MICROPROCESSORS AND THEIR APPLICATIONS
J. Tiberghien, G. Carlstedt, J. Lewi (eds.)
North-Holland Publishing Company
© EUROMICRO 1979

THE CONCURRENT PROGRAMMING LANGUAGE CAP AND THE μ-PROCESSOR ORIENTED CAP CAD-SYSTEM

Franz J. Rammig

Lehrstuhl Informatik I
Universität Dortmund
D-4600 Dortmund 50

This paper describes an integrated μ-processor oriented CAD system which has been designed and implemented at the University of Dortmund. The system offers the concurrent programming language CAP (similar to PL/1) as specification language. This language is used within the CAD system to describe
- machines to be designed,
- algorithms representing peripheral controllers,
- virtual machines for which software has to be cross-developed.

The paper includes a discussion of the major features of the language inclusive concurrency and structured concurrent programming.
The basic building blocks of the system (compiler, binder, interpreter, debugger, machine-model generator, documentation generator, code generators) are described and some applications are presented.

1) General

At the University of Dortmund, in cooperation with a german office-computer manufacturer (Kienzle Apparate GmbH, Villingen), we are developing and implementing an integrated μ-processor oriented CAD system for processor design, firmware implementation and controller design.
The system includes the basic components of a CAD system, supporting
- specification
- verification/evaluation
- documentation
- implementation.

We tried to design a unique system instead of collecting a couple of individual design aids. E. g. the same language is used as well to specify algorithms to be processed within controllers as to model the μ-processor that has been selected to process these algorithms. And we use the same debugger system for verification of processor models and to verify programs running on such models.

The basic components of our CAD system are:

- CAPCOM:	compiler CAP to CAPID (intermediate language)
- CAPLIN:	binder CAPID$^+$ to CAPID
- CAPSIM:	interpreter for CAPID/ RT-simulator
- CAPDOC:	documentation generator
- CAPTEST:	interactive debugging system
- CAPVM:	processor model generator
- CAPCCL:	optimizing cross code generator

First versions of CAPCOM, CAPSIM, CAPDOC, CAPTEST, CAPCCL are operational now, CAPLIN is scheduled to be completed late in 1979, CAPVM runs in an experimental state.
The main applications are:

- processor design:
Designs are formulated in CAP, compiled by CAPCOM, eventually integrated by CAPLIN and simulated with the aid of CAPSIM.
The final design as well as intermediate states are documented via CAPDOC.

- software cross development:
The target machine is formulated in CAP (eventually with the aid of CAPVM) and simulated by CAPSIM. The program to be verified/evaluated is stored into the simulated main store and thereby executed. This happens under control of CAPTEST.

- controller design:
The algorithm representing the task of a μ-processor based controller is formulated in CAP and translated into code of a selected processor via CAPCOM and CAPCCL. A high-level verification may take place via CAPSIM and CAPTEST. The design is documented with the aid of CAPDOC.

2) Specification support

The specification language in our CAD system is the language CAP /Ra1/. CAP stands for Concurrent Algorithmic Programming Language. This language is similar to PL/1 (we did not want to add an additional storey to the Tower of Babel) but offers a lot of important additional features including description of real-time behaviour, adequate data types, interrupt handling and structural description.
The most important feature of the language CAP is the ability to describe in an as well natural as concise manner concurrency.

This is done on the basis of a slightly modified Petri Net model. Surely, PL/1 may also be used to describe cooperating sequential processes (for a Petri Net based compile time analysis of multi tasking PL/1 programs see /HE1/) but CAP is intended to be a language to describe highly concurrent systems too.

2.1 Short description of the language CAP

CAP has been designed to be suitable as well for firmware implementation as for hardware description on various levels of abstraction including PMS-level, RT-level and IC-level. This implies that, similar to most register transfer languages /BA1/, CAP can be used as nonprocedural language. This means that the ordering of statements describing the operation of the system are not attached to any meaning. Similar to Dijkstra's "Guarded Commands" /DI1/ statements are associated with an explicite condition for execution of the operation described by the statements.
(In many cases procedural programming is much more comfortable. Therefore procedural programming is supported by CAP as well. In particular structured concurrent programming is the favorite programming style in CAP.)
The above conditions associated with statements are the firing of an associated Petri Net transition.

