PB-152 Hansen & Madsen: Denotational Semantics and High-Level Petri Nets

ISSN 0105-8517

Formal Semantics by a Combination of Denotational Semantics and High-Level Petri Nets

Niels Damgaard Hansen
Kim Halskov Madsen

DAIMI PB-152
September 1982

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

FORMAL SEMANTICS BY A COMBINATION OF DENOTATIONAL SEMANTICS AND HIGH-LEVEL PETRI NETS

Abstract

Denotational semantics has proved to be an excellent tool for the specification of nearly all kinds of declarations and commands in sequential languages, but the description of concurrent processes is in practice nearly impossible.

High-level Petri nets, on the other hand, have proved their value in the specification of communication and synchronization of concurrent processes.

We propose to combine the two models into a single approach, where denotational semantics is used to build up environments and to describe store transformations, while Petri nets are used to describe sequencing and communication.

This paper has been presented at the 3rd European Workshop on Theory and Applications of Petri nets, Varenna, Italy, September 1982 . The proceedings of the workshop will be published as a volume in Informatik Fachberichte, Springer-Verlag.

1. INTRODUCTION.

The purpose of the paper is to introduce a semantic model based on a combination of denotational semantics [Gordon] and high-level Petri nets [Genrich, Lautenbach], [Jensen]. The basic idea behind our approach is due to Kurt Jensen.

Denotational semantics has proved to be an excellent tool for the specification of nearly all kinds of declarations and commands in sequential languages. Even complicated manipulations of stores and environments are described in a natural way. The description of sequential control structures is handled by continuations, which have a very nice underlying mathematics but yields rather abstract descriptions difficult to use for non-experts. The description of concurrent processes is in practice nearly impossible.

High-level Petri nets, on the other hand, has proved its value in the specification of sequential and especially concurrent processes. Communication and synchronization are described in a natural way. It is, however, difficult to handle complex data structures and the description of even simple commands, such as assignments, gets only little support from the net structure itself. The description of scope rules is in practice nearly impossible.

As indicated above denotational semantics and high-level Petri nets are complementary semantic tools, in the sense that the qualities of one, to a very large degree, coincide with the weaknesses of the other. We propose to combine the two models into a single approach, where denotational semantics is used to build up environments and to describe store transformations, while Petri nets are used to describe sequencing and communication. A semantics of a language is, in our approach, given along the lines used in denotational semantics, i.e by defining a set of :

- syntactic domains (such as declarations and commands)
- syntactic clauses (in BNF-form)
- semantic domains (such as stores, environments and high-level Petri nets)
- semantic functions (defining a syntax-directed mapping from syntactic domains into semantic domains)

The main difference between our approach and denotational semantics is the inclusion of high-level Petri nets as a semantic domain. This enables us to avoid the use of continuations, since jumps and control structures can be represented by branches and loops in the nets. Nets can be bound into environments and the semantic functions of programs and commands map into nets. The inscriptions of the nets, on places, transitions and arcs, are obtained by means of other semantic functions. The marking of a program-net contains a token for each sequential process. The position of the token acts as a program counter representing the current progress of execution, while the colour attached to the token represents the current store of the process.

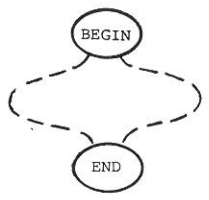
The approach is intended mainly to contribute to the specification of concurrent languages, although it can be used for sequential languages too. It is in section 2 introduced by means of a little sequential language, MINIPASC, inspired by, but not being a pure restriction of Pascal [Jensen, Wirth]. Section 3 defines a concurrent language, MINICONC, designed as an extension of MINIPASC in the direction of Concurrent Pascal [Hansen]. The small languages reflect to a large degree the substantial features of the corresponding large languages. The reader is assumed to be familiar with Pascal and Concurrent Pascal.

