI problems made easy by using process statements. An AND node in an AND-OR tree may be written in a way similar to the breadth first problem. Suppose we wish to join two boards with nails. We can sprout processes which get a hammer to the right hand, a nail to the left hand, and the two boards on a workbench. When all three processes are completed the nail may then be hammered into the boards. Hence there are three sprout statements (for the hammer, nail and boards) followed by a join statement (join all three processes) followed by the action of pounding the nails into the boards.
An OR node is slightly more difficult (figure 6). Suppose the boards may be nailed, glued or screwed together. Three processes may be sprouted which report whether each task is ready to be performed (have the materials been obtained?...). The report is transmitted by means of a cause statement. The first one to report ready represents the method chosen to join the boards. The board-joining process waits for this report through an interrogate statement.

Let us now return to our robot problem. To model this problem we need to know how long, in real time, the fill-bucket process needs to complete its task. Some means of keeping track of process real time is therefore necessary. SAIL glaringly has no such facility. It is not difficult to suppose how this could be implemented especially since so many other languages, particularly ALGOL based SIMULA, include a scheduler that accounts for process delays. In the spirit of SAIL the fill-bucket process could first calculate the time at which the bucket should become full based on the volume of the bucket and the rate at which water is flowing into the bucket (determined by the setting of the valve). The fill-bucket process could then suspend itself at high priority perhaps using SUSPME with an argument equal to the difference of the calculated time and the present time. When it comes time to awaken this process it immediately becomes active due to its high priority unless it had been cancelled earlier (perhaps by the robot shutting off the valve) in which case no flood is caused. Notice that the robot may or may not have found his sponge since the two processes are independent. In fact, while looking for the sponge the robot may have been warned (by the fill-bucket process) that the bucket was near full and
thereby may have dropped what he was doing.

We now discuss the implementation of frames and smart backtracking in SAIL. We do this with the help of the familiar missionary-cannibals example (figure 7). In this simple problem we keep a frame for each person and the rowboat since these entities cross the river and therefore change their frames of reference depending on the state of the problem. The person frame consists of five fields. These are the type field (missionary, cannibal, blank), the name field (miss1, miss2, ...), the side (of the river) field (1 is the start side, 0 is the goal side), the ismissionary field (1 if this person is a missionary otherwise 0), and the iscannibal field (1 is this person is a cannibal otherwise 0). These frames are constructed in SAIL by means of records. Each created frame is pointed to by an item defined in line 1200. The frames are created in lines 11400 to 12300 (blank frames which represent vacant seats are used for convenience and will not be discussed). The frames are given initial values in lines 12600 to 18600. The numberofmissionaries and numberofcannibals matrices are also defined for convenience. These tell how many of each type are on each side.

The actual river crossing takes place in lines 9110 to 10900. First, the positions of all persons are printed. Then, a person is chosen from the set of 10 people and another from the remaining 9. If the preconditions for crossing and postconditions after crossing are met the chosen people may cross the river by means of a call to procedure takeaction.
A close look at the precondition, postcondition and takeaction procedures demonstrates the use of the created records. Matching procedure "preconditions" succeeds if the following conditions are met: the two chosen persons must be on the same side as the rowboat (lines 3800, 3900), the two chosen persons must not be of the same type as the two that made the last crossing (lines 4000 to 4300), either the new number of missionaries on the boat side is zero (lines 4400 to 4600) or the new number of missionaries is greater than or equal to the new number of cannibals on the boat side. The reader may examine "postconditions" and "takeaction" on his own. Note that it is very easy to cross the river using the 1,0 representation of sides. The fact that numbers can be used with no effort can be a great help in much of the problem solving we wish to do.

The missionary-cannibal solution can be made much more efficient than presented above if river crossers are picked from a set of possibilities constructed to rule out redundant choices. In this problem redundant choices are easy to define; in each group of two persons to be tested for consistency, if the first person chosen is X and the second person Y then the type of X should not be the same as the type of any previous X picked on the current crossing and the type of Y should not be the same as the type of any previous Y. This means the possibilities sets of X and Y should be adjusted after each pair is picked. This may be accomplished in SAIL by means of matching procedures "adjustpossibilitiesx" and "adjustpossibilitiesy" as seen in figure 8. This constant revision of the possibilities sets to eliminate redundant choices is what we refer to as smart backtracking. In this example the adjusting procedures use text
arrays to save the tried types of person. Each new pick is tested against the array entries. The procedures succeed if the type had not been recorded. Upon succeeding at the adjusting procedures execution continues to check on preconditions and postconditions as before. A failure in an adjusting procedure causes a new pick of two persons from the possibilities sets. The possibilities sets and the save arrays are reinitialized at the outset of each new river crossing.

