DELTA PROJECT
REPORT NO. 12

ISSN 0105-8517

DELTA SEMANTICS DEFINED BY PETRI NETS

by

Kurt Jensen
Morten Kyng
Ole Lehrmann Madsen

DAIMI PB-95
March 1979
(revised version)

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

Preface

This report is identical to an earlier version of May 1978 except that Chapter 5 has been revised. A new paper : "A Petri Net Definition of a System Description Language", DAIMI, April 1979, 20 pages, extends the Petri net model to include a data state representing the program variables.

DELTA SEMANTICS DEFINED BY PETRI NETS

Kurt Jensen,
Morten Kyng, and
Ole Lehrmann Madsen

Abstract

Delta is a language designed for general system description. It is partly built upon Simula, but is more than a programming language, since it contains several features, which cannot be implemented on a computer system. E. g. a continuous time concept, concurrency between an unbounded number of components and the possibility of using predicates to specify state changes.

In this paper a formal semantics for Delta is defined and analysed using Petri nets.

Petri nets was chosen because the ideas behind Petri nets and Delta coincide on several points.

A number of proposals for changes in Delta, which resulted from this work, are also reported here, whereas a number of different extensions to the Petri net formalism may be found in [Jensen 78] (DAIMI TR-5).

TABLE OF CONTENTS

Chapter 5 :

Chapter 6 :

Chapter 1

INTRODUCTION

This paper defines, by use of Petri nets, a formal semantics for parts of the system description language Delta.

Delta was developed at the Norwegian Computing Center in 1973-75, and is partly built upon Simula. It is more than a programming language, in that it contains several features, which cannot be implemented on a computer system (e.g. a continuous time concept, concurrency between an unbounded number of components and the possibility of using predicates to specify state changes).

A formal syntax and a "semi formal" semantics for the language is defined in [Delta 75]. The semantics is defined in terms of partly a verbal description and partly an "idealised interpreter".

As a consequence of this definition method and because of the number of new concepts introduced, the exposition in [Delta 75] lacks clarity and stringency in several aspects. It thus seemed reasonable to assume that the understanding of the language could be enhanced by a formal definition of its semantics, and that the language itself could be improved by means of the insight gained from the formal definition.

As a tool in this semantic specification Petri nets seemed adequate, primarily because they support the study of concurrency as a discipline in its own right, and not as just an extra complication calling for further generalizations of the theory of sequential programs (such as e.g. interleaving).

Petri nets have received widespread use as models for systems containing asynchronous concurrent actions.

As stated in [Peterson 77] Petri nets have been used to describe, analyse and design a big spectrum of different hardware and software products.

The ideas behind Delta and behind Petri nets coincide on several points:

1. A system is considered as a number of components (processes acting in concurrency).

2. At discrete moments of time a set of events may cause an abrupt change in the system state.

3. In the open time intervals between events the system is described by a set of conditions satisfied by the system state.

4. System descriptions should be hierarchical, allowing further details to be added at succeeding levels.

1. to 4. summarize common ideas behind Delta and Petri nets. It is debateable to what degree the Petri net formalism, at present, satisfies 4.

To be fair we should also mention that there are some major differences between Delta and Petri nets:

(A) Delta has a global time concept.
Petri nets have not.

(B) In Delta events are mutually exclusive while Petri nets allow concurrent events if they operate on different resources (conditions).

On the other hand (A) and (B) may be taken as an indication of areas in which Delta's present semantics can be improved using experience from Petri nets.

Other uses of Petri Nets as a semantic model of languages may be found in [Lauer & Campbell 75] and [Pearl 78] where the semantics for path expressions and a process control language respectively are defined.

This paper assumes no special prerequisites.

It can be used

- as an introduction to Delta defining a semantics for the main part of the language. Used in this connection it would be preferable to read it "in concurrency" with [Delta 75].
- as an introduction to Petri nets and a major example of their use.

The rest of this paper is organized as follows:

Chapter 2 is an introduction to Delta and the subset of Delta (called Mini-Delta) for which we define a semantics.

Chapter 3 introduces Petri nets and the different extensions used. This chapter is essentially identical to the first five sections in [Jensen 78], in which a number of extensions to Petri nets are formally defined and compared.

Chapter 4 defines a semantics for Mini-Delta "equivalent" to the semantics defined in [Delta 75].

In chapter 5 we discuss different proposals for changes to the semantics defined in chapter 4.

Chapter 6 concludes the paper, points out some open problems and indicates directions for future work.

The different extensions to Petri nets reported in chapter 3 (and [Jensen 78]) are due to Kurt Jensen. All other chapters in this paper document joint work of all three authors.

We want to thank Antoni Mazurkiewicz and Kristen Nygaard for providing the initial inspiration for this work. We are grateful to Erik Meinecke Schmidt for many helpful discussions and comments. Moreover we have received helpful comments from Petter Håndlykken, Brian Mayoh and Peter Mosses.

Chapter 2

MINI-DELTA, A SUBSET OF THE DELTA LANGUAGE

The purpose of the Delta-project at the Norwegian Computing Center has been to develop a conceptual framework for conceiving systems, and an associated system description language for understanding and communicating about systems.

In this work SIMULA has been an important starting point. Besides being a programming language SIMULA is also a simulation language and it turns out that in many applications the experience gained by making a SIMULA description (i. e. a program) of the system considered is more useful than the actual simulation results ([Nygaard 73], [Simula 70]).

However, being a programming language, SIMULA imposes a number of restrictions on the system describer. He/she is forced to conceive a system as a model corresponding to the execution of a quasiparallel program, i. e. the basic description tool is the algorithm. Furthermore a set of algorithms used to describe a system has to be merged into one sequence of actions during program execution.

