Systems with universal subsystems, realization and application

BRUNO BUCHBERGER* and BERNHARD QUATEMBER**
**Institut für Mathematik*
***Institut für Statistik und Informatik*
Johannes Kepler Universität, Linz, Austria

Progress in Cybernetics and Systems Research, Vol. VII, Hemisphere Publ. Corp. 1978

1. INTRODUCTION

In this paper we study a possible realization of systems that are composed of, theoretically, infinitely many and, practically, a large number of subsystems. Each of these subsystems

- is identical
- essentially operates autonomously, however,
- at certain moments may communicate with other subsystems in its neighbourhood,
- is universal (in the sense of algorithm theory or, more practically, in the sense of computer science), and
- contributes to a common goal.

Thus, these systems share essential features with well-known other types of systems. On the other hand, however, they are distinct with respect to at least one feature from the systems studied so far. For instance:

Cellular spaces [1] consist of a large number of subsystems that communicate with subsystems in the neighbourhood. However, no single subsystem is universal. Furthermore, the operation of the whole system is synchronized.

Combinatorial switching circuits [2] may well be composed of identical components (see [3]) that operate asynchronously. Again the components are not universal.

Parallel computers [4] consist of a large number of processors that, normally, are not universal and do not operate autonomously.

Computer networks (see, for instance, [5]) consist of highly autonomous universal subsystems. However, only a few computers are combined in one system. Also, the communication between the subsystems occurs only for organizing the distribution of independent tasks among the subsystems, not for contributing to some common goal.

Hierarchical systems in the sense of Mesarovic [6] consist of subsystems with a certain degree of autonomous behaviour contributing to a common goal. However, no emphasis is on universality and homogeneity. The same characteristics hold for structured software systems [7].

Universality of the subsystems is an essential enrichment of systems. Of course, a combination of universal systems cannot produce more than a universal system. However, the availability of universal subsystems may drastically influence the complexity of problem solving in a universal system (see Section 4).

2. SYSTEMS WITH UNIVERSAL SUBSYSTEMS

The concept of a system with universal subsystems (SUS) is a slight generalization of the concept of computer-trees introduced in [8].

The basic component ("module") of a SUS is a sequential universal automaton [9] with finitely many additional "in-lines" and "out-lines" (see Figure 1).

Every in-line and every out-line consists of: a data line; a sensing line + sensor bit; and a setting line (see Figure 2).

In fact, so far in-lines cannot be distinguished from out-lines. The difference can only be made after having introduced the identifier modification described in the later paragraphs. An out-line of one module may be connected with an in-line of some other module in the following way (see Figure 3).

An arbitrary number (even a potentially infinite number) of modules may be interconnected in this way in order to form SUS's of different structures such as trees and nets like this: (see Figure 4).

We assume the modules to be programmed in an ALGOL-like language. Then the functional characteristics of the in- and out-lines may be explained by describing the semantics of two additional language features:

(1) Sensor Instructions:

```
if S1 then ...;  if S2 then ...;  ...
if T1 then ...;  if T2 then ...;  ...
U1:=true; U1:=false; U2:=true; U2:=false; ....
V1:=true; V1:=false; V2:=true; V2:=false; ...
```

These instructions are used for communication with the sensor bits which are supposed to be designated like this: (see Figure 5).

Notice that each sensor bit has two designations depending on whether it is referenced by its own module for reading or by a neighbouring module for setting and resetting.

(2) Identifier (Address) Modification

Every identifier x is available in k+1 issues x, x', x", ... , $x^{(k)}$ where k is the number of out-lines in the modules. A module C in the system, by means of identifiers of the types x, x', x", ... , $x^{(k)}$ has access to its own storage and (via the data lines) to the storage of its first, second, ... , k-th neighbour C', C", ... , $C^{(k)}$, respectively. Furthermore, if module C addresses a storage region of its neighbour $C^{(i)}$ by identifier $x^{(i)}$ then module $C^{(i)}$ may address the same