It should be remarked, that this is the very first attempt to use the new approach. It should be evaluated remembering the long time, and the many efforts, used to get denotational semantics to its current state. If our approach turns out to be valuable, much work remains to be done. On the practical side notation and auxiliary functions have to be improved, while on the theoretical side analysis methods and the well-definedness of the recursive functions involving net domains have to be established.

2. MINIPASC.

In this section we define the syntax and semantics of a small sequential language MINIPASC inspired by Pascal. The syntax is defined in Backus-Naur form and the semantics is defined by a combination of denotational semantics and high-level Petri nets. In the part described by means of denotational semantics we use as far as possible the same notation and auxiliary functions as in [Gordon].

For each kind of commands, C, we define a subnet with two distinguished places called BEGIN and END:



When a token is present at the BEGIN-place the command is ready for execution, whereas the finish of this execution is represented by a token present at the END-place. The colours, attached to the token at these two places, represent the store before and after the execution. If not explicitly indicated the colour set attached to places is Store, and the predicate attached to transitions is $s \neq errs$. Unless explicitly mentioned all places are initially unmarked. Sequencing of two commands is obtained by identifying the END-place of the first command with the BEGIN-place of the second.

In Denotational semantics we use the *-operator defined in [Gordon] p.47f:

f*g is the function corresponding to first doing f, if f produces an error then error is the result of f*g otherwise the result of f are "fed to" g. f*g is like $g \circ f$ except that

i) Errors are propagated.
Suppose f: D_1->[D_2+{error}] and g: D_2->[D_3+{error}] then
f*g: D_1->[D_3+{error}] is defined by

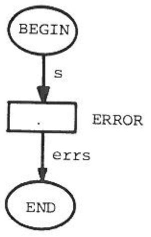
$$f*g = \lambda x.(fx = error)\text{->}error,\ g(fx)$$

ii) g may be curried.
Suppose f: D_1->[[$D_2 \times D_3$]+{error}] and
g: D_2->D_3->[D_4+{error}] then

$f*g : D_1 \rightarrow [D_4+\{error\}]$ is defined by:

$f*g = \lambda x.(fx=error)\rightarrow error,(fx=(d',d''))\rightarrow gd'd''$

In nets errors are handled in two ways. If an error occurs in evaluating the functions attached to the arcs of the net, the token colour is transformed into a special error-store, errs, which cannot be used in any transition firing. If an error occurs during the construction of a subnet by means of the semantic function *C*, a special error-net, errn, is included as a part of the obtained program-net:



In the following we define the semantics of MINIPASC along the lines used in [Gordon]. That is, by defining syntactic domains, syntactic clauses, semantic domains and semantic clauses.

SYNTACTIC DOMAINS.

- Primitive syntactic domains.

Ide	- the domain of identifiers I.
Bas	- the domain of basic constants B. The domain includes true and false .
Opr	- the domain of binary operators O.

- Compound syntactic domains.

Pro	- the domain of programs P.
Com	- the domain of commands C.
Dec	- the domain of declarations D.
Exp	- the domain of expressions E.

SYNTACTIC CLAUSES.

P ::= program D;C

C ::= begin $C_1;C_2;$ .... $;C_n$ end | I:=E | read (I) | write (E) |

$I_p(I_A)$ | if E then C_1 else C_2 | while E do C

D ::= $D_1;D_2;$ ... $;D_m$ | var I | const I=E | procedure $I_P(I_F)$;D;C

E ::= B | I | E_1O E_2

where $n > 0$ and $m \geq 0$.

The expression in the constant declaration const I=E, is composed of constants only.

SEMANTIC DOMAINS.

- Primitive semantic domains.

Bv	- the domain of basic values e. The domain includes the truth values true and false, and an error-value, errv.
Loc	- the domain of locations l. The domain includes two special locations "input" and "output".
Net	- the domain of high-level Petri nets n.

- Compound semantic domains.