Delta tries to avoid "computer-imposed" restrictions. In a Delta description various ways of describing system properties can be combined:

- algorithms/predicates. In addition to algorithms which describe actions that explicitly make changes in the system state, predicates can be used to specify actions that implicitly change the system state.

- discrete/continuous. Discrete as well as continuous changes of the system state can be modelled. The latter by describing system properties which depend on time and change continuously while time is increasing.

- quasiparallel/parallel. It is possible to describe a system of components operating truly in parallel.

- formal/informal. It is possible to have a combination of a formal and an informal description. The latter by using natural language.

In this chapter we introduce some of the Delta concepts (section 2. 1) and a subset of the Delta language called Mini-Delta (section 2. 2). Section 2. 2 contains a formal definition of the context free syntax of Mini-Delta and an informal (and incomplete) definition of its semantics. In chapter 4 we present a formal definition of a semantics for Mini-Delta. For a complete description of Delta, the reader is referred to [Delta 75].

This chapter is based on [Delta 75] and we shall often refer to specific pages in this report. We sometimes use a modified terminology.

As Mini-Delta is a subset of Delta we shall not distinguish between Delta and Mini-Delta when this causes no confusion.

2. 1 Delta Concepts

The system considered for a description is called the referent system. The person making the description is called the system reporter. In a communication process a Delta system description may be used to generate a model system on for example, a black board or on a piece of paper or in one's mind. The person generating the model acts as a "generalized computer", a Delta system generator, executing a Delta system description. In [Delta 75] an idealised Delta system generator is defined which is able to generate canonical model systems. The idealised system generator defines the semantics of Delta.

In [Delta 75, p. 15] a system is defined in the following way:

"A system is a part of the world, which we choose to regard as a whole, separated from the rest of the world during some period of consideration, a whole which we choose to consider as containing a collection of components, each characterized by a selected set of associated data items and patterns, and by actions which may involve itself and other components".

AN OVERVIEW OF DELTA SYSTEMS

The components of a Delta system are called objects. All objects may execute actions in concurrency. Besides actions, an object is characterized by a set of data items and patterns (by patterns we mean types, functions, procedures and classes).

The state of an object is given by the value of its associated data items and its stage of execution. The state of a Delta system is the total set of states for all objects and the values of some data items associated with the system (the variable TIME described below is an example of such a system item).

The system state may be changed by means of usual algorithms. It may also be changed by means of predicates. When a predicate is imposed, upon the system state, parts of the state which correspond to the free variables in the predicate are assigned values in such a way that the predicate is satisfied.

Objects may interact by means of interrupts. That is, one object may force another object to execute some specific actions.

A Delta system description describes a system during some period of time. The variable TIME (or MODEL TIME) is used to model time in the referent system.

The execution of a Delta system takes place in two different modes:

- concurrent mode. All objects are executing actions. Changes in the system state are described by means of predicates, one for each operating object. That is, the action executed by an object specifies a predicate to be imposed upon the system state. The value of TIME is continuously increased. Predicates may refer to the value of TIME, which means that the state of the system is changed as TIME is increased.

- event mode. Only one object is executing actions. Changes in the system state may be described using algorithms as well as predicates. The value of TIME does not change.

The normal situation is that the system is in concurrent mode where the system states are assumed to model the referent system. Such states are called representative states. At some specific moment of TIME an event may take place and the execution shifts to event mode. An event appears if e.g.

- an object wants to stop imposing its present predicate upon the system state,

or

- a new object enters the system.

In event mode the objects may send interrupts. Event mode ends when

- the object wants to impose a new predicate upon the system concurrent with the other objects

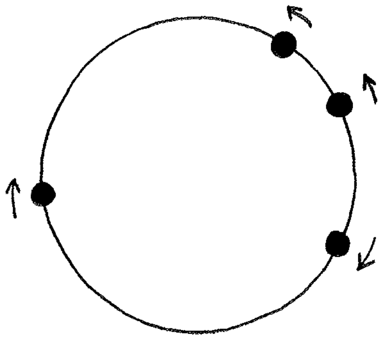
or

- the object has no more actions to execute.

In the following example we illustrate the concepts introduced.

EXAMPLE

We describe four balls which follow 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). An observer may place a "wall" in front of a ball and in this way negate its velocity. We assume that no other forces influence the system, i.e. no friction, no gravitation and no loss of energy. Furthermore we assume that the balls have the same mass and size.



First we give a rather informal description of the system where we merely introduce the objects in the system and the kind of actions they perform. The only data attributes being specified are the position on the orbit for each ball. Following the Delta conventions formal language elements are written in capital letters with key words underlined and informal language elements (i.e. not specified in detail) are written in small letters.

```
    OBJECT  BEGIN

    CLASS BALL :
         OBJECT BEGIN
             QUANTITY POSITION : REAL ;

             TASK BEGIN
                 REPEAT
                 (*
                    WHILE no-crash LET {POSITION = new-position}
                                        DEFINE POSITION ;
                    INTERRUPT BALLS [crashing-neighbour]
                                BY exchange-velocity ;
                    await ;
                 *)
             END TASK ;
         END BALL OBJECT ;

    BALLS : ARRAY [0:3] OF REF BALL ;

    REF OBSERVER :
         OBJECT BEGIN
             TASK BEGIN
                 REPEAT
                    (*
                       WHILE observing LET {observe};
                       INTERRUPT BALLS [selected-ball]
                                  BY change-direction ;
                    *)
             END TASK
         END OBSERVER OBJECT ;

    TASK BEGIN
         BALLS [0] : - NEW BALL ;
         BALLS [1] : - NEW BALL ;
         BALLS [2] : - NEW BALL ;
         BALLS [3] : - NEW BALL ;
         WHILE TRUE LET {TRUE} ;
    END TASK ;

END  OBJECT
```

Comments:

2 - 13 : Declares a category of BALL-objects which have a common structure.

