TR-79-024
Department of Computer Science
University of North Carolina

Syntax Directed Symbolic Execution

Carlo Ghezzi* and Mehdi Jazayeri
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina 27514,
USA

---

*On sabbatical leave from Istituto di Elettrotecnica ed Elettronica, Politecnico di Milano, Milan, ITALY

Syntax Directed Symbolic Execution

Abstract

A syntax directed formulation of symbolic execution is presented. The purpose of the presentation is to clarify the issues involved in the implementation of a symbolic execution system.

Key words and phrases: Symbolic execution; program testing; programming systems; syntax directed translation; software reliability

1. INTRODUCTION

Program testing is an important area of software engineering which has received a great deal of deserved attention recently (1). It appears that some combination of techniques from the program verification and the program testing areas will eventually emerge as the method of choice for software validation. One promising program testing method is based on symbolic execution (2).

The approach is founded on the assumption that it is impossible to test a program exhaustively on all possible data. One can, however, run the program with symbolic values for the input data and produce symbolic values for output data. In executing a program symbolically, one also derives symbolic expressions for conditions that ensure the execution of different paths in the program. The results of symbolic execution may be used in many applications including source-level debugging, test data generation, program documentation, proofs of correctness and detection of semantic errors that cannot be found statically.

Many approaches to symbolic execution have been proposed and several systems exist that are capable of executing programs symbolically (3-5). The purpose of this paper is to provide a formal description of symbolic execution. We present the top-down development of a symbolic execution system and

point out the important design decisions that distinguish the existing approaches from one another. The purpose of this paper is thus not to present a new testing methodology but rather to provide a unified framework for understanding symbolic execution and for comparing the different techniques.

The major characteristic of the formulation of symbolic execution presented here is that it is syntax directed (6). There are many reasons for choosing this approach. One is that tools developed this way can be easily modified to work for different programming languages. This is precisely the attraction of syntax directed compilation. Another and more important reason is that we believe it is important to have a representation of programs which is suitable for different program processing tools such as editors, translators, debuggers, optimizers, verifiers, etc. Syntax trees can serve as this unifying representation if all the tools are syntax directed (7-8).

The remainder of this paper is organized as follows. In the next section we discuss the major components of a symbolic execution system. In section 3 we give our syntax directed formulation of symbolic execution. Section 4 contains three examples to sharpen the ideas presented and section 5 offers some conclusions based on this work.

2. COMPONENTS OF A SYMBOLIC EXECUTION SYSTEM

Figure 1 illustrates the major components of a symbolic execution system. We have included the user as a component in order to emphasize the interactive nature of this system. It is the user who will direct the courses of action to be taken. In fact the formulation we present would often not terminate if the user were not present.

Symbolic evaluator is the component which scans the program and executes each statement. The size and complexity of expressions (boolean and otherwise) that the evaluator is required to evaluate grow quite rapidly. It is for this reason that a simplifier module would be helpful. This in itself could be as powerful a system as MACSYMA. In the absence of such a sophisticated simplifier, this is another place where the user can be relied upon to perform the simplifications. Our emphasis in this paper is on the symbolic evaluator component. We assume the existence of a simplifier.

