The implementation of the Computer Hardware Description Language CAP and its applications

Franz J. Rammig

University of Dortmund

Abstract:
This paper describes the current implementation and major applications of the language CAP (formerly DIGITEST II /RA 1/). CAP stands for Concurrent Algortithmic Programming Language, showing the central underlying concept of this language: The integration of an algorithmic programming language and Petri Nets.
Since 1975 we have implemented a large amount of software around CAP.
In this paper these components of the CAP CAD-system are discussed. The major applications of the system are:
- *processor design*
- *software design for not yet realized processors*
- *controller design.*

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 CAD-system for processor design and firmware implementation.
The system includes the basic components of a CAD-system, supporting
- specification
- verification/evaluation/modification
- documentation
- implemenation.

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.

2) Specification support

The specification language in our CAD system is the language CAP /RA 2, RA 3/ (formerly DIGITEST II/RA 1/). 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 a natural as well as concise manner, concurrency. This is done on the basis of a slightly modified Petri Net model.

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 CAP can be used as nonprocedural language /BA 1/. This means that the ordering of statements describing the operation of the system is not attached to any meaning. Statements are associated with an explicit 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 /RO 1/ we have different types of transitions /RA 4/: An AND-transition (usual Petri Net transition), an asymetric OR-transition, a DECIDER-transition and transitions for blockstructuring (CALL, BLKHEAD, BKLEND, RETURN).
Blocks are generally activated as concurrent activities, a monitor concept is provided. Places may have 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 /YO 1/ 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 transition:

CH1436-5/79/0000-0138$00.75 © 1979 IEEE

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 rule like FIFO, LIFO, Round Robin).
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 potentially 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 <label 2>.
RETURN-transition:
ON(<label2>, <label2>):<label2>:
This is a special AND-transition with two input-places and one output-place. It identifies a back synchronization 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. Note, that the BLKHEAD-transition models a special monitor.
BLKEND-transition:
ON(<label>):END;
This explicit description of transitions although being the basic concept of CAP is not used in most cases. Preferable is the implicit 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:
- bit strings of arbitrary length and direction
- character strings 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 notation) n-dimensional arrays and arbitrary structures are available. Of course scattered data carriers with individual names for the parts are possible. 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, i.e. it gets implicitly 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), ¬@(COINC), ¬(NOT)

Besides NOT they may be used either as dyadic or as monadic (APL-reduction) operators.
- relational: =, =, < , > , <= , >=
- string: || (concatenation), substr (substring)

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 assignments to a "IMPLICIT"-variable means a connection, otherwise a transfer.
Constants are written in the PL/1 way besides bit strings where the more comfortable XPL-notation /HO 1/ has been chosen. I/O is done by simply referencing FILE-variables.

2.1.3 Example of a CAP-program with explicit description of the controlling Petri Net (see appendix 1)

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>

<simple DO> ::= DO {SEQUENTIAL | PARALLEL | CONCURRENT} <terminator>

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

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

This means a usual loop. The relative ordering of the data manipulations within the group can be specified 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.

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

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 explicitly be specified. A procedure may consist out of a single (usually compound) <data manipulation>. In this case we call it "structured procedure".

We showed /RA 5/ that

a) every structured CAP-program (this is a CAP-program consisting only of structured procedures) is - deadlock free
- proper terminating
- residue free

b) on the other hand for every CAP-program having these characteristics there is a semantic equivalent structured CAP-program.

2.1.5 Example of a structured ISP-like CAP-program

(see appendix 2)

2.1.6 Additional features of CAP

Technology paramenters may be specified in a 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.

Examples:

```
A := SHL(C&D, 5) DELAY(UP 10 DOWN7);
```

If the new value of A is greater than the old there is a delay of 10 time units, if it is less the delay is 7 units.

```
DO PARALLEL DELAY(20);
    A := B;
    B := A;
END;
```

This describes a simple swap (DO PARALLEL denotes synchronized parallelism, DO CONCURRENT would have resulted in a data conflict in this example).

