STRUCTURED PARALLEL PROGRAMMING WITH A HIGHLY CONCURRENT PROGRAMMING LANGUAGE

Franz J. Rammig
University of Dortmund
Federal Republic of Germany

Abstract

The Concurrent Algorithmic Programming language (CAP) has been designed as programming language for highly concurrent processes. This means that in CAP concurrency in as basic as sequentialism is.

The basic concept of CAP is to implement a linguistic correspondence to (timed) interpreted Petri Nets. In addition we intended to design a language which should be as similar to PL/1 as possible, as we didn't want to add an additional storey to the Tower of Babel.

In this paper we will describe the basic concept of the language CAP. From this we will derive the class of Petri Nets (based on LOGOS /RO1/) which is suitable for our purpose. With the aid of these so-called CAP nets we will define what are well behaving control structures (safe, proper terminating, reusable).

On the other hand the problem of structuring concurrent programs is considered. As structured programming is very close related to a hierarchical process (stepwise refinement, top down design) CAP programs which we call structured ones are defined by an inductive construction. It is shown that elementary structured CAP programs are well behaving and that well behaving is preserved by every induction step. By this we get the result that every structure CAP program is well behaving.

The question is whether the class of structured CAP programs covers this class.

This question can be answered by two results:

a) Syntactical incompleteness:
   There are well behaving CAP programs that are not structured ones.
b) Semantical complementeness
   For every well behaving CAP program there is an equivalent structured CAP program, where we use a very strong definition for equivalence.

Finaly an overview of implemented CAP software (compiler, interpreter, debugger, documentation generator, family of codegenerators) will be given.

1. Basic Concens of CAP

While there are known a couple of programming languages for cooperating sequential processes (e.g. Concurrent Pascal, Modula) there is a lack of languages for highly concurrent systems. Such systems are of increasing importance in various fields such as hardware design, customers VLSI design, highly distributed systems, multiprocessor systems with a large number of processes.

CAP is intended to cover both application areas. To describe cooperating sequential processes a monitor concept and a handshaking mechanism has been integrated into the language with a foundation in the basic concept of the language. This basic concept can be characterised as linguistic correspondence to (timed) interpreted Petri Nets. Petri Nets have been chosen for various reasons:

- There is a rich and well founded theory on Petri Nets (see /PNC/).
- Petri Nets offer a model that is intuitivly understood by an arbitrary user.
- As there is a graphical representation of Petri Nets understanding of, and communication about concurrent processes is supported very well.

In addition a neutral problem analysis gives the some result:
To describe highly concurrent processes a procedural approach seems to be inadequate.

CAP is a nonprocedural language, i.e. the ordering of statements is not attached to any meaning. Statements are associated with an explicit condition for execution of the operation described by the statements. A statement ("triggered Operation") of a nonprocedural language has the form
$P(B_t) \longrightarrow A$

where $B_t \subset B$, B set of elementary conditions, $P(B_t)$ a predicate on B_t. The datamanipulation A is executed whenever $P(B_t)$ becomes true. The datamanipulation realizes a total mapping $a : D_t \rightarrow D_t'$ with $D_t, D_t' \subset D$, D set of elementary data (variables) inclusive conditions.

In a statement of this form control and datamanipulation are not separated strictly as the conditions are part of the data. To achieve more transparent designs the trend goes to a strict separation, however.

In this case we get statements of the form:
$P(B_t) \rightarrow B_t'$, A with $B \cap D = \emptyset$. I.e. the manipulation of the conditions due to statement execution is not longer implicitly part of data manipulation, but is result of an explicitely given mapping $c : B \times D \rightarrow B$. Note that we need data-elements as potential arguments, as there may be a data dependent control flow.

Such statements reflect very closely the idea of interpreted Petri Nets if we assume that all statements are interpreted independetly.

A statement stands for an interpreted transition t, B for the set of places, B_t the set of input places of t, B_t' the set of output places of t, $P(B_t)$ the firing condition, $c : B \times D \rightarrow B$ the firing rule and $a : D_t \rightarrow D_t'$ the interpretation.

A little problem arises concering data dependent control flow. Within Petri Nets this is usually expressed using places with forward conflict. To preserve our scheme we introduce a data-dependent transition instead of places with foreward conflict.

Because of symmetry we replace places with backward conflict by another kind of transition. It must be part of the firing rule of such a transition to resolve backward conflicts in some way. This may be done by a priority rule. But now we are forced to introduce firing rules that are dependent on the marking of the output-laces as well.

Therefore every place has a finite or infinite capacity so that free capacity may be interrogated.

