ISSN 0105-8517

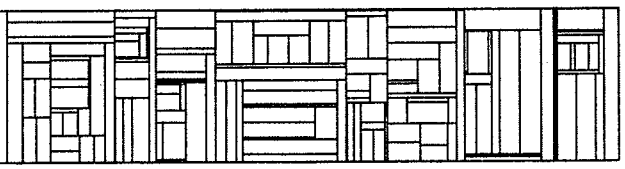
PETRI NETS AND SEMANTICS OF SYSTEM DESCRIPTIONS

by

Kurt Jensen
Morten Kyng

DAIMI PB-116
November 1981 (second edition)

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



PETRI NETS and SEMANTICS OF SYSTEM DESCRIPTIONS

by

Kurt Jensen and Morten Kyng

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C
Denmark

Abstract

This paper discusses the use of Petri nets as a semantic tool in the design of languages and in the construction and analysis of system descriptions. The topics treated are:

- Languages based on nets.
- The problem of time in nets.
- Nets and related models.
- Nets and formal semantics.
- Parallel program verification and nets.

Introduction

The purpose of this paper is to discuss some experiences with the use of Petri nets as a semantic tool. The material presented is from an ongoing research project, described in [Jensen, Kyng and Madsen, 79c]. The material is thus still under development and subject to change. The major part of the research project consists of the joint development of a system description language and a formal semantics for the language by means of a syntax-directed translation from system descriptions (programs) into net-models.

Our system description language is based on the Delta-language developed at the Norwegian Computing Center. The semantics of Delta was originally defined in a semi-formal way only, by means of an abstract interpreter described in the language itself [Holbæk-Hanssen, Håndlykken & Nygaard, 75]. In 1977 we defined the control-part of Delta (with minor restrictions) by means of a semantics, which was based on extended Petri nets [Jensen, Kyng & Madsen, 79a], [Jensen, 78]. This produced a number of proposals for corrections and simplifications of the original design. It turned out that many changes proposed to obtain simpler nets, also fulfilled the original design goals more satisfactorily. The simplifications implied that a stronger set of proof rules could be obtained. Moreover it would now be possible to rewrite the original abstract interpreter to less than half of its original size.

The work described above encouraged us to initiate a more radical redesign of the Delta-language and to use a net-model as a tool in the design process. This work is still in progress, and fragments of the evolving Epsilon-language have been presented in [Jensen, Kyng & Madsen, 79b].

The present paper is divided into five sections. In section 1 we informally introduce essential parts of Epsilon by means of an example and we discuss the scope of the language. In section 2 we discuss different aspects of time in nets in relation to language semantics. In section 3 we describe a net-model called concurrent-systems. In section 4 we define the semantics of some Epsilon-constructs by means of concurrent-systems. In section 5 we discuss how to prove properties of systems described in Epsilon.

The aim of the present paper is to provide a short intuitive introduction to our work on the five topics treated. Thus we focus on the presentation of ideas, at the expense of technical details. Readers, being interested further in particular aspects of our work, are referred to our more detailed papers contained in the reference list.

1. A LANGUAGE BASED ON NETS

In this section we introduce by means of an example some of the essential concepts of the system description language Epsilon. Epsilon is a successor of the Delta-language [Holbæk-Hanssen, Håndlykken & Nygaard, 75]. An earlier version of Epsilon is described in [Jensen, Kyng & Madsen, 79b]. The original Delta-language was developed without the use of net theory, but most of the main ideas behind the language design coincide with the system-view of general net theory.

System description

Epsilon is a system description language and not a programming language. We shall thus not limit ourselves to algorithms in the usual sense. We consider the much larger world of system description, where the systems may contain human beings, machines, physical/chemical processes, etc.

Such systems may develop continuously over a period of time. Although some systems cannot be adequately described using the classical time-concept, there are many situations where the existence of a global, totally-ordered time-scale shortens the description and enhances its clarity. Thus we allow the use of global time, but do not enforce it.

As already mentioned, Epsilon is not a programming language, and it is in fact not possible to implement it on a digital computer system. Thus efficiency of implementation is not a meaningful concern. We do not consider how to execute individual steps in algorithms or how to solve the equational systems specified in the language.

It is of interest to develop a programming language which is as far as possible consistent with the implementable parts of Epsilon. Together the two languages could be used to specify an edp-system and its environment, such as user-procedures and edp-department, and to program the edp-system itself. To some extent this consistency exists between Epsilon and the Beta systems programming language, [Kristensen, Madsen, Møller-Pedersen & Nygaard, 79]. In [Kyng, 76] it is described how to simulate a limited class of Delta systems on a computer.

Epsilon concepts

An Epsilon system consists of a number of objects, which each executes a sequence of actions. Objects may synchronize their actions via variables (e. g. the predefined global variable "time") or by means of direct communication as in [Hansen, 78], [Hoare, 78], [Ichbiah et al, 79], and [Kristensen, Madsen, Møller-Pedersen & Nygaard, 79].

Objects may execute two different kinds of actions called equational-actions and event-actions. An equational-action is specified by an equation over a set of variables together with a list of changeable variables. When a number of objects concurrently execute equational-actions the corresponding equations constitute a set of effective equations. The effective equations determine the values of the changeable variables (i. e. the union of the changeable variables associated with effective equations). Equations may involve (explicitly or implicitly) the predefined variable "time". The set of effective equations must be kept satisfied continuously as long as the corresponding actions are being executed. Summing up, equational-actions may involve continuous state transformations, the actual transformation may depend on the actions executed concurrently with the one considered (but only within the limits of the specified equation), an equational-action is executed over a period of time and its termination may depend on the actions of other objects.

An event-action is specified by a sequence of normal algorithmic statements. It is indivisible and instantaneous (with respect to the variable "time"). The effect of an event-action does not depend on concurrently executed actions. However, in some cases two objects jointly execute one event-action, this is illustrated in the example below.

Some language constructs: an example

The main language constructs to be discussed in this paper will be introduced by an example describing four balls following a circular orbit. The balls may move in both directions or stand still. Elastic collisions may appear between the balls. Two balls which collide will exchange their velocity (speed and direction). It is assumed that no external forces influence the system, i.e. no friction, no gravitation and no loss of energy. All balls have the same mass.