File	= Bv *	- files i.
Sv	= Bv + File	- storable values v.
Ev	= Loc + Bv	- expressible values e.
Proc	= Net × Loc	- procedures p.
Dv	= Ev + Proc	- denotable values d.
Env	= Ide -> [Dv + {unbound}]	- environments r.
Store	= [Loc -> [Sv+{undef}]] + {errs}	- stores s.

SEMANTIC FUNCTIONS.

$\mathcal{B}$: Bas -> Bv

$\mathcal{O}$: Opr -> [Bv×Bv] -> Store -> [Bv×Store]

$\mathcal{P}$: Pro -> File -> Net

$\mathcal{D}$: Dec -> Env -> Env

$\mathcal{R}$: Exp -> Env -> Store -> [Bv×Store]

$\mathcal{E}$: Exp -> Env -> Store -> [Ev×Store]

$\mathcal{C}$: Com -> Env -> Net

$\mathcal{B}$ and $\mathcal{O}$ are defined by

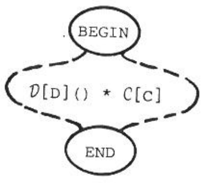
$$\mathcal{B}[B] = \begin{cases} e & \text{iff B is a literal for e} \\ \text{errv} & \text{otherwise} \end{cases}$$

$$\mathcal{O}[O](e_1,e_2)s = \begin{cases} (e,s) & \text{iff } e_1 O\ e_2 \text{ evaluates to e} \\ \text{error} & \text{otherwise} \end{cases}$$

where error=(errv,errs)

SEMANTIC CLAUSES FOR PROGRAMS.

(P) $\mathcal{P}$[<u>program</u> D;C]i =



The subnet denotes the net indicated by the inscription. This notation is used in the following.
The BEGIN-place of the program-net is initially marked by s=(i, ε / input,output). We consider input and output to be part of the store s. Initially the only used locations in the store s are "input" and "output" which have contents i and the empty file respectively. The declarations D are collected in the empty environment ().

SEMANTIC CLAUSES FOR DECLARATIONS.

$\mathcal{D}$[D]r is the environment r updated by the declaration D.

(D1) $\mathcal{D}[D_1;D_2; \ldots ;D_m]r = \begin{cases} \mathcal{D}[D_1]r*\mathcal{D}[D_2]* \ldots *\mathcal{D}[D_m] & \text{iff } m>0 \\ r \quad \text{iff } m=0 \end{cases}$

(D2) $\mathcal{D}$[<u>var</u> I]r = r[new r/I]
where the auxiliary function new returns a location not used in r.

(D3) $\mathcal{D}$[<u>const</u> I=E]r = $\mathcal{R}$[E]r() * λe s. r[e/I]
The identifiers in E may not include variables hence E is evaluated in the empty store ().

(D4) $\mathcal{D}$[<u>procedure</u> $I_P(I_F)$;D;C]r =

new r * λl.r[($\mathcal{D}$[D]r[l/I_F] * $\mathcal{C}$[C],l)/I_P]

At declaration time we bind to I_P a pair (n,l), where n is the net corresponding to the procedure-body C and l is the location corresponding to the parameter I_F. We define net(r I_P) = n and loc(r I_P) = l, i.e. the projections on the first and second components of r I_P, respectively. Similar projection are used for other cartesian products.
Recursive procedures are not allowed.

SEMANTIC CLAUSES FOR EXPRESSIONS.

(R) $\mathcal{R}$[E]r = $\mathcal{E}$[E]r * deref
where the auxiliary function
deref : Ev×Store ->Bv×Store
is defined by
deref e s = isLoc e -> (s e = undef -> error,(s e,s)),(e,s)
where the auxiliary function, isLoc, tests for membership in the semantic domain Loc. We shall without definition use other auxiliary functions which in a similar way test for membership of other semantic domains.