2.1.1 Control structures in CAP (basic concept)

Similar to LOGOS Control Graphs /RO1/ we have different types of transitions /RA2/: An AND-transition (nearly the usual Petri Net transition), an asymmetric OR-transition, a DECIDER-transition and transitions for blockstructuring (CALL, BLKHEAD, BLKEND, RETURN). A similar set of transitions is used in E-nets /NO1/. The main difference is that in E-nets one has tokens associated with attributes while in CAP nets one has transitions associated with attributes.
Blocks are generally activated as concurrent activities, a monitor concept is provided.
Places may have any fixed finite or an infinite capacity and the firing -rule of every transition type includes the condition, that every output-place of a transition must be marked below its capacity. (Note that by this we have boolean and integer places in the sense of /YB1/ and in addition bounded integer places.)
A general nonprocedural CAP program consists out of a set of statements, each in accordance with the following syntax:
<interpreted transition> ::=
<transition><data manipulation>

Transitions are described using label-variables as identifiers for places and ON-conditions to identify the type of transitions:

AND- transition:
ON(&(<label>≺,<label>≻$^{0:n}$)):≺<label>:≻$^{0:n}$
This transition is firable if every input-place (identified by the <labels> within the ON-condition) has at least one token and every output-place (the <labels> after the ON-condition) is marked below its capacity. If the transition fires, it withdraws a token from every input-place and puts one into every output-place.

OR-transition:
ON(|(<label>≺,<label>≻$^{0:n}$)):≺<label>:≻$^{0:n}$
This transition is firable if at least one input-place has at least one token and every output-place is marked below its capacity. If the transition fires, it withdraws a token from the leftmost marked input-place and puts one into every output-place. (For special applications we offer in addition to this fixed priority rule event-dependent rules like FIFO, LIFO, Roun Robin. Oviously all these priority rules make sense only if there is at least one bounded outputplace of the OR-transition considered.)

DECIDER-transition:
ON(<label>):IF <expression> THEN <label1>:
ELSE <label2>:
In this case also both output-places have to be marked below their capacity. Otherwise the net would become potentielly unsafe.

CALL-transition:
ON(<label1>):<label2> CALL <label3>;
This is a special AND-transition with one input-place and two output-places. The calling activity proceeds via <label2>.

RETURN-transition:
ON(&<label1>,<label2>):<label3>:
This is a special AND-transition with two input-places and one output-place. It identifies a backsynchronization of a concurrent activated task with the calling task.

BLKHEAD-transition:
ON CALL(<label>≺,<label>≻$^{0:n}$):<label>:
PROCEDURE...
This transition has an additional input-place which is not transparent to the user. This input-place controls the activation state of the procedure. The BLKHEAD-transition is firable, if this intransparent input-place is marked and there is at least one request, i.e. at least one transparent input-place is marked. Again

we have a priority rule.

BLKEND-transition:

```
ON(<label>):END;
```

Note that BLKHEAD-BLKEND-pairs model a special monitor. The original philosophy /HO1/ however has been modified in the sense that not only the request-coordination but also the manipulation of the resource is handled by the monitor. (The resource is treated as abstract data type!)
This explicite description of transitions although being the basic concept of CAP is not used in most cases. Preferable is the implicite generation of transitions via structured concurrent programming (see below).

2.1.2 Data manipulation in CAP

Data manipulation is done by PL/1 like constructs.
Every used variable has to be declared.
The basic data types are:

- bitstrings of arbitrary length and direction
- characterstrings of arbitrary length, direction and code
- integers of arbitrary length and various representations including two-complement, one-complement, unsigned integer, packed decimal
- floating point variables.