4 : Declares a variable with the name POSITION of type REAL .

5 - 12 : A TASK describes the actions to be executed by the BALL-objects.

6 - 11 : The BALL-objects repeat infinitely the execution of the imperatives enclosed by (* and *).

8 : The WHILE - imperative is executed in concurrent mode. The predicate "POSITION = new-position" is imposed upon the system state. This has the effect that the value of POSITION is constantly updated to have the same value as "new-position". The execution of the WHILE-imperative continues until the condition "no-crash" becomes false.

9 : The INTERRUPT - imperative is executed in event mode. The BALL-object with which the collision happens is forced to execute the action "exchange-velocity".

10 : Await (see section 5. 1, p. 102) is executed in concurrent mode. The Ball-object waits for an interrupt to execute. Afterwards it continues with the imperative following await.

14 : The array BALLS denotes the four BALL-objects.

15 - 24 : A singular OBSERVER object is declared and generated.

18 - 22 : The OBSERVER may at will select and interrupt one of the BALL-objects in order to make it change direction.

25 - 31 : The TASK of the system object initiates the system.

30 : Having initiated the system, the system object continues infinitely to impose the predicate TRUE upon the system state.

A typical situation in the system will be that all four balls move following the circular orbit and that the observer merely "observe"s them. In this situation the BALL-objects will execute the WHILE-imperative at line 8 and the OBSERVER executes the imperative at line 20, all in concurrency. The predicates' "POSITION= new-position" and "observe" are concurrently imposed upon the system state while TIME is continuously increased.

If two balls collide, events will take place. When they collide the condition "no-crash" will become false in the two BALL-objects which collide. The mode of execution changes to event mode. Each of the colliding BALL-objects will in turn send an interrupt (line 9) to the other BALL-object. These interrupts will be received at line 10. The effect of line 9-10 will be that the two colliding BALL-objects will exchange velocity. Hereafter the system continues in concurrent mode as before the events.

An event will also happen if the OBSERVER decides to change the direction of one of the balls (the condition "observing" becomes false). The mode of execution shifts to event mode and the OBSERVER interrupts the "selected-ball". The BALL-object will in this situation be at line 8 and it will execute the action "change-direction" and return to line 8. Here the execution of the WHILE-imperative is (temporarily) stopped because of an interrupt. In the previous paragraph the execution of the WHILE-imperative was stopped because of its condition being false.

We have assumed that three or more objects will not collide at the same moment of TIME just as the observer will not place the wall in front of a BALL at the same moment of TIME as it collides with another BALL.

Below we give a more detailed description of the system. We have added more data attributes in order to specify how velocity and position of each object varies as TIME increases and when collisions appear. Note, however, that some language elements still are informal.

```
1      OBJECT BEGIN
       CLASS BALL :
3         OBJECT BEGIN QUANTITY N : INTEGER ;
4            QUANTITY VELOCITY, POSITION, P0, T0  : REAL ;

5            TASK PROCEDURE EXCHANGE-VELOCITY :
6            TASK BEGIN QUANTITY V : REAL ;
7               VELOCITY : = V;  P0 : = POSITION;  T0 : = TIME;
8            END EXCHANGE-VELOCITY TASK;

9            TASK PROCEDURE CHANGE-DIRECTION;
10           TASK BEGIN
11              VELOCITY : = -VELOCITY;  P0 : = POSITION;  T0 : = TIME;
12           END CHANGE-DIRECTION TASK;

13           BOOLEAN FUNCTION NO-CRASH:
14           BEGIN  NO-CRASH: = NOT (
15             ((POSITION - BALLS [N⊖1]. POSITION = distance) AND
               (VELOCITY < BALLS [N⊖1] VELOCITY)) OR
16             ((BALLS [N⊕1]. POSITION - POSITION = distance) AND
               (BALLS [N⊕1]. VELOCITY < VELOCITY )))
17           END  NO-CRASH ;

18           TASK BEGIN QUANTITY I : INTEGER;
19              P0 : = POSITION ;  T0 : = TIME ;
20              REPEAT
21                (*
22                  WHILE NO-CRASH
                       LET {POSITION = P0+VELOCITY * (TIME-T0)}
                       DEFINE POSITION ;
23                  I : = crashing-neighbour ;
24                  INTERRUPT BALLS [I]
                        BY BALLS [I]. EXCHANGE-VELOCITY
                           PUT (* V : = VELOCITY *);
25                  await
26                *)
27           END TASK
28        END BALL OBJECT ;
```

```
29      BALLS : ARRAY [0:3] OF REF BALL;

30      REF OBSERVER :
31          OBJECT BEGIN
32              TASK BEGIN QUANTITY I : INTEGER ;
33                  REPEAT
34                  (*
35                      WHILE observing LET {observe} ;
36                      I : = select-ball ;
37                      INTERRUPT BALL[I] BY BALL[I].CHANGE-DIRECTION;
38                  *)
39              END TASK
40          END OBSERVER OBJECT;

41      TASK BEGIN
42          BALL[0]:- NEW BALL PUT (* N:= 0; VELOCITY:=vo;
                                                   POSITION:= po *)
43          .
44          .
45          .
46          WHILE TRUE LET {TRUE}
47      END TASK ;
48    END  OBJECT
```

3 - 4 : The data attributes have been extended.

5 - 8 , 9 - 12 : The actions EXCHANGE-VELOCITY and CHANGE-DIRECTION have been specified by TASK procedures.

13-17: NO - CRASH has been specified as a boolean function. A collision may appear with its right neighbour (15) or left (16) respectively. Distance depends on the size of the balls and the circumference of the orbit.

19 : P0 and T0 are auxiliary variables used to store the value of POSITION and TIME at the last collision with another ball or wall.