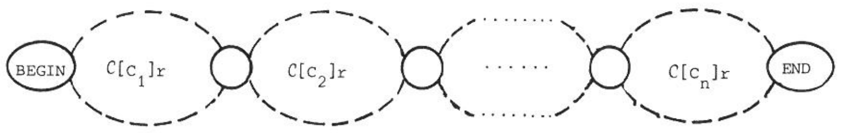
(E1) $\mathcal{E}$[B]rs = ($\mathcal{B}$[B],s)

(E2) $\mathcal{E}$[I]rs = rI = unbound -> error, (rI,s)

(E3) $\mathcal{E}[E_1 \text{ O } E_2]r = \mathcal{E}[E_1]r * \lambda e_1 . \ \mathcal{E}[E_2]r * \lambda e_2 . \ \mathcal{O}[O](e_1, e_2)$

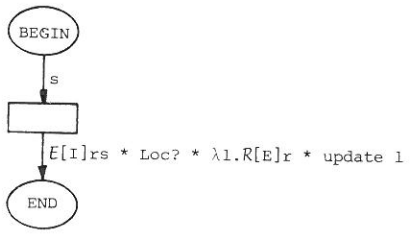
SEMANTIC CLAUSES FOR COMMANDS.

(C1) $\mathcal{C}$ [begin $C_1; C_2; \ldots ; C_n$ end]r =



The END-place of the net corresponding to C_i is identified with the BEGIN-place of the net corresponding to C_{i+1}.

(C2) $\mathcal{C}$ [I:=E]r =



where the auxiliary functions

Loc? : Ev -> Store -> Ev×Store

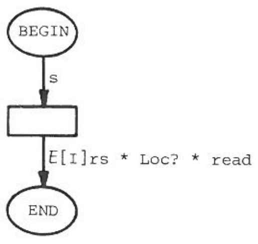
update : Loc -> Sv -> Store -> Store

are defined by

Loc? e s = isLoc e -> (e,s), error

update l v s = s[v/l]

(C3) $\mathcal{C}$ [read (I)]r =



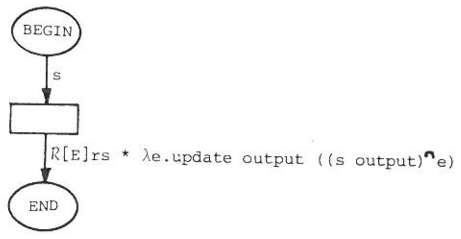
where the auxiliary function

read : Loc -> Store -> Store

is defined by

```
read l s = null(s input) -> errs, update l (head(s input)) s
                                * update input (tail(s input))
```

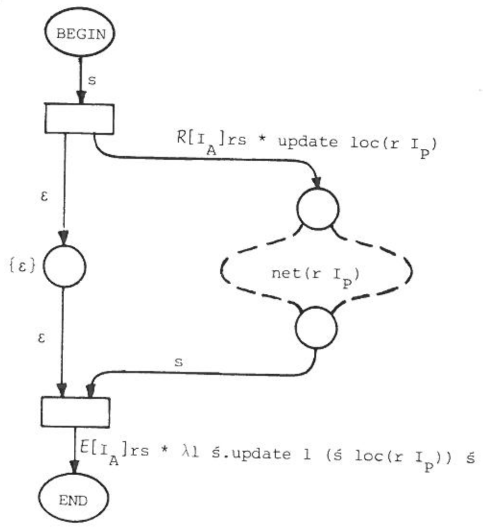
(C4) $\mathcal{C}$[<u>write</u> (E)]r =



where ⌢ denotes list-concatenation.