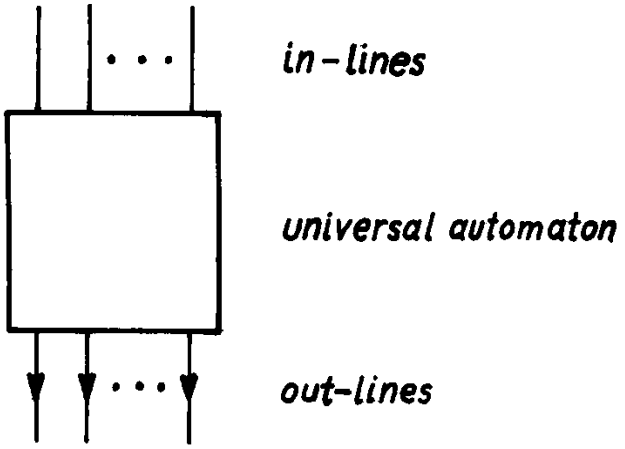

Figure 1. One module

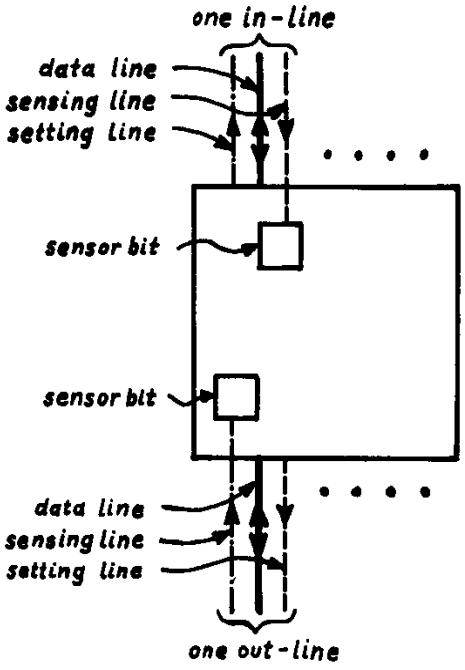

Figure 2. Structure of in- and out-lines

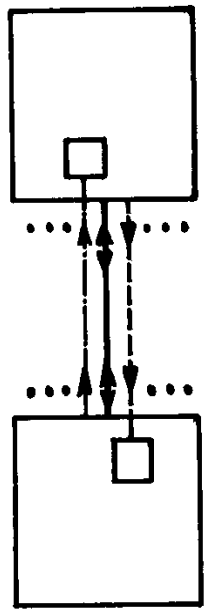

Figure 3. Connection between in- and out-lines

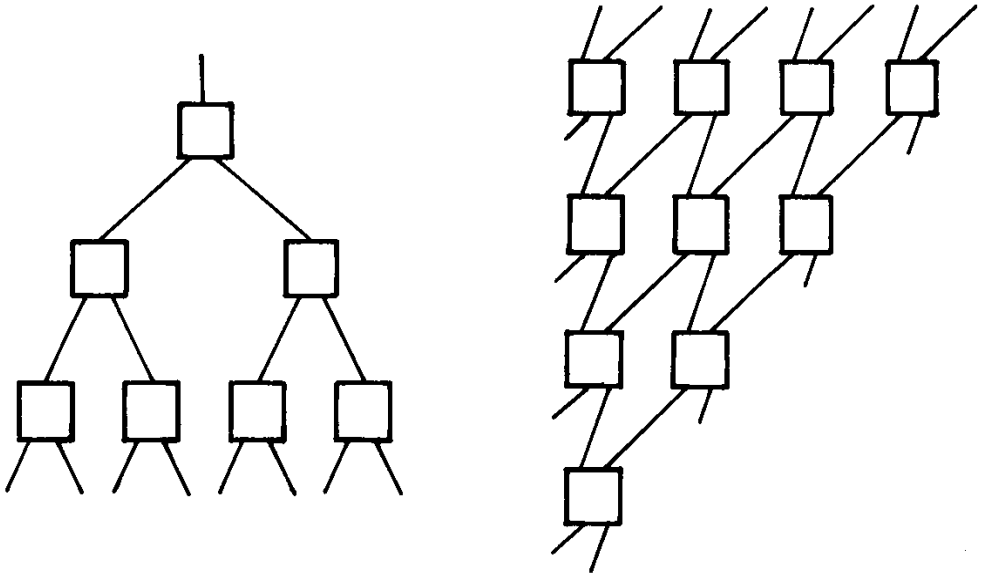

Figure 4. Different types of SUS's

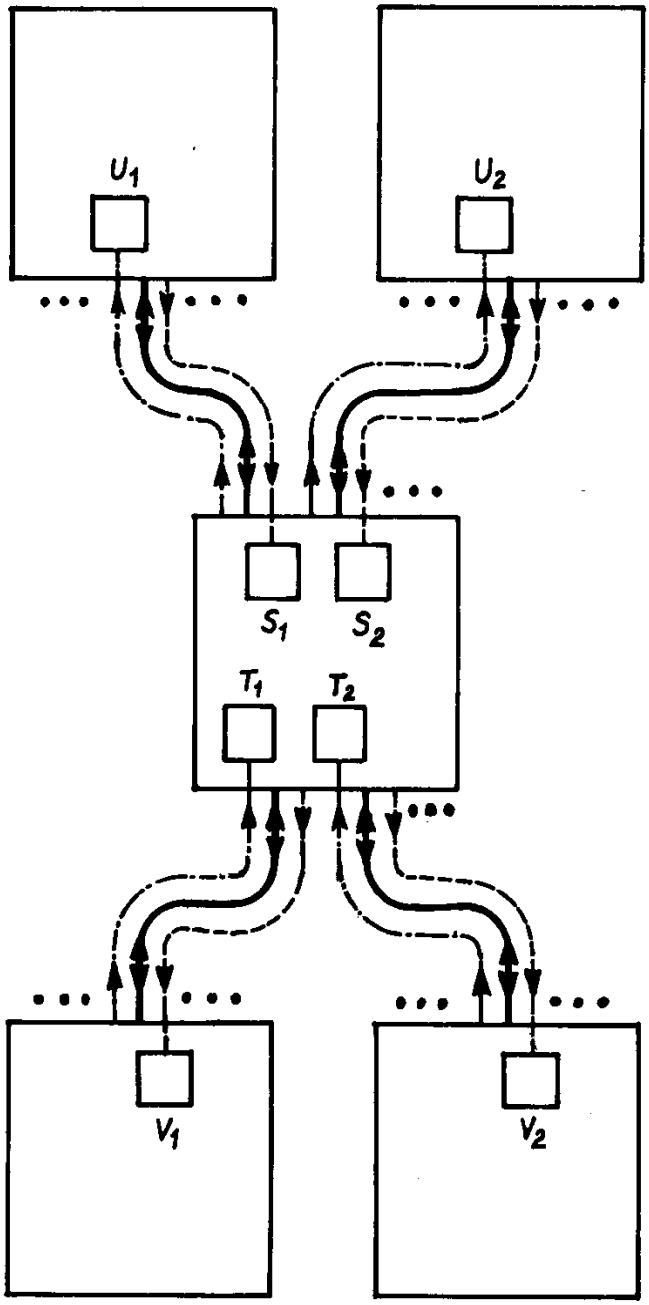

Figure 5. Designation of sensor bits

region by identifier x.

In a SUS all modules are started simultaneously and operate autonomously. However, by means of the two additional language features described above, at certain points of the computation neighbouring modules may communicate. Concurrent access to the same storage region is supposed to be excluded by an appropriate use of the sensor instructions.

Further, we add the following language feature:

(3) Computed goto:

goto a;

This jump-instruction uses the numerical value of variable a as a label. In order to avoid ambiguities we admit integer labels exclusively.

3. REALIZATION

It is easy to realize (finite approximations to) SUS's with present-day hardware components.

The modules may be realized by an appropriate combination of the following components: a microprocessor (μP), a data memory (DM), a program memory (PM), an interface (IF) -- containing the sensor bits and realizing the address modification.

The structure of one module is as follows (without loss of generality we consider only two in-lines and two out-lines): (see Figure 6).