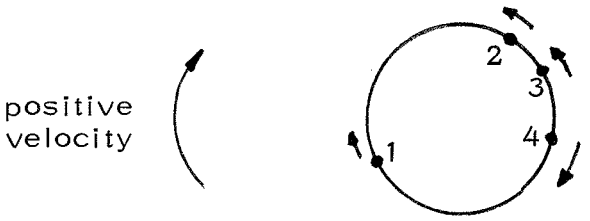


Figure 1.1

This system can be described in a number of different ways, depending on e.g. the purpose of the description and the language used. One such description has just been presented using normal english and a figure (fig. 1.1). In fig. 1.2 we present a more formal description using Epsilon. The purpose is to illustrate some language-constructs, and we will not discuss the specific choices made for this description.

The system described in fig. 1.2 consists of four ball-objects with identical specifications (3-18). Each ball-object has four variables of type real (4), which describe position, velocity, position of last collision, and time of last collision. And two constant references, initialised to its left and right neighbour respectively (5). After initialisation (6) each ball-object executes a guarded-statement (7-17). The guarded-statement specifies (7) an equation and a variable, pos, to be determined by the equation, when the ball-object is moving without colliding with its neighbours. A collision with the left neighbour is due to occur when the two balls are adjacently positioned (9: pos ≈ left.pos) and when the velocity of the left neighbour is greater than that of the object itself (9: vel < left.vel), cf. fig. 1.1. We allow more than two ball-objects to collide at the same moment of time and we synchronize

```
1   system
2         ball(1..4):
3         object
4               pos, vel, pos0, time0: real;
5               left, right: ref ball constant := ball(this ⊖ 1), ball(this ⊕ 1);
6               initialisation of variables;
7               let pos = pos0 + vel × (time - time0) determine pos
8               when
9                     □ pos ≈ left.pos, vel < left.vel, with left →
10                                          do vel := left.vel
11                                          end;
12                                          pos0, time0 := pos, time
13                    □ pos ≈ right.pos, vel > right.vel, with right →
14                                          do vel := right.vel
15                                          end;
16                                          pos0, time0 := pos, time
17              end let
18        end ball
19  end
```

Figure 1.2

their collisions pairwise (9: with left). When a collision occurs with the left neighbour, the two boolean expressions in (9) are true and the object-synchronization "with left" is matched with the corresponding "with right" in the left neighbour. Execution of the equational-action (7) is terminated and the statements following the arrow are executed (10-12). The assignment between "do-end" (10-11) describes part of an event-action executed jointly with the part described between "do-end" in the object which matched the "with left" (i.e. the "do-end" (14-15) in the left neighbour). The effect of the joint event-action is to exchange the velocities of the two colliding ball-objects. After this event-action the object assigns to pos_0 and $time_0$ (12). Then the equational-action described by (7) is resumed.

The net of the ball-system

In the net-semantics, which we describe further in section 3, execution of event-actions corresponds to the firing of transitions. Each transition has an attached expression containing a guarding boolean expression (prefixed by "when") and an assignment (prefixed by "do"). This is similar to the approach in [Keller, 76] and [Mazurkiewicz, 77]. Execution of equational-actions is represented by tokens on the corresponding places (e.g. a token on the place "MOVE" in fig. 1.3). Each place has an attached expression specifying an equation and a list of variables to be determined by the equation. The equation is prefixed by "let" and the variables by "det" (for "determine"). By default omitted guards and equations are always satisfied. Omitted assignments are dummy-actions.

In section 4 we discuss how to translate an Epsilon description into a net. In this section we merely present the net of the ball-system (fig. 1.3). The dashed lines indicate the subnet corresponding to a single ball-object. Initially all four BEGIN-places are marked with a single token each and all other places are unmarked. When a transition gets concession it is forced to fire at that moment of "time" (unless it looses concession by the firing of another transition), cf. section 2.

The use of boolean expressions in the guarded-statement (9 and 13) resembles the use in guarded commands [Dijkstra, 75]. Conflicts are resolved non-deterministically. There is, however, a profound difference between our guarded-statement and a normal algorithmic control-statement. Execution of the guarded-statement consists of a continuous state-transformation (specified by the equation), combined simultaneously with tests of the conditions determining the duration of the state-transformation. Execution of an algorithmic control-statement alternates between transformations and tests. This difference is enhanced by the fact that the single equation specified in the example above may in general be replaced by a sequence of equational-actions all supervised by the same set of guards, cf. section 4.