Before wrapping up we present, as promised, coroutine implementation in SAIL. The two constructs that apply here are the suspend and the resume statements. When a coroutine needs to call another coroutine it resumes the called routine to ready status (presumably the called coroutine is presently suspended). It then suspends itself. When it is resumed execution begins at the first statement following the suspend. One way communication between the coroutines is possible at the time of resume-suspend. The number of coroutines interacting with each other is not limited to two; the suspend and resume statements take on arguments which name the process to be suspended or resumed.

This paper has presented a flavor of how SAIL may be used to solve some problems considered to be in the sphere of AI. Although SAIL is apparently a useful AI language due to the large number of AI techniques it covers, its relative efficiency and its readability, it does fall short of being a godsend. Consider the robot problem again. Suppose the robot turns the valve slowly and erratically. The precise level of water in the bucket may be found at time t by
integrating the function that describes the valve opening from the initial time to time \( t \) (the initial level of fluid and the precise contents are presumed kept in the bucket's frame). This requires continuous communication between the turn-valve process and the fill-bucket process. Continuous communication between processes is impossible in single processor machines on which SAIL is implemented. Satisfactory results may be possible by approximating the communicated function over a small real time interval. This approach, however, creates two problems: the required high frequency of communication between processes tends to eat up time (there may be many approximately continuously communicating processes active at one time) and it is difficult to judge, sometimes, how small the approximating interval should be set so that errors introduced to the approximation will not have deleterious effects on the model. One may be tempted to choose a very coarse approximation and not be too concerned with the resulting error since the level of the bucket may be satisfactorily described as empty, half full, full or whatever. This approach may be all right the first time the bucket is filled or emptied but repeated fillings will tend to accumulate the error until at some point the statement expressing the level will be in error. It is not hard to find other processes to which these observations apply.

Other problems with SAIL are the usual space limitation problem which makes AI problems other than illustrative examples difficult to implement and the single processor implementation which although successful at simulating concurrent processes fails to produce a system speedy enough for many AI problems.
```plaintext
00001 /* Solution of the four queens problem in theorem prover style */
00010 begin "queen"
00020 integer item one,two,three,four;
00030 integer array positionincolumn[1:4]; integer ii;
00040 integer itemvar X1,X2,X3,X4; set NUMBERS;  
00050
00060 matching recursive procedure findrow(integer itemvar ROW;integer COL);
00070 begin
00080   integer i;
00090   for i = 1 step 1 until COL-1 do
00100     if positionincolumn[i]=datum(ROW) or
00110       positionincolumn[i]+i=datum(ROW)+COL or
00120       positionincolumn[i]-i=datum(ROW)-COL then fail;
00130     positionincolumn[COL] := datum(ROW);
00140     succeed;
00150   end;
00160
00170 datum(one) := 1;
00180 datum(two) := 2;
00190 datum(three) := 3;
00200 datum(four) := 4;
00210 NUMBERS := {one,two,three,four};
00220
00230 foreach X1,X2,X3,X4 such that
00240   X1 in NUMBERS and findrow(X1,1) and
00250   X2 in NUMBERS and findrow(X2,2) and
00260   X3 in NUMBERS and findrow(X3,3) and
00270   X4 in NUMBERS and findrow(X4,4) do begin
00280     for ii = 1 step 1 until 4 do print(positionincolumn[ii]);
00290     print(" ");
00300   end;
00310 end;
```
00003  /* Solution of four queens problem in "depth first" style */
00004  /* Note that one need only change line 3400 for the solution */
00005  /* on any size chess board */

