Franz J. Rammig
University of Dortmund
F.R.G.

# Petri-Net based description, analysis and simulation of concurrent processes

## 1. Abstract:

### 1.1 The language CAP

While there are a couple of programming languages which allow the representation of parallelism /1,2,5/ there is a lack for languages which allow in an as well disciplined as general manner the description of concurrency.
A well defined mathematical description method for concurrent processes is given by the Petri net model /3,7,9/. It fulfils the above requirements and in addition is widely used and extremely easy to understand.
This concept is integrated into the language CAP in a very elegant way:
Lables are used as places, where one has in mind that a labelled statement puts a token into its label(s) after it has been processed. On the other hand the processability of a statement is controlld by "On-conditions" on labels. Every statement is processed if and only if its "On-condition" has become true. At this time, in accordance with the firing-rule (="On-condition"), "tokens" are withdrawn from certain labels used within the "On-condition" and placed into the labels of the statement.
In addition various parameters may be associated to transitions thus offering a description power similar to the Macro-E-Nets /8/. On the other hand there are a lot of language-constructs to support structured programming.

### 1.2 The analysis of CAP-programs at compile-time

Since a couple of years Petri nets are object of research. For some rather restricted subclasses of Petri nets sufficiant an necessary conditions for the topology of a net to be well-formed are known /3,4,7/. Under a well-formed net we understand a net without dead-locks being safe (no loss of information) and residue-free (independent from a certain history). Related to results by Herzog an Yoeli /4/ for a subclass of Petri nets we prove that there are six conditions for the topology of a net of the subclass of Petrinets which are used to describe the control-structure of CAP-programs so that the net is well-formed.

By this we can decide at compile-time without simulation whether a CAP-program describes a life, safe and residue-free system (i. e. a useful one) or not.
After having defined, what we mean ba a "Structured CAP-program" we will have the fine result that every structured CAP-program is well-formed.

### 1.3 The simulation of systems described in CAP

As CAP allows the description of systems down to a level of specification which is comparable with the bit-level of digital systems inclusive the detailled description of the real-time-behaviour, for the processing of CAP one needs a system similar to a simulator for digital circuits. On the other hand also very global descriptions on a high level of abstraction are possible.
This implies that the run-time system for CAP must be extremely adaptive. We tried to solve this problem by a very flexible and powerful tabel driven and event oriented simulator with the event-mechanism being directly adopted from the Petri net concept of CAP.

## 2. Interpreted Petri Nets

PN:= (S,T,F) is called Petri Net : <=>
S finite, non emty set (set of places),
T finite, non emty set (set of transtions),
$S \cap T = \emptyset$,
$F \subseteq S \times T \cup T \times S$,

$$\bigwedge_{X \in S \cup T} \bigvee_{Y \in S \cup T} ((X,Y) \in F \vee (Y,X) \in F)$$

Places may contain tokens up to a certain capacity, defined by a mapping
cap: $S \rightarrow \mathbb{N} \cup \{\infty\}$ called capacity-distribution.

The distribution of tokens at a certain point of time i is described by a mapping $m_i : S \rightarrow \mathbb{N} \cup \{ o \}$. Such a mapping is called marking. There must be:

$$\bigwedge_{i \in \mathbb{N}} \bigwedge_{s \in S} m_i (s) \leq cap (s)$$

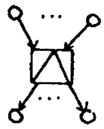
Transitions may fire in accordance to a certain firing rule. The firing of a transition produces a marking $m_i'$ out of a marking $m_i$ (single-step behaviour).
For every transition $t \in T$ there may be an associated function $f_t:D_t \rightarrow I_t$ mapping a certain domain $D_t$ into a certain image-range $I_t$. There may be:

$$\bigvee_{t \in T} \bigvee_{t' \in T} D_t \cap D_{t'} \neq \emptyset \vee D_t \cap I_{t'} \neq \emptyset \vee D_{t'} \cap I_t \neq \emptyset \vee I_t \cap I_{t'} \neq \emptyset$$

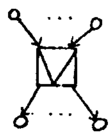
(potential data conflicts). Note that this implies the existence of another graph, called data-graph.

We define that a function $f_t$ will be executed iff its associated transition t has become firable. After the termination of the function's execution the transition will fire.
Similar to LOGOS-Control-Graphs /10/ in our nets the set of transitions is partitioned into seven classes.
The classes are characterized by their firing-rules and in some cases by restrictions of use.
Both will be defined only in an informal way within this paper.