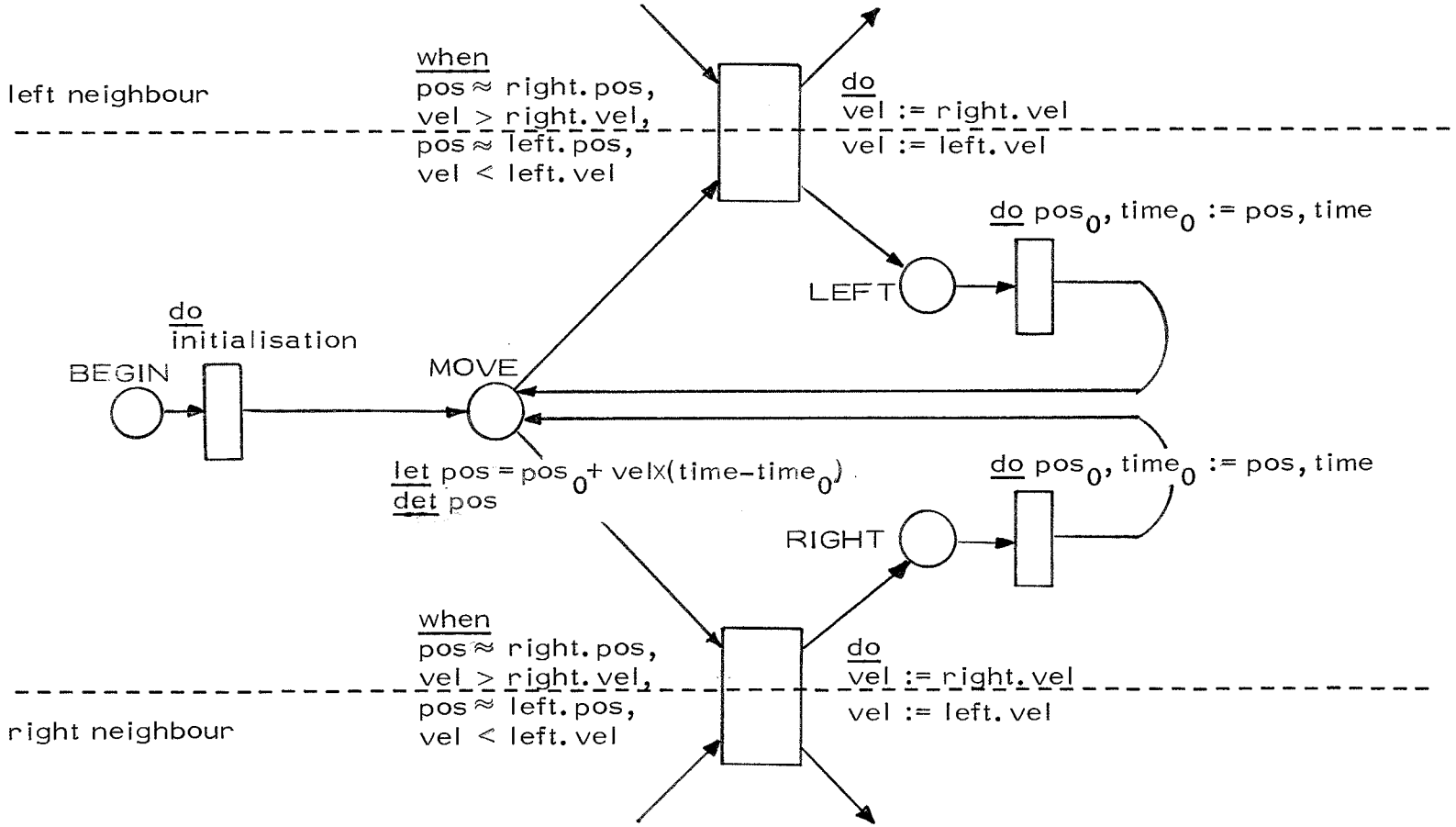

left neighbour
when
pos ≈ right.pos,
vel > right.vel,
pos ≈ left.pos,
vel < left.vel
do
vel := right.vel
vel := left.vel
do pos0, time0 := pos, time
LEFT
do
initialisation
BEGIN
MOVE
let pos = pos0 + velx(time-time0)
det pos
do pos0, time0 := pos, time
RIGHT
when
pos ≈ right.pos,
vel > right.vel,
pos ≈ left.pos,
vel < left.vel
do
vel := right.vel
vel := left.vel
right neighbour

Figure 1.3

2. TIME IN THE NET-MODEL

When we describe a system it is possible to apply a time-concept in at least two different ways. Either "time" is explicitly available, e.g. as a variable, and "time" may be used to describe the development of the system, as we did with the ball-system in section 1. In this case things change - at least with respect to the net-model - because "time" increases. This mode of description is used in classical physics. Alternatively "time" may be treated as a derived concept, which is not a priori available. Then "time" increases because things happen.

Further choices to be made regarding the time-concept include:

- is "time" considered continuous or discrete, and
- is "time" considered global (totally-ordered) or local ("relativistic")?

In our net-model we support the use of an explicit, newtonian (i.e. continuous, global) "time". Such a time-concept has proved its usefulness in a large number of applications, as mentioned in section 1. We are not - at least at present - considering the possibility of supporting a continuous, local "time".

Although we support an explicit, newtonian time-concept, it is possible for the person making a system description to decide to treat "time" as a derived concept, i.e. to make an Epsilon description which does not use the predefined variable "time". Such a description corresponds to a net which behaves like traditional Petri nets [Petri, 76]. Transitions, which have concession, are allowed to fire, but they are not obliged to. Thus we have an asynchronous model (although objects may communicate synchronously, as described in section 1).

The use of a newtonian time-concept in Epsilon is supported in the net-model by the variable "time". The behaviours of nets where this variable is used alternate between a continuous mode, where no transition has concession, i.e. the marking is constant, but the value of "time" is continuously increased, and an instantaneous mode, where the value of

"time" remains fixed, while the marking is changed by the firing of transitions. The duration of each continuous period is as short as possible, i.e. it ends when at least one transition gets concession. The duration of each instantaneous period is as long as possible, i.e. it continues until no transition has concession. Two transition firings belonging to the same instantaneous period are said to be time-equal. Concurrent firings are always time-equal (but the reverse is not true).

A continuous period ends by the firing of (at least) one transition, which did not have concession during the period. Since the marking has not changed, this implies that the value of the guard of the transition has changed.

When "time" is used, the model is synchronous, in the sense that the beginning of an instantaneous period is synchronized with "time" as described above. In some respects this resembles "synchronized Petri nets" defined in [Moalla, Pulou & Sifakis, 78]. In their model transitions are divided into external and internal transitions. Each internal transition is dominated by exactly one external transition. When an external transition firing occurs it is followed by a maximal firing sequence of internal transitions dominated by it. Then the system waits for the next external transition firing and so on. In our model the increase of "time" plays the role of external transition firings, and each increase dominates each (internal) transition.

The firing of transitions during an instantaneous period is still asynchronous in the normal Petri net sense. In fact there is no technical difference between the firing rule for nets corresponding to descriptions with and without the use of "time". The expressions attached to places and transitions in a net, simply do not involve "time" if the corresponding description does not. Each behaviour (firing sequence) of such a net contains only one instantaneous period. In terms of the analogy with "synchronized Petri nets" we may say that when a dead marking is reached, there are no external transitions to revive it.

There is, however, important differences between the traditional Petri net concepts such as liveness and deadlock, and those needed to study nets using "time". For example the existence of an infinite sequence of time-equal transition firings probably indicates an error if the net involves "time". And, as illustrated in section 5, when we prove properties of an Epsilon system using "time", we are primarily interested in markings corresponding to continuous periods, i.e. markings which are dead during the period.

3. THE MODEL: CONCURRENT-SYSTEMS

In this section we describe the semantic model, which is used to define the Epsilon-language. While Petri nets are excellent models for the control-flow of a language, they are less suited as models for state transformations in the data part. To remedy this situation we augment Petri nets with a data-part containing a set of variables.

The model, called concurrent-systems, consists of a triple: (CON, INT, INIT). The control-part, CON, is a condition/event-net. The interpretation, INT, specifies a set of variables and attaches an expression over some of the variables to each place and each transition. The initial state, INIT, specifies an initial marking and initial values for the variables.