(C5) $\mathcal{C}$[$I_P(I_A)$]r = isProc(r I_P) AND isLoc(r I_A) -> n, errn

where n =

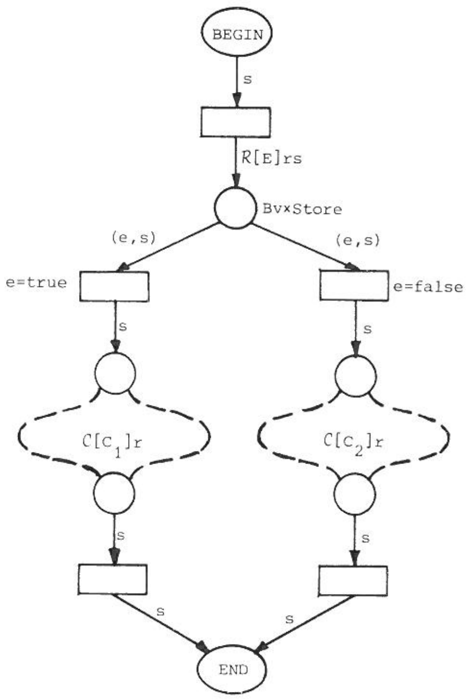

BEGIN
s
$\mathcal{R}[I_A]$rs * update loc(r I_P)
ε
{ε}
net(r I_P)
ε
s
$\mathcal{E}[I_A]$rs * λl ś.update l (ś loc(r I_P)) ś
END

If the same procedure is called at several different places only a single copy of net(r I_P) is included in the program-net.

The place with colour set {ε} guarantees that we return to the correct place.

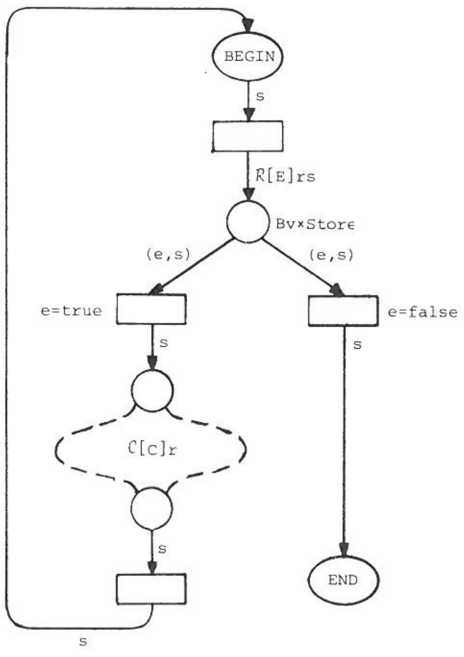
The parameter passing mechanism is "call by value and result".

(C6) $\mathcal{C}$[<u>if</u> E <u>then</u> C_1 <u>else</u> C_2]r =



In this and the following net the transitions may fire only when their predicates are satisfied.

(C7) $\mathcal{C}$[<u>while</u> E <u>do</u> C]r =



3. MINICONC.

In this section we define the syntax and semantics of a small concurrent language, MINICONC. MINICONC is an extension of MINIPASC described in the preceding section. In this section we specify only the extensions and modifications made. MINICONC reflects to a large degree the substantial features of Concurrent Pascal.

SYNTACTIC DOMAINS.

. Compound syntactic domains.
Ent - the domain of entry procedure declarations, Y.

SYNTACTIC CLAUSES.

C ::= <u>initmonitor</u> I_M | <u>initprocess</u> $I_P(I_A)$ | $I_M.I_E(I_A)$ |
<u>delay</u> (I_Q) | <u>continue</u> (I_Q) | ...

D ::= <u>monitor</u> I_M;D;Y;C | <u>process</u> $I_P(I_F)$;D;C |
<u>queue</u> $I_{Q1}, I_{Q2}, \ldots, I_{Qm}$ | ...

Y ::= $Y_1;Y_2;\ldots;Y_n$ | <u>procedure</u> <u>entry</u> $I_E(I_F)$;D;C

where $n > 0$ and $m \geqq 0$

The program clause defines an anonymous parameterless process called the initial process, and its purpose is to declare and initialize other processes and monitors. Each process and monitor can only be declared and initialized by the initial process, and it can only be initialized once.

