Concurrent Languages in Distributed Systems
G.L. Reijns and E.L. Dagless (editors)
Elsevier Science Publishers B.V. (North-Holland)

DESIGN AIDS FOR HIGHLY DISTRIBUTED HARDWARE

Franz J. Rammig
Universität-Gesamthochschule-Paderborn
Paderborn, Fed. Rep. of Germany

This paper describes how the design process for highly distributed hardware is supported by tools that have been developed and implemented by the author and his collegues. These tools include a hardware description language which is especially tailored towards loosely coupled distributed systems, a simulator for this language and a synthesis algorithm for the control structure of systems described in this language. Both, nMOS and CMOS VLSI circuits may be synthesized by this algorithm.

INTRODUCTION

The design of distributed hardware consisting of numerous loosely coupled, concurrently operating modules is a complicated task. It is rarely supported by traditional design tools like hardware description languages, simulators, synthesis algorithms. Distributed, concurrently operating hardware on the other hand is nothing but an implementation (realization) of a concurrent algorithm (the interpretation algorithm for instructions) that has to be formulated in a concurrent description language. So the key point to the problem is a proper hardware description language that allows the formulation of concurrent algorithms. Looking at a broadband language that is able to accompany the design process over various levels of abstraction this power is requested especially at the system level and the processor level (algorithmic level). At lower levels (register transfer, gate, switch) the nature of the system to be organized as distributed concurrent one is no longer visible. So when discussing hardware description languages we will concentrate on language features for the system level and the algorithmic level.
Concurrency is a feature of the control structure while distribution is a structural feature. Therefore we will mainly discuss concepts for the formulation of control structures and to structure a description (and by this the described system) into a number of cooperating modules.
Finally a system has not only to be described but also to be synthesized. From the point of view of this paper only the realization of the intermodule communication (the global control) is of interest. A possible realization technique for this task is presented.

A MODEL FOR CONCURRENCY : INTERPRETED PETRI NETS

Petri nets /PE1/ are an easy to understand, simple, general, and a mathematically well investigated model of concurrent processes. In this paper we will not discuss ordinary Petri nets but the special descendant used as the semantic foundation of the hardware description language CAP/DSDL /RA1/.
CAP nets are transition/event nets with a heterogenous set of transitions (firing rules) in contrast to the pure Petri net approach where we have only a single one. Other heterogenous nets can be found elsewhere in the literature (e.g. /HY1/, /YG1/, /NO1/, /RO1/).

Def. 1

PG = (P,T,E) is called Petri net graph :<=>
P finite set (of "places")
T finite set (of "transitions")
$E \subset P \times T \cup T \times P$
$P \cap T = \emptyset$
$\forall x \in P \cup T : \exists y \in P \cup T : (x,y) \in E \vee (y,x) \in E$

Def. 2

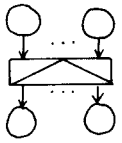
PN = (PG,m_0,R) is called Petri net :<=>
PG = (P,T,E) is Petri net graph
$m_0 \in M = \{m \mid m : P \to \mathbb{N}_0\}$ (initial marking)
$R \in \{r \mid r : T \to f_T\}$ with
$\forall t \in T : (f_t : M \to M)$ (firing rule of t)

While in ordinary Petri nets there is exactly one firing rule for all transitions, we will consider six types of firing rules.

Def. 3

Let be PN = $((P,T,E),m_0,R)$ a Petri net, $t \in T$,
${}^\cdot t := \{p \in P \mid (p,t) \in E\}$, $t^\cdot := \{p \in P \mid (t,p) \in E\}$
The transition t is called AND-transition :<=>
1) t is firable under marking m :<=>
$\forall p \in {}^\cdot t : m(p) > 0$
2) $f_t : M \to M$ is called firing of t :<=>
$f_t(m(p)) = m(p) - 1 \iff p \in {}^\cdot t \wedge t$ firable
$f_t(m(p)) = m(p) + 1 \iff p \in t^\cdot \wedge t$ firable
$f_t(m(p)) = m(p)$ else
Let A(PN) $\subset$ T denote the set of all AND-transitions of a Petri net PN. This type of transition corresponds to the transition in ordinary Petri nets directly.