The expression attached to a transition specifies a boolean expression (guard), which must be satisfied to obtain concession (in addition to the normal condition for concession), and a transformation, which is carried out on some of the variables, when the transition fires. The concurrency-relation for places and transitions is defined as for traditional Petri nets, except that two transitions to be concurrent also have to use disjoint sets of variables. In our figures the guard is prefixed by "when" and the transformation by "do".

The expression attached to a place specifies an equation and a set of changeable variables. In our figures the expression is prefixed by "let" and the variable list by "det" (for "determine"). When the place is marked the changeable variables are assigned values such that the equation is satisfied (together with the equations of other marked places). It should be noted that expressions attached to places may involve variables which change continuously. These expressions are needed to define the semantics of the equational-actions of Epsilon. The algorithmic statements of the language can be described by attaching expressions to transitions only, cf. [Keller, 76], [Mazurkiewicz, 77].

When a concurrent system is in continuous mode (cf. section 2) the set of equations associated with the marked places is continuously satisfied. by changing the values of some variables. When the system is in instantaneous mode the values of the variables cannot change between transition firings. Thus the equation-system is only solved once for each step in a firing sequence.

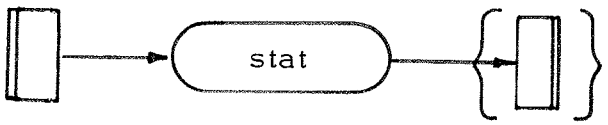
If a situation is reached, where the set of equations associated with marked places has no solution, we consider this as a descriptional error and we say that the concurrent-system is ill-behaved.

A formal definition of concurrent systems and their behaviours may be found in [Jensen, Kyng & Madsen, 79b].

4. LANGUAGE SEMANTICS DEFINED BY NETS

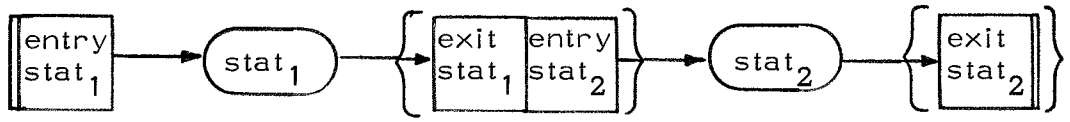
In the present section we define in terms of concurrent-systems the semantics of a few kinds of Epsilon-statements and discuss the limitations and shortcomings of our present approach. The use of a net-model has been an aid in developing a better understanding of concurrent processes and their communication and of the relation between continuous equational-actions and instantaneous event-actions (see section 1).

Each Epsilon-statement is represented by a concurrent-subsystem with exactly one transition distinguished as an entry and zero or more transitions distinguished as exists. In the general case the exit-transitions are partitioned into several sets depending on where to continue execution. Normal GOTO's are not allowed, but it is possible to skip the rest of a guarded-statement. In this paper we only consider exit-transitions leading to the immediately following statement. We depict the concurrent-subsystem representing a statement in the following way:



The entry-transition and the exit-transitions are indicated by vertical bars, to the left and right respecitvely. The braces indicate that there may be zero or more exit-transitions. The oval symbol named "stat" denotes an open subnet.

Sequential composition of two statements, "$stat_1$; $stat_2$", corresponds to the composition of the two concurrent-subsystems as shown below. Each exit-transition of $stat_1$ is concatenated with a copy of the entry-transition in $stat_2$. Concatenation of two transitions will be done only when the second transition has no guard. Such a concatenation yields a single transition, where the guard is taken from the first of the original transitions and the transformation is the sequential composition of the two original transformations. Concatenation is associative.

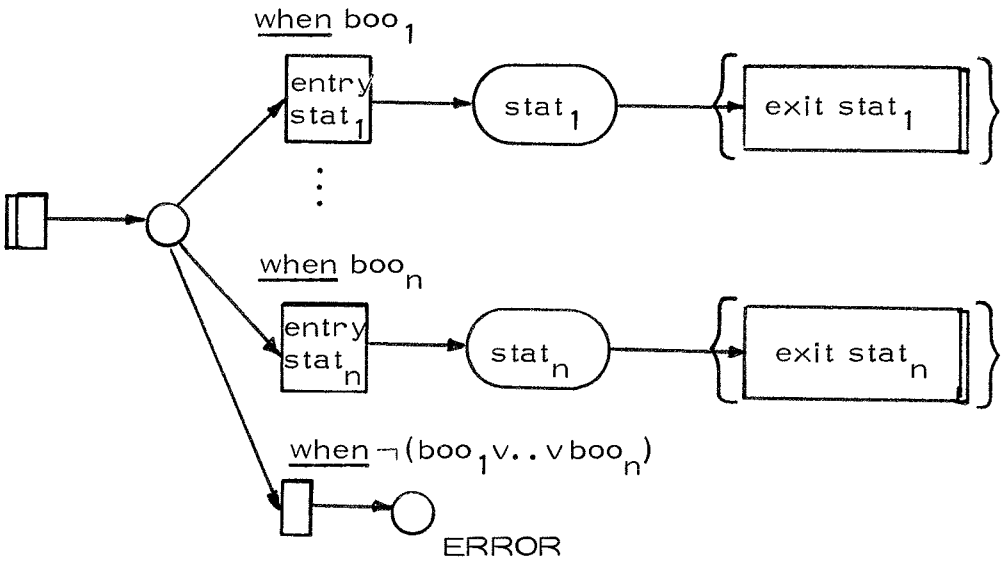


Before we proceed with the definitions of Epsilon-statements, we present the concurrent-systems of some - hopefully more familiar - statements: Dijkstra's guarded commands [Dijkstra, 75]. (These are not part of our language.)