22 : The predicate "new position" has been specified in detail. The value of POSITION is continuously changed as TIME increases.

24 : Note that the action EXCHANGE-VELOCITY to be forced upon BALLS[I] is the one local to BALLS[I]. The PUT-clause is a call by value parameter transfer.

□

We hope that by now the reader has an intuitive understanding of some of the main ideas behind Delta. In the following subsections we give a more detailed description of the semantics of Delta. These subsections may be skipped during a first reading.

OBJECTS

An object consists of:

- an object head containing the patterns, and the values of the data items of the object,
- a stack of activities containing the stage of execution of the object, and
- an agenda containing interrupts being sent to the object but which have not yet been executed.

DATA ITEMS AND PATTERNS

A data item is a named entity and can be either a quantity or a reference [Delta 75, p. 16]:

- a quantity is the association of a name and a constant or variable value which is a state observed through a measurement. The set of possible states is defined by a type.
- a reference is the association of a name and a constant or variable value being one specific object in the system or no object (NONE).

Quantities and references correspond to usual programming language elements such as variables and constants. It is the value aspect which is important. Quantities are used to represent values of abstract properties of an object. For example the age, hair-colour, weight, etc. of a person object may be described as quantities. References are used as names of objects. This corresponds to pointers or references in programming languages. Objects represent real physical objects such as persons, tables, etc. whereas quantities represent abstract properties of such objects.

An *action entity* represents the execution of a related set of actions (like the activation record and code of a block activation or of a procedure call in Algol).

An infinite set of elements with identical structure is called a *category.* A *pattern* defines a category:

- a *class pattern* defines a category of objects,
- a *procedure* defines a category of action entities,
- a *function* defines a category of action entities for computing a value, and
- a *type* defines a category of quantities.

A pattern does not in itself introduce any objects, action entities or quantities in the system but may be used by other language constructs to generate such ones. Objects, action entities or quantities generated in this way are said to be *category defined.*

Some of the above mentioned language constructs may contain the description of the object, action entity or quantity without referring to a pattern declaration. Objects, action entities or quantities generated in this way are said to be *singular*.

In Algol 60 a procedure call referring to a procedure declaration generates a category defined action entity, whereas an inner block generates a singular action entity.

Quantities, references and patterns are called the attributes of the object. The generation of an object implies the generation of an object head representing the attributes of the object.

A Delta system is organised as a nested structure of objects. Any object may have internal objects. The system as a whole is represented by the system object.
A subsystem is a set of objects consisting of an object (bound) and all its internal objects at all levels (content). A subsystem is represented by its bound.

A litter is a set of objects consisting of the system object or a category defined object (primary object) and all its singular first level internal objects plus their singular first level internal objects etc. (secondary objects).

There is a bijective correspondance between subsystems and bounds and between litters and primary objects.The set of litters is a partition of the set of all objects.

The system object and category defined objects are primary objects. Singular objects except the system object are secondary objects.

TASKS, ACTIVITIES AND INTERRUPTS

A task consists of a set of data items, quantities and patterns (except classes) and a sequence of actions. A task corresponds to a procedure or inner block in Algol 60 . Each object may have an associated task, its prime task . The generation of an object implies the generation of an action entity representing its prime task. The object immediately begins executing the actions of its prime task and is said to be operating. When all actions in the prime task are executed, the object becomes terminated.

When an object terminates all objects in its subsystem are also terminated.

An action in a task may be a task in itself. The action then includes the generation of a new action entity representing the task and the execution of the actions in this new action entity (a procedure call or an inner block). In this way the actions of an object are organized as a stack in the usual way as e. g. in Algol 60. Such a stack of action entities is called an activity.

Objects can communicate by means of interrupts. An object A may interrupt an object B with a given task. Object B will then temporarily postpone its activity and start executing the interrupting task enforced upon it by A. The interrupting task may itself contain the execution of new tasks and is thereby creating a new activity. When the interrupting activity is completed, the object resumes the interrupted activity. An interrupting activity may again be interrupted. Thus the total state of execution for an object may be described by a stack of activities. The reason for not viewing the whole collection of action entities as one stack is that the action entities inside an activity are logically connected whereas there may be no logical connection between activities.

An interrupt may or may not penetrate the resistance of the current action of the receiver. If the current action resists the interrupt, the interrupting task will remain appended to a "waiting list" called the agenda of the object. The interrupting task will be executed as soon as the receiver starts executing an action which is penetrated by the interrupt. A priority system is used to decide when an interrupt penetrates.

ACTIONS AND IMPERATIVES

A task may contain two kinds of actions:

- a concurrency action
  is executed concurrently with actions executed by other objects. Some concurrency actions may be interrupted.

- event action. No other object is executing any actions (changing the system state) while an event action is executed. An event action cannot be interrupted.

Actions are described by imperatives which are classified as either concurrency imperatives, event imperatives, composite imperatives or task imperatives.

A concurrency imperative describes a concurrency action. The entry into and the exit from a concurrency imperative are event actions.

An event imperative describes an event action.

A composite imperative is either a compound imperative (a sequence of imperatives), a conditional imperative or a repetition imperative. A composite imperative selects imperatives, e.g. sequencing (IMP_1;IMP_2) or testing some condition (IF B THEN (*IMP*)). Selection is an event action.

A task imperative is a procedure call or an inner block. It involves generation of an action entity, execution of the actions in this action entity, return from the action entity and removal of the action entity. Generation, initiation of execution of actions, return and removal are event actions.

In a usual programming language the state of the system (program execution) can be changed by using assignment imperatives. The assignment imperative is also part of Delta and is classified as an event imperative. No assignment imperative can be executed in concurrent mode, thus only predicates can specify state changes in concurrency imperatives.