As in PL/1 (using the same notion) n- dimensional arrays and arbitrary structures are available. Variables may be associated with various attributes. The most important attribute is the "IMPLICIT"-attribute. A variable with this attribute is comparable with a "terminal" in CDL /CH1/ i.e. it gets implicitely a new value every time a variable on which it depends on gets a new value.
Variables may be initialized or superimposed on other variables or fixed addresses.
There are the following operators:

- arithmetic:
  +, -, *, /, mod, **. They get their concrete meaning by the operands they are used with.
- logic:
  & (and), | (or) ,@(exor) \&(nand)
  \| (nor), \@(coincidence), \(not)
  Besides "not" they may be used either as dyadic or as monadic (APL-reduction) operators.
- relational: =,\= ,<, >, <=, >=
- string: ||(concatenation), substr

Note that in most cases we have chosen the PL/1-symbol for operators. There is a well defined natural precedence between the operators. Expressions may be formed in the usual way. The string operators may be used on the left side of an assignment too. also multiple assignment is possible. The assignment-symbol is := (like ALGOL). In the sense of CDL /CH1/ assignments to an "IMPLICIT"-variable means a connection, otherwise a transfer. Constants are written in the PL/1 way besides bitstrings where the more comfortable XPL-notion /MK1/ has been chosen. I/O is done by simply referencing FILE-variables.

2.1.3 Example of a CAP-program with explicite description of the controlling Petri Net.

See fig. 1-3 for a CAP program with two concurrent cyclic processes with synchronization,
the controlling CAP-net
and an equivalent Petri Net (for construction see /RA2/).

2.1.4 Structured concurrent programming

Up to now we only introduced assignments as <data manipulation>. We now introduce compound statements for <data manipulation> with inherent control structure.

```
<data manipulation> ::= <assignment>
                      | <if>
                      | <call>
                      | <terminator>
                      | <group>
```

<if> and <call> have the same syntax and semantics as in PL/1. Note that in respect to both aspects they are slightly different from the DECIDER-transition or the CALL-transition resp..
A <terminator> may be simply a ";" denoting an empty statement. Most interesting is <group>:

```
<group> ::= <grouphead><groupbody> END;

<groupbody> ::= «<data manipulation>»1:n
<grouphead> ::= <simple DO>
              | <while>
              | <case>
              | <replication>
                      SEQUENTIAL
<simple DO> ::= DO  { PARALLEL   } <terminator>
                      CONCURRENT
```

The meaning is self explanentatory. Note that in concurrent groups there is no synchronization in contrary to parallel groups.

```
                   SEQUENTIAL
<while>::= DO    { PARALLEL    } WHILE
                   CONCURRENT
             <expression><terminator>
```

This means a usual loop. The relative ordering of the data manipulations within the group can be specified either as sequential or parallel or concurrent.

```
<case> ::= DO CASE <expression><terminator>
```

This group is defined as in XPL, i.e. if expression has the value n the n+1-th data manipulation within the group will be executed.

```
                         SEQUENTIAL
<replication> ::= DO {PARALLEL    }
                         CONCURRENT
              SEQUENTIAL
  <indexrange>{PARALLEL   }<terminator>
              CONCURRENT
```

In this case the relative ordering of the <data manipulation> within the group as well as the ordering of the application of the different indices can be specified. <indexrange> is defined similar to PL/1. Note, that within a compound <data manipulation> no transition can explicitely be specified. A procedure may consist out of a single (usually compound) <data manipulation> . In this case we call it a "structured concurrent procedure".
We showed /RA3/ that

a) every structured CAP- program (this is a CAP-program consisting only out of structured procedures) is
   - deadlock free
   - proper terminating
   - reusable.
b) on the other hand for every CAP net having these chracteristics there is a semantic equivalent structured CAP program.

(For a constructive proof of a similar result for "Control Nets for Asynchronous Systems" /HY1/ see /BSY/.)

2.1.5 Example of a structured CDL-like CAP program

See fig.4 for a example that describes in CAP the complementer example of Y. Chu /CH1/. The CDL origin implies that there is no asynchronous parallelism.

2.1.6 Additional features of CAP

Technology parameters may be specified in an descriptive or postulative manner.
As descriptive specification we implemented a delay parameter while as postulative specification we have limitation parameters for time and memory consumption.
The postulative specifications are important control inputs for our optimizing code generator while the descriptive specifications are used for simulation and analysis. Technology parameters may be included in every statement- <terminator>, thus offering dedicated specification.
Another important feature is the ability to specify interrupt structures in a detailled manner.