The statement

"if $boo_1 \rightarrow stat_1$
⋮
$\square\ boo_n \rightarrow stat_n$ $(n \geq 1)$
fi"

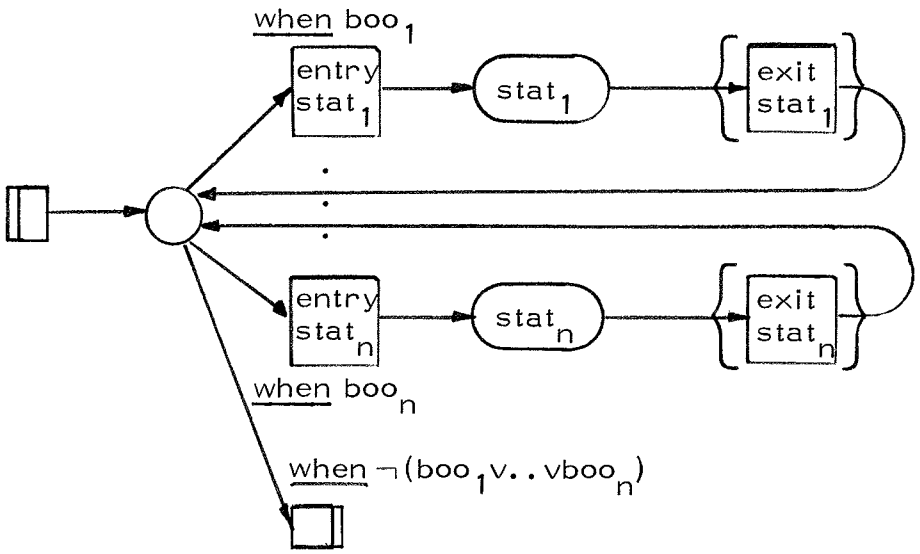
is represented by the concurrent-system



The statement

"do $boo_1 \rightarrow stat_1$
⋮
$\square$ $boo_n \rightarrow stat_n$
od" $(n \geq 1)$

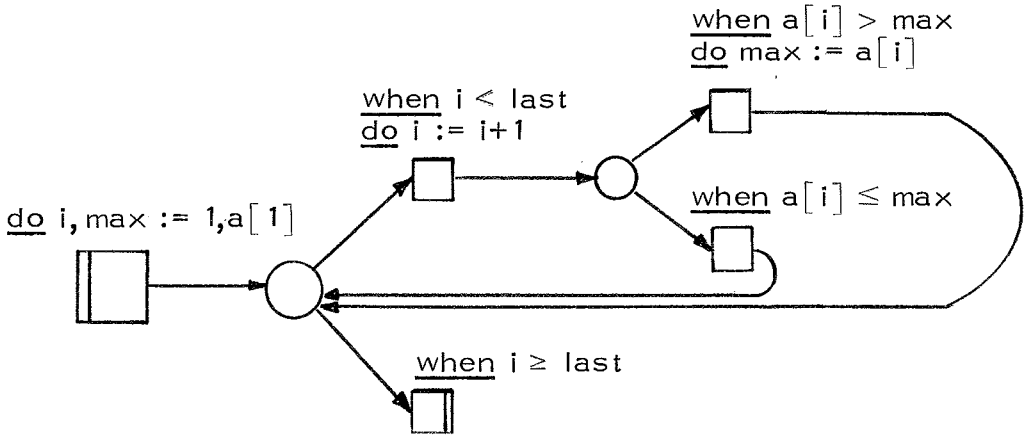
is represented by the concurrent-system



Next consider the following program-segment intended to find the largest value in the array a[1..last]:

"i := 1; max := a[1];
do i < last → i := i + 1;
  if a[i] > max → max := a[i]
  $\square$ a[i] ≤ max → skip
  fi
od"

Assignment-statements and skip-statements are represented by single transitions being both entry and exit. We get the following concurrent-system from the above program-segment (the error-branch of the if-statement has been omitted since its guard reduces to false):



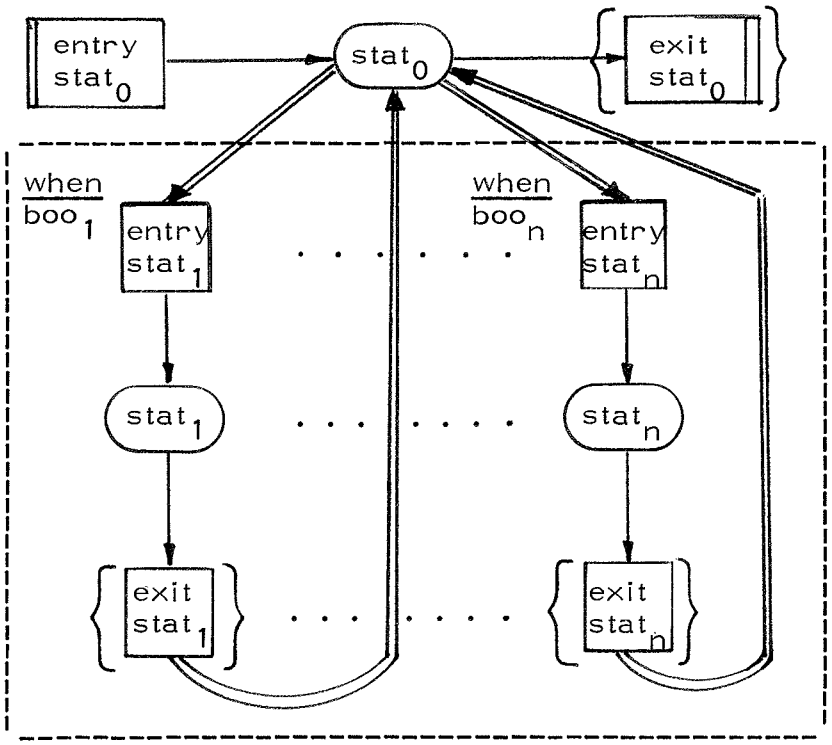
In this example we considered algorithmic statements, thus we attached expressions to the transitions only.

Now we return to the Epsilon-statements. A guarded-statement with boolean expressions only in the guards, i.e.:

$$
\begin{array}{l}
\text{"}stat_0 \\
\underline{\text{when}} \\
\qquad \square\ boo_1 \rightarrow stat_1 \\
\qquad\quad \vdots \qquad\qquad\qquad\qquad (n \geq 0) \\
\qquad \square\ boo_n \rightarrow stat_n \\
\underline{\text{end}}\text{"}
\end{array}
$$

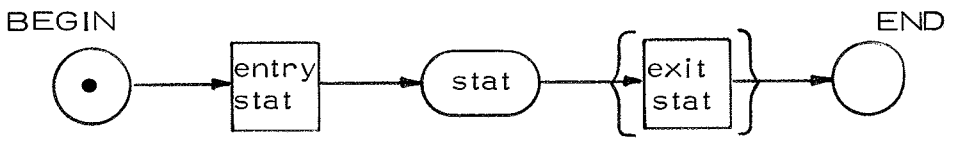
is represented by the following composition of concurrent-subsystems. The thick arrows and the dashed rectangle indicate that a copy of the subsystem representing the when-part is attached to each place in the subsystem representing $stat_0$. If $stat_0$ describes a single equational-action, the corresponding

subsystem contains only one place and the thick arrows could be replaced by normal arrows. If $stat_0$ describes several equational-actions they are all supervised by the guards in the when-part. The concurrent-subsystem representing $stat_0$ contains several places and a copy of the concurrent-subsystem inside the dashed rectangle must be attached to each place.



