SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 935–942 (1980)

Modified Petri Nets as Flowcharts for Recursive Programs

SIEGFRIED WENDT
Department of Electrical Engineering, University of Kaiserslautern, P.O.B. 3049, 6750 Kaiserslautern, West Germany

SUMMARY

The problem of graphic representation of recursive program structures is discussed. On the basis of the stack mechanism, modified Petri nets are introduced which only slightly differ from the common state machine flowcharts, but have much more representation power.

KEY WORDS Flowchart Petri net Recursion Stack

INTRODUCTION

There are many people designing software who get more insight into program structure by looking at a flowchart instead of reading a program source. The common flowchart techniques, however, do not allow us to represent recursive program structures. This is quite obvious, since these flowcharts are only special versions of finite state machine graphs and therefore are restricted to the representation of regular sequences. The representation power of flowcharts can be much increased by introducing potentially infinite storage for the flowchart tokens. Since the usual generators for recursively defined sequences use a stack, it is reasonable to look for a way to introduce the stack mechanism into the flowchart. A rather simple solution has been found which will now be described. Since this solution is based on Petri nets, the description is preceded by a short introduction to the definition of such nets.

PETRI NETS

A Petri net is a directed graph with two types of nodes, an initial marking of nodes of one type and a firing rule to change the marking.[1] The nodes of the markable type are called places, the others are called transitions. Only nodes of different types can be connected by directed arcs. In graphic representations, places are drawn as circles and transitions as rectangles. Figure 1 shows an example of a net with 6 places and 6 transitions. The marking is the assignment of a non-negative integer to each place; the places can be viewed as containers for tokens, and the marking as the actual contents.

The firing rule determines how a new marking can be derived from a given marking. The rule consists of two parts: the first part defines a necessary condition which enables a transition to fire, and the second part defines how the firing of a transition changes the marking. A transition is enabled to fire if each input place contains at least one token; an input place of a transition is a place which is connected to the transition by an arc

0038–0644/80/1110–0935$01.00

Received 9 February 1979
Revised 7 May 1980

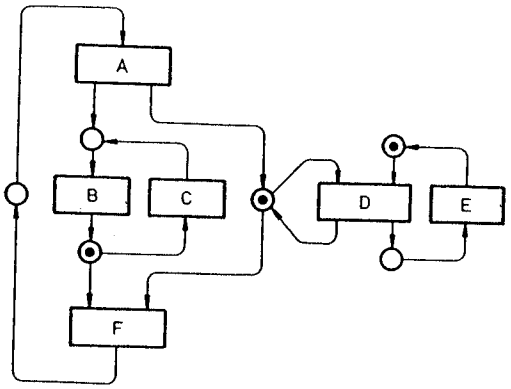

Figure 1. Example of a Petri net

pointing to the transition. The firing of a transition changes the marking as follows: Each input place loses one token and each output place gains one token; the marking of a place which is both input and output place stays unchanged.

A net is said to contain a conflict if an enabled transition can be disabled by the firing of another transition. Figure 1 shows such a conflict: the given marking enables the transitions C, D and F, and the firing of F will disable C and D.

GENERAL DESCRIPTION OF THE METHOD

The change of the program counter's value during the execution of a sequential program corresponds to the flow of a single token along selected paths of the flowchart. Each call of a routine not only shifts this so-called processor token from the calling program to the called routine, but also generates a so-called stack token which is put on a stack place—labelled S—within the calling program in order to store the return position (see Figure 2). Since the called routine can call another routine and so on,