Some actions use TIME in the sense that the value of the variable TIME is continuously increased during the execution of these actions.

Actions which may take TIME are called time consuming. Actions which take no TIME are called instantaneous. Only concurrency actions may be time consuming. Event actions are always instantaneous.

All time consuming actions may be interrupted and all interruptable actions are time consuming as the interrupting task may contain time consuming actions. Thus actions are interruptable if and only if they are time consuming.

The following is an example of a concurrency imperative which may start a time consuming action:

```
WHILE TIME < T LET {X = f(TIME)} DEFINE X;
```

This means that the value of X is constantly changed in such a way that X = f(TIME) while TIME < T. When TIME = T, the execution of the imperative is completed. All concurrency imperatives are variants of the above imperative in the sense that they contain a predicate to be imposed upon the system state and a duration clause/designational clause describing the condition for the action to continue.

THE OPERATION OF A DELTA SYSTEM

In the previous subsection we have seen that a Delta object may execute two kinds of actions, concurrency actions and event actions. When the idealised system generator [Delta 75, ch. 11] executes a Delta system this takes place in two different modes:

- concurrent mode. Each operating object is executing a concurrency action, thereby imposing a predicate upon the system state. The total set of these predicates is called the set of effective predicates. The system is said to be in a concurrent state. The value of TIME is continuously increased and as the effective predicates may depend on TIME, the state of the system may also change continuously. Each concurrency action being executed has a condition for it to continue. At some moment of TIME an event may take place. This happens in the following situations: 1) some concurrency actions continuation-condition becomes false, 2) a new object is generated, 3) an interrupt is penetrating, or 4) an object terminates. In all cases the mode of execution then shifts to event mode.

- event mode. Only one object executes actions. All other operating objects are stopped in the execution of a concurrency action. The value of TIME is fixed. The actions being executed by the active object is an event and consists of a sequence of event actions:

(i) If the object is operating then exit from a concurrency action else (the object is not operating then it has just been created and it will) start to execute actions in its prime task (and thereby become operating).

(ii) It then continues to execute event actions until it

(iii) either has executed the entry to a concurrency action or has no more actions to execute (it then terminates and is no longer operating).

The mode of execution then shifts back to concurrenct mode. An event brings the system from one concurrent state to another concurrent state. The states occuring during an event are called event internal states.

If two or more objects are to execute an event at the same moment of TIME then these events will take place in a nondeterministic order. If one event implies another event these will obviously occur in their logical sequence, Note that the concurrent mode occurs between any two events and this implies that the set of effective predicates will be imposed upon the system state, even if the events take place at the same moment of TIME. This means that the system state may be changed between two events.

Suppose that an object A is executing a task containing the following sequence of imperatives:

```
L1 : WHILE B1 LET PRED1 ;
         e1 ;
         IF B2 THEN (* e2 *)
         ELSE (* e3 ;
L2 :            WHILE B3 LET PRED2 ;
              *) ;
L3 : WHILE B4 LET PRED3
```

B1 - B4 are boolean expressions, PRED1 - PRED3 are predicates, e1 - e3 are event imperatives, and WHILE B LET PRED are all concurrency imperatives, and L1-L3 are labels.

Suppose that the system is in concurrent mode and that A is executing L1. Let the set of effective predicates be EP. (then PRED1 is in EP). Suppose that B1 becomes false then A is to execute an event. The following sequences of event actions may take place.

```
Exit from L1 ; e1 ;
   test of B2 ;
      true : (* e2 ;
                  test of B4 ;
                  true : (* entry to L3 *)
                  false : (*continue after L3 *)
            *)
      false : (* e3 ;
                  test of B3 ;
                  true : (* entry to L2 *)
                  false : (* test of B4 ;
                               true : (* entry to L3 *)
                               false : (* continue after L3 *)
                         *)
             *)
```

If both B2 and B 4 are true the set of effective predicates becomes (EP - { PRED1 }) ∪ { PRED3 } and this is the set of predicates which will be imposed upon the system state.

Recall that system states which have a meaningful interpretation in the referent system are called representative states. Concurrent states should always be representative whereas event interval states need not be.

If the system is observed during the TIME interval $[T_0, T_1]$, $-\infty < T_0 \leq T_1 \leq \infty$ the state of the model system can be described in the following way:

The TIME axis $[T_0, T_1]$ can be divided into a (possible infinite) number of open intervals

$$]T_0, S_1[,]S_1, S_2[, \ldots$$

where T_0, S_1, S_2, ... are the real numbers defining the moments of TIME when events take place. In each TIME interval a fixed number of predicates are satisfied by the system state. At TIME T_0, S_1, S_2, ... this set of predicates is changed. In addition the events may also have an effect on the system state.

If in some TIME interval the set of predicates cannot be fulfilled, then the system description is said to be erroneous..

2.2 The Mini-Delta Language

In this section we introduce Mini-Delta which is a subset of the Delta Language. We have focussed upon Mini-Delta as it contains the most interesting parts of Delta, which are:

- the ability to specify a system as a nested structure of objects operating in concurrency,
- communication between objects in form of interrupts, and
- the specification of actions in form of concurrency imperatives where predicates are used to specify the state of the system.

Furthermore it is possible to declare a data part of each object in form of quantities and references. As we are not modelling quantities we shall not use any fixed syntax for quantities. As a consequence of this we shall not decide upon the syntax of expressions and predicates.

The assignment statements have been excluded. The effect of this is that it is not possible to describe a change of state in the system by an algorithm. This can only be done by using predicates. Nearly all other event imperatives and compound imperatives have been included. We have excluded some language elements which are necessary if Mini-Delta should be used in practice, that is e.g. the virtual concept, parameters, priorities, and prefixing.