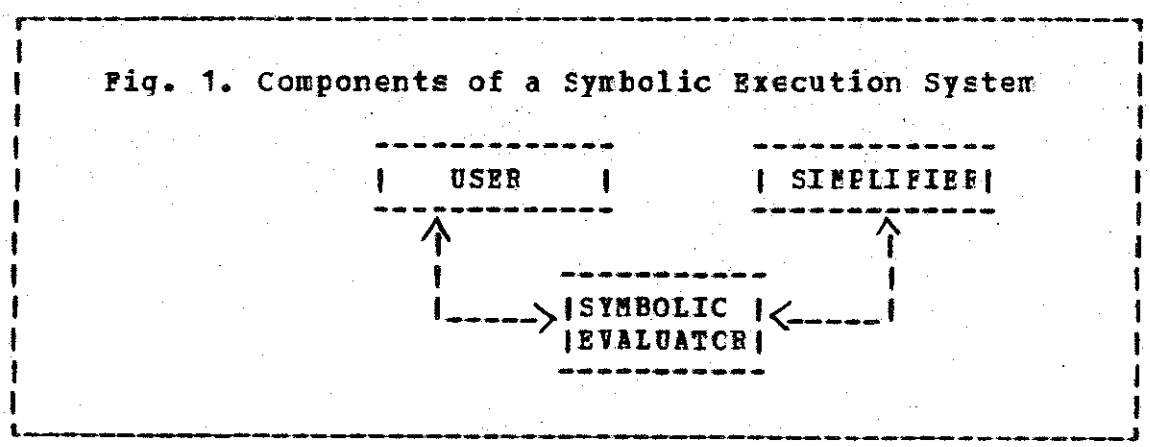


Fig. 1. Components of a Symbolic Execution System

3. SYNTAX DIRECTED FORMULATION OF SYMBOLIC EXECUTION

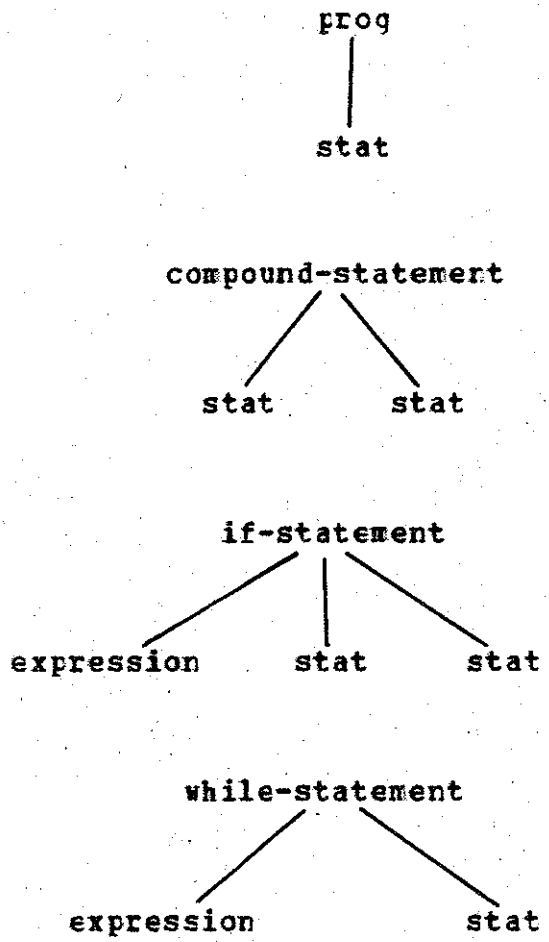
In this section we present in detail the workings of the Symbolic Evaluator. Since the system is syntax directed we have to assume a syntax for the language. Figure 2 gives the grammar that is assumed by the algorithms which follow. Some rules are left unspecified for simplicity. It is really immaterial what these rules are; they do not affect our results. Figure 3 gives the syntax trees corresponding to the first four productions of the grammar. The ambiguity of the grammar does not affect the results preserted here; we can use any of possibly many syntax trees generated for a program. Figure 4 gives an example program and its syntax

Fig. 2. Fragment of a sample grammar

```
1 prog -> statement
2 statement -> statement ; statement
3 statement -> if expression then statement
                                  else statement
                fi
4 statement -> while expression
                 do statement od
5 statement -> assignment-statement
6 statement -> read-statement
7 statement -> write-statement
```

tree.

Fig. 3. Syntax trees for the example language



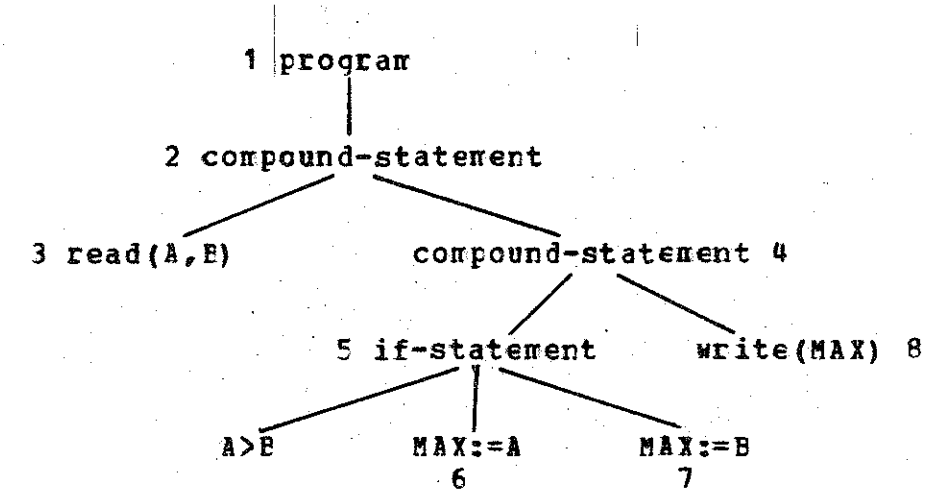
We next discuss the structure of the evaluator.