As we now have special transitions for conflicts, consequently places are restricted to have only one input transition and one output transition. Nevertheless the resulting Petri Nets are not Marked Graphs as we have distinct firing rules.

We restrict ourselves to processes that may be of arbitrary internal structure but have one definite beginning and one definite ending. Such processes (kinds of nets) can be composed very neatly. However we need a kind of a monitor whenever a subnet shall reflect a time-shared resource.

The above considerations lead to a kind of modified Petri Nets which is very close to LOGOS control graphs /RO1/. In fact CAP-Graphs use nearly the same set of transitions, with more general firing rules, a little more restricted use of blockhead/blockend and more rules concerning the global structure. A more detailed discussion of CAP-nets will follow under 2..

Formally basic CAP statements are similar to "Guarded Commands" /DI1/. Similar transition types as in CAP-nets are known from Macro-E-Nets /NO1/. The main difference is that in CAP we have no attributes at tokens but in contrary attributes at transitions. When restricting on AND-, OR-, IF-transitions (OR without priority), on transitions with degree (1,2) or (2,1) and on places with infinite capacity we get the nets in /HY1/.

The question is now, how to make a programming language out of this concept, and not only any programming language but one that is very similar to PL/1.

The description of data manipulation can be done very easily using PL/1 assignment statements. The problem is to integrate Petri Nets into the language.

To do this we have to describe places and transitions. For places we simply use labels, and transitions are described with the aid of ON-conditions on labels.

A typical statement of a (unstructured) CAP program looks like:
ON(A,B) : C : D : E : RESULT := ARGUMENT1 + ARGUMENT2;
There is a transition with inputplaces A,B and outputplaces C,D,E. As data manipulation we have a simple addition.

When this transition becomes firable it may fire. When it fires it first computes its datamanipulation and then manipulates its input- and output-labels with respect to its firing rule.

2. An Introduction into CAP nets

2.1 Def.:

APN := (S,T,F) is called Petri Net Graph :<=>

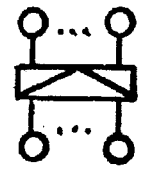
S finite nonempty set of places
T finite nonempty set of transitions
$F \subset S \times T \cup T \times S$, $S \cap T = \emptyset$
$\forall x \in S \cup T : \exists y \in S \cup T : ((x,y) \in F \vee (y,x) \in F)$

As usual by $^\bullet t$ is denoted the set of input places of transition t, by $t^\bullet$ the set of output places of t. Input places and output places are ordered, the position of an input place s of a transition t is given by $id._t(s)$, $id_t.(s)$ gives the position of an output place s. Places may contain tokens, where the capacity of places may be finite:
cap : $S \rightarrow N \cup \{\infty\}$ (capacity distribution)
m : $S \rightarrow N \cup \{0\}$ (total marking)
By M we denote the set of all markings of a given Petri Net.

2.2 Def.: (AND transition)

Let $A \subset T$. For every $a \in A$ holds:
a is firable under marking m : <=>
$(\forall s \in {}^\bullet a : m(s) > 0) \wedge (\forall s \in a^\bullet : m(s) < cap(s))$.
$f_a : M \to M$, $f_a(m) = m'$ is called firing of a : <=>
a is firable and $\forall s \in {}^\bullet a : m'(s) = m(s)-1$
$\forall s \in a^\bullet : m'(s) = m(s)+1$
$m'(s) = m(s)$ otherwise.

Symbol:

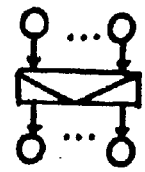


In the language we write:
$ON(\&(e_1,\ldots,e_n) : o_1 : \ldots o_m :$ or simply:
$ON(e_1,\ldots,e_n) : o_1 : \ldots o :$

2.3 Def.: (OR-transition)

Let $O \subset T$. For every $o \in O$
o is firable under marking m :<=>
$(\exists s \in {}^\bullet o : m(s) > 0) \quad (\forall s \in o^\bullet : m(s) < cap(s))$.
$f_o : M \to M$, $f_o(m) = m'$ is called firing of o : <=>
o is firable and $m'(s) = m(s)-1 :<=> s \in {}^\bullet o \wedge id_{.o}(s) = \min\{id_{.o}(s) m(s) > 0\}$
$m'(s) = m(s)+1 :<=> s \in o^\bullet$
$m'(s) = m(s)$ otherwise

Symbol:



In the language we write:
$ON(|(e_1,\ldots,e_n)) : o_1 : \ldots o_m :$

In addition we have other priority rules besides the above defined one as FIFO, LIFO etc. This shall not be considered in this paper.