Each process has exactly one monitor as parameter (except the initial process). Entry procedures and queue variables may only be declared within a monitor; queue variables can only be referenced from inside the monitor, whereas the entry procedures can only be referenced from outside the monitor.

SEMANTIC DOMAINS.

. Compound semantic domains.

Menv = Ide -> Proc	- monitor environments, y.
Mon = Net×Menv	- monitors, m.
Prcs = Menv -> Net	- processes, p.
Que = Ide*	- queues, q.
Dv = Ev + Proc + Mon + Prcs + Que + Ide	- denotable values, d.

SEMANTIC FUNCTIONS.

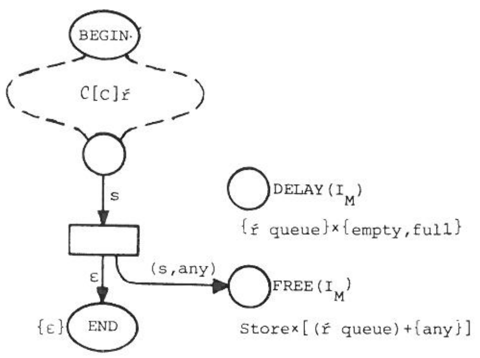
$\mathcal{Y}$: Ent -> Env -> Mon

SEMANTIC CLAUSES FOR DECLARATIONS.

(D5) $\mathcal{D}$ [monitor I_M;D;Y;C]r =

$\mathcal{D}$ [D] (I_M/monitor) * λŕ. $\mathcal{Y}$[Y]ŕ * λy.r[(n,y)/I_M]

where n =



The identifier "monitor" is used for remembering the name of the monitor.
The monitor environment, y, contains for each entry procedure the corresponding net and parameter location.
There is a FREE-place for each monitor, and the token-colours at this place are pairs. The first component represents the store of the local variables of the monitor, while the second component may take the names of the queue variables and the reserved identifier "any" as values. The marking of the FREE-place indicates which processes are allowed to use the monitor: no token indicates that the monitor is already occupied, (s,any) indicates that any process may enter it, while (s,I_Q) indicates that only

the process delayed at I_Q may enter.

There is also a DELAY-place for each monitor. This is always marked with exactly one token for each queue variable in the monitor. The second component of the token-colour indicates whether the queue variable is empty or not. The initial marking of the DELAY-place is :

$m_0(\text{DELAY}(I_M)) = \{(I_{Q1},\text{empty})\}+\{(I_{Q2},\text{empty})\}+ \dots +\{(I_{Qm},\text{empty})\}$.

(D6) $\mathcal{D}[$ <u>process</u> $I_P(I_F);D;C\ ]r = r[(\lambda y.\mathcal{D}[D](y/I_F) * \mathcal{C}[C]\ )/I_P]$

When the process is initiated it is passed a monitor environment, y, which contains the entry procedures of the actual monitor parameter.

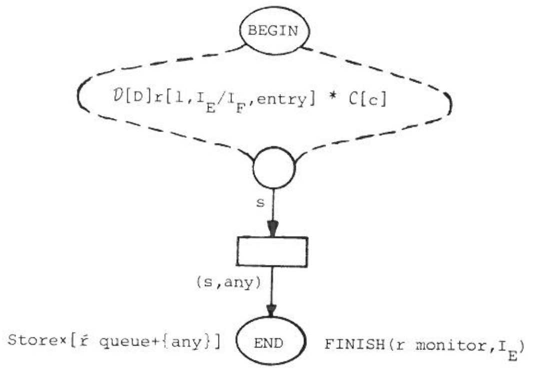
(D7) $\mathcal{D}[$ <u>queue</u> $I_{Q1},I_{Q2}, \dots ,I_{Qm}\ ]r = \begin{cases} r[\ ()/\text{queue}\ ] & m=0 \\ r[\ (I_{Q1},I_{Q2}, \dots ,I_{Qm})/\text{queue}\ ] & m>0 \end{cases}$

A sequence of queue-identifiers is bound to the reserved identifier "queue".