3.1 SYNTAX DIRECTED SYMBOLIC EVALUATOR

The description here emphasizes the conceptual organization of the evaluator rather than a precise and detailed implementation. We make extensive use of data and procedural abstraction.

Fig. 4. Sample program and syntax tree

```
read(A,B);
if A>B
then MAX:= A
else MAX:= B
fi;
write(MAX)
```



The symbolic evaluator is a multipass algorithm, each pass being a recursive traversal of the syntax-tree. When a node is visited at each pass, the evaluator performs actions which depend on the statements described by that node.
Each node N of the syntax tree has a number of attributes:

- type, which indicates the syntactic construct represented by the node (compound-statement, while-statement, etc.);

- inherited environment, which represents the symbolic environments inherited from the statements that in some computations are executed before the construct represented by N;

- synthesized environment, which represents the symbolic environments which hold after the execution of the construct represented by N.

To specify these attributes, the record notation N.type, N.inh and N.syn will be used in the paper. In an implementation, each node N might be accessible through a pointer P. Similarly, P.inh (P.syn) can be a pointer to the inherited (synthesized) environment and assignment of environments can be implemented as pointer assignments.

An environment is represented by a set of pairs; the elements of a pair are a boolean expression (called a constraint) and a table of program variables and their symbolic values. The constraint specifies the condition under which a particular control path is executed and the table contains symbolic expressions for all the variables assigned along that path. The environment contains this information about all control paths thus far examined. A constraint that is identically false indicates that the associated path cannot be traversed in any computation; a true constraint, on the other hand, indicates that the path is always traversed (for example, the first statement of a program usually has a true constraint). Finally, an environment that is empty indicates that nothing has been computed. After each pass is executed, the synthesized environment for the root of the tree represents the result of symbolic execution of a number of program control paths.

The following procedure, multipass, performs symbolic execution by a multiple scan of the tree.

```
procedure multipass;
    for all nodes N do
            N.inh <- empty
    od;
    let R be the root of the syntax tree;
        R.inh <- {(T,empty)};
    while the user requests a new pass do call pass(R) od
end multipass
```

Procedure pass which performs the actual scanning of the tree, produces the result of each pass for the user.

```
procedure pass(R:node);
     let S be R's son;
     S.inh <- R.inh ∪ S.inh;
     call process (S); {symbolically execute the statements
          which derive from S using S.inh as the inherited environment};
     R.syn <- S.syn;
     display R.syn to the user;
     R.inh <- empty; S.syn <- empty
        {These assignments are in preparation for the next pass}
end pass
```

Procedure process calls on a specialized procedure depending on the statement type.

```
procedure process(N:node);
    case N.type of
     compound-statement: call compound-statement(N);
     if-statement: call if-statement (N);
     while-statement: call while-statement (N);
     assignment: call assignment-statement(N);
     read-statement: call read-statement(N);
     write-statement: call write-statement(N)
    esac
end process
```

The actual processing of statements is performed by the specialized procedures of which there is one for each statement (node) type. The following procedures give the flavor of these procedures. It is this set of procedures that differs when a different language is to be treated.

```
procedure compound-statement (CS:node);
    let S1,S2 be the left and right sons of CS respectively;
    S1.inh <- CS.inh ∪ S1.inh;
    call process (S1);
    S2.inh <- S1.syn ∪ S2.inh;
    call process (S2);
    CS.syn <- S2.syn;
    CS.inh <- empty; S1.syn <- empty; S2.syn <- empty
 end compound-statement

procedure if-statement (IS:node);
       let E be the expression which appears in the condition
       let S1, S2 be the nodes corresponding to the true and the
         false branch, respectively;
       S1.inh <- combine (IS.inh,E) ∪ S1.inh;
       S2.inh <- combine (IS.inh,not E) ∪ S2.inh;
         {function combine is described below}
       call process (S1);
       call process (S2);
       IS.syn <- S1.syn ∪ S2.syn;
       IS.inh <- empty; S1.syn <- empty; S2.syn <- empty
end if-statement
```

All inherited attributes are initialized to empty to indicate that nothing has been evaluated yet. During each pass all the alternative branches of condition statements are symbolically executed. Therefore, one single pass performs a symbolic execution of all paths if the program is loop free.

```
procedure while-statement(WS:node);
        let E be the expression which appears in the condition;
        let S be the son of WS corresponding to the loop body;
        S.inh <- combine(WS.inh,E) ∪ S.inh;
        call process (S);
        WS.syn <- combine(WS.inh,not E)
        WS.inh <- S.syn;  S.syn <- empty
end while-statement
```