Symbol



Def. 4

Let be PN = ((P,T,E),m_0,R) a Petri net, t ε T, ˙t, t˙ defined as in Def. 3.

The transition t is called OR-transition :<=>

1) t is firable under marking m :<=> ∃ p ε ˙t : m(p) > 0

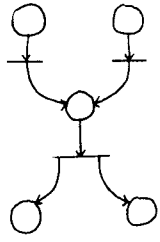
2) f_t : M → M is called firing of t :<=>

$f_t(m(p)) = m(p) - 1$ <=> p ε ˙t m(p)>0 ∧ t firable

$f_t(m(p)) = m(p) + 1$ <=> p ε t˙ ∧ t firable

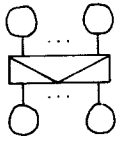
$f_t(m(p)) = m(p)$ else

This transition corresponds in ordinary Petri nets to a subnet like :



Let O(PN) ⊂ T denote the set of all OR-transitions of a Petri net PN.

Symbol :



A third and forth type of firing rules will be defined as Def. 6 and Def. 7.

Def. 5

IPN = (PN,I,D) is called interpreted Petri net :<=>

PN = ((P,T,E),m_0,R) Petri net,

I ε {i | i : T → O ∪ {λ}} with

O = {o | o : dom(o) ⊂ X^D → codom(o) ⊂ X^D}

D manysorted set (of "data objects")

A pair IF = (f_t,i_t) will be called interpreted firing.

For technical reasons within this paper we restrict ourselves on such interpreted Petri nets where only OR-transitions of degree (1,1) and ARBITER/DECIDER-transitions have a nonempty interpretation. Of course that is no semantic restriction.

Def. 6

Let be IPN = (((P,T,E),m_0,R),I,D) an interpreted Petri net, $t \in T$, $^\cdot t = \{p_i\}$, $t^\cdot = \{p_{false}, p_{true}\}$, $i(t) = d \to d$, $i(t)(d) = d$, value(d) $\in$ {true, false} .
The transition t is called DECIDER-transition :<=>

1) t is firable under marking m : = $m(p_i) > 0$

2) $f_t : M \to M$ is called firing of t :<=>

$f_t(m(p_i)) = m(p_i) - 1$

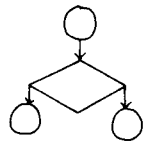
$f_t(m(p_{true})) = m(p_{true}) + 1$ <=> t firable $\wedge$ d = true

$f_t(m(p_{false})) = m(p_{false}) + 1$ <=> f firable $\wedge$ d = false

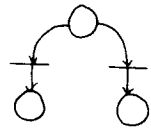
$f_t(m(p)) = m(p)$ else

Let D(IPN) denote the set of all DECIDER transitions of IPN.

Symbol :



In ordinary Petri nets this transition corresponds to a subnet like :



This type of transition can easily be generalized to a "case"-type transition (n branch decider). The CAP/DSDL compiler makes use of this possibility.

Def. 7

Let be IPN = (((P,T,E),m_0,R),I,D) an interpreted Petri net, $t \in T$,
$^\cdot t = \{enable, req_i \mid i = 0,\ldots,n\}$, $t^\cdot = \{run, ret_i \mid i = 0,\ldots,n\}$,
$i(t) = d \to d$, $i(t)(d) = d$, value(d) $\in \{\{req_i \mid i = 0,\ldots,n\} \to [0,n], biject.\}$.
The transition t is called BLKHD-transition :<=>

1) t is firable under marking m :<=>

$m(enable) > 0 \wedge \exists p \in \{req_i \mid i = 0,\ldots n\} : m(p) > 0$

2) $f_t(m(enable)) = m(enable) - 1$

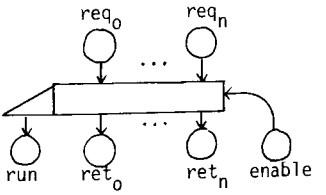
$f_t(m(run)) = m(run) + 1$

$f_t(m(req_i)) = m(req_i) - 1_{} <=> \{m(req_i)>0 \wedge (value(d))(req_i) = \max\{(value(d))(req_j) \mid j=0,\ldots,n\}$

$f_t(m(ret_i)) = m(ret_i) + 1$

Let B(IPN) denote the set of all BLKHD-transitions of IPN. This type of transition is a special case of an arbiter where the arbitration is carried out between the request places req_1 to req_n.