The address modifications may be realized as follows: We divide the set A of addresses that are not needed for addressing PM or for other special purposes into three disjoint subsets A_0, A_1, and A_2 of equal size and define two bijective mappings:

$$f_1 : A_1 \to A_0$$

$$f_2 : A_2 \to A_0 .$$

(For example: $A := \{0,\ldots,3071\}$,
$A_0 := \{0,\ldots,1023\}$,
$A_1 := \{1024,\ldots,2047\}$,
$A_2 := \{2048,\ldots,3071\}$,
$f_1(a) := a - 1024$,
$f_2(a) := a - 2048$.)

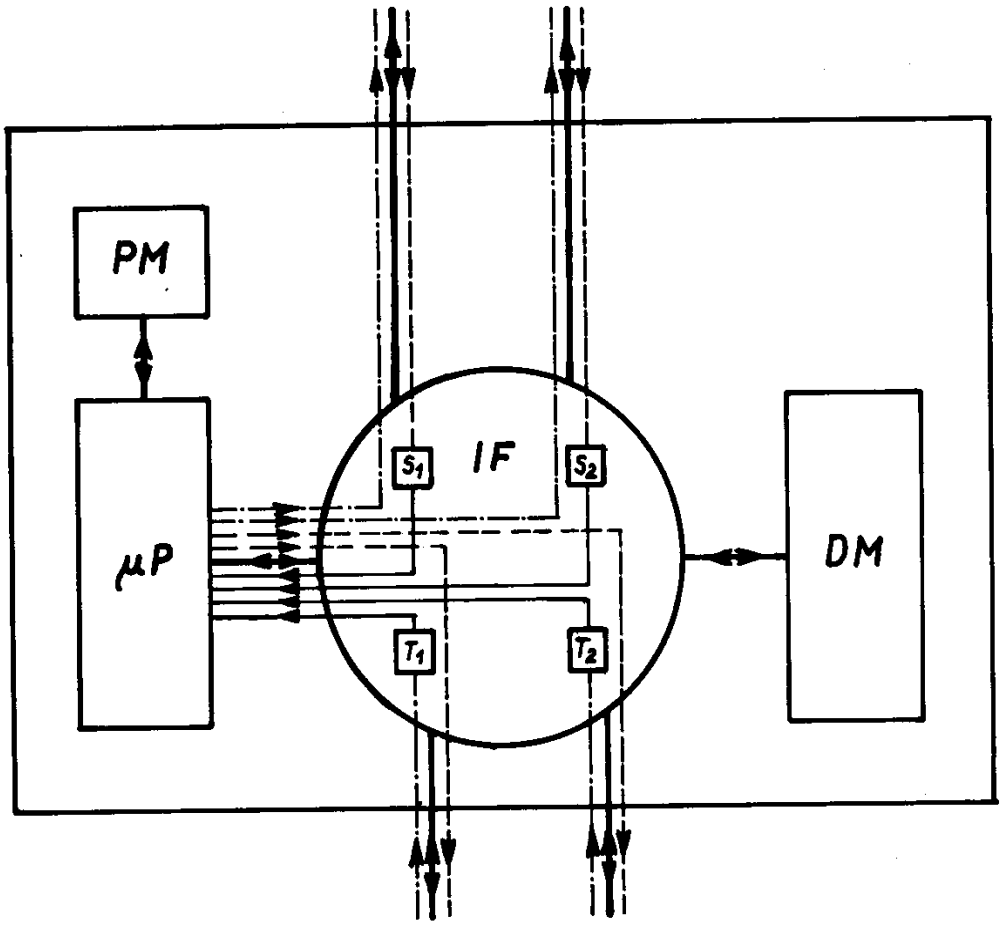

Figure 6. Structure of one module

The IF must realize the following function:

(1) An address a produced by the μP is analysed in the IF. Accesses to the DM of its own module or to the DM's of the first and second neighbour are, then, carried out according to the following rule:
  If $a \in A_0$, then location a of the DM of its own module is accessed.
  If $a \in A_1$, then location $f_1(a)$ of the DM of the first neighbour is accessed.
  If $a \in A_2$, then location $f_2(a)$ of the DM of the second neighbour is accessed.
(2) An address a arriving at the IF via an in-line remains unchanged.

The sensor bits may be implemented by using bits of a fixed storage location outside DM.

A concrete realization of this type has been carried out in collaboration with ICA-Company (Vienna) using the KIM-1 microprocessor (see [10]).