In a while-statement node, WS, we store the effect of going through the loop one more time in WS.inh; WS.syn is for the case when the condition is false and the loop is not iterated. Thus after the first pass, WS.syn carries the environment indicating no loop executions at all. WS.inh holds the environment for exactly one loop iteration.

```
procedure assignment-statement (AS:node);
     AS.syn <- AS.inh;
     for each pair in AS.syn do
        if there is an entry in pair.table for the
            variable being assigned
        then update its symbolic value
        else create an entry for the variable and
            initialize its symbolic value
     fi
     od;
     AS.inh <- empty
end assignment-statement
```

The above procedures are for typical statement types. One other statement deserves mention: the read-statement. The reading of a variable can be treated just like an assignment to the variable except that the value to be assigned has to be manufactured by the system. For each "read X" statement, the value assigned will be xi where i-1 is the number of times a value has been read into X prior to the execution of this read statement. Thus x1, x2,... will be the successive values assigned to X by successive read statements.

And finally, the function combine which has been used in the above procedures for manipulating constraints is given below.

```
function combine (ENV:environment, EXP:expression):environment;
   for each pair in ENV do
          X <- symbolic evaluation of EXP using symbolic
               values of variables stored in pair.table;
        pair.constraint <- pair.constraint and X
   od;
   return (ENV)
end combine
```

3.2 ANALYSIS AND REFINEMENTS

The approach to symbolic execution presented above is meant to be straightforward and easily-understood. It is not meant to be efficient and it is not. In this section we discuss some important aspects of the above formulation and some refinements of it that can affect the efficiency of an implementation. We also contrast and compare the different approaches to symbolic execution.

The first factor which contributes to the efficiency of the above approach is the degree of interaction with the simplifier. In principle, the simplifier can be invoked only after each pass, in order to discard any pairs in the synthesized environment of the root that have false constraints. However, a considerable reduction of processing time and storage space in the management of environments can be

gained if the simplifier is called after each time that function combine is invoked. We call a pair whose corstraint has the value false an infeasible computation, since the condition that would enable the traversal of the control path cannot be satisfied. Symbolically executing the program with such environments is wasteful and can be avoided if a simplifier is used. The user can also provide this service interactively if a powerful simplifier is not available.

A second way to increase the efficiency of the evaluator is to avoid visiting every node at every pass, as the present evaluator would do even if processing certain nodes can be known not to yield any new information. In some cases, the symbolic execution of a pass may assure that symbolically executing certain nodes in the next pass will not synthesize any new environments (i.e. the synthesized environment will be empty). In particular, if the program does not contain any loops, every pass after the first will fail to produce any new information since the synthesized environment of the root will be empty. This, of course, is the direct result of the fact that in the first pass all the control paths will have been examined.

As given here, the multipass algorithm can only halt as a consequence of a user command. This reflects the intrinsi-

cally non-terminating process of symbolically executing programs in general. It is reasonable, however, to try to automatically stop the process if it can be guranteed that more passes will not produce any new information. This would happen after all program paths have been examined. Although in general this condition cannot be expected to occur, it will occur for programs without loops (after one pass) and for programs with only indexed loops. In other words, the process is indeed terminating for programs which have only a finite number of control paths.

This can, in fact, be considered to be a special case of the general problem which we call redundant computations. One could imagine cases where one more pass would produce new information, but not for all control paths. That is, in general, one would like to avoid processing those control paths for which no new information will result. As a special case, this would avoid an entire pass once all paths have been exhausted.

To accomplish this, we could use a global variable ACTIVE, as well as a boolean-valued attribute active at each node. The purpose of a node's active attribute is to indicate whether further processing of the node should be attempted (true value) or not (false value). The purpose of the global ACTIVE is to mark the nodes' active attributes cor-

rectly. Procedure process is modified to test a node's active attribute and "process" the node only if it is active. Before the first pass, multipass would initialize all active attributes to true and the global ACTIVE to false. During each pass, after the synthesized environment of a node is computed, the current value of ACTIVE is assigned to the node's active attribute. Just after the call to procedure process in procedure while-statement, ACTIVE is set to true if S.syn is not empty. The descendants of a while node are also marked as active if the node is so marked. To start any pass after the first, ACTIVE is set to false.