The overall delay is 20 time units.

Another important feature is the ability to specify interrupt structures in a detailled manner.

2.2 Translation of CAP

We implemented in a very short time (appr. 1,5 years) a compiler for CAP. This compiler translates CAP source-programs into a 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 a RT-simulator. Both use CAPID as input. CAPID consists out of two major parts: a list-structured description of the controlling Petri Net and a description of the associated data manipulation in postfix polish. Additional tables serve as description of technology parameters. CAPID has been designed in such a way that it can be directly interpreted very effectively by our RT-simulator.

We use a syntaxdirected LR-1 parser with an excellent error correcting facility with good diagnostics.

The compiler runs currently on a DEC-10, a SIEMENS 7.738 an an IBM 370. It is written in SIMULA.

3. Support of verification and modification

We have to consider two different tasks:

a) the simulation of designs which have been specified in CAP
b) the verification and modification of programs running on virtual machines, the machines being described in CAP

Although interesting compile-time verification algorithms may be implemented, up to now we concentrated ourselves on dynamic verification aids 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 as well suitalbe 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 PDP8, the INTEL 8085 and the TI990.
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 (table 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 can also simulate parallel control flows 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, cause the control flow to continue at a different state than the state being interrupted.
Within our simulator the occurrence 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 except that their firing delays the firing of other transitions. In addition an interrupt-transition may force a reorganization of the marking in a way that would be impossible in respect to the net-topology.

4. Support of documentation

Our CAD system has been designed to be used in an industrial 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 /NS 1/-diagrams 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 refinement 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 gnerates 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

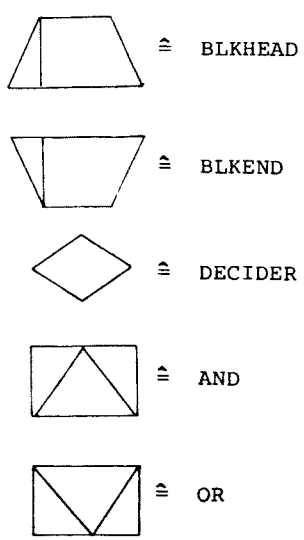
A very ambitious part within our CAD-system was the implementation of a goal-processor independent optimizing compiler for CAP under special consideration of µ-processors as goal-processors.
An important application field for CAP is the design of µ-processor-based controllers (i.e. the algorithm representing 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 limiations in controller design, we had to implement a compiler with sophisticated optimization.
The code generating process is done on several steps, where we remain processor independent as long as possible. Appendix 3 demonstrates the code generating process with the aid of an extremely simple example. Adding a new goal processor to the system means simply providing some tables describing this processor.
In a first step we translate CAPID into another intermediate language (CCLID). This is the code of a virtual three-address machine. It covers all µ-processors available now and a lot of other processor too. During this step happens the first part of optimization, i.e. all processor independent optimization like loop unrolling, dead code elimination, redundant subexpression elimination, etc.. Controlled by some processor dependent tables in a second step code for the selected processor is generated.