(Y1) $\mathcal{Y}[\ Y_1;Y_2; \dots ;Y_n\ ]r = (\mathcal{Y}[Y_1]r)[\mathcal{Y}[Y_2]r] \dots [\mathcal{Y}[Y_n]r]$

(Y2) $\mathcal{Y}[$ <u>procedure</u> <u>entry</u> $I_E(I_F);D;C\ ]r = \text{new}\ r * \lambda 1.(\ (n,1)/I_E)$

where n =

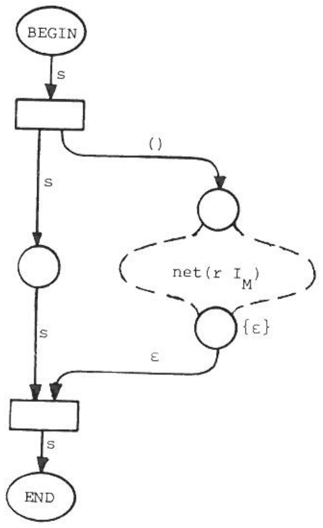


The identification "FINISH(r monitor,I_E)" of the END-place is used whenever the monitor is left due to the execution of a continue command, cf. clause (C12). The colour set attached to the FINISH-place is identical to the colour set attached to the FREE-place (cf. (D5)). The identifier "entry" is used for remembering the name of the entry procedure.

SEMANTIC CLAUSES FOR COMMANDS.

(C8) $C[$ initmonitor $I_M]r$ = $isMon(r\ I_M) \rightarrow n,\ errn$

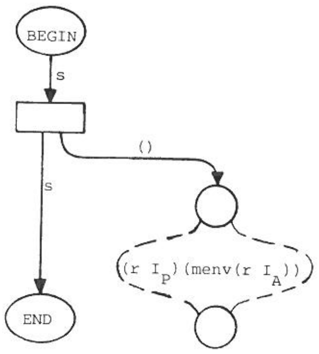
where n =



(C9) $C[$ initprocess $I_P (I_A)]r$ =

$(isPrcs(r\ I_P))$ AND $(isMon(r\ I_A)) \rightarrow n,\ errn$

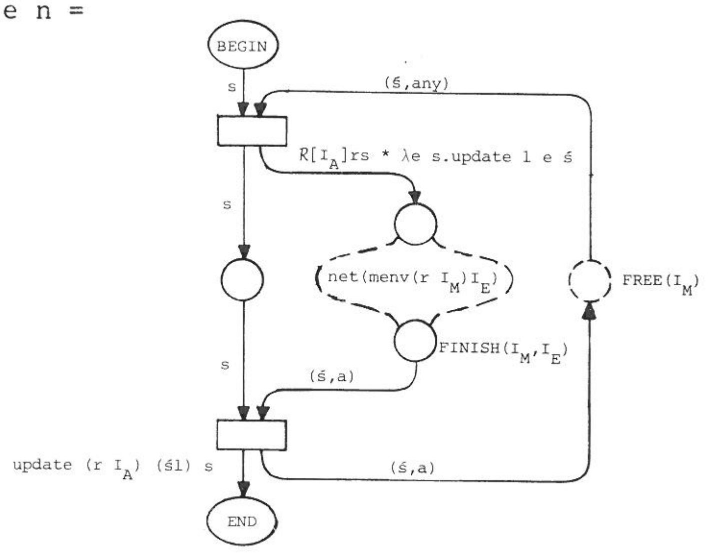
where n =



() is the empty store.