00100  begin "queen2"
00200    integer array positionInColumn[1:8]; integer k;
00400  recursive procedure queen(integer N);
00500    begin
00600      integer i,j,ii; label loop;
00700      if N=0 then begin
00800        for ii _ 1 step 1 until k do print(positionInColumn[ii]);
00900        print(" ");
01000      end
01100      else begin
01200        for i _ 1 step 1 until k do begin
01300          for j _ k step -1 until N+1 do
01400            if positionInColumn[j]=i or
01500              positionInColumn[j]+j=N+1 or
01600                positionInColumn[j]-j=1-N
01700                  then go to loop;
01800            positionInColumn[N] __ i;
01900            queen(N-1);
02000        end
02100      end;
02200  end;
03400  k __ 4;
03500  queen(k);
03600  end;

Figure 2
/** Solution of the four queens problem in "depth first" style **/
/** using list L as a stack structure to record queen placement **/

begin "queen3"
require 8 new!items;
integer k; list L;

recursive procedure queen(integer N);
begin
integer i; integer array itemvar X; label loop;
integer array itemvar Y; integer array Q[1:2];

if N=0 then begin
foreach X such that X in L do print(datum(X)[2]);
print(" ");
end
else
for i = 1 step 1 until k do begin
foreach X such that X in L do
if datum(X)[2]=i or
datum(X)[1]+datum(X)[2]=N+i or
datum(X)[1]-datum(X)[2]=N-i then go to loop;
Y = new(Q);
datum(Y)[1] = N;
datum(Y)[2] = i;
put Y in L before 1;
queen(N-1);
remove 1 from L;
delete(Y);

loop;
end;
end;

k = 8;
queen(k);

end;

figure 3
begin "queen4"
integer RUNME; integer array posi[0:4];
require 400 newItems;
recursive procedure breadthfirst(integer COL; integer ROW; integer array position);
begin
integer ii; integer array positionincolumn[0:4];
itemvar P1,P2,P3,P4;
for ii = 0 step 1 until 4 do positionincolumn[ii] = position[ii];
for ii = 1 step 1 until COL-1 do
  if positionincolumn[ii]=ROW or
    positionincolumn[ii]+ii=ROW+COL then terminate(myproc);
positionincolumn[COL] = ROW;
for ii = 1 step 1 until 4 do print(positionincolumn[ii]);
print(" ");
terminate(myproc);
end:
P1 = new; P2 = new; P3 = new; P4 = new;
sprout(P1,breadthfirst(COL+1,1,positionincolumn),RUNME);
sprout(P2,breadthfirst(COL+1,2,positionincolumn),RUNME);
sprout(P3,breadthfirst(COL+1,3,positionincolumn),RUNME);
sprout(P4,breadthfirst(COL+1,4,positionincolumn),RUNME);
join({P1,P2,P3,P4});
end;
RUNME = 1;
breadthfirst(0,0,posi);
end;
begin "queen5"
thickness 0.4
begin
integer RUNME; integer array posi[0:4];
require 400 new items;

recursive procedure depthfirst(integer COL; integer ROW;
integer array position);

begin
integer ii; integer array positionincolumn[0:4];
itemvar P1,P2,P3,P4;
for ii _ 0 step 1 until 4 do positionincolumn[ii] _ position[ii];
for ii _ 1 step 1 until COL-1 do
  if positionincolumn[ii]=ROW or
     positionincolumn[ii]-ii=ROW-COL or
     positionincolumn[ii]+ii=ROW+COL then return;
positionincolumn[COL] _ ROW;
if COL=4 then begin
  for ii _ 1 step 1 until 4 do print(positionincolumn[ii]);
  print(" ");
  return;
end;
P1 _ new; P2 _ new; P3 _ new; P4 _ new;
sprout(P1,depthfirst(COL+1,1,positionincolumn));
join([P1]);
sprout(P2,depthfirst(COL+1,2,positionincolumn));
join([P2]);
sprout(P3,depthfirst(COL+1,3,positionincolumn));
join([P3]);
sprout(P4,depthfirst(COL+1,4,positionincolumn));
join([P4]);
end;
RUNME _ 1;
depthfirst(0,0,position);
end;
SPROUT(p1,grab(righthand,hammer));
SPROUT(p2,grab(lefthand,hammer));
SPROUT(p3,on(workbench,boards));

JOIN({p1,p2,p3});
POUND(hammer,nail,boards);

An example of an AND node in SAIL

SPROUT(p1,nail(sueevent,boards));
SPROUT(p2,glue(sueevent,boards));
SPROUT(p3,screw(sueevent,boards));
winner = interrogate(sueevent,WAIT);
FOREACH p SUCH THAT p IN {p1,p2,p3} AND
   NOT p=winner DO TERMINATE(p);