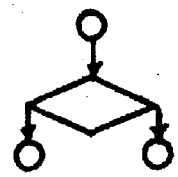
2.4 Def.: (DECIDER transition)

Let $D \subset T$. For every $d \in D$ holds:
$|{}^\bullet d| = 1$, $|d^\bullet| = 2$
For every decider transition d there is a predicate P_d on data.
d is firable under marking m :<=>
$(\forall s \in {}^\bullet d : m(s) > 0) \wedge (\forall s \in d^\bullet : m(s) < cap(s))$.
$f_d : M \to M$, $f_d(m) = m'$ is called firing of d :<=>
d is firable and $m'(s) = m(s)-1 :<=> s \in {}^\bullet d$

$$m'(s) = m(s)+1 :<=> \begin{cases} id_{d^\bullet}(s) = 0 \wedge P_d = \text{"true"} \\ id_{d^\bullet}(s) = 1 \wedge P_d = \text{"false"} \end{cases}$$

$m'(s) = m(s)$ otherwise

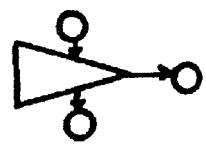
Symbol:



In the lanugage we write:
ON(e_1) : IF P_d THEN o_0:
ELSE o_1:

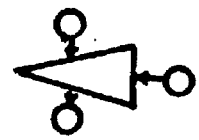
This transition may be generalized very easy to a CASE style transition. Special AND-transitions are used to call well defined subnets (blocks) and to synchronize their ready signal with the control flow of the calling block:

C-transition:



Representation in the language:
ON(e_1) : o_1 : .CALL o_2 :

R-transition:



Representation in the language:
ON(<-e_2,e_1) : o_1 :

2.5 Def.: (Blockhead transition)

Let $H \subset T$. For every $h \in H$ holds:
$|{}^\bullet h| > 1$, $|h^\bullet| = |{}^\bullet h|$
o is firable under marking m : <=>
$\forall s \in {}^\bullet h, id_h(s) = 0 : m(s) > 0$;

$\exists s \in {}^\bullet h, id_h(s) \neq 0 : m(s) > 0$;

$\forall s \in h^\bullet : m(s) < cap(s)$.
$f_h : M \rightarrow M$, $f_h(m) = m'$ is called firing of h : <=>

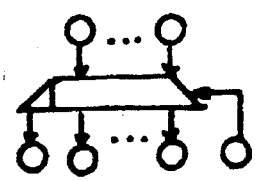
h is firable and

$m'(s) = m(s)-1$: <=> $s \in {}^\bullet h \wedge id_h(s) = 0 \vee$
$s \in {}^\bullet h \wedge id_h(s) = \min\{id_h(s) \neq 0 \mid m(s) > 0\}$

$m'(s) = m(s)+1$: <=> $s \in h^\bullet \wedge id_h(s) = 0 \vee$
$s \in h^\bullet \wedge id_h(s) = \min\{id_h(s) \neq 0 \mid m(s) > 0\}$

$m'(s) = m(s)$ otherwise

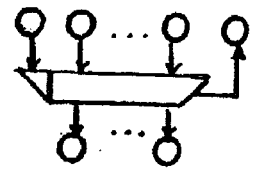
Symbol:



Again, we support other priority rules, too. The inputplace with $id._h = 0$ controls the activatability, the activity starts via the outputplace with $id_h. = 0$. The other places serve to identify and preserve the caller.

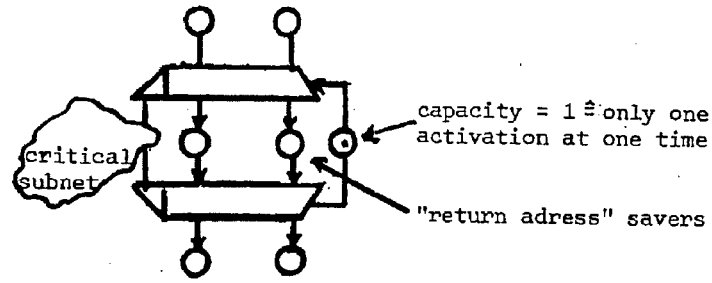
As blockend is used a symmetric transition with the same firing rule:

Blockend transition:



Representation in the language:
$ON(e_1)$: END

It should be noted, that the pair Blockhead-Blockend form a monitor-mechanism guarding a critical region. At any time only one reference to the net within this pair can be activated.