A further optimization can be effected in the way the descendants of a while node are marked active. The straightforward way is to simply rescan all such nodes and mark them. A less inefficient way is to have a global variable GO. The decision whether to process a node N is now (GC or active). During a pass, whenever an active while node is encountered, GC is set to true. This will ensure that the node's children will be processed even if they are marked inactive. We therefore do not need to rescan the node's children simply to mark them. GC is initialized to false at the beginning of every pass.

Further refinements of the approach are possible and in some cases desirable. We have treated the table component of environments as an abstract object into which objects can be inserted and from which objects can be retrieved. The particular representation chosen can affect the performance of the system. DISSECT (3) uses a linear list of variables ordered on the instruction counter. A better approach is to use lexicographic ordering of variables to speed up the search. The reason (3) uses the instruction counter ordering is because of arrays. We propose to use the instruction counter ordering only for the array elements within each array entry. That is, the array name is used to find a location in a lexicographically oredered table; this location holds a pointer to a list of array elements ordered on the instruction counter.

Another important aspect of any symbolic execution system is how it treats loops. We have decided to examine loops through multiple passes. Cheatham et al (5) have the very interesting approach of deriving a recursive equation which describes the effect of the loop. No iteration is necessary in their scheme. The approach is not applicable in general but they claim that practical programs are handled adequately.

Another refinement possible with respect to loops has to do with the application of symbolic exection to error detection. In particular, assume an inherited environment of a loop node which has the constraint C1 is used to execute the node and an environment is synthesized with constraint C2. If C2 implies C1, it is guaranteed that the loop will not terminate. This information would be valuable to the programmer and he should be informed of it.

Another error that is easily detected is use of a variable before it has been assigned. This can be checked when an attempt is made to search the table for the value of a variable and it does not exist. Processing of this environment should be terminated at this point with an appropriate message to the user.

Our scheme can also be modified slightly to indicate at the end of each pass not only the information about the control paths examined but also what fraction of control paths were traversed. This information would be useful in deciding whether to perform another pass.

A final feature that would be useful for a programmer and can be incorporated into our system is the ability to force the symbolic execution of certain paths. This could be accomplished by calling pass, not with the root node, but

the node that marks the beginning of the path to be examined. The inherited environment of the node must, of course, be initialized first and some other minor modifications to pass are also necessary.

4. EXAMPLES

In this section we give three examples of the use of the symbolic evaluator. The first is a program with no loops; the second is a program with two loops but no arrays; the final example illustrates how arrays may be treated. The examples should help clarify the workings of the evaluator.

Example 1: straightline program.

The table below shows the values of the attributes for the nodes of the program tree in Figure 4.

step	node	attribute	value
1	2	inh	{(T,empty)}
2	3	inh	"
3	3	syn	{(T,[A:a1,B:b1])}
4	4	inh	"
5	5	inh	"
6	6	inh	{(a1>b1,[A:a1,B:b1])}
7	6	syn	{(a1>b1,[A:a1,B:b1,MAX:a1])}
8	7	inh	{(a1¬>b1,[A:a1,B:b1])}
9	7	syn	{(a1¬>b1,[A:a1,B:b1,MAX:b1])}
10	5	syn	{(a1>b1,[A:a1,B:b1,MAX:a1]), (a1¬>b1,[A:a1,B:b1,MAX:b1])}
11	5	inh	empty
12	6	syn	empty
13	7	syn	empty
14	8	inh	{(a1>b1,[A:a1,B:b1,MAX:a1]), (a1¬>b1,[A:a1,B:b1,MAX:b1])}
15	8	syn	{(a1>b1,[A:a1,B:b1,MAX:a1]), (a1¬>b1,[A:a1,B:b1,MAX:b1])}
16	4	syn	{(a1>b1,[A:a1,B:b1,MAX:a1]), (a1¬>b1,[A:a1,B:b1,MAX:b1])}
17	4	inh	empty
18	5	syn	empty
19	8	syn	empty
20	2	syn	{(a1>b1,[A:a1,B:b1,MAX:a1]), (a1¬>b1,[A:a1,B:b1,MAX:b1])}
21	2	inh	empty
22	3	syn	empty
23	4	syn	empty
24	1	syn	{(a1>b1,[A:a1,B:b1,MAX:a1]), (a1¬>b1,[A:a1,B:b1,MAX:b1])}
26	2	syn	empty