4. APPLICATION

Our main motivation for developing the concept of SUS's is the desire to exploit the parallelism inherent in the recursive formulation of certain algorithms in order to speed them up. The adequate shape of SUS's for this application is the (binary) tree. We have investigated this application in [8] by giving a number of typical examples which show that all types of recursions may be executed on SUS's in a natural way.

We first briefly summarize the discussion given in [8]. Then we pursue some aspects of this application in more detail.

Consider the tautology problem which is recursively defined by

$$\begin{aligned} taut(t,n) \leftrightarrow\ & n=0 \wedge true(t) \vee \\ & n\geq 1 \wedge taut(\, subst(t,n,1),\ n-1) \wedge \\ & \quad\ taut(\, subst(t,n,0),\ n-1) \end{aligned}$$

where we used the following abbreviations:

taut(t,n)	-	the boolean expression t contains no more than n variables and is a tautology,
true(t)	-	the boolean expression t does not contain any variables and the evaluation of t yields the truth value 1,
subst(t,n,x)	-	the result of substituting the truth value x for the n-th variable in the boolean expression t.

This definition of the tautology problem may equally well be read as a recursive formulation of an algorithm for solving the problem. The execution of this recursive algorithm on an ordinary computer uses a stack mechanism which artificially sequentializes the simultaneous procedure calls taut(subst(t,n,1),n-1) and taut(subst(t,n,0),n-1). As a consequence, the execution time of this algorithm on a normal computer is $O(2^n)$. In fact, no essentially better algorithm is known for this problem at present time (see [11]).

However, we can solve the problem in O(n) steps on the binary computer tree (Figure 4) by the following program:

```
1: if ¬S1 then goto 1;
   if n=0 then
            begin  y:= true(t); U1:= true; goto 5
            end;
   if n≥1 then
            begin
              t':= subst(t,n,1); t":= subst(t,n,0);
              n':= n":= n-1;
              V1:= V2:= true;
            2: if ¬(T1∧T2) then goto 2;
              y:= y'∧y";
              U1:= true; goto 5
            end;
5: ;
```

If this program is loaded to all modules of the tree and all modules are started simultaneously, a "computational wave" is generated in the tree which first propagates downward until level n of the tree and then contracts again. Propagating downward it distributes sub-tasks to the modules, contracting upward it combines partial results in order to yield the final solution. The total amount of time for this computation is O(n).

Of course the gain in time complexity must be purchased by an equivalent explosion of hardware complexity. (Note that this is very similar to what can be observed when passing from serial to parallel adders or from serial to parallel multipliers, see [3].)

In practical applications we can realize only a finite number N of levels in the tree. In this case, at the N-th level we must call a sequential procedure true(t,n) instead of true(t) which solves the tautology problem for the ultimate value of n (see [8]). The time complexity of the algorithm then is = 2^{n-N}. Hence,

$$\frac{\text{computation time on ordinary computer}}{\text{computation time on computer tree}} = \frac{2^n}{2^{n-N}} = 2^N.$$

This means that in any case we have a constant speed-up of 2^N. If, for instance, N = 15 (which seems to be realistic for present-day hardware) then a constant speed-up of approximately 30,000 is possible. This corresponds to what has been achieved by hardware improvement in one computer generation. Even though we cannot really convert $O(2^n)$ into O(n), we could thus achieve a "one-generation speed-up" by introducing a new computer concept instead of developing new computer hardware.

Let us now pursue the above example in more detail. At the bottom level of the tree we have to execute the procedure true(t). Such a procedure is polynomial in the length of t (see [11]). However, we again can define a recursive procedure for this problem which may easily be translated into a computer-tree program that exploits the parallelism contained in the recursive formulation:

```
true(t):= { true,                          if t=1
          { false,                         if t=0
          { ¬true( opd1(t) ),              if t is a negation,
          { true( opd1(t) ) ∧ true( opd2(t) ),
          {                                if t is a conjunction,
          { true( opd1(t) ) ∨ true( opd2(t) ),
          {                                if t is a disjunction,
```

where opd1(t) and opd2(t) denote the first and second operand of the boolean expression t, respectively.