(C10) $\mathcal{C}[\ I_M . I_E (I_A)\ ]r$ = (isMon(r I_M)) AND (isLoc(r I_A)) ->
(isProc(menv(r I_M) I_E) -> n, errn), errn

where n =

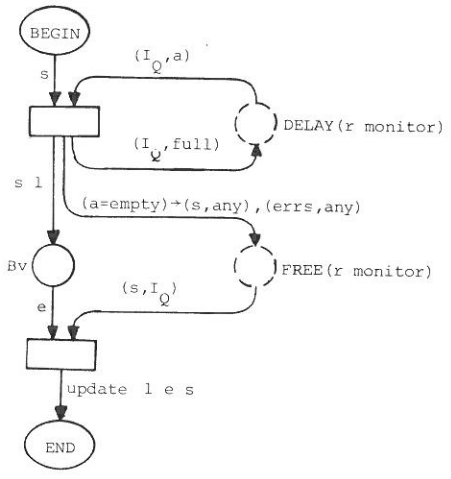


and l = loc(menv(r(r monitor))(r entry)).

menv(r(r monitor)) is the monitor environment. When passed the entry procedure identifier, (r entry), it returns a pair (n,l) where l is the location corresponding to the entry procedure parameter.

(C11) $\mathcal{C}[$ <u>delay</u> $(I_Q)\ ]r$ = (I_Q IN (r queue)) -> n, errn

where n =

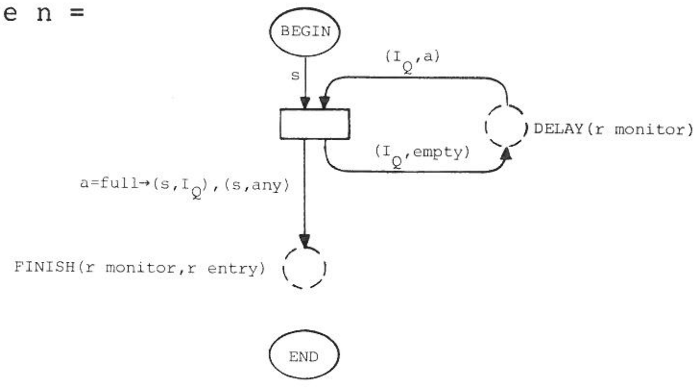


and l = loc(menv(r(r monitor))(r entry)) cf. (C10).

A delay on a nonempty queue variable causes an error-marking of the FREE-place. Otherwise the parameter value is preserved, and the monitor is left open.

(C12) C[<u>continue</u> (I_Q)]r = $(I_Q$ IN (r queue)) -> n, errn

where n =


BEGIN
s
(I_Q,a)
DELAY(r monitor)
(I_Q,empty)
$a=\text{full}\rightarrow(s,I_Q),(s,\text{any})$
FINISH(r monitor,r entry)
END

and l = loc(menv(r(r monitor))(r entry)) cf. (C10).

A continue on a nonempty queue variable causes the calling process to leave the monitor and hand over the monitor to the waiting process. If the queue-variable is empty the calling process returns from the monitor leaving it open. In both cases the rest of the entry procedure is skipped. Therefore no arc leads to the END-place of this command; instead an arc leads to the FINISH(r monitor,r entry)-place, which is the END-place of the entry procedure (cf. (C10)).

Acknowledgements.

We are grateful to Kurt Jensen for stimulating discussions and many helpful comments.

References.

[Genrich, Lautenbach]
Genrich, H. and Lautenbach, K.: System modelling with high-level Petri nets.
Theoretical Computer Science 13 (1981) 109-136.

[Gordon]
Gordon, G.: The Denotational Description of Programming languages.
Springer Verlag, New York 1979.

[Hansen]
Hansen, P.B.: The Architecture of Concurrent Programs.
Prentice Hall, New Jersey 1977.

[Jensen]
Jensen, K.: High-level Petri nets.
DAIMI PB-151, 1982.

[Jensen,Wirth]
Jensen, K. and Wirth, N.: Pascal user manual and report.
Springer Verlag, New York 1975.