Symbol :



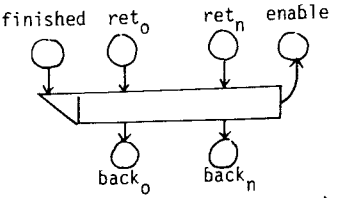
Def. 8

Let be IPN = (((P,T,E),m_o,R),I,D) an interpreted Petri net, $t \in T$,
$^\bullet t$ = {finished, ret_i | i = 0,...,n} , $t^\bullet$ = {enable, $back_i$ | i = 0,...,n} .
The transition t is called BLKEND-transition :<=>

1) t is firable under marking m :<=>
   m(finished) > 0 $\wedge \exists$ p $\in$ {ret_i | i = 0,...,n} : m(p) > 0
2) f_t : M → M is called firing of t :<=>
   f_t(m(finished)) = m(finished) - 1
   f_t(m(enable)) = m(enable) + 1
   f_t(m(ret_i)) = m(ret_i) - 1 } <=>m(ret_i) > 0
   f_t(m($back_i$)) = m($back_i$) + 1

Let N(IPN) denote the set of all BLKEND-transitions of IPN.

Symbol :



BLKHD and BLKEND transitions are generated as pairs by the CAP/DSDL compiler where the respective places named "enable" and "ret_i" are identified. This monitor mechanism is produced for every function or procedure declared in a CAP/DSDL source description, modelling a time-shared hardware device with own internal control.

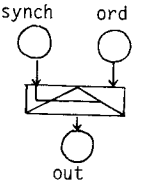
Def. 9

Let PN = ((P,T,E),m_o,R) be a Petri net, $t \in T$, $^\bullet t$ = {synch, ord} , $t^\bullet$ = {out} .
The transition t is called AT-transition :<=>

1) t is firable under marking m :<=> m(synch) > 0
2) f_t : M → M is called firing of t :<=>
   f_t(m(synch)) = m(synch) - 1
   f_t(m(ord)) = m(ord) - 1 } <=> m(ord) > 0
   f_t(m(out)) = m(out) + 1

Let Y(PN) denote the set of all AT-transitions of PN.

Symbol :



AT-transitions are generated by the CAP/DSDL compiler whenever an AT-construct is included in the CAP/DSDL source program. This construct allows the designer to specify control flows that are synchronized with arbitrary data events (e.g. edges of clock signals).

Now we have available the building blocks of the nets to be considered. In order to get nets that can easily be implemented we introduce some global restrictions: First of all, having introduced special types of transitions to model places with backward and forward conflict it no longer makes sense to have places with connectivity degree other than (1,1). Restrictions on the interpretation have already been mentioned. Finally we will restrict our model on 1-safe nets. As the nets obtained are a subclass of the nets used in CAP/DSDL we will call the resulting nets safe restricted CAP nets (SRCN).

Def. 10

Let be IPN = $(((P,T,E),m_0,R),I,D)$ an interpreted Petri net.

IPN is called SRCN :<=>

1) $T = A(IPN) \dot\cup O(IPN) \dot\cup D(IPN) \dot\cup B(IPN) \dot\cup N(IPN) \dot\cup Y(IPN)$

2) $\forall p \in P : |{}^\cdot p| = |p^\cdot| = 1$

3) $\forall p \in P : \forall m \in M : m(p) \in \{0,1\}$

4) $\forall t \in T : t \notin D(IPN) \vee t \notin B(IPN) \vee t \notin A(IPN) \cap \{t \in T \mid \text{degree} = (1,1)\}$
$\Rightarrow i(t) = \lambda$

FROM CONCEPT TO LANGUAGE : CAP/DSDL

CAP/DSDL (Concurrent Algorithmic Programming Language/Digital Systems Description Language) is a broadband hardware description language that covers the entire bandwidth from system level via algorithmic level, register transfer level, gate level, down to switch level. Its basic idea is to take interpreted Petri nets (CAP nets in our case) and to build a PASCAL oriented hardware description language around it. From PASCAL is taken the overall per module organization of CAP/DSDL and also much of the notation. Every module in CAP/DSDL is organized in the following way:

- Module head with interface description (mandatory)
  - Definition of constants (optional)
  - Definition of object types (optional)
  - Declaration of objects (optional)
  - Definition of "assertions" (optional)
  - Declaration of interrupt service routines (optional)
  - Description of the controlled part (optional)
  - Description of the control part (mandatory)

- Module head with interface description:
Modules (unfortunately denoted by the keyword procedure in CAP/DSDL) serve as the basic encapsulation construct. They constitute a scope for identifiers and hide all objects that are not mentioned in an export list away from the outside world. A module is not a block in the ALGOL sense, i.e. all local objects are static ones that exist and maintain values independently from the activation status of the module. Parameter passing is always by reference but the direction (in, out or inout) has to be specified.

- Definition of constants, definition of object types, declaration of objects
A hardware description language has to provide a set of constants, of object types, and of operators that is different from that of general purpose languages like PASCAL. Data objects in CAP/DSDL are based on bitstrings where each "bit" may carry one of 127 values at a time (powerset of L0, L1, M0, M1, H0, H1, ZZ without the empty set, where the prefix letter denotes the signal strength: L for "low impedance", M for "medium impedance", H for "high impedance", see /LR1/).
Of course there exists a variety of language constructs to define constants and to structure data objects (arrays, records). Types of modules may be defined as well as types of data objects and special objects for Petri net places and for interrupt signals. This special feature will be discussed later. As operators we have the complete set of logical operators to be used either in a dyadic manner (excepting not) or in a monadic one (as reduction operator excepting not). Arithmetic and relational operators are offered, both for unsigned and two's complement arithmetic. Stringhandling is carried out using concatenation and substring operators. In addition a rich set of predefined builtin functions is available, e.g. for shift/rotate, timing, or to describe special MOS features like bidirectional transfer gates in a precise manner. Of course there is strong type checking.

- Definition of "assertions"
In CAP/DSDL it is possible (and recommended) to formulate invariant "assertions" about a system to be modelled. Whenever such an invariant is violated, an action as formulated by the user is caused.
Example: (uptime is a standard builtin function of CAP/DSDL while WRITE, DATA_RDY and SETUP are user variables in this example, their correlated behaviour to be monitored)

```
not (WRITE & uptime(WRITE) - uptime(DATA_RDY) <= SETUP) =>
error (' setup condition violated ')
```

- Declaration of interrupt service routines
Event driven control is a natural approach in distributed systems. Interrupt systems are described in CAP/DSDL via interrupt signals (objects of type interrupt with domain "set","not set"), interrupt service routines and some specific operations (sint(x), wait(x), enable(x), disable(x)). A module is sensitive to an interrupt signal x if it contains an interrupt service routine for signal x. So by the declaration of interrupt service routines not only the reaction to an interrupt is defined but also the static interruptability (which may restricted further dynamically using disable/enable operations)

- Description of the controlled part/description of the control part
A hardware system typically can be separated into a controlled part (data path) and a controlling one. This separation principle is reflected in CAP/DSDL by offering two distinct parts to be used in a description.

In the part for the data path two phenomena are described: The value behaviour of non storing data objects (variables with type prefix implicit in CAP/DSDL) is defined and guarded register transfers (or more complex operations, but guarded in any case) are formulated. A CAP/DSDL description of a system may consist mainly of this part, as shown in the example below.

Example: (3 bit ripple counter at RT-level)

```
  :
var CNT, T1, T2, T3: bit; CLK: implicit bit;
  :
impdef
  CLK := not CLK delay (50)
  at up (CNT & CLK) do T1 := T1 := not T1;
  at up (CLK & not T1 do T2 := not T2;
  at up (CLK & not T2 do T3 := not T3;
```

This part of a description can also be viewed at as a control structure based upon events on data objects instead of being based upon identified control events. By the CAP/DSDL software system such guarded operations are treated as isolated subnets with a unique entry place that is marked by the respective event on data variables.
The control part of a system is described by formulating a concurrent algorithm. This may be done either by using the usual(straightforward extended) algorithmic constructs (seqbegin...end, conbegin...end, parbegin...end, while...do, repeat... until, case...of, if...then...else, for...seqto...do, for...conto...do, for... parto...do) or by formulating directly a controlling CAP-net. The first alternative is to be preferred as, by exclusive use of this alternative, a deadlock-free, reusable and safe system can be ensured /RA2/. For obvious reasons this alternative is called "structured concurrent programming" in CAD/DSDL. It has been shown /RA2/ that every deadlock-free, reusable, and safe CAP-net can be represented by a semantically equivalent structured concurrent CAP/DSDL program.