In the general case the guarded-statement may have exit-transitions other than those of $stat_0$, but this will not be discussed in the present paper.

An object is described by the following concurrent-system, where stat is the actions of the object. Initially only the BEGIN-place is marked.



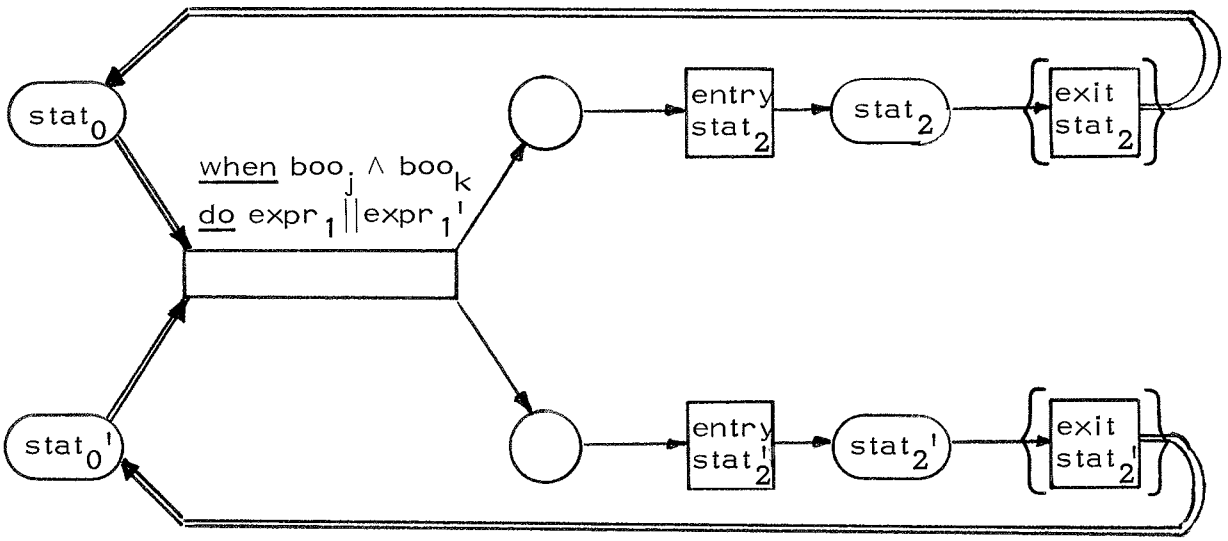
Now we consider the situation where one or more of the guards in a guarded-statement specifies synchronization with another object directly (not via variables), i.e.:

```
"stat0
 when
    ⋮
    ▯ booj, with S' → do
                          stat1
                      end;
                      stat2
    ⋮
 end"
```

This j'th branch in object S has to be selected together with a similar branch in object S':

```
"stat0'
 when
    ⋮
    ▯ book, with S → do
                         stat1'
                     end;
                     stat2'
    ⋮
 end"
```

We shall only allow the situation where $stat_1$ and $stat_1'$ are parts of an event-action, and update disjoint sets of variables. The two branches are then represented by the following concurrent-subsystem. The expression "do $expr_1 || expr_1'$ " means joint execution in the following sense: First the new values to be assigned to variables changed by $stat_1$ ($expr_1$) or $stat_1'$ ($expr_1'$) are found. Then these values are assigned to the variables (in non-deterministic order).



The construction above is also applicable when more than two objects synchronize their (event-)actions. Our approach is at present inspired mainly by [Hansen, 78], [Ichbiah et al, 79], [Kristensen, Madsen, Møller-Pedersen & Nygaard, 79], and thus by [Hoare, 78].

An Epsilon system containing a number of objects is represented by the union of the concurrent-subsystems representing each object. For each possible directly synchronized communication (i.e. each matching set of with-statements) the corresponding transitions are joined in the manner described above. When each object contains several with-statements addressing the same object(s) we have an exponential blow-up in the number of transitions representing direct communication.

Syntax-directed translation of Epsilon-descriptions into concurrent-systems could be formalized using an attribute grammar with concurrent-systems as attributes. It would then be possible to make a compiler, which uses concurrent-systems as target code. This is similar to the use of Net-Attributed-Grammars in [Hruschka & Kappatsch, 79].

Next we consider some of the limitations and shortcomings of our semantic model.

Petri nets are static in the sense that it is not possible to add or remove subnets. The use of recursive procedures and Simula-like classes of objects calls for concurrent-systems with an infinite number of identical subsystems representing different procedure/object-invocations.

To be concurrent two transitions must use disjoint sets of variables. Thus it is not possible, from the net-structure alone, to deduce whether two transitions are concurrent or not. We could represent each variable by a place, but this would generate large unstructured nets with many side-conditions. It is often difficult to make a clear distinction between control-part and data-part. It is for instance not obvious how to represent (dynamic) object-references.

As mentioned earlier in this section heavy use of direct communication may blow up the number of transitions, even with a static object structure.

Some of the problems described above can be overcome by the use of a net-model where information can be attached to tokens [Genrich & Lautenbach, 79], [Jensen, 79, 80]. It would then be possible to equip each token, acting as a control-pointer, by a colour which represents the data state of the corresponding object.

We have not yet established a satisfactory set of methods for the analysis of concurrent-systems. We return to this in the next section.

5. ANALYSIS OF SYSTEM DESCRIPTIONS

In the previous sections we have discussed how a net-model can be used to describe the semantics of programming and system description languages. We defined a class of net-models, called concurrent-systems, and a syntax-directed translation mapping each program or system description into a concurrent-system. The use of a formal model improves the language-design and may be of help, when presenting the language.

In this section we discuss how to analyze the concurrent-systems obtained from programs or system descriptions. A number of existing methods of net-analysis can be used for our model (e.g. [Lautenbach, 75], [Keller, 76], [Kwong, 77], [Mazurkiewicz, 77]). We shall not repeat the descriptions of these methods, but focus on how analysis can benefit from the distinction between continuous-mode and instantaneous-mode (cf. section 2).