CAUSE(sueevent,THIS_PROCESS);

An example of an OR node in SAIL

Figure 6
begin "me"

repeatclass person (string type,name; integer side,ismissionary,iscannibal);
repeatclass river (integer side);
record-pointer (river) rowboat;
record-pointer (person) item M1,M2,M3,C1,C2,C3,B1,B2,B3,B4;
set people,redset;
integer array numbermissionaries[0:1],numbercannibals[0:1];

class recursive procedure crosriver(string lastx,string lasty);
begin

integer totalmissionarycross,totalcannibalcross,n,j;
record-pointer (person) itemvar x,y,z;

procedure printpersonpositions;
begin
integer n;

n = 0;
foreach Z such that Z in {M1,M2,M3,C1,C2,C3} and
(person:side[datum(Z)]=1) do begin
print(person:name[datum(Z)]," ");n = n+1;end;
for j = n+1 step 1 until 5 do print(" ");
print(" ");
foreach Z such that Z in {M1,M2,M3,C1,C2,C3} and
(person:side[datum(Z)]=0) do
print(person:name[datum(Z)]," ");
outstr('12&'15);
end;

matching procedure preconditions;
begin
if (person:side[datum(X)]=river:side[rowboat]) and
(person:side[datum(Y)]=river:side[rowboat]) and
(not((person:type[datum(X)]=lastx and
person:type[datum(Y)]=lasty) or
(person:type[datum(X)]=lasty and
person:type[datum(Y)]=lastx) ) ) and
(numbermissionaries[river:side[rowboat]] -

person:ismissionary[datum(X)] -

person:ismissionary[datum(Y)] = 0 or
numbercannibals[river:side[rowboat]] -

person:iscannibal[datum(X)] -

person:iscannibal[datum(Y)] geq

numbermissionaries[1-river:side[rowboat]] +

person:ismissionary[datum(X)] +

person:ismissionary[datum(Y)] geq

numbercannibals[1-river:side[rowboat]] +

person:iscannibal[datum(X)] +

person:iscannibal[datum(Y)] or

numbermissionaries[-river:side[rowboat]] +

person:ismissionary[datum(X)] +

person:ismissionary[datum(Y)] = 0 )
then succeed else fail;
end;

matching procedure postconditions;
begin
if (numbermissionaries[-river:side[rowboat]] +

person:ismissionary[datum(X)] +

person:ismissionary[datum(Y)] geq

numbercannibals[-river:side[rowboat]] +

person:iscannibal[datum(X)] +

person:iscannibal[datum(Y)] or

numbermissionaries[-river:side[rowboat]] +

person:ismissionary[datum(X)] +

person:ismissionary[datum(Y)] = 0 )
then succeed else fail;
end;
procedure takeaction;
begin
  numbermissionaries[river:side[rowboat]]_=
    numbermissionaries[river:side[rowboat]]-
    totalmissionarycross;
  numbermissionaries[1-river:side[rowboat]]_=
    numbermissionaries[1-river:side[rowboat]]+
    totalmissionarycross;
  numbercannibals[river:side[rowboat]]_=
    numbercannibals[river:side[rowboat]]-
    totalcannibalcross;
  numbercannibals[1-river:side[rowboat]]_=
    numbercannibals[1-river:side[rowboat]]+
    totalcannibalcross;
  person:side[datum(X)]_1-person:side[datum(X)];
  person:side[datum(Y)]_1-person:side[datum(Y)];
  river:side[rowboat]_1-river:side[rowboat];
end;
if numbermissionaries[0]+numbercannibals[0]=6 then return;
printpersonpositions;
foreach X such that X in people do begin
  reset _ (people-{X});
  foreach Y such that Y in redset and
  postconditions do begin
    totalmissionarycross_
    person:ismissionary[datum(X)] +
    person:ismissionary[datum(Y)];
    totalcannibalcross_
    person:iscannibal[datum(X)] +
    person:iscannibal[datum(Y)];
takeaction;
crossriver(person:type[datum(X)],person:type[datum(Y)]);
takeaction;
end;
end;
people _ {M1,M2,M3,C1,C2,C3,B1,B2,B3,B4};
datum(M1) _ newrecord(person);
datum(M2) _ newrecord(person);
datum(M3) _ newrecord(person);
datum(C1) _ newrecord(person);
datum(C2) _ newrecord(person);
datum(C3) _ newrecord(person);
datum(B1) _ newrecord(person);
datum(B2) _ newrecord(person);
datum(B3) _ newrecord(person);
datum(B4) _ newrecord(person);
rowboat = new record (river);
river: side [rowboat] = 1;