The semantic foundation of CAP/DSDL is given by CAP-nets. The structured constructs are viewed as templates of special nets that can be nested recursively. Consequently the CAP/DSDL compiler maps every control structure onto a CAP-net and our synthesis algorithm implements such nets directly in silicon.

FROM DESCRIPTION TO HARDWARE : THE CAP/DSDL SYNTHESIZER

Various methods for a direct implementation of interpreted Petri nets have been proposed in the literature. Some of them are centralized, using a diode matrix or a PLA structure /PA1/, /KI1/. Such approaches have the deficiency that a central control unit is obtained that has to distribute control signals all over the system. This causes a layout with considerable high crossover complexity, especially if λ is scaled down. Most approaches make use of flip-flops in order to model a place, representing the actual marking by the flip-flop's state /DTM/, /KI1/, /PA1/.
In our opinion this approach has two main deficiencies:
- As input places have to be emptied when a transition fires, we need two connections for every place-to-transition edge of the net.
- As redundant information is stored, redundant flip-flops are neccessary. This results in a waste of chip area.

Therefore we prefer a "non return to zero" (NRZ) encoding of the markings /RA3/. This approach was first investigated by Patil and Dennis /PD1/. In addition our implementation is a distributed one.

In our approach a place is represented by a simple line and transitions by certain circuits.

Let l(p) denote the line representing a place p and let be t a AND-transition, $p \in {}^{\cdot}t$.

- t becomes firable if an arbitrary edge occures on l(p) and for all other places $p' \in {}^{\cdot}t$ the last edge on l(p') has been the same as this one on l(p).
- In order to model firing , transition t toggles the value on all $l(p^+)$ with $t^+ \in t^{\cdot}$ and holds this value. Furthermore internally it toggles its encoding of all its input places.

Note that we now have a one phase (non return to zero) protocol between places and transitions. The storing capability has been moved from the places to the transitions. In transitions we have to store two kinds of information:

- the actual coding of the input places
- the actual value of the output places.

All this information can be stored in a single flip-flop independently from the number of input and output places. This is true because

- at every point of time the value of all lines representing output places of a transition is identical (but may be interpreted in different ways by different transitions that have such places as input places),
- we can assume that at every point of time all input places of a transition are encoded identically with respect to the considered transition,
- we can use the same flip-flop in order to hold the value of the output lines and to remember the actual coding of the input lines as this coding is toggled if and only if the transition fires. By firing and only by firing a transition toggles the value of its output lines.

We will not discuss the building blocks of our implementation method in this paper. They are described in detail in /BR1/ (both, nMOS and CMOS realization). Just to give an idea of the simplicity of the approach we will sum up the principles:

a) An AND-transition with n inputs and m outputs is modelled by a n-input C-gate. Using the nMOS realization of /MC1/, 2n+4 transistors are needed.

b) An OR-transition with n inputs and m outputs is modelled by a n-input coincidence gate. Following an idea of /ED1/, 3n-3 transistors are needed for a nMOS realization.

c) A DECIDER-transition with one enable input, a fully decoded m-bit control input and m outputs is realized with the aid of one pulse generator and m gated toggle flip-flops. This implementation needs 12m+5 transistors for a nMOS realization.

d) A two-way ARBITER (the heart of the BLKHD-transition) consists of 3 toggle flip-flops, a pulse generator and some additional circuitry. A nMOS realization costs 58 transistors. A n-way ARBITER costs 1/2(n+35n+38) transistors and finally a

BLKHD-transition for n requests needs $1/2(n^2+41n+32)$ transistors.

e) A BLKEND-transition for n requests is implemented using one pulse generator, n toggle flip-flops and some additional circuitry, needing 18n+2 transistors in total.