For a specific language the syntax-directed translation maps into a small subclass of concurrent-systems only. As an example the semantics described in section 4 can be shown to guarantee that the concurrent-system representing an object is a state machine with exactly one token. In more complicated languages (e.g. the one described in [Jensen, Kyng & Madsen, 79b]) this need not be the case; but for any well-designed language it will be possible to infer a number of system properties from the mere fact, that we deal with concurrent-systems representing programs or system descriptions. Such proofs can be done by the use of structural induction [Stoy, 77] and the invariant method [Lautenbach, 75]. They need only be carried out once for each language, not for each program or system description.

The division into continuous-mode and instantaneous-mode can be utilized in the analysis of concurrent-systems. Each system state consists of a marking and values of the variables. We are primarily interested in the system states which occur in continuous-mode. It is an essential design goal of Epsilon that these system states are representative in the sense that they are abstractions of states in the modelled system. In contrast to this, we allow the system states of instantaneous-mode to be states, which may have no counterparts in the modelled system. In the

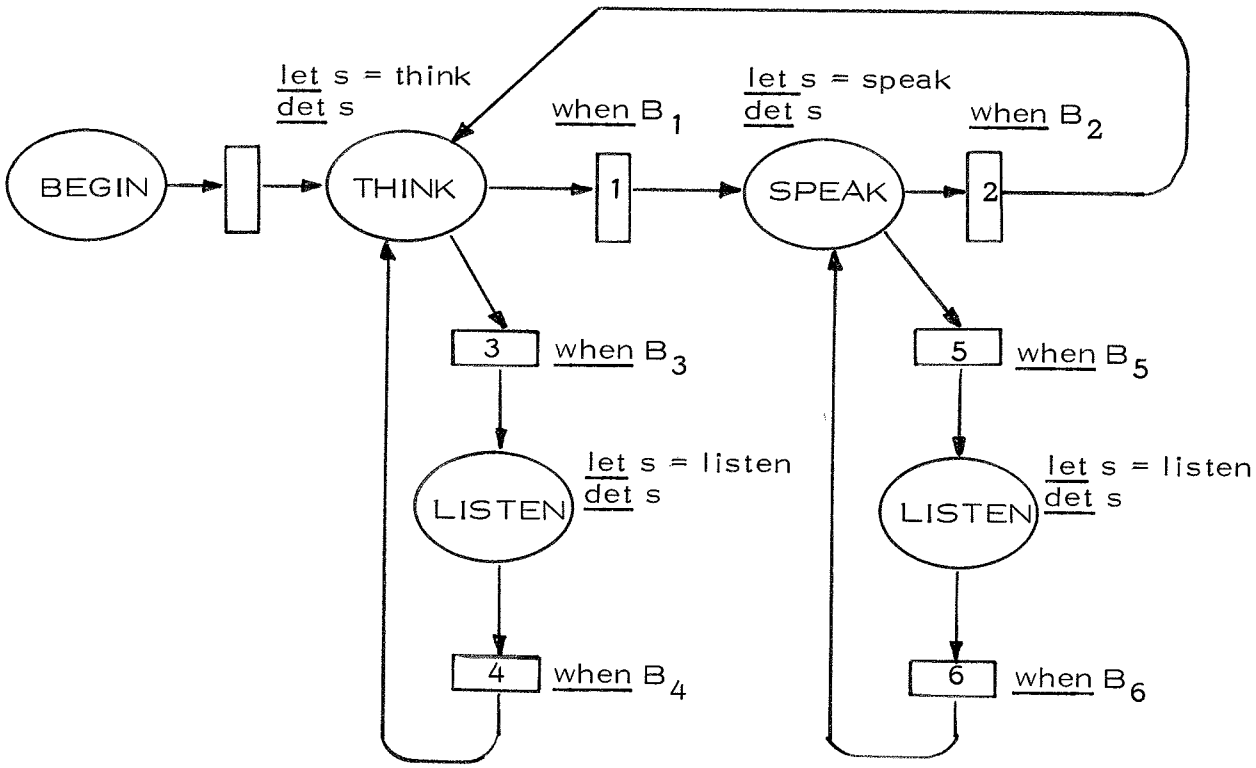
ball-example of section 1, each collision is described by the firing of three different transitions. There exist system states where the velocities of the two colliding balls have been updated "according to the collision", but where one or both balls remain to update the variables recording position and time of the last collision. We consider such system states to be without any physical counterparts.

From the description of concurrent-systems (in section 3) it immediately follows that all system states must be (forward) reachable. Furthermore if the system state is dead (i.e. no transition has concession) it belongs to continuous-mode and it should be representative. Whether the representative states include other states (i.e. states of instantaneous-mode) has to be decided for each Epsilon-description/concurrent-system by determining if any of the states of instantaneous-mode are abstractions of states in the modelled system. This process of determining a set of representative states and our insistence on being interested in these states only (at least during analysis) often drastically reduces the number of system states which have to be considered.

In the ball example of section 1, a token on any of the places labelled by BEGIN, LEFT or RIGHT gives concession to the transition which has the marked place as input-place. Thus the system states with these markings need not be representative, and we decide that they are not. We conclude that each representative state has a marking where the MOVE-places are marked for all four objects (called a MOVE-state). Finally we have to decide whether or not (some of) the non-dead MOVE-states should be considered as representative. If not, the representative states are exactly the states of continuous-mode, that is the states where no collisions occur. If we include the non-dead MOVE-states this means that we consider these states, where a collision has been "recognized, but not executed", as an abstraction of states in the modelled system. As illustrated below the implication of this is that we may consider a collision between more than two balls as composed of a set of pairwise collisions.

Given an occurrence-net for a concurrent-system the distinction between representative and non-representative states could be captured by a mapping analogous to a net-morphism. All transitions fired to get from one representative marking to the next is mapped (together with the interconnecting places) into a single transition, where the expression describe the total effect of all the original transition firings. The concurrent-system, which is the image of such a net-morphism, would be an occurrence-net, where the system states are exactly the representative states of the original occurrence-net. In the ball-example, a collision between more than two ball-objects will be represented by a closed subnet in the corresponding occurrence-net. If we demand that representative states are dead, such a closed subnet will be mapped into a single transition. If we include the non-dead MOVE-states in the set of representative states, then such a closed subnet will be mapped into a smaller one, where each transition represents a pairwise collision.