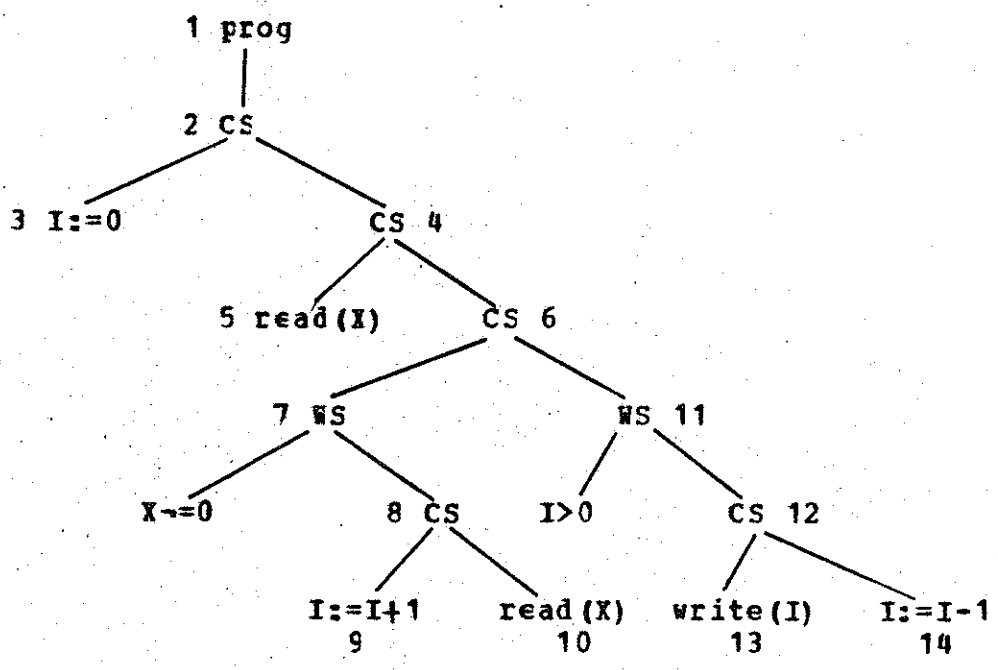
At the end of the pass the values of attributes at all nodes except the root are empty.

The output produced covers all control paths. It indicates that the program result depends on the relationship of the values read for A and B.

Example 2: Program with loops.

The following program and its syntax tree show the behavior of the system on programs with loops.

```
I:=0;
read (X);
while X¬=0
do I:=I+1;
     read (X)
od;
while I>0
do write (I);
     I:=I-1
od
```



step	node	attribute	value
0	1	inh	{(T,empty)}
1	2	inh	"
2	3	inh	"
3	3	syn	{(T,[I:0])}
4	3	inh	empty
5	4	inh	{(T,[I:0])}
6	5	inh	"
7	5	syn	{(T,[I:0,X:x1])}
8	5	inh	empty
9	6	inh	{(T,[I:0,X:x1])}
10	7	inh	"
11	8	inh	{(x1¬=0,[I:0,X:x1])}
12	9	inh	"
13	9	syn	{(x1¬=0,[I:1,X:x1])}
14	9	inh	empty
15	10	inh	{(x1¬=0,[I:1,X:x1])}
16	10	syn	{(x1¬=0,[I:1,X:x2])}
17	10	inh	empty
18	8	syn	{(x1¬=0,[I:1,X:x2])}
19	8	inh	empty
20	9	syn	empty
21	10	syn	empty
22	7	syn	{(x1=0,[I:0,X:x1])
23	7	inh	{(x1¬=0,[I:1,X:x2])}
24	8	syn	empty
25	11	inh	{(x1=0,[I:0,X:x1])}
26	12	inh	{(0>0,[I:0,X:x1])} = empty
27	13	inh	empty
28	13	syn	empty
29	13	inh	empty
30	14	inh	empty
31	14	syn	empty
32	14	inh	empty
33	12	syn	empty
34	12	inh	empty
35	13	syn	empty
36	14	syn	empty
37	11	syn	{(x1=0,[I:0,X:x1])}
38	11	inh	empty
39	12	syn	empty
40	6	syn	{(x1=0,[I:0,X:x1])}
41	6	inh	empty
42	7	syn	empty
43	11	syn	empty
44	4	syn	{(x1=0,[I:0,X:x1])}
45	4	inh	empty
46	5	syn	empty
47	6	syn	empty
48	4	syn	{(x1=0,[I:0,X:x1])}
49	4	inh	empty
50	5	syn	empty
51	6	syn	empty

52	2	syn	{(x1=0,[I:0,X:x1])}
53	4	inh	empty
54	5	syn	empty
55	6	syn	empty
56	1	syn	{(x1=0,[I:0,X:x1])}
57	1	inh	empty
58	2	syn	empty

At the end of pass 1, the only nonempty attributes are the synthesized environment of node 1 and the inherited environment of node 7. The latter attribute will cause the next pass to exercise the next iteration of the loop. The reason that the inherited environment of node 13 is empty is that in this program it is impossible to have an execution of the second loop without an iteration of the first.