As a tool for system description, Mini-Delta is still very useful, also compared to Delta. By means of Mini-Delta one can make a high level system description where one only considers the structure of the system, the interaction between objects and the shift between states which are all representative. In Mini-Delta the important thing is not how a given state is imposed on the system state, but which predicates the state of the system should fulfill at any TIME. A more refined system description specifying the transistions between representative states would have to use full Delta.

The syntax of Mini-Delta is defined using the metalanguage proposed by N. Wirth in [Wirth 77]. It is an extension of BNF to contain regular expressions on the right-side of a production. Nonterminals are written as identifiers possibly containing a hyphen (-). Terminals are enclosed in quotes ("), e.g. "BEGIN". ::= is written as = , alternative as | . A clause enclosed by { } may appear zero or more times. A clause enclosed by [ ] is optional. Clauses may be grouped by inclosing them in ().

```
(1)   Mini-Delta-System-Description =
            Object-Description
```

This describes the system object.

```
(2)   Object-Description =
      "OBJECT" "BEGIN"
                {Object-Attribute-Declaration ";"}
                [Task-Description]
            "END" "OBJECT"
```

Describes an object, its attributes and its prime task.

```
(3)   Object-Attribute-Declaration =
           Pattern-Declaration
      |    Data-Declaration
```

```
(4)   Pattern-Declaration =
           Class-Declaration
      |    Task-Procedure-Declaration
```

```
(5)   Class-Declaration =
           "CLASS" Identifier ":"
                      Object-Description
```

Declares a category of objects having common properties as described by object description. Identifier is the title of the class.

```
(6)   Task-Procedure-Declaration =
           "TASK" "PROCEDURE" Identifier ":"
                     Task-Description
```

Declares a category of tasks. Identifier is the title of the task procedure.

```
(7)   Task-Description =
           "TASK" "BEGIN"
              {Task-Attribute-Declaration ";"}
              Imperative {"; " Imperative}
           "END" "TASK"
```

Describes a task by its attributes and its actions.

```
(8)   Data-Declaration =
           Quantity-Declaration
      |    Reference-Declaration
      |    Singular-Reference-Declaration
```

```
(9)   Quantity-Declaration =
           "QUANTITY" Identifier {"," Identifier}
                          ":" Quantity-Type
```

Declares a list of quantities of the same type. The syntax of Quantity-Type will not be specified.

```
(10)  Reference-Declaration =
            "REF" Identifier {"," Identifier}
                  ":" Class-Title
```

Declares a list of references with a given qualification. The reference(s) may have as value any object in the class with the name Class-Title or the value noobject (NONE).

```
(11)  Singular-Reference-Declaration =
            "REF" Identifier ":" Object-Description
```

Declares a singular object which is constantly referenced by the reference Identifier. The object is generated when its enclosing object is generated.

```
(12)  Task-Attribute-Declaration =
            Quantity-Declaration
      |     Reference-Declaration
      |     Task-Procedure-Declaration
```

Note that a task cannot have internal objects.

```
(13)  Imperative =
            Concurrency-Imperative
      |     Event-Imperative
      |     Composite-Imperative
      |     Task-Imperative

(14)  Concurrency-Imperative =
            Time-Concurrency-Imperative
      |     Instant-Concurrency-Imperative

(15)  Time-Concurrency-Imperative =
            Duration-clause  Property-Clause
            [Priority-clause]
            [Postponement-clause]
            [Resumption-clause]
```

The Duration-Clause describes the condition for the action to continue. The Property-Clause describes the predicate to be imposed upon the system state. The Priority-Clause describes the actions resistance against being interrupted. The syntax of Priority-Clause shall not be specified.

```
(16) Duration-Clause =
             Empty
      |      "WHILE" Boolean-Expression
      |      "PASSINGLY"
```

Empty means that the action will continue forever. "WHILE" means that the action will continue until the Boolean-Expression becomes false. The syntax of Boolean-Expression will not be specified. "PASSINGLY" means that the action will continue as long as there are penetrating interrupts. In all three cases the action will be completed if the objects encloser terminates just as it may be interrupted.

```
(17) Property-Clause =
             "LET" "{" Predicate "}"
             ["DEFINE" Variable {"," Variable} ]
```

Defines a predicate over variable quantities. The variables after DEFINE are the ones which may be changed in order to satisfy the predicate (the free variables). The syntax of predicate and variable will not be specified.

```
(18) Postponement-Clause =
             "EXIT" Compound-Imperative
```

Describes actions to be executed before a possible interrupting task is executed.

```
(19) Resumption-Clause =
             "REENTRY Compound-Imperative
```

Describes actions to be executed after a possible interrupting task has been executed.

```
(20)  Instant-Concurrency-Imperative =
              Designational-Clause [Property-Clause]
```

Describes an instantaneous and thus noninterruptable concurrency action. The execution of the action imposes a predicate upon the system state.

```
(21)  Designational-Clause =
      |       "PAUSE"
      |       "ADVANCE"
      |       "CONCLUDE"
      |       "TERMINATE"
```

"PAUSE" means that the object will proceed with the imperative following the PAUSE-imperative. "ADVANCE" may only be used within a postponement or resumption clause and means that the imperative containing the clause shall be completed immediately. "CONCLUDE" means that the current activity shall be concluded. "TERMINATE" means that the object shall terminate.

```
(22)  Event-Imperative =
              Reference-Name ":-" "NEW" Class-Title
      |       "INTERRUPT" Object-Expression
              "BY" Task-Procedure-Title
```

The effect of NEW is that a new object in class Class-Title is generated. The value of Reference-Name becomes that object. The effect of "INTERRUPT" is that the object denoted by Object-Expression is interrupted. Object-Expression will not be specified, it may e. g. be a reference name.

```
(23)  Composite-Imperative =
              Compound-Imperative
      |       Conditional-Imperative
      |       Repetition-Imperative
```

```
(24)  Compound-Imperative =
              "(*" Imperative {";" Imperative} "*)"
```

```
(25)  Conditional-Imperative =
            "IF" Boolean-Expression
            "THEN" Compound-Imperative
            ["ELSE" Compound-Imperative]

(26)  Repetition-Imperative =
            "REPEAT" Compound-Imperative
      |     "WHILE" Boolean-Expression
            "REPEAT" Compound-Imperative
      |     "REPEAT" Compound-Imperative
            "UNTIL" Boolean-Expression

(27)  Task-Imperative
            Task-Description
      |     "EXECUTE" Task-Procedure-Title
```

The first case generates and executes a singular task (inner block). The second case generates and executes a category defined task (a procedure call).