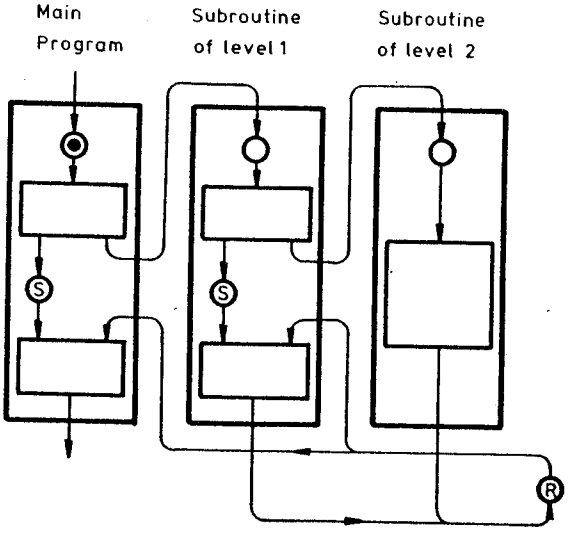

Figure 2. Nesting of subroutines

many stack tokens can be waiting on different stack places before the first return occurs. If there is no recursion, each of the different stack places contains at most one token, but in case of recursion, it is possible that there are stack places with more than one token on each. At each return the processor token is placed on the so-called return place—labelled R—from which there are paths to all return positions. If there are stack tokens waiting on different stack places, the conflict of where the processor token should go has to be resolved. This is done by introducing the 'age' as an attribute of the stack tokens. The age of a stack token says how long this token has already been waiting on its place. Now the conflict can be resolved such that whenever the processor token is on the return place, the transition with the 'youngest' stack token will fire.

In the literature on net theory, nets with rules to solve conflicts are called priority nets.[2]

EXAMPLES

The first example is the standard example for recursive programs: the computation of the factorial $n!$ of a non-negative integer n. Figure 3 shows two versions of the program in an ALGOL- or PASCAL-like notation. In the right-hand version, the data stack is explicitly declared. The semantics of the two stack operations are as follows:

POP(V): The top element is removed from the stack and its value is stored into the variable V.

PUSH(V): An element with the value of the variable V is placed on top of the stack; v is not changed.

In the left-hand version of Figure 3, the stack mechanism is hidden. In order to understand this program, one must know that each call of the procedure causes a new memory allocation for the local variables which implies a *push* for the old values of these variables. The corresponding *pop* occurs implicitly at each return from the procedure.

In a graphic representation of a procedure call the stack operations for the data variables must be shown explicitly because in a graph the call can only be a simple 'Jump to subroutine'. Such a jump means a transfer of control and a saving of the return address, nothing else. Therefore, the flowchart in Figure 4 corresponds to the right-hand version of Figure 3.

A stack place which is drawn as a double circle has a potentially infinite capacity for tokens.

The second example deals with the generation of graphics code and its execution. The graphics code is a program, i.e. a sequence of graphics instructions, and needs a display processor for execution. The graphics instructions are classified into four categories:

1. Instructions which can be interpreted by a hardware display device; for example '*Draw vector* Δx, Δy' or '*Write alphanumeric symbol A*';
2. Instructions for the modification of the graphics program counter; for example '*Jump to location L*' or '*Jump to graphics subroutine S*';
3. Instructions which must be expanded by a macro expander; for example '*Draw a circle* x_0, y_0, r' which must be expanded into a sequence of vector instructions as a piecewise approximation of the circle;
4. Stop instruction.

An expansion generated by the macro expander may itself contain other, lower level macros which again have to be expanded. Each expansion has the form of an individual graphics program including a stop instruction.

```
PROGRAM MAIN
   LOCAL VARIABLES
      N;
   BEGIN
      WRITE('N=');
      READ(N);

      CALL FAC(N);

      WRITE('N!=', N);
   END;

PROCEDURE FAC(J);
   LOCAL VARIABLES
      M;
   BEGIN

      IF J>0 THEN
         BEGIN
            M:=J-1;


            CALL FAC(M);


            J:=J*M;
         END
      ELSE  J:=1;

   END;
```

```
GLOBAL VARIABLES
   J, M, N;
   STACK;

PROGRAM MAIN


   BEGIN
      WRITE('N=');
      READ(N);
      PUSH(N);
      CALL FAC;
      POP(N);
      WRITE('N!=', N);
   END;

PROCEDURE FAC;


   BEGIN
      POP(J);
      IF J>0 THEN
         BEGIN
            M:=J-1;
            PUSH(J);
            PUSH(M);
            CALL FAC;
            POP(M);
            POP(J);
            J:=J*M;
         END
      ELSE  J:=1;
      PUSH(J);
   END;
```

Figure 3. Example for direct recursion: computation of the faculty of an integer

Generation and execution of graphics program code is done by the program system which is shown in Figure 5 in an ALGOL- or PASCAL-like notation. The main program provides the original graphics program and has it executed by the virtual display processor. The virtual display processor fetches and decodes the graphics instructions. Instructions of categories 1 and 2 are called simple instructions and can be 'executed' by the virtual display processor itself, either by transferring them to the peripheral hardware display device or by modifying the graphics program counter's value. Macro instructions are transferred to the macro expander which returns the expanded code to the virtual display processor for execution. Figure 6 shows the corresponding flowchart.