2.6 Def.: (CAP net, semiformally)

A Petri Net $N := (S,T,F)$ is called CAP net $:\Leftrightarrow$

1) $T \subset A \cup O \cup D \cup C \cup R \cup H \cup E$
2) $H \cap E = \emptyset$
3) $\forall s \in S : |{}^\bullet s| < 2, \quad |s^\bullet| < 2$
4) $\forall t \in T : {}^\bullet t \cap t^\bullet = \emptyset$
5) $k : H \to E$, total, one to one
   $e = k(h) :\Leftrightarrow$
   5.1) $s' \in e^\bullet, id_e.(s') = 0, s \in {}^\bullet h, id._h(s) = 0 \Rightarrow s = s'$
   5.2) $\forall s \in {}^\bullet e, id._e(s) \neq 0 : \{s \mid s \in h^\bullet \wedge id._e(s) = id_h.(s)\} \neq \emptyset$
   5.3) $|h^\bullet| = |{}^\bullet e|$
6) every $s \in {}^\bullet h$, $id._h(s) \neq 0$ is outputplace of a C-transition or h is the single outermost blockhead transition, every $s \in e^\bullet$, $id_e.(s) \neq \emptyset$ is inputplace of a R-transition or e is the single outermost blockend transition.
7) every block is synchronized with the block by which it is called.
8) $s \in {}^\bullet h \cap e^\bullet \Rightarrow cap(s) = 1$
9) there is no recursive call.
10) the Petri Net Graph without the inputplaces s of the single outermost blockhead with $id._h(s) \neq 0$ and the outputplaces s of the single outermost blockend with $id_e.(s) \neq 0$ is strongly connected.

2.7 Def.: (initial marking, final marking, proper terminating, safe, reusable, well behaving)

Let be I the unique outermost blockhead trasition, Q the unique outermost blockend transition.
A marking m_I of a CAP net is called initial marking :<=>

$$\forall s \in S : \begin{cases} m_I(s) = 1 <=> (h \in H : s \in {}^{\bullet}h \wedge id._h(s) = 0 \vee s \in {}^{\bullet}I \\ m_I(s) = 0 \quad \text{otherwise} \end{cases}$$

A marking m_f of a CAP net is called final marking :<=>

$\exists s \in S, \quad s \in Q^{\bullet}: m_f(s) > 0$

Let be M_f the set of final markings.

A CAP net N is called to be safe :<=>

$\forall m \in M$, reachable from m_I: $\forall s \in S : m(s) < 2$

A CAP net N ist called proper terminating :<=>
From every marking m, reachable from m_I, a final marking is reachable.

a CAP net is called reusable:<=>

$$\forall n \in M_f, \text{ reachable from } m_I : \forall s \in S : \begin{cases} m(s)=1 \Leftrightarrow \exists h \in H : s \in {}^{\bullet}h \wedge id._h(s)=0 \vee s \in E^{\bullet} \\ m(s) = 0 \quad \text{otherwise} \end{cases}$$

The behaviour of a reusable CAP net is independent from its activation history.
A CAP net N is called well behaving :<=>
it is safe, proper terminating and reusable.

3. Structured CAP Programming

Strict nonprocedural programming is comparable with extremly GOTO-oriented programming. Therefore a more disciplined way of concurrent programming is introduced now. The idea is to construct CAP programs out of a few basic building blocks. These building blocks have a special syntactical notation and correspond to specific subnet structures. The syntactic notation is similar to the notation used in PL/1.

It will be shown later, that the constructs which are investigated in this paper are in some kind "sufficient". This does not mean, that they are "sufficiently convenient". For practical applications one misses constructs like "bridges" (synchronisation of two concurrent activities) or loop with multiple exits. But within this paper we investigate an extremly restrictive but "sufficient" set of constructs.

3.1 Elementatry Structured CAP Program

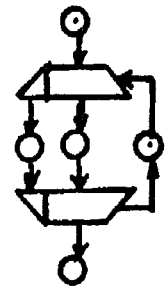
The most elementary syntactically correct CAP program looks like:

```
ON CALL(e ,...,e ) : o  : PROCEDURE
   ON(o ) : f  :;
ON(f ) : END;
```

We allow that this may be replaced by simply:

```
ON CALL(e ,...,e ) : PROCEDURE;
   ;
END;
```

This is called the elementary structured CAP program. It corresponds to the following CAP net:



Formally:
.(In the following we will have to identify a specific subset of the set of places within CAP net graphs, the set of Transparent Places, TP. We will add the component TP to the definition of CAP net graphs and CAP nets.