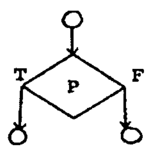
If t ε A then t may have an arbitrary number of input-places and output-places. An And-transition is firable if all input-places are marked and if every output-place is marked below its capacity. If an And-transition fires, it withdraws a token from every input-place and places a token into every output place. We will use the following graphical representation for And-transitions:



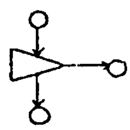
If t ε O then again t may have an arbitrary number of input-places and output-places. An Or-transition is firable if at least one marked input-place is marked and if every output-place is marked below itscapacity. If an Or-transition fires, it withdraws a token from its left-most marked input-place and places a token into every output-place We will denote Or-transitions by the following graphical representation:



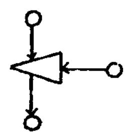
If t ε D then t has exactly one input-place and two output-places. A decider is firable if the input-place is marked and both output-places are marked below their capacity. If a decider fires it withdraws a token from its input-place and in accordance with the value of an associated predicate it places a token into one of its output-places. Graphical representation:



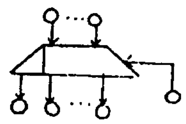

The remaining four types of transitions are uses to describe block structures. The calling of a block is performed by a special one-input-two-output And-transition (C-transition) with the graphical representation:



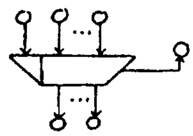
Note that the calling block remains active. As a consequence of this we need another special And-transition, (R-transition), one with two input-places and one output-place to synchronize the termination-signal of the called procedure with the control-flow of the calling block at a particular location. Graphical representation:



Blocks are bracketed by a pair consisting out of a Blockhead-transition (H-transition) and a Blockend-transition (E-transition). A H-transition may have an arbitrary number of input-places, one for every reference-source to this particular block and one additional input-place, called the "feedback-place". There is one output-place for every input-place besides the feedback-place to preserve the information about the calling location and one additional place (the leftmost one) to initialize the activities of the block. A H-transition is firable if the feedback-place and at least one other input-place are marked below its capacity. If it fires it withdraws a token from the feedback-place and from the leftmost remaining input-places and places a token into the special output-place thus initializing the activities of the block and into the particular output-place associated to the selected input-place to preserve the source of preference. Graphical representation:



The E-transition is defined in dual manner. Its graphical representation is:



## 3. The philosophy of CAP

Before going into more details of CAP-graphs we briefly will point out how the Petri-Nets are integrated into our language CAP. This is done in a very simple and, as I state, natural way:
As in goto-free programming there is no other meaningful use for labels they are used to denote places. One has in mind that a labelled statement puts a token in its label(s) after it has been processed. By a statement in this context we mean either an assignment-statement or a DO- group of statements. The processability of a statement is controlled by "On-conditions" on labels. A statement is processed if and only if its "On-condition" has become true. At this time, in accordance with the firing-rule (type of "On-condition") as explained above, "tokens" are withdrawn from certain labels used within the "On-condition" and placed into the labels of the statement.

For the different transitions of CAP-graphs we have the following constructs in CAP:

And-transition:

ON (&($i_1$,...,$i_n$)):$O_1$:....:$O_m$: <Statement>; or simply:

ON ($i_1$,...,$i_n$):$O_1$:....:$O_m$: <Statement>;

Or-transition:

ON (|($i_1$,...,$i_n$)):$O_1$:....:$O_m$: <Statement>;

Decider-transition:

ON ($i_1$):IF (P) THEN $o_1$ : <Statement>;
ELSE $o_2$ : <Statement>;

C-transition:

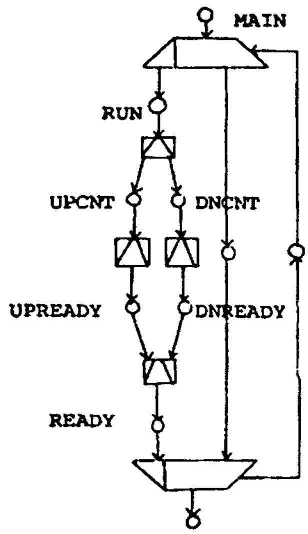
ON ($i_1$) : $O_1$: CALL $o_2$;