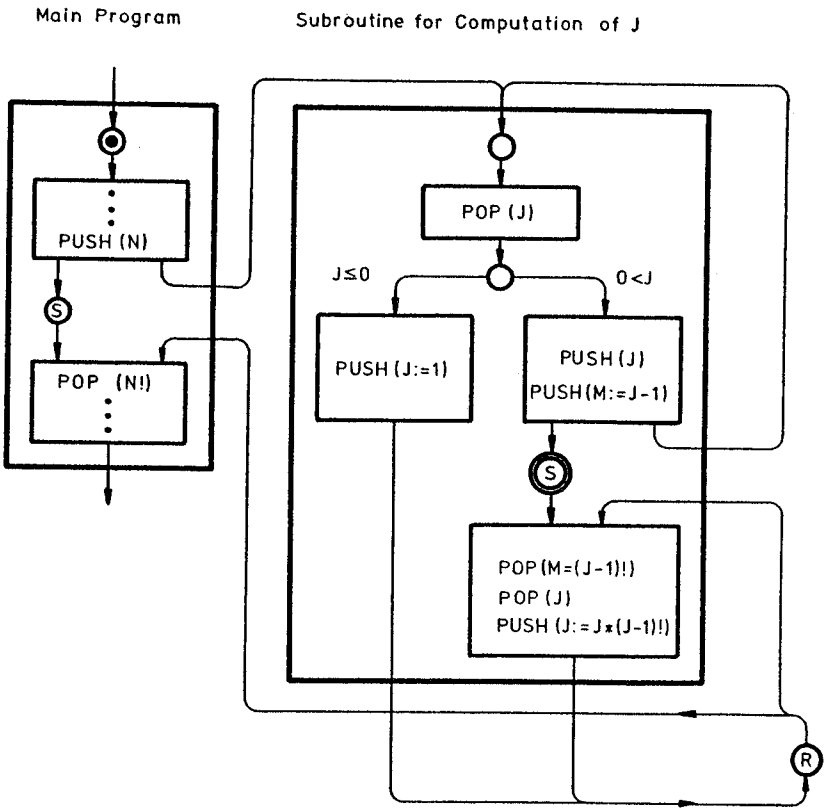

Figure 4. Flowchart to Figure 3

```
GLOBAL VARIABLES
   GRAPHPROG[1..MAX];
   GRAPHPC, NEXTFREE, GRAPHINSTR;
   STACK;

PROGRAM MAIN
   BEGIN
      GENERATE OR READ GRAPHICS CODE
      INTO GRAPHPROG[1..ENDOFPROG];
      GRAPHPC:=1;
      NEXTFREE:=ENDOFPROG+1;
      CALL VIRTDISPLPROC;
   END;


PROCEDURE VIRTDISPLPROC;
   BEGIN
      FETCH: GRAPHINSTR:=GRAPHPROG[GRAPHPC];
             GRAPHPC:=GRAPHPC+1;
             IF GRAPHINSTR IN SET OF
                SIMPLEINSTR THEN
                BEGIN
                   EXECUTE GRAPHINSTR;
                   GOTO FETCH;
                END;
             IF GRAPHINSTR IN SET OF
                MACROINSTR THEN
                BEGIN
                  PUSH(GRAPHPC);
                  CALL MACROEXPANDER;
                  POP (GRAPHPC);
                  GOTO FETCH;
                END;
   END;

PROCEDURE MACROEXPANDER;
   BEGIN
      EXPAND GRAPHINSTR
      INTO GRAPHPROG[NEXTFREE..ENDOFEXP];
      PUSH (NEXTFREE);
      GRAPHPC:=NEXTFREE;
      NEXTFREE:=ENDOFEXP+1;
      CALL VIRTDISPLPROC;
      POP (NEXTFREE);
   END;
```