Def.: (Elementary Structured CAP Net)

Let be CN the set of all CAP nets.
$E \in CN$, $E = (P,T,F,TP,m_0)$ is called Elementary Sturctured CAP Net :<=>
$P := \{s_1,s_2,s_3,s_4,s_5\}$
$T := \{t_1 \in H,\ t_2 \in E\}$
F is given by $^\bullet t_1 = \{s_4,s_1\}$
$t_1^\bullet = \{s_2,s_3\}$
$^\bullet t_2 = \{s_2,s_3\}$
$t_2^\bullet = \{s_4,s_5\}$

Prop.: E is well behaving

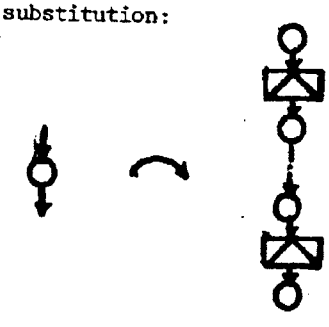
Proof: $[m>$ is given by $(1,0,0,1,0) \rightarrow (0,1,1,0,0) \rightarrow (0,0,0,1,1)$
I.e. E is proper terminating, safe and reusable.

3.2 Sequential Substitution

Any empty statement (a statement consisting only out of a ";") may be replaced by:

```
DO SEQUENTIAL;
   ;
   .
   .   m empty statements
   .
   ;
END;
```

This corresponds to the following net substitution:



Formally:

Def.: (SEQ-Substitution)

Let be SCN the set of all structured CAP net graphs
seq: SCN → SCN seq(O) = (N), O := (PO,TO,FO,TPO),
N := (PN,TN,FN,TPN)
is called SEQ-Substitution :<=>
PN := (PO \ {s}) ∪ {$sn_1,\ldots sn_{i+1}$} with s ∈ TPO arbitrary, i ∈ ℕ
TN := TO ∪ {$tn_1,\ldots,tn_i$}
FN := (FO \ FO' ∪ FN'
FO' := {$(t_1,s) \in FO$} ∪ {$(s,t_2) \in FO$}
FN' := {$(t_1,sn_1),(sn_{i+1},t_2)$} ∪
{$(tn_j,sn_{j+1}),(sn_j,tn_j) \mid j = 1 : i$}
TPN := (TPO \ {s}) ∪ {$sn_1,\ldots,sn_{i+1}$}

Prop.: O with initial marking well behaving => N with initial marking well behaving

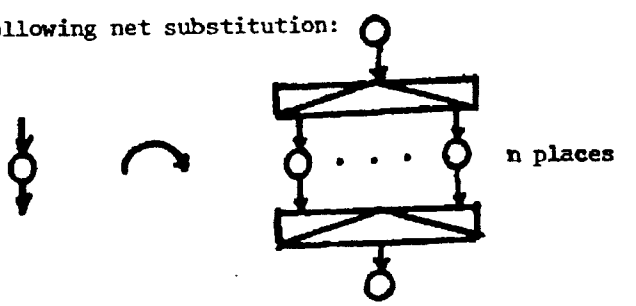
Proof.: O well behaving => ∃m : m(s) = 1 ∧[m(s)>∩ $M_f \neq \emptyset$
=> ∃m' : m'($sn_1,\ldots,sn_{i+1}$) = (1,0,...0)
[m'> = (1,0,...0) → (0,1,...0) → ... → (0,0,...,1) → ...
I.e. seq preserves well behaving.

3.3 Concurrent Substitution

Any empty statement may be replaced by:

```
DO CONCURRENT;
   ;
   .
   .    m empty statements
   .
   ;
END;
```

This corresponds to the following net substitution:



Formally:

Def.: (CON-Substitution)

con : SCN → SCN, con(O) = (N), O := (PO,TO,FO,TPO),
N := (PN,TN,FN,TPN)
is called CON-Substitution :<=>
PN := (PO \ {s}) ∪ {$sn_1,\ldots,sn_{i+1}$}, s ∈ TPO arbitrary, i ∈ ℕ
TN := TO ∪ {tn_1,tn_2}
FN := (FO \ FO') ∪ FN'
FO' := {$(t_1,s) \in FO$} ∪{$(s,t_2) \in FO$}
FN' := {$(t_1,sn_1),(sn_i,t_2)$} ∪ {$(sn_1,tn_1),(tn_2,sn_i)$} ∪
{$(tn_1,sn_j),(sn_j,tn_2) \mid j = 2 : i-1$}

TPN := (TPO \ {s}) ∪ {sn_1,...,sn_i}

Prop.: O with initial marking well behaving =>
N with initial marking well behaving