R-transition:
ON ($\neg i_1, i_2$): $O_1$: <Statement>;
Blockhead-transition:
ON CALL ($i_1, \ldots, i_n$) : $o_1$: PROCEDURE .....;
Blockend-transition:
ON ($i_1$): $o_1$: END;

Note that neither the feedback-place of blocks nor the preserving of the reference-source are expressed explicitely.
The notation of <Statement> is very similar to the notation of PL/1. In fact we did not want to design a totally new language but one that should be as similar to a well known language as possible. The usage of PL/1 for this purpose had pragmatic reasons.
The following extremely small example should illustrate the idea of the language CAP:

```
ON CALL (MAIN): RUN: PROCEDURE;
   DCL (A,B,C) FIXED,
       (RUN, UPCNT, DNCNT, UPREADY, DNREADY, READY) LABEL;
   ON (RUN): UPCNT : DNCNT :   A := Ø;
   ON (UPCNT) : UPREADY :      B := A + 1;
   ON (DNCNT) : DNREADY :      C := A - 1;
   ON (DNREADY, UPREADY):READY:A := B + C;
   ON (READY) : END MAIN;
```

The control-structure of this CAP-program with its two concurrent activities, synchronized at the end is given by the following CAP-graph:




## 4. Structured CAP-programs

To allow to write concurrent programs which are even easier to understand as the above example we introduce additional features into the language CAP.
This is done by DO-groups where we distinct between

- simple DO-groups,
- loop-DO-groups,
- case-DO-groups,
- replication-DO-Groups.

Each of these DO-groups may describe a sequential, concurrent or parallel grouping.
We mean that two statements are executed in parallel if the initation of the operations takes place at exactly the same point of time.
We will explain here only the meaning of some examples of DO-groups:

DO;$S_1$;...;$S_n$; END;

By this we mean simply parallel execution of the statements $S_1$ until $S_n$.

ON ($L_1$): $L_2$ : DO CONCURRENT; $S_1$;....; $S_n$; END;

By this we mean the following:

ON ($L_1$) : $HL_1$:...$HL_n$:;
ON ($HL_1$): $HHL_1$ : $S_1$;
⋮
ON ($HL_n$): $HHL_n$ : $S_n$;
ON ($HHL_1$,..$HHL_n$) : $L_2$:;

Finally by

ON ($L_1$) : $L_2$ : DO SEQUENTIAL; $S_1$;....;$S_n$; END;

we mean:

ON ($L_1$) : $HL_1$ : $S_1$;
ON ($HL_1$) : $HL_2$ : $S_2$;
⋮
ON ($HL_n$) : $L_2$ : $S_n$;

<u>Loop DO-groups:</u>

ON ($L_1$) : $L_2$ : DO [ {SEQUENTIAL / CONCURRENT} ] WHILE (P); $S_1$: ... : $S_n$; END;

By this we mean:

ON (|($HL_1$, $L_1$)) : $HL_2$ :;
ON ($HL_2$) : IF (P) THEN $HL_1$: DO [ {SEQUENTIAL / CONCURRENT} ];
$S_1$;.....;$S_n$;
END;
ELSE $L_2$:;

<u>Case-DO-groups:</u>

ON ($L_1$): $L_2$: DO CASE (P); $S_1$;....; $S_n$; END;

By this we mean:

ON ($L_1$): IF (P=0) THEN $HL_0$ : $S_1$;
ELSE $HL_{00}$ :;
ON ($HL_{00}$) : IF (P=1) THEN $HL_1$ : $S_2$;
ELSE $HL_{11}$:;
⋮
ON ($HL_{n-2n-2}$) : IF (P=n-1) THEN $HL_{n-1}$:$S_n$;
ELSE $HL_n$;
ON (|($HL_0$;....;$HL_n$)) : $L_2$:;

As a special case of the above if n has the value 2 we also may write:

ON ($L_1$) : $L_2$ : IF (P) THEN $S_1$;
ELSE $S_2$;

We now define that

```
ON CALL (L1) : PROCEDURE ....; S1; END;
```

is equivalent to

```
ON CALL (L1) : HL1 : PROCEDURE.....;
   ON (HL1) : HL2 : S1;
   ON (HL2) : END;
```

Finally we define that if the R-transition for a procdure-call is omitted then

```
ON (L1) : L2: CALL L3; will be substituted by
ON (L1) : HL2 : CALL L3;
ON (←L3, HL2) : L2 :;
```

Now, clearly, we may write CAP-programs without any use of labels besides procedure-names. Such programs are called "structured CAP-programs". Not that this exactly meets Dijkstra's philosophy. The above little CAP-example may be replaced by an equivalent structured one:

```
ON CALL (MAIN) : PROCEDURE;
   DCL (A,B,C,) FIXED;
   DO SEQUENTIAL;
   A : = Ø ;
   DO CONCURRENT;
      B : = A + 1;
      c : = A - 1;
   END;
   A : = B + C;
  END;
END MAIN;
```

## 5. Analysis of CAP-programs

To be able to state our main results about the analysis of CAP-programs we have to define some additional features of CAP-graphs. We will do this in a rather informal way:

Let ts be a place or transition.