In general, all computations that are recorded in the synthesized environment of the root node at the end of pass i, correspond to a total of i-1 loop iterations. In this example, at the end of pass 1, no iterations of loops are recorded. However, the inherited environments of while nodes retain the information necessary for the next pass to exercise the next loop iterations. Thus at the end of pass 2, we could have the results of

- zero iteration of the first loop and one iteration of the second,
- one iteration of the first loop and zero iteration of the second.

But the trace of pass 2 which is shcwn below, shows that both of these computations are infeasible. This is indicated by the empty evircnment synthesized for the root node.

step	node	attribute	value
1	2	inh	empty
2	3	inh	empty
3	4	inh	empty
4	5	inh	empty
5	6	inh	empty
6	7	inh	{(x1¬=0,[I:1,X:x2])}
7	8	inh	{(x1¬=0 and x2¬=0,[I:1,X:x2])}
8	9	inh	"
9	9	syn	{(x1¬=0 and x2¬=0,[I:2,X:x2])}
10	9	inh	empty
11	10	inh	{(x1¬=0 and x2¬=0,[I:2,X:x2])}
12	10	syn	{(x1¬=0 and x2¬=0,[I:2,X:x3])}
13	10	inh	empty
14	8	syn	{(x1¬=0 and x2¬=0,[I:2,X:x3])}
15	8	inh	empty
16	9	syn	empty
17	10	syn	empty
18	7	syn	{(x1¬=0 and x2=0,[I:1,X:x2])}
19	7	inh	{(x1¬=0 and x2¬=0,[I:2,X:x3])}
20	8	syn	empty
21	11	inh	{(x1¬=0 and x2=0,[I:1,X:x2])}
22	12	inh	{(x1¬=0 and x2=0,[I:1,X:x2])}
23	13	inh	"
24	13	syn	"
25	13	inh	empty
26	14	inh	{(x1¬=0 and x2=0,[I:1,X:x2])}
27	14	syn	{(x1¬=0 and x2=0,[I:0,X:x2])}
28	14	inh	empty
29	12	syn	{(x1¬=0 and x2=0,[I:0,X:x2])}
30	12	inh	empty
31	13	syn	empty
32	14	syn	empty
33	11	syn	{(x1¬=0 and x2=0 and 1¬>0,...)}=empty
34	11	inh	{(x1¬=0 and x2=0,[I:0,X:x2])}
35	12	syn	empty
36	6	syn	empty
37	6	inh	empty
38	7	syn	empty
39	11	syn	empty
40	4	syn	empty
41	4	inh	empty
42	5	syn	empty
43	6	syn	empty
44	2	syn	empty
45	2	inh	empty
46	3	syn	empty
47	4	syn	empty
48	1	syn	empty
49	1	inh	empty
50	2	syn	empty

We will not show the trace of pass 3 but by now it should be clear what will be accomplished. As can be seen, what is left in the inherited attributes of while statements will be carried through the next iteration of the loops. The final result of pass 3 is that the synthesized environment of the root node displayed will be {(x1¬=0 and x2=0,[I:0,X:x2])} which corresponds to one iteration through each loop.

Example 3: Program with arrays.

This example is intended to show how arrays may be handled within our scheme. The basic idea is to have one entry for the entire array in each table. This entry records the values for all array elements assigned so far. Insertions and retrievals into this entry are handled in a last in first out order to ensure that the most recent value assigned to an array element is retrieved. If array indices were all constants or known quantities, handling arrays would require little additional effort. The situation is, however, more complicated when, as is usually the case, array indices are expressions that depend on input values. In such cases, a reference to, say, A[x1], is ambiguous in the sense that any one of the array elements may be being referenced (as long as the constraint is satisfied.) The example below shows how these problems are solved in our scheme.

PROGRAM

```
1 X[1]:=0; X[2]:=0; X[3]:=0;
2 read(I,J);
3 X[I]:=3; X[J]:=5;
4 read(M);
5 X[1]:=X[M];
6 X[M]:=X[J];
7 A:=X[M];
8 if A=0
  then A:=1
  else A:=-1
  fi
```

For this example, we will not show the syntax tree, nor the entire trace. By now the reader should be familiar with much of the detail. We have numbered the program statements and in the following will show the important steps in the symbolic execution of the program by referring to these numbers.