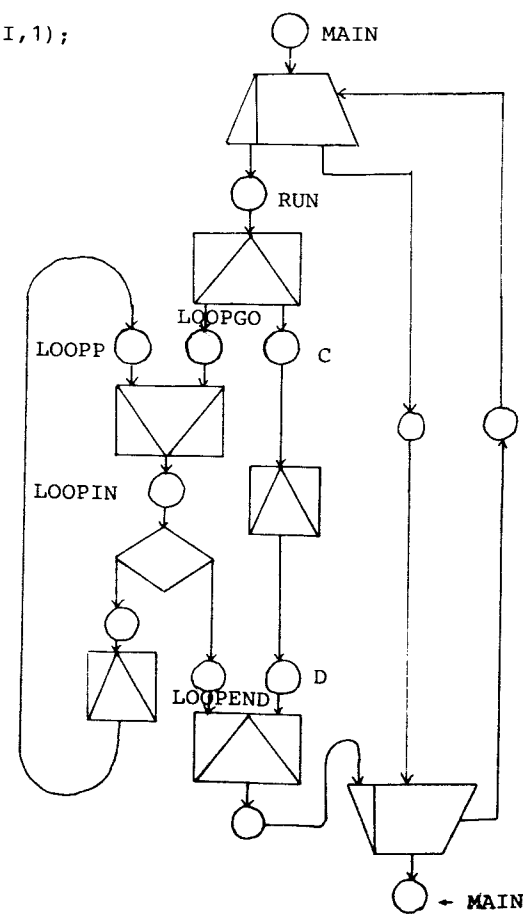
These tables include a description of the instruction set and a definition of the semantics of the instruction expressed with the aid of CCLID. These tables are produced manually or by a code-generator-generator which we are just implementing /CA 1/. Having generated code for the selcted processor we check whether any limitations are violated. In this case additional processor dependent optimization algorithms (e.g. code motion, optimized register assignment) are activated unless there is no more violation or the restriction show to be unfullfillable.
Up to now we have available the controlling tables for the 8085 and the 8048 (generated manually) while late in 1979 the code gnerator generator will be available which makes adding a new processor type to the system a relative simple task.

Appendix 1)

```
ON CALL(MAIN): RUN: PROCEDURE;
   DCL(B,A) CHAR(EBCDIC,80), /* EBCDIC string */
      I FIXED(8,TC)
   DCL FILE(SYSIN, SYSOUT) CHAR(80); /* I/O ports */
   ON(RUN)? LOOPGO: C: B,A := SYSIN; /*Fork, input, multiple assignment*/
   ON(C): D: SUBSTR(A,0,10)|| SUBSTR(A,11,70) := A; /*rotate*/
   ON(|(LOOPGO,LOOPP)): LOOPIN: I := 1;
   ON(LOOPIN): IF I≠80 THEN LOOPT: I := I+1;
                       ELSE LOOPEND:;
   ON(LOOPT): LOOPP: SUBSTR(B,I,1) := 'T'||SUBSTR(B,I,1);
ON(LOOPEND,D): END;
```

Controlling Petri Net for above example:





Appendix 2) Part of PDP-8 description

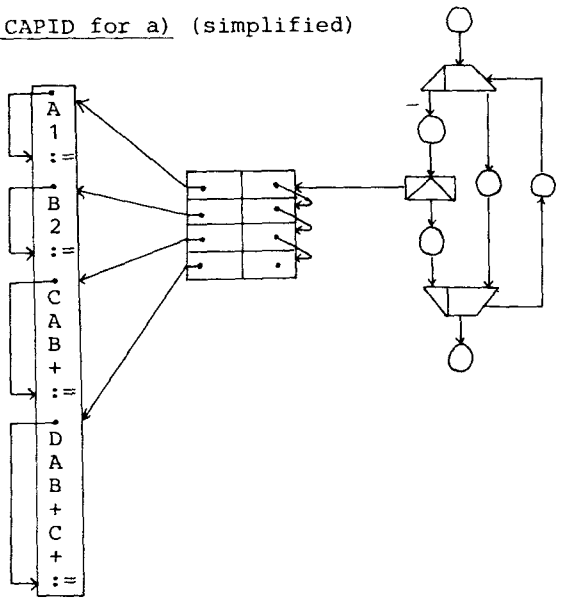
```
ON CALL (PDP8): PROCEDURE;
   DCL AC FIXED(12,TC), /*two's complement arithmetic*/
       L BIT(1),
       PC FIXED(12,TC),
       (RUN, I_STATE, IO_PULSE(3) BIT(1);
   DCL 1 INSTRUCTION, /*scattered register*/
         2 PAGE_ADDRESS FIXED(7,UN),
         2 IND_BIT BIT(1),
         2 PAGE_O_BIT BIT(1),
         2 OP BIT(3);
   DCL M("(3)10000") FIXED(12,TC), /*address space specified as octal number*/
       .
       .       /*additional declarations, omitted here*/
       .;
   ON CALL(ADDRESSING): PROCEDURE;
      /*description of addressing mechanism, ommitted here*/
   END;

   DO SEQUENTIAL WHILE RUN; /*instruction cycle*/
      IF I REQUEST & I_STATE
         THEN DO PARALLEL;
                 M(0) := PC; I_STATE := "0"; PC := 1;
              END;
         ELSE DO SEQUENTIAL;
                 DO PARALLEL;
                    INSTRUCTION := M(PC); PC := 1;
                 END;
                 CALL ADDRESSING;
                 DO CASE INSTRUCTION.OP;
                    /*case 1: and*/ AC := AC & M(Z);
                    /*case 2: add*/ L || AC := (L ||AC)+("0"||M(Z));
                    /*case 3: isz*/ DO SEQUENTIAL;
                                       M(Z1) := M(Z)+1;
                                       IF M(Z1)=0 THEN PC := PC+1;
                                    END;
                    /*case 4: dca*/ DO PARALLEL;
                                       M(Z) := AC;
                                       AC := 0;
                                    END;
                       .
                       .
                       .
                 END; /*of case*/
              END; /*of sequential*/
           END; /*of instruction cycle*/
        END; /*of PDP8 description*/
```