$ts^{\bullet} := \{ ts' \mid (ts,ts') \in F \}$, $^{\bullet}ts := \{ ts' \mid (ts', ts) \in F \}$

Let m be a marking. Then by [m> we denote the marking-class of m, i. e. the set of markings reachable from m. Let P be a path from ts to ts'. By S(P) we mean the set of places on this path, by T (P) the set of transitions on this path.

By W (ts, ts') we mean the set of pathes from ts to ts'.

There must exist a one-to-one mapping $b : H \to E$ mapping those transition onto one another which shares feedback-places and reference-source preserving places.

Compatible with the ALGOL-scape-of-variables the sets of places and transitions are partitioned into classes belonging to blocks.

By T (h) we mean the set of transition belonging to the block with H-transition h.

S (h) is defined similary.

There is exactly one outermost block. Let (h,e) be the bracket of the outermost block: $^{\bullet}s = \emptyset \iff s \in {}^{\bullet}h$ and s is no feedback-place, $s^{\bullet} = \emptyset \iff e^{\bullet}$ and s is no feedback-place.

$$\bigwedge_{s \in S} \; |{}^{\bullet}s| \le 1 \wedge |s^{\bullet}| \le 1$$

This does not imply that we are working with "marked graphs" as we have more complicated transitions. In fact the subclass of Petri-Nets treated here is even more general than "Simple Petri-Nets". Let BB be the set of block-brackets (h,e).

$$\bigwedge_{(h,e)\in BB} \; \bigwedge_{t\sigma' \in T(h) \circ S(h)} W(h,t) \neq \emptyset \wedge W(ts,e) \neq \emptyset$$

Finally we don't allow reentrance and recursion.

A marking $m_I$ is called <u>initial marking</u> : <=>

$$\bigwedge_{s \in S} \quad \begin{array}{l} m_I(s) = 1 \Leftrightarrow {}^{\cdot}s = \emptyset \text{ or } s \text{ is "feedback-place"} \\ m_I(s) = 0 \quad \text{otherwise} \end{array}$$

A marking $m_F$ is called <u>final marking</u> <=>

$$\{ s \in S \mid s^{\cdot} = \emptyset \wedge m_F(s) > o \} \neq \emptyset$$

A CAP-graph is called <u>safe</u> : <=>

$$\bigwedge_{m_I} \; \bigwedge_{m \in [m_I>} \; \bigwedge_{s \in S} \quad m(s) \leq 1$$

A CAP-graph is called <u>life</u> : <=>

$$\bigwedge_{m_I} \; \bigwedge_{m \in [m_I>} \; \bigvee_{m' \in m_F} \quad m' \in [m>$$

A CAP-graph is called <u>residue-free</u> : <=>

$$\bigwedge_{m_I} \; \bigwedge_{m \in [m_I> \cap \{m_F\}} \quad m(s) = 1 \Leftrightarrow s^{\cdot} = \emptyset \vee s \text{ s "feedback-place".}$$

A CAP-graph is called <u>well-formed</u> : <=>
It is safe, life and residue-free.

A CAP-graph is called <u>local safe</u> : <=>

$$\bigvee_{n \in N} \; \bigwedge_{s \in S} \quad cap(s) < n$$

A CAP-graph is called <u>potential unsafe</u> : <=>

$$\bigwedge_{s \in S} \quad cap(s) = \infty \text{ or } cap(s) = 1,\ cap(s) = 1 \Leftrightarrow s \text{ is "feedback-place".}$$

A CAP-graph is called <u>unblocked</u> : <=>
There is only one block within the net.