Chapter 3

PETRI NETS

The theory of Petri nets was founded by Carl Adam Petri in his thesis [Petri 62]. Petri nets have received widespread use as models for systems containing asynchronous concurrent actions.

The basic concepts of the theory are introduced in [Petri 73], [Petri 75], [Petri 76] and [Holt & Commoner 70]. [Peterson 77] contains a survey and an elaborate bibliography of much of the existing work on the subject (see also [Genrich & Thiagarajan 78] which is a reply to Peterson's paper from two of Petri's nearest co-workers).

This chapter is divided into four sections. Section 1 introduces Petri nets. The definitions are illustrated by two simple examples.

Section 2 and 3 define the different extensions to Petri nets, which are used in the Delta semantics defined in chapter 4.

Petri nets may be infinite. In section 4 our field of interest is restricted to finite markings and finite firings.

This chapter is essentially identical to the first five sections in [Jensen 78].

That paper moreover contains a definition of hyper Petri nets, where the rules for concession and firing are defined by attaching a set of functions to each transition. The modelling powers of Petri nets, extended Petri nets and hyper Petri nets are defined, compared and found identical.

3.1 Petri Nets

A Petri net is a 4-tuple pn = (S, T, PRE, POST) where

1. S, T are sets (possibly infinite)
2. PRE, POST are relations on SxT (possibly infinite)
3. $S \cup T \neq \emptyset$
4. $S \cap T = \emptyset$
5. $PRE \cap POST = \emptyset$
6. range $(PRE \cup POST) = T$

Elements of S are called places and elements of T transitions.
A marking of a Petri net, pn, is a function

$m: S \longrightarrow \{0,1\}$

A place s is marked in m iff m (s) = 1, s is unmarked iff m (s) = 0.

Petri nets can be interpreted in a vast number of different ways (see [Petri 75]). Basic for them all is that places represent conditions, which may hold or not, and transitions represent events (actions), which may occur.

When a certain specified combination of conditions is satisfied (places marked) a given event is enabled (has concession). Then the event may occur (transition fires), but it is not obliged to do so. If the event occurs it changes some of the conditions (a new marking is derived).

Different transitions may fire concurrently if they are independent (has different conditions).

Some papers on Petri nets (e.g. [Peterson 77]) allow places to contain multiple markings. This corresponds to a function

$$m: S \longrightarrow \mathbb{N}_o$$

where INo is the set of all nonnegative integers.

Other papers exclude isolated places, i.e. they require that

$$\text{domain } (PRE \cup POST) = S$$

In this paper we will only use binary markings ($S \longrightarrow \{0, 1\}$) and we will allow isolated places.

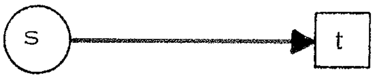
A Petri snapshot is a pair ps = (pn, m) where pn = (S, T, PRE, POST) is a Petri net and m a marking of pn.

A Petri snapshot can be represented graphically as a directed graph with two different kinds of nodes:

○ for places

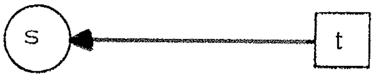
□ for transitions

There is an arc from place, s, to transition, t:



iff (s, t) $\in$ PRE.

There is an arc from t to s:



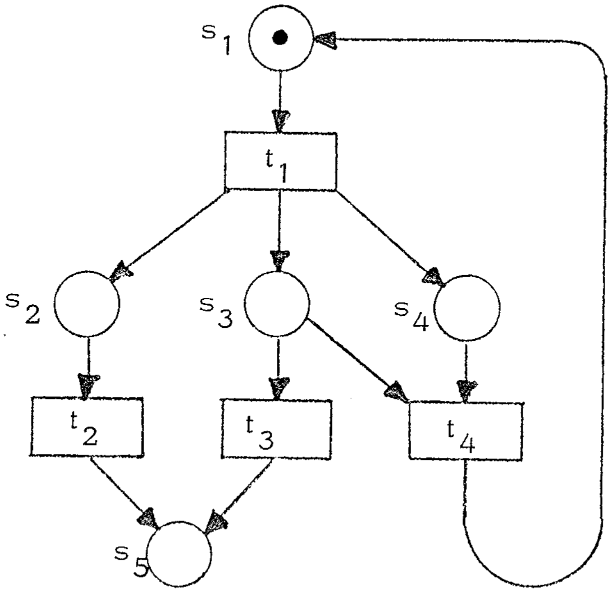
iff (s, t) $\in$ POST

The marking is represented by a token (dot) in the places which are marked.

Example 1.