2.2 Translation of CAP (Component CAPCOM)

We implemented in a very short time (appr. 1.5 years) a compiler for CAP. This compiler translates CAP source programs into an intermediate language (CAPID). This method has been chosen as we are processing CAP programs in different ways.
We have implemented as well an optimizing code generator for a variety of μ-processors as an interpreter serving as well as RT-simulator. Both use CAPID as input.
The CAP compiler is a two pass compiler with an optional additional step to generate a formatted printout of the source text including level-count, nesting-count, procedure identifications and a highly sophisticated xref-tabel which not only indicates the use of a variable but also identifies the way of use (left hand side or right hand side of an assignment, declaration, use within control expression etc.).
We use a syntaxdirected LR(1) parser with an excellent error correcting facility with good diagnostics.
In addition contextsensitive dependencies are checked carefully (e.g. type compatibility).
The compiler runs currently on a DEC-10 and a IBM 370. It is written in SIMULA.

3. Support of verification/evaluation (components CAPSIM, CAPTEST, CAPVM, CAPLIN)

We have to consider two different tasks:

a) the verification/evaluation of CAP programs
b) the verification/evaluation of programs running on virtual machines, the machines being described in CAP (see fig.5).

Although interesting compile-time verification algorithms may be implemented, up to now we concentrated ourselves on dynamic verification via simulation.
For this purpose we implemented an interpreter for CAPID as CAP runtime system.
This tool is very flexible and transparent to the debugger system which we have implemented too. This debugger system is well suitable for debugging CAP as for debugging arbitrary programs running on arbitrary virtual machines described in CAP. Up to now, besides a couple of hypothetical processors, we have CAP-models of the PDP-8, the INTEL 8085 and the TI 990. In addition we already have,generated by CAPVM the overall control structure of the MCS 6500, the NS PACE and the INTEL 4004.
The debugger includes a variety of commands like tracing, setting of variables, inspecting of variables, setting and inspecting of clocks etc..
The CAP runtime system is at the same time an excellent RT-simulator. Petri Nets being the basic concept for the potential concurrent control flow within CAP programs, we consequently designed a discrete-event-oriented simulation system. An event in this context is defined as firing of a Petri Net transition. The structure of the Petri Net is reflected precisely in an internal data structure (tabel driven simulation).

Let T be a transition, t_i be a point of system time. Assume T has become firable at t_i. The first step is to calculate the point of time $t_j \geq t_i$ the transition will fire. For this time t_j the event representing the firing of T is generated and stored into a properly organized queue. The associated data-operations (if present) are initiated immediately. They may occur at different points of time within the period$[t_i, t_j]$ the transition is activated. If system-time has reached the scheduled firing-time t_j of the transition T, the firing of the transition T is simulated according to its firing rule. As a consequence some transitions (inclusive T!) may become firable. Each transition T' which has become firable is handled in the same manner as T.
Our simulator is not only capable to handle "clean" Petri Nets as described above but also can simulate parallel control flows being distorted by interrupts. In the language CAP different types of interrupts can be programmed. Included are interrupts with programmable priority and interrupts that, after execution of the interrupt handling routine, cause the control flow to continue at a different state than the state being interrupted.
Within our simulator the occurence of interrupts is controlled by a programmable random-generator. Interrupts are represented as special transitions being capable of activating themselves. They are processed like normal transitions. In addition an interrupt-transition may force a reorganzation of the marking in a way that would be impossible in respect to the net-topology. (For a detailled discussion of interrupts in concurrent systems see /GR1/.)

4. Support of documentation (component CAPDOC)

Our CAD system has been designed to be used in an idustrial environment. This implies that documentation is of great importance. As stated above, structured concurrent programming is the favorite programming style for CAP. To emphasize this, our system offers documentation aids especially for structured concurrent programs. This is done by a system generating modified Nassi-Schneiderman-diagrams /NS1/ out of structured CAP programs. The modification is due to the additional concurrency feature of CAP. The programmer can control the level of abstraction very easy by special control statements. By this he can produce a hierarchy of documents at various levels of abstraction either during the design process of stepwise refinment or after completion out of the final program.
As there are cases, where unstructured programming is useful, this programming style is supported too. For this purpose we implemented a system that generates a Petri Net representation of the control flow. Practical considerations implied however that the representation of this Petri Net is very unusual.