Theorem: Let N be a potential unsafe, unblocked CAP-graph.
N well-formed <=> B1 and B2 and B3 and B4 and B5 and B6

with

$$B1 : \Leftrightarrow \left(j \in A \Rightarrow \bigwedge_{s_i \in {}^{\bullet}j} \; \bigvee_{P_i \in W(I,s_i)} j \notin T(P_i)\right)$$

$$B2 : \Leftrightarrow \Big(f \in A \cup O \wedge u \in O \wedge P_1 \in W(f,u) \wedge P_2 \in W(f,u) \wedge S(P_1) \cap S(P_2) = \emptyset \Rightarrow$$
$$\bigvee_{j \in A \cap T(P_1)} \; \bigvee_{t \in T(P_2)} \; \bigvee_{P_3 \in W(t,j)} S(P_1) \cap S(P_3) = \emptyset$$
$$\vee \bigvee_{j' \in A \cap T(P_2)} \; \bigvee_{t' \in T(P_1)} \; \bigvee_{P_4 \in W(t',j')} S(P_2) \cap S(P_4) = \emptyset \Big)$$

$$B3 : \Leftrightarrow \Big(d \in P \wedge j \in A \wedge s_1, s_2 \in {}^{\bullet}j \wedge \bigvee_{P_1 \in W(d,s_1)} \wedge \bigvee_{P_2 \in W(d,s_2)} \wedge S(P_1) \cap S(P_2) = \emptyset$$
$$\Rightarrow \bigvee_{f_1 \in (A \cup O) \cap T(P_1)} \; \bigvee_{f_2 \in (A \cup O) \cap T(P_2)} \; \bigvee_{P_3 \in W(f_1,s_2)} \; \bigvee_{P_4 \in W(f_2,s_1)} S(P_3) \cap S(P_1) = \emptyset \wedge S(P_4) \cap S(P_2) = \emptyset$$

$$B4 : \Leftrightarrow \Big(f \in A \cup O \wedge d \in P \wedge j \in A \wedge P \in W(I,f) \wedge s \in {}^{\bullet}j \wedge P_1 \in W(f,s)$$
$$\wedge P_2 \in W(f,d) \wedge P_3 \in W(d,j) \wedge S(P_1) \cap S(P_2) = \emptyset \wedge$$
$$S(P_1) \cap S(P_3) = \emptyset \wedge S(P) \cap (S(P_1) \cup S(P_2) \cup S(P_3)) = \emptyset$$
$$\Rightarrow \bigvee_{t \in T(P_2) \cup T(P_3)} W(d,t) \neq \emptyset \wedge \bigvee_{P_4 \in W(d,t)} S(P_4) \cap S(P_3) = \emptyset \Big)$$

$$B5 : \Leftrightarrow \Big( C \text{ circle} \wedge f \in (A \cup O) \cap T(C) \Rightarrow \bigwedge_{\substack{s,s' \in f^{\bullet} \\ s \neq s'}} \; \bigvee_{j \in A \cap T(C)} \; \bigvee_{\substack{P_1 \in W(s,j) \\ P_2 \in W(s',j)}} S(P_1) \cap S(P_2) = \emptyset \Big)$$

$$B6 : \Leftrightarrow \Big(C \text{ circle} \wedge j \in A \cap T(C) \Rightarrow \bigwedge_{\substack{s,s' \in {}^{\bullet}j \\ s \neq s'}} \; \bigvee_{f \in (A \cup O) \cap T(C)} \; \bigvee_{\substack{P_1 \in W(s,j) \\ P_2 \in W(s',j)}} S(P_1) \cap S(P_2) = \emptyset \Big)$$

(see fig. 1 - 6)

Corollary Let N be a local safe, unblocked CAP-graph:
N well-formed <= B1 and B2 and B3 and B4 and B5 and B6

Corollary (informal) Let N be a CAP-graph:
N well-formed <=> Every CAP-graph constructed out of N by the following procedure is well-formed.
If N'' is called by a block N' and N'' communicates with its environment only via its block-brackets then isolate N'' and "bypass" any reference to N'' by replacing the C- and R-transitions by And-transitions and the block by a place shared by these two transitions.
If N'' communicates in another way with its environment then there may be only one reference to this block. Then eliminate its block-nature by proper replacing of C-, R-, H- and E-transitions by And-transitions. (see fig. 7 - 8)
Corollary Every structured CAP-program is well-formed.
The last result is an extremely important one. It states that besides all the well-known advantages of structured programming the most important potential misbehaviours of concurrent programs besides data-conflicts are avoided by restricting ourselves on structured CAP-programs. In addition it is extremely easy to check whether a CAP-program is structured or not as there must only be checked the absence of labels.