Figure 5. Example for indirect recursion: interpretation of a display program

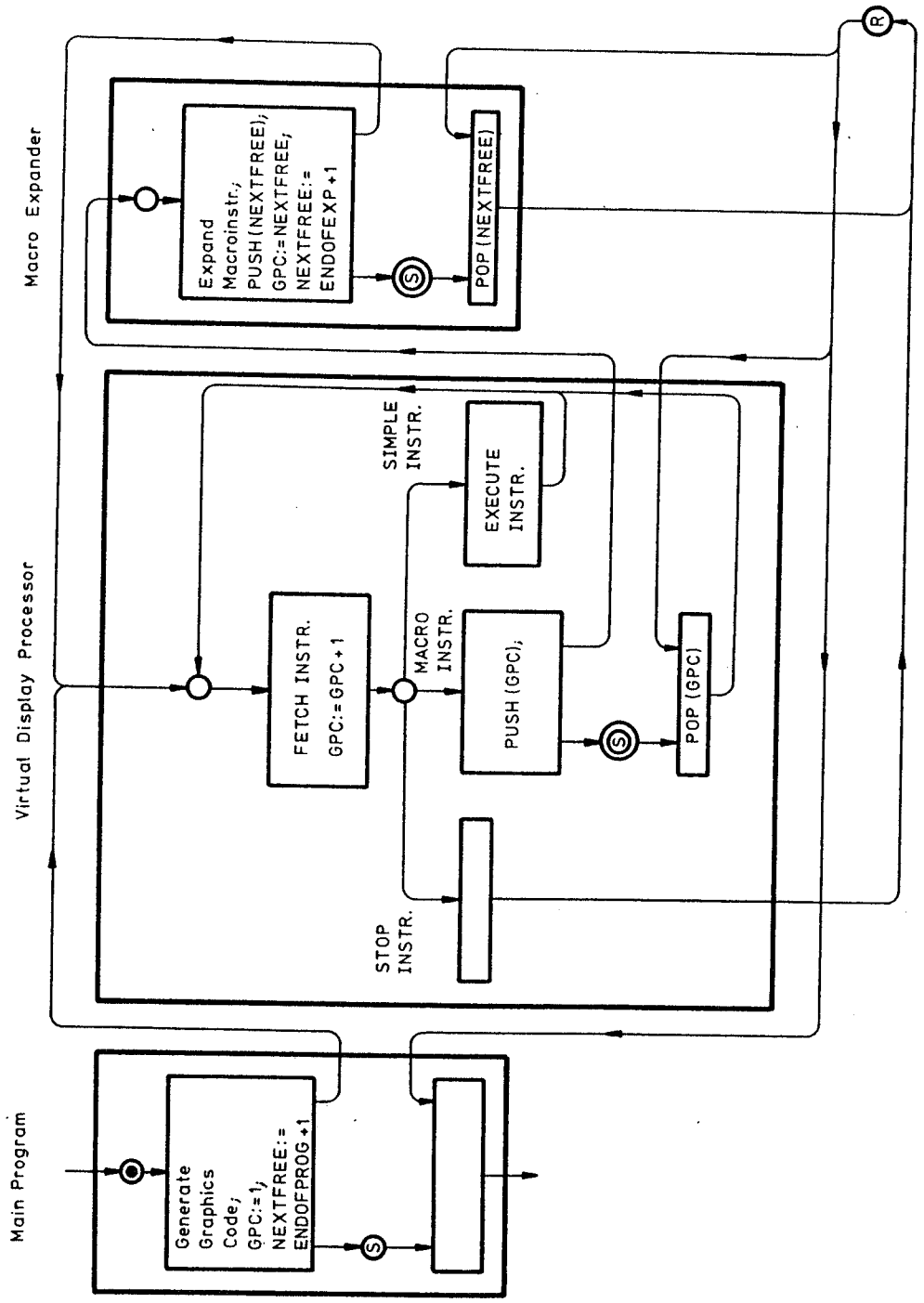

Figure 6. Flowchart to Figure 5

REFERENCES

1. C. A. Petri, *Kommunikation mit Automaten,* Schriften des IIM, No. 2, Bonn, 1962.
2. M. Hack, *Decidability Questions for Petri Nets,* PhD thesis, Department of Electrical Engineering, MIT, 1975.