A corresponding computer-tree program is:

```
3: if ¬ S1 wait;                                        *)
   if t=1 then
          begin  y:= true; U1:= true; goto 4
          end;
   if t=0 then
          begin .....
          end;
   s:= true;
   while t is a negation do
          begin t:= opd1(t); s:=¬s
          end;
   if t is a conjunction then
          begin  t':= opd1(t); t":= opd2(t);
                 V1:= V2:= true;
                 if  (T1∧T2) wait;
                 y:= y'∧y"; if ¬s then  y:= ¬y;
                 U1:= true; goto 4
          end;
   if t is a disjunction then
          begin .....
          end;
4:;
```

From this example we see that we can carry out the evaluation process in a time that is linear in the number of nested boolean operations in t. Furthermore, we wished to demonstrate how we can avoid the branching of the computation in the case where t is a negation. In general this technique may be applied whenever a recursive procedure is called only once in a certain part of the procedure body. This technique permits a saving in the number of modules needed in a given computation.

We now can combine the two programs for taut and true in a single one using the computed goto in order to switch control from one program part to the other. (Note that we cannot simply load the program for true to the n-th and the subsequent levels of the tree because the level n depends on the input data):

```
1: if¬ S1 wait;
   goto a;                                              **)
2: if n=0 then goto 3;
   if n≥1 then
          begin
            t':= ...; t":= ...; n':= n":= n-1;
            a':= a":= 2; V1:= V2:= true;
            if ..... goto 4
          end;
3: if t=1 then
          begin  y:= true; U1:= true; goto 4
          end;
   if t=0 then ...;
   s:= true;
   while t is a negation do ...;
   if t is a conjunction then
          begin  t':= opd1(t); t":= opd2(t);
                 a':= a":= 3;
                 V1:= V2:= true;
                 if ....
          end;
   if t is a disjunction then ...;
4: ;
```

*) abbreviation for: 3: if S¬then goto 3;
**) a must be initialized with 2.

Of course, by the techniques given above we could also implement one of the more sophisticated methods for solving the tautology problem. These methods analyse the syntactic structure of t from the very beginning instead of first generating all combinations of truth values (see, for instance, [12]).

The above program for true also demonstrates one of the major practical problems using computer-trees: the computation of the procedure true may propagate very far in one branch of the tree whereas it terminates after a few levels in some other branches. Thus, in a concrete implementation where we have only a fixed number N of levels a given computation might be impossible, though there would still be a lot of modules available in the system. This problem could be solved if we had a hardware mechanism that would permit us to flexibly interconnect the modules during execution time.

There are two extreme possibilities for realizing such a mechanism: a time-shared bus for all modules or a complete interconnection between all modules. The first possibility does not work because it would be a bottleneck in the system that would artificially sequentialize the computational process which should be parallel. The second possibility is not realistic because of its complexity. Therefore a compromise has to be found. Recently, optimistic proposals in this direction have been made (see [13]).

The above example also is good for showing another practical problem: If a module calls its two offsprings it has to transmit the formal parameters which may be considerably long. Although, theoretically, this may only multiply the time complexity by a constant factor, in practice, such a factor can play an important role. We would not need transmitting formal parameters if we had a common storage for the whole system which could

be accessed (at least for reading purposes) by all modules simultaneously. In this case we could simply transmit pointers to the information instead of the information itself. The same argument applies to the program memory. At present such memories are not available (see, however, again [13]).

In the above example one module combines the solutions of two subproblems in order to form the final solution of the problem. In some cases we actually need only the solution to one of the subproblems although we must start trying to solve all of them. This is typical for non-determinism in algorithms. Of course, we can also implement this type of control by an appropriate use of the sensor instructions. We can show this by considering the satisfiability problem:

> given a boolean expression t of n variables,
>
> decide whether t is satisfiable (i.e., whether there is an assignment of the truth values 1 and 0 to the variables of t such that the evaluation of t yields 1)