5. Support of implementation (components-CAPCOM, CAPLIN, CAPCCL)

A very ambitious part within our CAD-system is the implementation of a goal-processor independent optimizing cross compiler for CAP under special consideration of μ-processors as goal-processors.
As there is a special paper on this topic within this volume /CA1/ I only will give a rough summary.
An important application field for CAP is the deign of μ-processor based controllers (i.e. the algorithm representing the task of a controller is given in CAP).
This algorithm has to be implemented on an arbitrary μ-processor (or a multi-μ-processor system in a future version).
As typically there are very restrictive time and memory limitations in controller design, we had to implement a compiler with sophisticated optimization.
The code generating process is done in several steps, where we remain processor independent as long as possoble. Even the basic optimization is done without consideration of a special goal processor. Processor specific optimization runs only if a time- or memory-restriction is violated.
Adding a new goal processor to the system means simply providing some tabels describing this processor. Even this task is supported by an automated generation process.

6. Acknowledgement

This project is managed in cooperation with Kienzle Apparate GmbH, Villingen, West Germany; partly funded within "3. DV Förderungsprogramm" by the government of the Federal Republic of Germany. Besides the author the system has been designed by H. J. Appelrath, P. Cazacu, G. Fischer, T. Kunze, U. Mayer, G. Szwillus and a couple of excellent students.
We thank Mr. Walter, Kienzle GmbH for supporting the implementation of the system, Mr. Heiß, Kienzle GmbH for formulating machine models in CAP and Mr. Hayer, Kienzle GmbH for helpful discussions about the documentation of unstructered CAP programs.

Fig.1) Example of a nonprocedural CAP program (simplified output of formatter)

```
C A P        FORMATTED  LISTING


LINE   LV NT

00100  0  0  ON CALL(REQ1,REQ2):RUN:PROCEDURE(UB,PERCENTAGE);                      PR1
00200  1  0     DCL (UB,PERCENTAGE,INRECCNT,OUTRECCNT) FIXED, A CHAR(EBCDIC,80);
00300  1  0     DCL (RUN,WCS,WR,WF,WCP,RCS,TR,TS,TP,RCP,FULL) LABEL;
00400  1  0     DCL FILE (SYSIN,SYSOUT) CHAR(80);
00500  1  0     ON(RUN):WCS:RCS: OUTRECCNT,INRECCNT := 0;
00600  1  0     ON(!(RCP,RCS)):TS: A := SYSIN;
00700  1  0     ON(TS):TR:TP: SUBSTR(A,0,10)!!SUBSTR(A,10,70) := A;
00800  1  0     ON(TP):RCP: INRECCNT := INRECCNT+1;
00900  1  0     ON(!(WCP,WCS)):WR: PERCENTAGE := 100*(UB OUTRECCNT)/UB;
01000  1  0     ON(WR,TR):WF: SYSOUT := A;
01100  1  0     ON(WF): IF PERCENTAGE>= 8 THEN FULL: ;
01200  1  0     ELSE WCP: ;
01300  0  0  ON(FULL):END;                                                          PE1
```

Fig.2) CAP-net for program of fig.1 (main cycles especially indicated

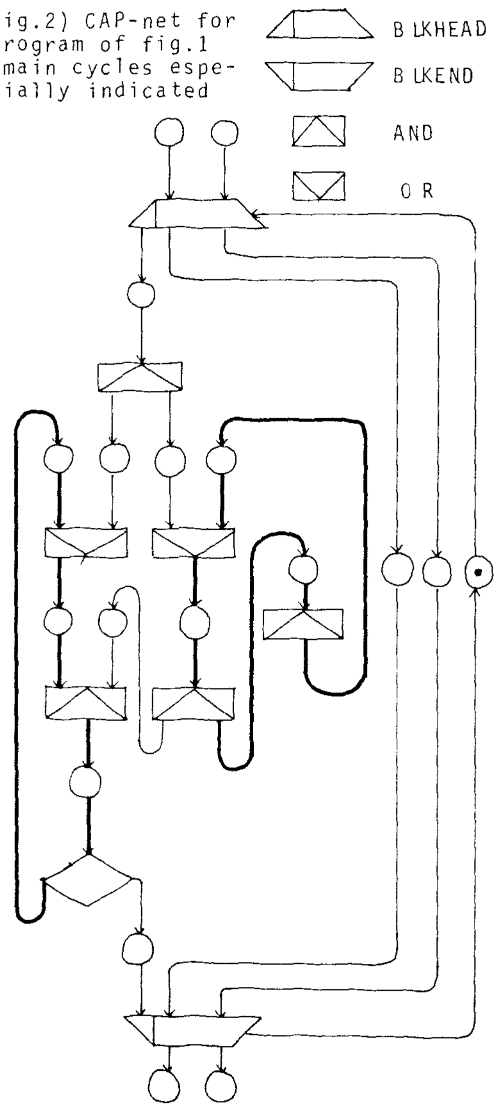


Fig. 3) Equivalent standard Petri Net for CAP net in fig. 2 (main cycles again especially indicated). It has been assumed that the capacity of every place of the CAP net is one.