Appendix 3) Extremely simple example to demonstrate the compilation process

a) CAP program:

```
ON CALL(ADD): PROCEDURE;
   DCL (A,B,C,D) FIXED(8,TC);
   DO SEQUENTIAL;
      A := 1;
      B := 2;
      C := A+B;
      D := A+B+C;
   END;
END;
```

b) CAPID for a) (simplified)



c) CCLID for a) (simplified)

```
LOAD A , 2
LOAD B , 2
ADD  C , A , B
ADD #1 , A , B
ADD  D , C , #1
```

d) 8085 code for a)

```
MVI b , 01H
MVI C , 02H
MOV A , B
ADD C
MOV D , A
MOV A , B
ADD C
MOV E , A
MOV A , E
ADD D
MOV H , A
```

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)

/BR1/ F.T. Bradshaw
"Directed Graph Models for Hardware/ Software Design"
in Proceedings of 1975 International Symposium on Computer Hardware Description Languages and their Applications, New York 3.9.-5.9.1975

/CA1/ P.G. Cazacu, R. Müller, M. Helms:
"Konstruktion eines Codegenerator-Generators"
(Construction of a Codegenerator-Generator) in German
Forschungsbericht Nr. 75 der Abt. Informatik, Universität Dortmund, 1979

/HO1/ W.M. McKeenman et al.:
"A Compiler Generator"
Prentice Hall 1970

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

/RA1/ F.J. Rammig:
"DIGITEST" II: An Integrated Structural and Behavioural Language"
in Proceedings of 1975 International Symposium on Computer Hardware Description Languages and their Applications, New York 3.9.-5.9.1975

/RA2/ 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, Univ. Dortmund, 1979

/RA3/ F.J. Rammig:
CAP manual (in German)
On demand

/RA4/ F.J. Rammig:
"Überlegungen zur Kontrollstruktur einer Computer-Hardware-Beschreibungs-Sprache" (About the Control-structure of a CHDL) in German
Forschungsbericht Nr. 62. der Abt. Informatik, Univ. Dortmund, 1978

/RA5/ F.J. Rammig:
"Zur Bedeutung des strukturierten nebenläufigen Programmierens im CAD-Bereich" (About Structured Concurrent Programming within the Field of CAD)
in Proceedings of "Systeme, Automaten, Schaltwerke, Haus Ahlenberg 8.-12.5.1978, B. Reusch (ed.)
Forschungsbericht Nr. 73 der Abt. Informatik, Univ. Dortmund, 1978

/RO1/ C.W. Rose:
"LOGOS and the Software Engineer"
FJCC 1972

/YO1/ M. Yoeli, Z. Barzilai:
"An Behavioural Description of Communication Switching Systems"
Technicon Israel Institute of Technology, Computer Science Dept., Technical Report No. 99, June 1977