A suitable program for the computer-tree is:

```
if ¬S1 wait;
if n=0 then
        begin  y:= true(t); U1:= true; goto 2
        end;
if n≥1 then
        begin
          t':= subst(t,n,1); t":= subst(t,n,0);
          n':= n":= n-1;
          V1:= V2:= true;
        3: if T1 then  if y' then
             begin y:= true; U1:= true; goto 2
             end;
           if T2 then  if y" then
             begin y:= true; U1:= true; goto 2
             end;
           if T1∧T2 then  if ¬(y'vy") then
             begin y:= false; U1:= true; goto 2
             end;
           goto 3;
           end;
2: ;
```

So far we have only considered examples where the tree structure seems to be the appropriate type of SUS. However, it seems to be natural to use other types of SUS's for other types of algorithms. For instance, dynamical programming seems to be a possible field of application for which a connection schema of the type shown in Figure 4, right-hand side, might be appropriate (see [11], p. 68). However, we have not analysed this type of application so far. Also we think that such patterns may well be simulated in array processors (see, for instance, [14]). On the other hand, as far as we can see, computer-trees cannot be implemented in array processors without loss of efficiency.

CONCLUSION

We introduced the theoretical concept of systems with universal subsystems, in particular computer-trees. Our main concern was the speed-up of algorithms by writing appropriate programs for computer-trees. In typical examples the gain in time complexity is exponential. This has to be purchased by an equivalent increase of hardware complexity.

Two major hardware problems were mentioned which have to be solved before we can gain a practical advantage from SUS's.

Recently, a number of similar proposals have been made in the literature (see [13-18]. In favour of our approach we would like to emphasize its simplicity and modularity and the easy availability of test hardware.

Acknowledgement: We thank Mrs. I. Chang for her help in preparing the English text.

REFERENCES

1. CODD, E. F., Cellular Spaces, Academic Press, 1968.
2. HARRISON, M. A., Introduction to Switching and Automata Theory, McGraw Hill, 1965.
3. LANGHELD, E., "Zellenlogik und algorithmischer Schaltungsentwurf I und II," Elektronik, Heft 1 (Jänner 1978), pp. 34-42, Heft 2, (Feber 1978), pp. 59-66.
4. ENSLOW, P. H., Multiprocessors and Parallel Processing, John Wiley, 1974.
5. ASHENHURST, R. L. and R. H. VONDEROHE, "A Hierarchical Network," Datamation, 1975.
6. MESAROVIČ, D. MACKO and Y. TAKAHARA, Theory of Hierarchical, Multilevel Systems, Academic Press, 1970.
7. WIRTH, N., Systematisches Programmieren, Teubner, 1975.
8. BUCHBERGER, B., "Computer-Trees and their Programming," Proc. Troisième Colloque de Lile "Les arbres en algèbre et en programmation," 16-18 February, 1978.
9. BUCHBERGER, B. and B. ROIDER, "Input/Output Codings and Transition Functions in Effective Systems," International Journal of General Systems, Vol. 4, 1978, in press.
10. BUCHBERGER, B. and J. FEGERL, "A Universal Module for the Hardware-Implementation of Recursion," Bericht Nr. 106, Institut für Mathematik, University of Linz, 1978.
11. AHO, A., J. E. HOPCROFT and J. D. ULLMAN, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
12. SMULLYAN, R. M., First-Order Logic, Springer, 1968.
13. SULLIVAN, H., T. R. BASHKOW and D. KLAPPHOLZ, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I and II," Conf. Proc., 4th Annual Symposium on Computer Architecture, ACM SIGARCH 5 (1977), 3, pp. 105-117.
14. HANDLER, W., "Aspects of Parallelism in Computer Architecture," in M. Feilmeier (ed.), Parallel Computers-Parallel Mathematics, North-Holland, 1977.
15. GLUSHKOV, V. M., M. B. IGNATYEV, V. A. MYASNIKOV, V. A., TORGASHEV, "Recursive Machines and Computing Technology," Proc. IFIP Congress, 1974.
16. SCHWENKEL, F., "Zur Theorie unendlicher Parallelprozessoren," Computer Science, 26, 1975.
17. GOSTELOW, A. and K., "A Computer Capable of Exchanging Processors for Time," IFIP Congress, 1977.
18. VORGRIMMLER, K. and P. GEMMAR, "Structural Programming of a Multiprocessor System," in M. Feilmeier (ed.) -- see reference 14 above.