## 6. Execution of CAP-programs (Simulation)

For simulation purpose we expand our model to timed interpreted Petri Nets. The implementation of timing will be explained with the aid of the associated language-constructs of CAP.
In CAP nearly every colon may be replaced nonrecursively be <Terminatior>;.
Besides other specifications, < Terminator >may contain a <Delay-spezification>.
This has the general form (shorthand notations are also allowed):

$$\begin{array}{l} \text{DELAY } (a_1\text{: UP } f_{u1}\ (\ldots),\ \text{DOWN } f_{d1}\ (\ldots)/ \\ \qquad \vdots \\ \qquad a_n\text{: UP } f_{un}\ (\ldots).\ \text{DOWN } f_{dn}\ (\ldots)/ \\ \quad \rightarrow r_1\text{: UP } f_{ur1}(\ldots),\ \text{DOWN } f_{dr1}(\ldots)/ \\ \qquad \vdots \\ \quad \rightarrow r_m\text{: UP } f_{urm}\!:\ldots),\ \text{DOWN } f_{drm}(\ldots)) \end{array}$$

The functions $f_{ij}$ may be arbitrary functions of arbitrary arguments. $a_1,\ldots,a_n$ must be argument-variables within the statement to which the delay-specification belongs, $v_1,\ldots,v_m$ result-variables respectively. Note that there may be a different delay-specification for every variable used within a statement and that the delays may differ for increasing or decreasing changes of values. The latter is very important for the description of digital switching circuits.
As the control-structure of CAP-programs is given by a Petri-net and as there are well-defined events within Petri-nets obviously we will execute CAP-programs by event-oriented simulation.
We have to distinct between control-events and data-events. A control-event takes place if a transition becomes firable while a data-event takes place if a new value is assigned to a data-variable.
Control-event may produce additional-events while data-events can't.
Let t be a transition with input-places $i_1,\ldots,i_k$ and output-places $o_1,\ldots,o_e$ with an associated delay-specification as mentioned above.

A control-event for this transition takes place if at point of time $t_0$ t becomes firable. Now new values are calculated for $v_1,\ldots,v_m$ out of the arguments $a_1,\ldots,a_n$. But not the value of $a_i$ at $t_0$ will be taken but the value of $a_i$ at $t_{0-f_{ui}(\ldots)}$ or $t_{0-f_{di}(\ldots)}$ depending whether the last assignment to $a_i$ was one increasing its value or decreasing it respectively. For every $r_i$ a data-event will be produced at a point of time $t_{0+x}$ where x is calculated out of the delay-specification in a similar way.

Finally the point of time when the next control-event, namely the firing of t will take place , is calculated as the maximum of these calculated output-delays. A consequence of the firing of t may be further control-events (other transitions may become firable).

## 7. Conclusion

Within this paper a language has been presented, which is very well-suited for the description of concurrent processes. The language is rather similar to PL/1 and is therefore very easy to understand for people who understand PL/1-programs. By a natural integration of Petri-nets concurrency can be described in a very concise, distinct and precise manner.
The clearness of CAP-programs may even be increased by writing structured CAP-programs. In contrary to most other models of concurrent processes we have necessary and sufficient conditions for the topology of the control-graph to check well-formedness. Therefore this feature may be checked at compile-time without any simulation!
As a special advantage structured CAP-programs are also well-formed.
Having a precise model for concurrency we can execute CAP-programs with the simulator. This simulator is powerful enough to allow the modelling of a very precise timing.

## 8. References


/1/ Brich Hansen, P.: Concurrent Pascal, A Programming Language for Operating System Design,
California Institute of Technology, Information Science, Techn. Report Nr. 10 (1974)

/2/ Burroughs Corp.: Burroughs B67oo/B77oo ALGOL Language Reference Manual, 5ooo649 (1974)

/3/ Herzog, O.: Zur Analyse der Kontrollstruktur von parallelen Programmen mit Hilfe von Petri-Netzen,
Berichte der Abteilung Informatik der Universität Dortmund Nr. 24 (1976)

/4/ Herzog, O. & Yoeli, M.: Control Nets for Asynchronous Systems, Part 1, Technion Haifa, Department of Computer Science, Techn. Report Nr. 74 (1976)

/5/ IBM Corp.: OS PL/1 Languages Reference Manual, GC33-ooo9-3 (1974)

/6/ IEEE Computer, December 1974: Hardware description languages.