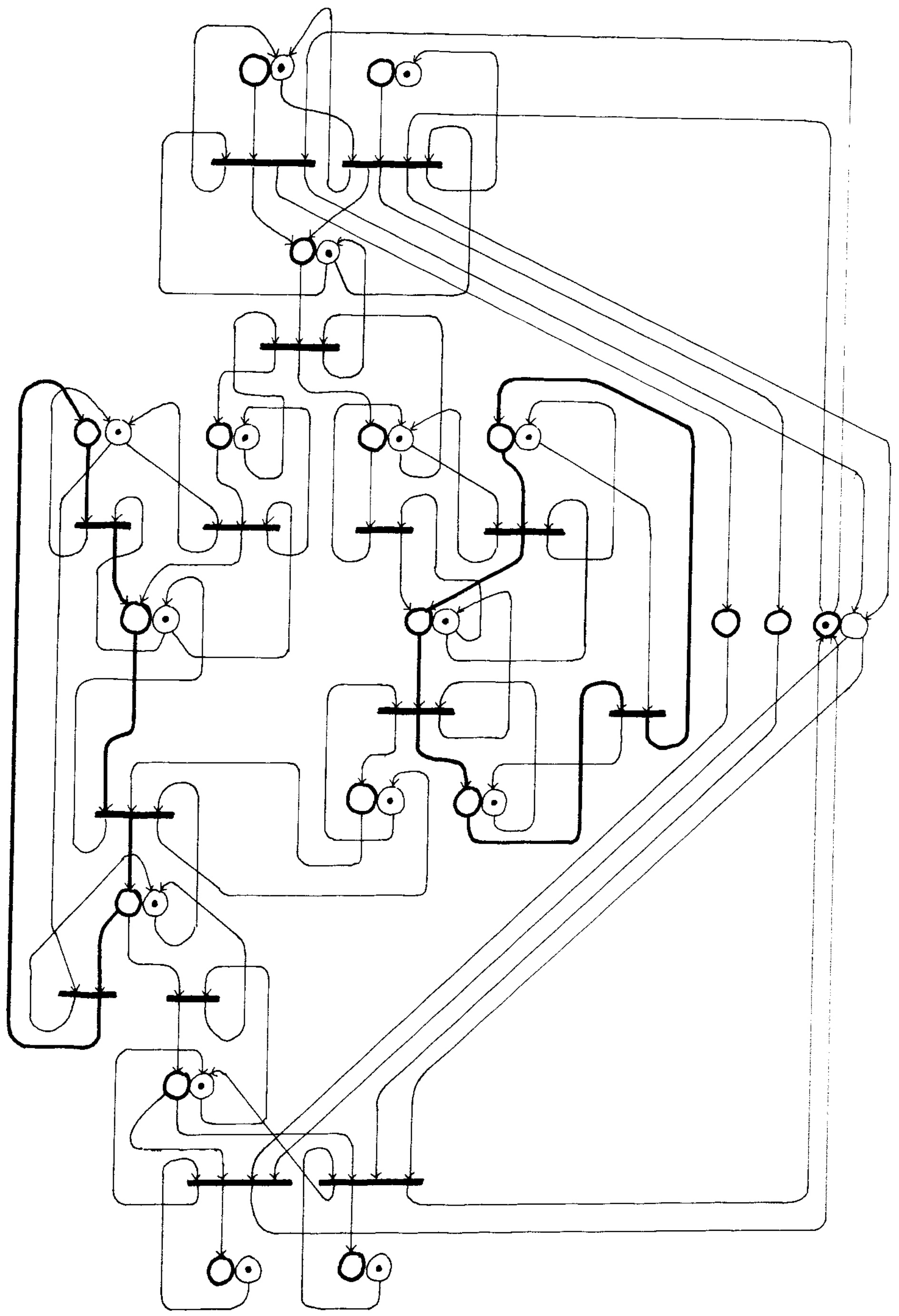
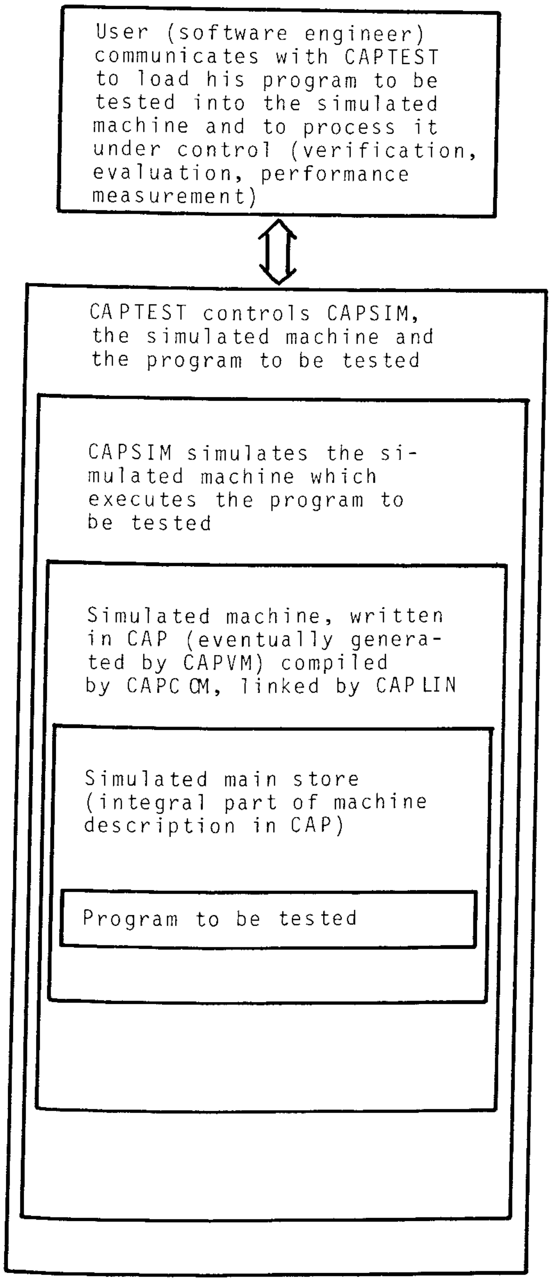