Proof.: O well behaving => ∃ m : m(s) = 1 ∧ [m> ∩ M_f ≠ ∅
=> ∃ m' : m'(sn_1,..,sn_i) = (1,0,...0)
[m'> = (1,0,...,0) → (0,1,...,1,0) → (0,...,0,1) →
I.e. con preserves well behaving

3.4 Conditional Substitution

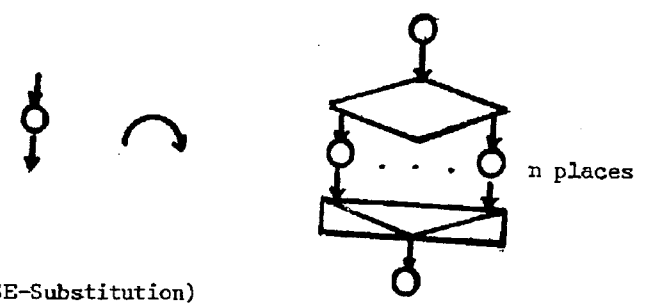
Any empty statement may be replaced by:

```
DO CASE;
   ;
   .
   .     m empty statements
   .
   ;
END;
```

or as special case by:

```
IF p THEN;
   ELSE;
```

This corresponds to the following net substitution:



Formally:

Def.: (CASE-Substitution)

con : SCN → SCN, case(O) = (N), O := (PO,TO,FO,TPO),
N := (PN,TN,FN,TPN)
is called CASE-Substitution :<=>
PN := (PO \ {s}) ∪ {sn_1,...,sn_i }, s ∈ TPO arbitrary, i ∈ ℕ
TN := TO ∪ {tn_1,tn_2}
FN := (FO \ FO') ∪ FN'
FO' := {(t_1,s) ∈ FO} ∪ {(s,t_2) ∈ FO}
FN' := {(t_1,sn_1),(sn_i,t_2)} ∪ {(sn_1,tn_1),(tn_2,sn_i)} ∪
{(t_1,sn_j),(sn_j,tn_2) | j = 2 : i-1}
TPN := (TPO \ {s}) ∪ {sn_1,...,sn_i}

Prop.: O with initial marking well behaving =>
N with initial marking well behaving

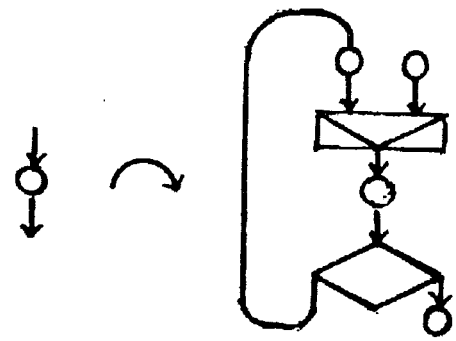
Proof.: O well behaving = ∃ m : m(s) = 1 ∧ [m> ∩ M_f ≠ ∅
=> ∃ m' : m'(sn_1,...,sn_i) = (1,0,...0)
[m'> = (1,0,...,0) → (0,0,...,0,1,0,...,0) → (0,0,...,0,1) →
I.e. case preserves well behaving

3.5 Iterative Substitution

Any empty statement may be replaced by:

```
DO WHILE;
   ;
END;
```

This corresponds to the following net substitution:



Formally:

Def.: (ITER-Substitution)

iter : SCN → SCN, iter(O) = (N), O := (PO,TO,FO,TPO)
N := (PN,TN,FN,TPN)
is called ITER-Substitution :<=>
PN := (PO \ {s}) ∪ {s_1,s_2,s_3,s_4} , s ∈ TPO arbitrary
TN := TO ∪ {tn_1,tn_2}
FN := (FO \ FO') ∪ FN'
FO' := {(t_1,s) ∈ FO, (s,t_2) ∈ FO}
FN' := {(t_1,s_1),(s_3,tn_1),(s_1,tn_1),(tn_1,s_2),(s_2,tn_2),(t_2,s_4),(t_2,s_3)}
TPN := TPO \ {s} ∪ {s_1,s_2,s_3,s_4}

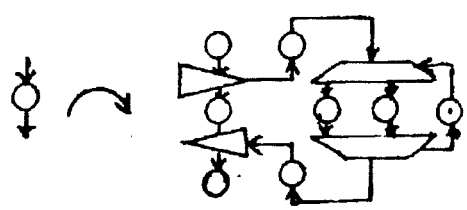
Prop.: O with initial marking well behaving =>
N with initial marking well behaving