As in other formal models it is often convenient to create a proof simultaneously with the description. To illustrate this we describe a system, where two persons each are able to think, speak and listen. Each person is described by the following concurrent-system:



In the most general situation, where nothing is assumed about the guards of transitions, a single instantaneous-period may contain loops (i.e. infinite firing sequences). One of these loops may be avoided by demanding for each object that B_1 (i.e. desire to stop thinking) cannot be true at the same value of "time" as B_2 (i.e. desire to start thinking).

Next we define the other guards by the following expressions, where "partner" is a reference to the other person-object.

$$B_3 \equiv B_5 \equiv \text{partner.s} = \text{speak}$$
$$B_4 \equiv B_6 \equiv \text{partner.s} \neq \text{speak}$$

It is then straightforward to prove that the following property is satisfied for all dead states and thus for all states in continuous-mode:

$$\text{s} = \text{speak} \Leftrightarrow \text{partner.s} = \text{listen}$$

Both person-objects may start to speak simultaneously by concurrent firings of the transitions 1. Then we may have an infinite sequence of firings in a single instantaneous-period by alternation between simultaneous firings of the concurrent transitions 5 and of the concurrent transitions 6. This corresponds to the situation where two polite persons both want to speak, but wait infinitely for each other.

This example illustrates a difference between concurrency-models and interleaving-models. In an interleaving-model there would be a non-deterministic choice selecting the person-object which has to listen (one of the simultaneously enabled transitions 5 would be fired first, thereby preventing the firing of the other). In a concurrency-model there is also the possibility where the two concurrent transitions fire concurrently, i.e. both person-objects start to listen.

References

[Dijkstra, 75]
Dijkstra, E. W.: Guarded commands, nondeterminacy and formal derivation of programs. Comm. ACM 18, 8 (August 1975), 453-457.

[Genrich & Lautenbach, 79]
Genrich, H. J. and Lautenbach, K.: System modelling with high-level Petri nets. Theoretical Computer Science 13 (1981), 109-136.

[Hansen, 78]
Hansen, P.B.: Distributed processes: A concurrent programming concept. Comm. ACM 21, 11 (November 1978), 934-941.

[Hoare, 78]
Hoare, C.A.R.: Communicating sequential processes. Comm. ACM 21, 8 (August 1978), 666-677.

[Holbæk-Hanssen, Håndlykken & Nygaard, 75]
Holbæk-Hanssen, E., Håndlykken, P. and Nygaard, K.: System description and the Delta language. Norwegian Computing Center, Oslo 1975.

[Hruschka & Kappatsch, 79]
Hruschka, P. and Kappatsch, A.: Net-attributed grammars. Gesellschaft für Elektronische Informationsbearbeitung, Aachen-Walheim, 1979.

[Ichbiah et al, 79]
Ichbiah, J.D. et al: Preliminary ADA reference manual. SIGPLAN Notices 14, 6, part A, June 1979.

[Jensen, 78]
Jensen, K.: Extended and hyper Petri nets. DAIMI TR-5, Computer Science Department, Aarhus University, August 1978.

[Jensen, 79]
Jensen, K.: Coloured Petri nets and the invariant-method. Theoretical Computer Science 14 (1981), 317-336.

[Jensen, 80]
Jensen, K.: How to find invariants for coloured Petri nets. Mathematical Foundations of Computer Science 1981, J. Gruska and M. Chytil (eds.), Lecture Notes in Computer Science vol. 118, Springer Verlag 1981, 327-338.

[Jensen, Kyng & Madsen, 79a]

Jensen, K., Kyng, M. and Madsen, O.L.: Delta semantics defined by Petri nets. DAIMI PB-95, Computer Science Department, Aarhus University, March 1979.

[Jensen, Kyng & Madsen, 79b]

Jensen, K., Kyng, M. and Madsen, O.L.: A Petri net definition of a system description language. Semantics of Concurrent Computation, Evian 1979, G. Kahn (ed.), Lecture Notes in Computer Science vol. 70, Springer Verlag 1979, 348-368.

[Jensen, Kyng & Madsen, 79c]

Jensen, K., Kyng, M. and Madsen, O.L.: Petri nets as a semantic tool. Special Interest Group, Petri Nets and Related System Models, Newsletter No. 3, November 1979, 6-9.

[Keller, 76]

Keller, R.M.: Formal verification of parallel programs. Comm. ACM 19, 7 (July 1976), 371-384.

[Kristensen, Madsen, Møller-Pedersen & Nygaard, 79]

Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B. and Nygaard, K.: BETA language proposal as of April 1979. DAIMI PB-98, Computer Science Department, Aarhus University, June 1979.

[Kwong, 77]

Kwong, Y.S.: On reduction of asynchronous systems. Theoretical Computer Science 5 (1977), 25-50.

[Kyng, 76]

Kyng, M.: Implementation of the Delta language interrupt concept within the quasiparallel environment of Simula. DAIMI PB-58, Computer Science Department, Aarhus University, August 1976.

[Lautenbach, 75]

Lautenbach, K.: Liveness in Petri nets. Interner Bericht ISF-75-02. 1, GMD Bonn, July 1975.

[Lipton, 75]

Lipton, R.J.: Reduction: A method of proving properties of systems of processes. Comm. ACM 18, 12 (December 1975), 223-243.

[Mazurkiewicz, 77]

Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI PB-78, Computer Science Department, Aarhus University, July 1977.

[Moalla, Pulou & Sifakis, 78]

Moalla, M., Pulou, J. and Sifakis, J.: Synchronized Petri nets: A model for the description of non-autonomous systems. Mathematical Foundations of Computer Science 1978, J. Winkowski (ed.), Lecture Notes in Computer Science vol. 64, Springer Verlag (1978), 374-384.

[Petri, 75]

Petri, C.A.: Interpretations of net theory. Interner Bericht 75-07, GMD Bonn, July 1975.

[Petri, 76]

Petri, C.A.: Non-sequential Processes. Interner Bericht ISF-77-05, GMD Bonn, June 1977, translation of a lecture at University of Erlangen-Nürnberg, June 1976.

[Stoy, 77]

Stoy, J.E.: Denotational semantics: The Scott-Strachey approach to programming language theory. The MIT Press, Cambridge, Massachusetts, 1977.