Fig.4) Example of a structured CAP program (output produced by formatter/xrefgenerator without procedurename-option, info character inicates the usage, D stands for declaration, L for left side of an assignment, R for right side, C for usage within a control expression)

```
C A P        FORMATTED  LISTING

LINE   LV NT

00100  0  0  ON CALL(COMPLEMENT):PROCEDURE;                                PR1
00200  1  0     DCL A BIT(5), C FIXED(3,UN) INIT(0),
00300  1  0         FINI BIT(1), S BIT( 3) INIT("100"),
00400  1  0         P BIT(1) IMPLICITE;
00500  1  0     DO PARALLEL;                                                  DO1
00600  1  1        P := \P DELAY(UP 10, DOWN 10);
00700  1  1        IF P THEN DO PARALLEL WHILE "1";                              DO2
00800  1  2           IF SUBSTR(S,0) THEN DO PARALLEL;                              DO3
00900  1  3              S := "010";
01000  1  3              A := SUBSTR(A,0)!!SUBSTR(A,1,4);
01100  1  3              C := C+1;
01200  1  2           END;                                                          DE3
01300  1  2           IF SUBSTR(S,1) THEN IF C=5 THEN S := "001";
01400  1  2           ELSE S := "100";
01450  1  2           ELSE;
01500  1  2           IF SUBSTR(S,2) THEN DO PARALLEL;                              DO4
01600  1  3              S := "000";
01700  1  3              FINI := "1";
01800  1  2           END;                                                          DE4
01900  1  1        END;                                                          DE2
02000  1  0     END;                                                          DE1
02100  0  0  END;                                                          PE1
                                    C A P        CROSS - REFERENCE  TABLE

IDENTIFIER             INFO CHARACTER / LINE NUMBER


A                      D   200
                       L  1000
                       R  1000  1000
C                      C  1300
                       D   200
                       L  1100
                       R  1100
COMPLEMENT             D   100
FINI                   D   300
                       L  1700
IMPLICITE              D   400
P                      C   700
                       D   400
                       L   600
                       R   600
S                      C   800  1300  1500
                       D   300
                       L   900  1300  1400  1600
SUBSTR                 C   800  1300  1500
                       R  1000  1000
```