Proof: O well behaving => ∃m : m(s) = 1 [m>∧M_f ≠ ∅
=> m' : m'(s_1,s_2,s_3,s_4) = (1,0,0,0)
∃m' = (1,0,0,0) → (0,1,0,0) → (0,0,0,1) → ...
(0,0,1,0)
I.e. iter preserves well behaving

3.6 Subblock Substitution

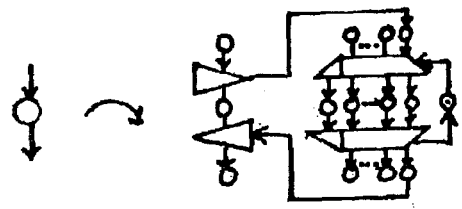
Any empty statement may be replaced by:
CALL a; (provided procedure a is declared and structured)

This corresponds to the following net substitution:



if it is the first reference

I.e. an Elementary Structured CAP-net has to be included into the net. It may be expanded due to its declaration afterwards.



otherwise

I.e. the subnet has already been included and only an additional "entry" has to be added.

Formally:

Def.: (BLK-Substitution)

blk : SCN → SCN, blk(O) = (N), O := (PO,TO,FO,TPO)
N := (PN,TN,FN,TPN)
is called BLK-Substitution :<=>
PN := (PO \ {s}) ∪ {s_i | i = 1 : 8}
TN := TO ∪ {$tn_1 \in C, tn_2 \in R, tn_4 \in H, tn_8 \in E$}
FN := (FO \ FO') ∪ FN'
FO' = {$(t_1,s) \in FO$, $(s,t_2) \in FO$}
FN' is given
1) $^{\bullet}tn_3 = \{s_6,s_4\}$, $^{\bullet}n_3 = \{s_6,s_7\}$, $^{\bullet}tn_4 = \{s_6,s_7\}$, $t^{\bullet}n = \{s_8,s_5\}$
2) $^{\bullet}tn_1 = \{s_1\}$, $^{\bullet}n_1 = \{s_4,s_2\}$, $^{\bullet}tn_2 = \{s_5,s_2\}$, $t^{\bullet}n_2 = \{s_3\}$
3) $t_i^{\bullet} = \{s_1\}$, $^{\bullet}t_2 = \{s_3\}$
TPN := (TPO \ {s}) ∪ {s_1,s_2,s_3,s_6}
blk is BLK2-Substitution :<=>
PN := (PO \ {s}) ∪ {s_1,s_2,s_3,s_4,s_5}
TN := TO ∪ {$tn_1 \in C$, $tn_2 \in R$}
FN := (FO \ FO') ∪ FN'
FO' = {$(t_1,s) \in FO$, $(s,t_2) \in FO$}
FN' = {$(t_1,s_1),(s_1,tn_1),(tn_1,s_4),(tn_1,s_2),(s_5,tn_2),(s_2,tn_2),(tn_2,s_3),(s_3,t_2),(s_4,t_3),(t_4,s_5)$}
with $t_3 \in TO \cap H$, $t_4 \in TO \cap E$
TPN := (TPO \ {s}) ∪ {s_1,s_2,s_3}

Prop.: O with initial marking well behaving =>
N with initial marking well behaving

Proof.:
a) BLK1-Substitution
The referenced block is an Elementary Structured CAP net. Therefore it may be replaced by a simple place preserving global behaviour. Then BLK1-Substitution is equivalent to CON-Substitution
b) BLK2-Substitution
By definition of H-transitions and E-transitions a multi-entry block behaves per reference like a single-entry block. Therefore BLK2-Substitution is equivalent to BLK1-Substitution.

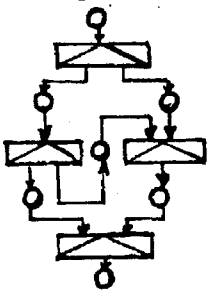
A CAP program is called structured CAP program iff it can be constructed by the above procedure.

3.7 Theorem

Every structured CAP program is well behaving.

Proof: Follows immediately from Prop. 3.1, 3.2, 3.3, 3.4, 3.5, 3.6.

The contrary is not true as ilustrated by the following example:



is not structured but well behaving

This means not that semantically the call of structured CAP programs is more restrictive than the class of well behaving CAP programs, as we have the following result: (A similar result for "bridge free" "Control Nets For Asynchronous Systems" /HY1/ with a constructive proof is given by /BSY/.)

3.8 Theorem:

For every well behaving CAP program N we have a structured CAP program N' with N' is prompt and hangup-free simulation of N.