After line 2 of the program, the environment is {(T,[I:i1,J:j1,(X[1]:0,X[2]:0,X[3]:0)])}. The value of the environment after line 3 becomes

{(T,[I:i1,J:j1,(X[1]:0,X[2]:0,X[3]:0,X[i1]:3,X[j1]:5)])}.

The next step shows that assignments to array elements are kept in instruction counter order. After line 5 the environment is

{(T,[I:i1,J:j1,M:m1,(X[1]:0,X[2]:0,X[3]:0,X[i1]:3,X[j1]:5,X-[1]:X[m1])])}. Note that the new value for X[1] does not merely replace the old one but is appended to the end of the array entry. The reason is that the new value of X[1] is not only invalidating the previously explicitly assigned value to X[1] but is also invalidating the implicitly assigned

values which in this case may be X[i1] and/or X[j1]. After appending the new X[1], we may of course delete the old explicit X[1] from the table.

After line 7, the environment is
{(T,[A:x[m1],I:i1,J:j1,(X[2]:0,X[3]:0,X[i1]:3,X[j1]:5,X[1]:-X[m1], X[m1]:X[j1])])}. The environment inherited by each component of the if-statement is the same as that inherited by the entire if-statement except for the constraint. Evaluating the constraint shows the tedium of dealing with arrays. The source of the complication is that the constraint must be in terms of input variables only. Thus it is at this time that we need to access the array values (in last in first out order, of course). The constrairt for the first component of the if statement is simply A=0 which we now must simplify. After much simplification, the constraint will be (j1=1 and m1¬=1 and m1¬=i1). For the else branch, it will be (j1¬=1 or j1=m1 i1=m1).

5. CONCLUSIONS

The syntax directed symbolic evaluation scheme described in this paper is not intended to give a complete description of the features a symbolic evaluator should posess, nor does it cover all the constructs which can be found in "real" programming languages. Rather, its purpose was to give a fla-

vor of what a syntax directed symbolic evaluatcr lcoks like and how evaluation strategies can be embedded in the scheme.

Additional language constructs could be handled by new procedures. For example, goto's can be treated as proposed in (8).

The scheme can also be enriched and specialized fcr different applications. Source level debugging for some classes of non-statically checkable errors can be easily accomplished. For example, using uninitialized variables or out of range array references can be trapped for some computations.

The symbolic evaluator itself can be seen as a component of the set of tools provided for program validation. Its output can either be used to synthesize test data (in such a case only the constraint part is used) or to prove assertions on the result of some computations.

One important feature that characterizes the symbolic evaluator proposed here is that it is syntax directed. Syrtax directed schemes provide a conceptual framework for designing and understanding a number of tools which can be built around a programming language in a coherent program development system. Traditionally, syntax directed schemes have

been proposed for describing translators. More recently, syntax directed formulations of editors (7) and optimizers (8) have also been investigated. It is our opinion that a syntax directed formulation of all the programming tools has a great advantage over ad-hoc techniques in terms of structure, reliability, modifiability, and portability.

ACKNOWLEDGEMENTS

The work of Mehdi Jazayeri was supported by National Science Foundation Grant No. MCS77-03729. Carlo Ghezzi was supported in part by CNR.

REFERENCES

(1) R. T. Yeh, Current trends in programming methodology, vol II: Program validation, Prentice Hall, Englewood-Cliffs, 1977,1-322.

(2) J. C. King, Symbolic execution and program testing, Comm. ACM, vol.19,no.7, July 1976,385-394.

(3) W. E. Howden, Symbolic testing and the DISSECT symbolic evaluationn system, IEEE Trans. Software Eng., vol.SE-3,no.4, July 1977,266-278.

(4) L. A. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. Software Eng., vol.SE-2,no.3, September 1976, 215-222.

(5) Cheatham, Holloway, and Townley, Symbolic evaluation and the analysis of programs, IEEE Trans. Software Eng., vol.SE-5,no.4, July 1979,402-417.

(6) Aho and Ullman, Principles of compiler design, Addison-Wesley,Reading, 1977,1-604.

(7) Ghezzi and Mandrioli, Incremental parsing, ACM TOPLAS, vol.1,no.1, July 1979,58-70.

(8) Babich and Jazayeri, The method of attributes for data flow analysi, Acta Informatica ,vol.10,no.3, December 1978,245-264.