Fig. 5) Application example :
Cross development of software


User (software engineer) communicates with CAPTEST to load his program to be tested into the simulated machine and to process it under control (verification, evaluation, performance measurement)
CAPTEST controls CAPSIM, the simulated machine and the program to be tested
CAPSIM simulates the simulated machine which executes the program to be tested
Simulated machine, written in CAP (eventually generated by CAPVM) compiled by CAPCOM, linked by CAPLIN
Simulated main store (integral part of machine description in CAP)
Program to be tested

7. References

/BA1/: M. R. Barbacci:
A Comparison of Register Transfer Languages for Describing Computers and Digital Systems
IEEE ToC Vol. C-24 No. 2 (1975)

/BSY/: Z. Barzilai, E. Strasbourger, M. Yoeli:
On Structured Parallel Programming
Technion- Israel Institute of Technology, Computer Science Department, Technical report No. 129 (1978)

/CA1/: P. Cazacu, H.-. Appelrath:
A High Level Programming Language and a Universal Optimising Cross-Compiler for Microprocessors
Proceedings of Euromicro '79, Göteborg (1979)

/CH1/: Y. Chu:
Introducing CDL
IEEE Computer, December 1974

/DI1/: E. W. Dijkstra:
Guarded Commands, Nondeterminacy and Formal Derivation of Programs
CACM Vol. 18, No. 8 (1975)

/HE1/: O. Herzog:
Zur Analyse der Kontrollstruktur von parallelen Programmen mit Hilfe von Petri-Netzen
Forschungsbericht Nr. 24 der Abt. Informatik der Univ. Dortmund (1976)

/HY1/: O. Herzog, M. Yoeli:
Control Nets for Asynchronous Systems, Part I
Technion- Israel Institute of Technology, Computer Science Department, Technical report No. 74 (1976)

/HO1/: C. A. R. Hoare:
Monitors: An Operating Systems Structuring Concept
CACM Vol. 17, No. 10 (1974)

/GR1/: K. Gröning:
Zur Darstellung und Simulation von Interrupts in parallelen Systemen durch Petri-Netze
Diplomarbeit, Abt. Informatik, Univ. Dortmund (1979)

/MK1/: W. M. McKeenman et al.:
A Compiler Generator
Prentice Hall (1970)

/NO1/: J. D. Noe, G. J. Nutt:
Macro E-nets for Representation of Parallel Systems
IEEE ToC Vol. C-22, No. 8 (1973)

/NS1/: I. Nassi, B. Shneiderman:
Flowchart Techniques for Structured Programming
SIGPLAN Notices, Nov. 1976

/RA1/: F. J. Rammig:
An Introduction to the Concurrent Algorithmic Programming Language CAP, or Looking at CAP with the Revised Ironman's Eyes
Forschungsbericht Nr 80 der Abt. Informatik der Univ. Dortmund (1979)

/RA2/: F. J. Rammig:
Überlegungen zur Kontrollstruktur einer Computer-Hardware-Beschreibungssprache
Forschungsbericht Nr. 62 der Abt. Informatik der Univ. Dortmund (1978)

/RA3/: F. J. Rammig:
Zur Bedeutung des strukturierten nebenläufigen Programmierens im CAD-Bereich
In: B. Reusch(ed.): Systeme, Automaten, Schaltwerke, Haus Ahlenberg Mai 1978. Forschungsbericht Nr. 73 der Abt. Informatik der Univ. Dortmund (1978)

/RO1/: C. W. Rose:
LOGOS and the software engineer
FJCC (1972)

/YB1/: M. Yoeli, Z. Barzilai:
On Behavioral Description of Communication Switching Systems
Technion- Israel Institute of Technology, Computer Science Department, Technical report No. 99 (1977)