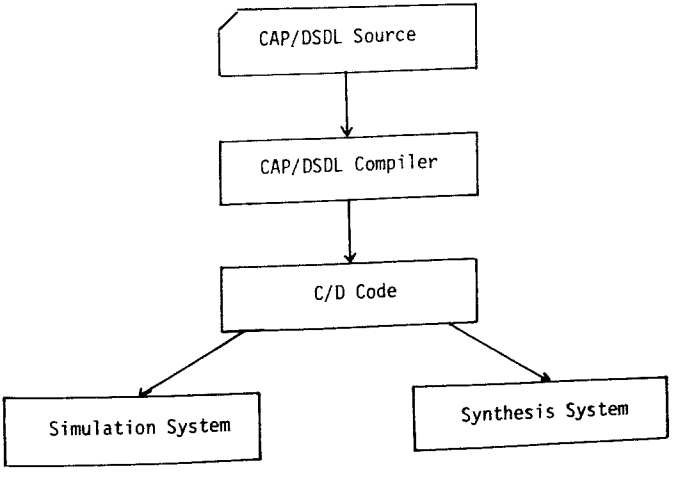
f) A AT-transition is just a special case of the BLKEND-transition and is implemented similarily with 20 transistors.

g) Data operations have to be sensitive for any edge on an "invoke"- input and have to acknowledge by toggling the value of an "acknowledge"-output. Such black boxes may be inserted into the net whenever it is requested by the concurrent algorithm to be implemented. (In practice just the opposite procedure is followed up: First the data manipulation modules are arranged following a data path approach and afterwards the circuitry for the implementation of the control structure is inserted, connecting "invoke"- and "acknowledge"-signals via implemented transitions due to the net structure.)

In the CAP/DSDL system all system services are based on a single format data base, called C/D-code. This is an internal representation of CAP/DSDL programs, organized as a couple of multiple linked lists. In the C/D-code the complete control structure of a CAP/DSDL description is already transformed into an interpreted CAP-net. Using this a remarkable part of the synthesis job to be done is carried out by the usual CAP/DSDL compiler, used for every application of CAP/DSDL.

Fig. 1 shows the overall organization of the control structure implementation algorithm.

Fig. 1



GRANULARITY OF CONCURRENCY : THE MODULARIZATION CONCEPT OF CAP/DSDL

Like MODULA, CAP/DSDL has included a module concept, but this is closer to the class-concept of SIMULA.
In CAP/DSDL the terms "module" and "procedure" are identified, i.e. every procedure describes a piece of hardware that has its own resources, its own control, and may be requested by several activities concurrently. The requests are served in a time shared manner, i.e. included in the procedure/module concept is a monitor concept with arbitration (see BLKHEAD/BLKEND-transitions).
Procedures may be defined as types. So an arbitrary number of instantiations of such a type may be created just by declaring "variables" of such a type.

Example

```
type memorycell =
     procedure memorycell (in operation : bit;
                           inout data : bit(8));
        const read = "Ø"; write ="1";
        var datacell : bit(8);
        conbegin
           case operation of
              read : data := datacell;
              write : datacell := data
           end
        end;

     memorybank =
     procedure memorybank ( in operation : bit;
                            in adr       : bit(1Ø);
                            inout data   : bit(8));
        var memory : array [Ø : 1Ø23] of memorycell;
        conbegin
           memory [adr] (operation,data)
        end;
  :
var memory : array [Ø : 1Ø23] of memorybank
  :
```

It should be mentioned that in CAP/DSDL procedure types may be generic objects. Another important feature of CAP/DSDL procedure types is that they may be declared as implementations of abstract data types (ADT) /LS1/, i.e. objects that export operations on data instead of data (-values). Such procedures therefore are called "export procedures" in CAP/DSDL and are comparable with classes in SIMULA.