person: type [datum (M1)] = "missionary";
person: name [datum (M1)] = "Miss1";
person: side [datum (M1)] = 1;
person: ismissionary [datum (M1)] = 1;
person: iscannibal [datum (M1)] = 0;

person: type [datum (M2)] = "missionary";
person: name [datum (M2)] = "Miss2";
person: side [datum (M2)] = 1;
person: ismissionary [datum (M2)] = 1;
person: iscannibal [datum (M2)] = 0;

person: type [datum (M3)] = "missionary";
person: name [datum (M3)] = "Miss3";
person: side [datum (M3)] = 1;
person: ismissionary [datum (M3)] = 1;
person: iscannibal [datum (M3)] = 0;

person: type [datum (C1)] = "cannibal";
person: name [datum (C1)] = "Cann1";
person: side [datum (C1)] = 1;
person: ismissionary [datum (C1)] = 0;
person: iscannibal [datum (C1)] = 1;

person: type [datum (C2)] = "cannibal";
person: name [datum (C2)] = "Cann2";
person: side [datum (C2)] = 1;
person: ismissionary [datum (C2)] = 0;
person: iscannibal [datum (C2)] = 1;

person: type [datum (C3)] = "cannibal";
person: name [datum (C3)] = "Cann3";
person: side [datum (C3)] = 1;
person: ismissionary [datum (C3)] = 0;
person: iscannibal [datum (C3)] = 1;

person: type [datum (B1)] = "\n";
person: name [datum (B1)] = "Blank";
person: side [datum (B1)] = 0;
person: ismissionary [datum (B1)] = 0;
person: iscannibal [datum (B1)] = 0;

person: type [datum (B2)] = "\n";
person: name [datum (B2)] = "Blank";
person: side [datum (B2)] = 0;
person: ismissionary [datum (B2)] = 0;
person: iscannibal [datum (B2)] = 0;

person: type [datum (B3)] = "\n";
person: name [datum (B3)] = "Blank";
person: side [datum (B3)] = 0;
person: ismissionary [datum (B3)] = 0;
person: iscannibal [datum (B3)] = 0;

person: type [datum (B4)] = "\n";
person: name [datum (B4)] = "Blank";
person: side [datum (B4)] = 0;
person: ismissionary [datum (B4)] = 0;
person: iscannibal [datum (B4)] = 0;
18800  numbermissionaries[1] _ 3;
18900  numbermissionaries[0] _ 0;
19000  numbercannibals[1] _ 3;
19100  numbercannibals[0] _ 0;
19200
crossriver(" "," ");
19400  end;

figure 7
/** Solution to missionary-cannibals problem using existing features **/
/** to provide for smart backtracking **/

{first section of program}

matching procedure adjustpossibilities;
begin
integer j;

j _ 1;
while j leq 3 and not savetype[j] = "d" do begin
if savetype[j] = person:typ[datum(Y)] then fail;

j _ j+1;
end;
savetype[j] _ person:typ[datum(Y)];
succeed;
end;

matching procedure adjustpossibilitiesx;
begin
integer j;

j _ 1;
while j leq 3 and not savetype[x]="d" do begin
if savetype[x]=person:typ[datum(X)] then fail;

j _ j+1;
end;
savetype[x] _ person:typ[datum(X)];
succeed;
end;

if numbermissionaries[0]+numbercannibals[0]=6 then return;

printpersonpositions;

possibilitiesx _ PHI;
foreach X such that X in people and
(person:side[datum(X)]=river:side[rowboat]) do
put X in possibilitiesx;

foreach X such that X in possibilitiesx and
adjutpossibilitiesx do begin
possibilities _ (possibilitiesx-X);
foreach Y such that Y in possibilitiesx and
adjutpossibilitiesy and
preconditions and
postconditions do begin

end;

crossriver(person:typ[datum(X)], person:typc[datum(Y)]);
takeaction;

crossriver(person:typ[datum(X)], person:typc[datum(Y)]);
takeaction;

end;