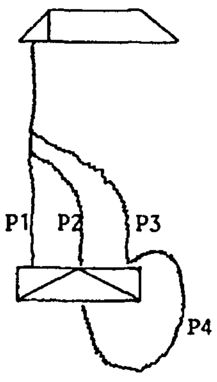
/7/ Lautenbach, K.: Exakte Bedingung der Lebendigkeit für eine Klasse von Petri-Netzen,
Berichte der Gesellschaft für Mathematik und Datenverarbeitung Bonn Nr. 82 (1973)

/8/ Noe, J.D. & Nutt, G.J.: Macro-E-Nets for Representation of Parallel Systems, IEEE Toc, C22 Nr. 8 (1973)

/9/ Petri,C.A.: Concepts of Net Theory, Mathematical Foundations of Computer Science, Proceedings of Symposium and Summer School, High Tatras, Sept. 3-8, 1973, pp 137-146

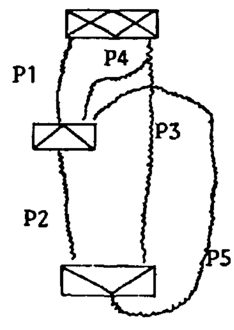
/1o/ Rose, C.W.: LOGOS and the software engineer, FJCC (1972)

Fig. 1 : Cond. 1



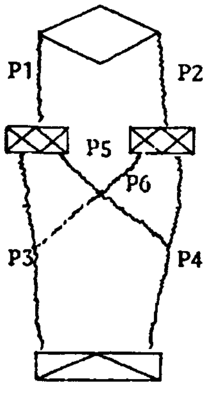

P1,P2,P3 must exist
P3 may not be substituted
by P4!

Fig. 2 : Cond. 2



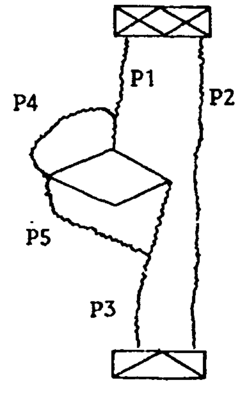

P4 must exist
P4 may not be substituted
by P5!

Fig. 3 : Cond. 3



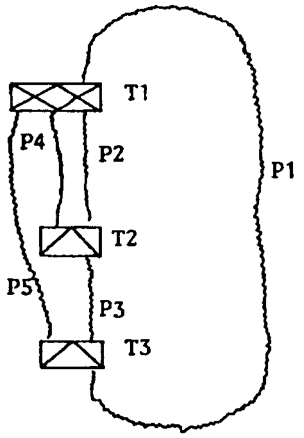

P5 and P6 must exist!

Fig. 4 : Cond. 4



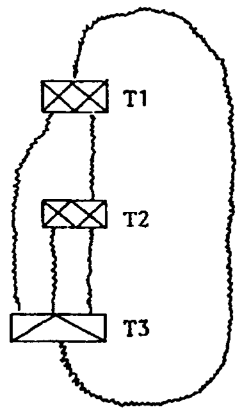

Either P4 or P5 must exist!

Fig. 5 : Cond. 5




T1 implies T2 and T3 .

Fig. 6 : Cond. 6




T3 implies T1 and T2

Fig. 7

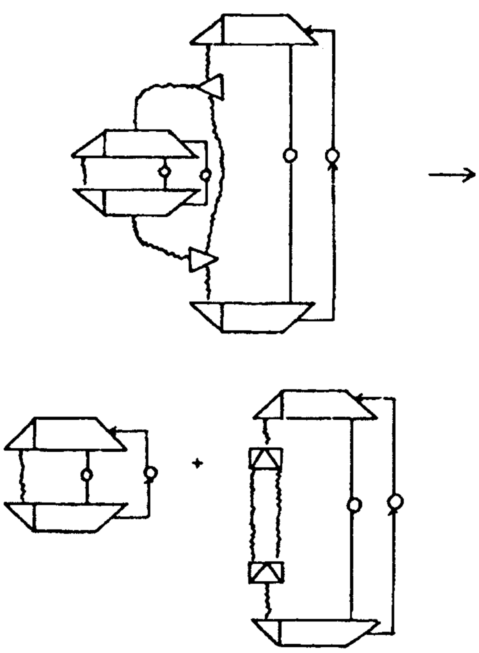


Fig. 8