Example:
(Implemented ADT "stack", implemented on array, generic with attributes type of "stackelement" and "size" of stack)

```
type stack =
     export (flush, pop, push) procedure stack
                               [stackelement : type; size : const] ;
        var stackcarrier : array [Ø : size] of stackelement;
        procedure flush;
           {procedure body}
        procedure pop (out data : stackelement; out empty :bit);
           {procedure body}
        procedure push (in data : stackelement; out full : bit);
           {procedure body}
        end;
  :
var largebytestack : stack [bit(8), 1Ø ØØØ] ;
    smallarraystack : stack [array [Ø : 16] of bit(32), 64] ;
```

The CAP/DSDL concept of modularization is comparable with the LMA concept /SC1/ and the concept of packet architectures /DE1/:
A system is constituted by a set of modules of different types that are invoked mutually in order to perform a certain action. After the requested action has been performed (and eventually the required data has been stored in well defined interface locations) an invoked module acknowledges to its invoker and deactivates itself (makes itself ready for a subsequent activation).
This organization concept may be nested to any depth and ADT procedures may be intermixed arbitrarily with ordinary ones. Regular structures are described using arrays and records of procedures (modules). Concurrency is not restricted to the cooperation of modules but is possible within modules as well. Our synthesis algorithm performs its realization task on every control structure given by a CAP-net.
Parts of a description that have not to be processed by the synthesizer can be encapsulated by using data driven control (usually used for synchronous submodules) or by a compound statement of type "begin...end".

REFERENCES

/BR1/ R. Brück
"Zur Synthese asynchroner Steuerwerke aus CAP/DSDL Beschreibungen"
(Synthesis of asynchronous controllers from CAP/DSDL descriptions)
in german
Diplomarbeit, Universität Dortmund, Abt. Informatik (1983)

/DE1/ J. B. Dennis
"Packet communication architecture"
Proceedings of 1975 Sagamore Conference on Parallel Processing (1975)

/DTM/ R. David, R. Tellez-Giron, E. Mitrani
"Emploi des cellules universelles pour la synthèse de systemes asynchrones
décrit par reseaux de Petri"
Digital Processes, 6 (1980)

/ED1/ C. Edwards
"Some novel exclusive OR/NOR circuits"
Electronic Letters (1975)

/FR1/ D. Frantz, F. Rammig
"The impact of an advanced CHDL on VLSI design"
Proceedings of ICCD'83, Port Chester, N.Y. (1983)

/HY1/ O. Herzog, M. Yoeli
"Control Nets for asynchronous systems, Part I"
Technion Haifa, Dept. of Computer Science, Techn. Report 74 (1976)

/KI1/ D. Kinniment
"Regular programmable control structures"
Proceedings of VLSI'81, Edinbourgh (1981)

/LR1/ K.-D. Lewke, F.J. Rammig
"Description and simulation of MOS devices in register transfer languages"
Proceedings of VLSI'83, Trondheim (1983)

/LS1/ B. Liskov, A. Snyder, R. Atkinson, C. Schaffert
"Abstraction mechanisms in CLU"
CACM, Vol.20, No.8 (1977)

/MC1/ C. Mead, L. Conway
"Introduction to VLSI systems"
Addison Wesley (1980)

/NO1/ J. Noe, G. Nutt
"Macro E-nets for representation of parallel systems"
IEEE ToC, Vol. C-22, No.8 (1978)

/PA1/ S. Patil
"Micro control for parallel asynchronous computers"
Proceedings of Euromicro '75 (1975)

/PD1/ S. Patil, J. Dennis
"The description and realization of digital systems"
Proceedings of 6th IEEE Annual Comp. Society Intern. Conf. (1972)

/PE1/ J. L. Peterson
"Petri Nets"
ACM Computing Surveys, Vol. 9 (1977)

/RA1/ F. J. Rammig
"Preliminary CAP/DSDL Language Reference Manual"
Forschungsbericht Nr. 129 d. Abt. Informatik d. Univ. Dortmund (1981)

/RA2/ F. J. Rammig
"Structured parallel programming with a highly concurrent programming language"
Atti di Congresso Annuale AICA '80, Bologna (1980)

/RA3/ F. J. Rammig
"An alternative approach to self-timed VLSI systems"
Forschungsbericht Nr. 131 d. Abt. Informatik d. Univ. Dortmund (1982)

/RO1/ C. Rose
"LOGOS and the software engineer"
Proceedings of FJCC '72 (1972)

/SC1/ M. Stefik, L. Conway
"Towards the principled engineering of knowledge"
The AI Magazine (1982)