Proof:

1. Let A := $\{a_1, \ldots, a_n\}$ be a λ-free, injective labelling of the transition of N (see /HA1/). As we have concurrency we have $L(N) \subset (\mathcal{P}(A))^*$ for the generated language.
   N well behaving => N safe => the set of possible total markings is finite
   => N finite nondeterministic automaton.
   => L(N) can be deribed by a regular expression over $\mathcal{P}(A)$
   The oprerations of regular expressions correspond directly to the substitutions for generating structured CAP nets if we allow multiple labels of transitions (denoting synchronized parallel execution).

2. Obviously this construction doesn't introduce hang-ups or additional loops. Therefore we get a hang-up free and prompt simulation.

It should be noted, that within this context we are speaking about uninterpreted nets. Indeed concerning any interpretation we would have to introduce additional variables to "construct" structured CAP-nets to simulate a CAP-net with multiple exits of loops or bridges (see /BSY, KO1/).

4. Implemented CAP software

CAP has been designed and implemented as specification language within a CAD system covering the field from firmware design to logic design. Especially supported by the CAP CAD-system is the construction of micro-processor based controllers.

Up to now we have impemented the following software:

4.1 CAP compiler CAPCOM

This system compiles CAP programs into an intermediate language called CAPID. It is a two step compiler with a LR-1 parser including an excellent error correction facility.

4.2 CAP interpreter CAPSIM

This is a virtual CAPID-machine The controlling Petri Net is modelled using

avent-oriented simulation.

4.3 CAP debugger CAPTEST

This system coopetates with CAPSIM and allows an interactive debugging either of CAP programs or of programs running on virtual machines that have been described in CAP. Up to now we have implemented such models of the TI990, INTEL 8085, INTEL 8048, DEC PDP8.

4.4 CAP documentation generator CAPDOC

A hierarchy of documents may be generated by this system during the top down design of a program, as it accepts also not fully formulated programs. In addition complete programs may be documented at various levels of abstraction under user control. As far as possible generalized Nassi-Shneiderman-Diagrams /NS1/ are generated.

4.5 CAP code generator family CAPCCL

An important application field for CAP is the design of micro-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 micro-processor or a multi-micro-system. 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 on several steps, where we remain processor independent as long as possible. Even the basic optimization is done without considering 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.

Appendix

a) A CAP programm: (for a language description see /RA2/)

```
on call (EXAMPLE1) : RUN : procedure;
    dcl (A,B) char (ebcdic, 80), I fixed;
    dcl file (SYSIN, SYSOUT) char (80);
    on (RUN) : LOOPGO : C :  B,A := SYSIN;
    on (C) : D :              substr (A, 0, 2) := '$$';
    on (|(LOOPGO, LOOPP)) : LOOPIN : I := I+1;
    on (LOOPIN) : if I≠80 then LOOPT : I := I+1;
                          else LOOPEND :;
    on (LOOPT)=LOOPP :       substr (B, I, 1) := 'T';
on (LOOPEND, D) : end;
```

b) Structured equivalent

```
on call (EXAMPLE1) : procedure;
    dcl (A,B) char (ebcdic, 80), I fixed;
    dcl file (SYSIN, SYSOUT) char (80);
    do sequential;
      B,A := SYSIN;
      do concurrent;
        substr (A, Ø, 2) := '$$';
        do I := Ø to 79 sequential;
           substr (B, I, 1) := 'T';
        end;
      end;
    end;
end;
```

References

/BSY/ Z. Barzilai, E. Strasbourger, M. Yoeli:
On structured parallel programming
Technion Haifa, Dept. of computer Science, Techn. Report 129
(1978)

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

/HA1/ M. Hack:
Petri Net Languages
MIT, Project MAC, Computation Structures Group Memo 124
(1975)

/HY1/ O. Herzog & M. Yoeli:
Control Nets for Asynchronous Systems, Part 1
Technion Haifa, Dept. of Computer Science, Techn. Report 74
(1976)

/KO1/ S. Rao Kosaraju:
Analysis of Structured Programs
Journal of Computer and System Sciences 9
(1974)

/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)

/PNC/ W. Brauer (ed.):
Net Theory and Applications
Springer
(1979)

/PR1/ L. Priese:
On the concept of simulation in asynchronous systems
4th ECMS, Linz (Austria)
(1978

/RA1/ F. Rammig: Überlegungen zur Kontrollstruktur einer Computer-Hardware-Beschreibungs-Sprache (german)
Universität Dortmund, Abteilung Informatik
(1978)

/RA2/ F. Rammig:
An Introduction to CAP or Looking at CAP with the Revised Iroman's Eyes
Universität Dortmund, Abteilung Informatik
(1979)

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