The graph

[3.1]



represents a Petri snapshot, ps_1 = (pn, m_1), consisting of a Petri net pn = (S, T, PRE, POST) where

$S = \{s_1, s_2, s_3, s_4, s_5\}$

$T = \{t_1, t_2, t_3, t_4\}$

$PRE = \{(s_1, t_1), (s_2, t_2), (s_3, t_3), (s_3, t_4), (s_4, t_4)\}$

$POST = \{(s_1, t_4), (s_2, t_1), (s_3, t_1), s_4, t_1), (s_5, t_2), (s_5, t_3)\}$

and a marking m_1, where

$m_1 (s_1) = 1$

$m_1 (s_2) = m_1(s_3) = m_1 (s_4) = m_1 (s_5) = 0$

□

Let s be a place and t a transition.

s is a precondition for t $\iff s \in \mathrm{PRE}_t \iff (s,t) \in \mathrm{PRE}$

s is a postcondition for t $\iff s \in \mathrm{POST}_t \iff (s,t) \in \mathrm{POST}$

s is a condition for t $\iff s \in \mathrm{COND}_t \iff (s,t) \in \mathrm{COND}$

where $\mathrm{COND} = \mathrm{PRE} \cup \mathrm{POST}$

PRE_t, POST_t and COND_t are unary relations on S. COND is a binary relation on SxT.

The definition of precondition, postcondition and condition can easily be generalized to cover sets of transitions instead of single transitions:

$$s \in \mathrm{PRE}_{T_1} \iff \exists t \in T_1\, [s \in \mathrm{PRE}_t]$$
$$s \in \mathrm{POST}_{T_1} \iff \exists t \in T_1\, [s \in \mathrm{POST}_t]$$
$$s \in \mathrm{COND}_{T_1} \iff \exists t \in T_1\, [s \in \mathrm{COND}_t]$$

A transition, t, has concession in a marking, m, iff all preconditions are marked and all postconditions are unmarked:

$$\forall s \in S \quad \begin{bmatrix} (s,t) \in \mathrm{PRE} & \Rightarrow & m(s) = 1 \\ (s,t) \in \mathrm{POST} & \Rightarrow & m(s) = 0 \end{bmatrix}$$

Two transitions are independent iff their conditions are disjoint.

A set of transitions, T_1, has concession in a marking m iff

(i) Each transition in T_1 has concession in m

(ii) The transitions in T_1 are pairwise independent

(iii) $T_1 \neq \emptyset$

If T_1 has concession in m then T_1 may fire. If T_1 fires then m ceases to exist and is replaced by a new marking, m', derived from m by unmarking all T_1's preconditions and marking all T_1's postconditions.

The rules defining concession and firing can be summarized as follows:

$$m \xrightarrow[pn]{T_1} m'$$

$$\Updownarrow$$

$$\begin{cases} \forall s \in S \quad \begin{bmatrix} s \in PRE_{T_1} & \Rightarrow m(s) = 1 \wedge m'(s) = 0 \\ s \in POST_{T_1} & \Rightarrow m(s) = 0 \wedge m'(s) = 1 \\ s \notin COND_{T_1} & \Rightarrow m(s) = m'(s) \end{bmatrix} \\ \forall t_1, t_2 \in T_1 \; [t_1 \neq t_2 \quad \Rightarrow COND_{t_1} \cap COND_{t_2} = \emptyset] \\ T_1 \neq \emptyset \; . \end{cases}$$

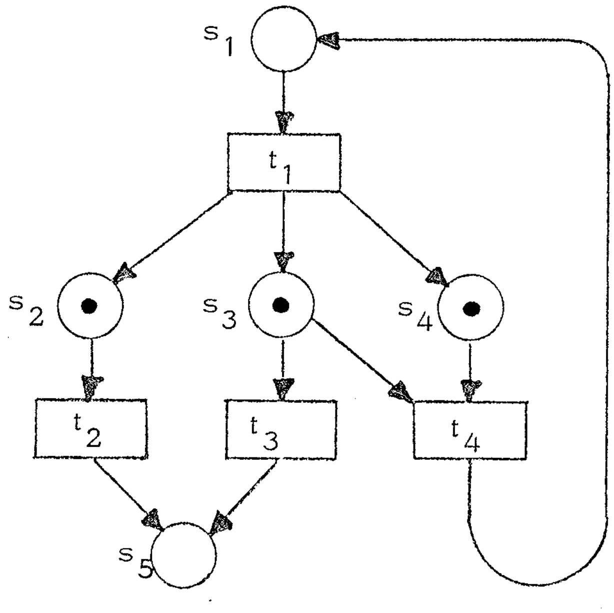
m' is <u>directy reachable</u> from m ($m \xrightarrow[pn]{} m'$) iff there exists a set of transitions T_1 such that

$$m \xrightarrow[pn]{T_1} m'$$

<u>Example 1. (continued)</u>

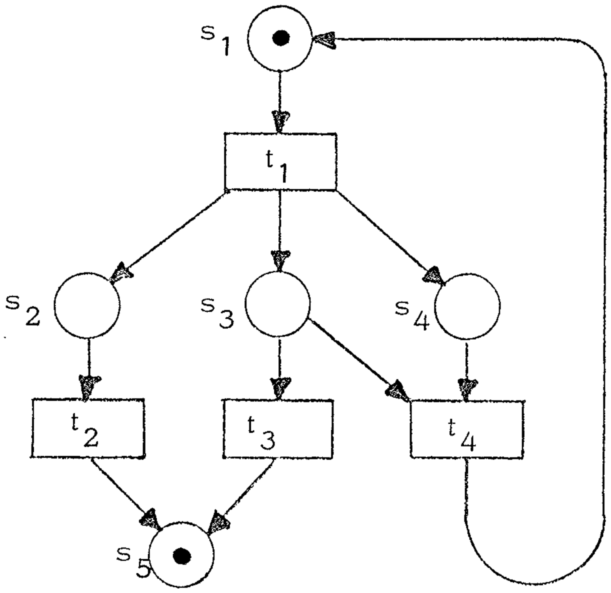
Transition $t_1$has concession in marking m_1 (see [3.1]). No other transition has concession in m_1. If t_1 fires, m_1 is replaced by a new marking m_2:

[3.2]



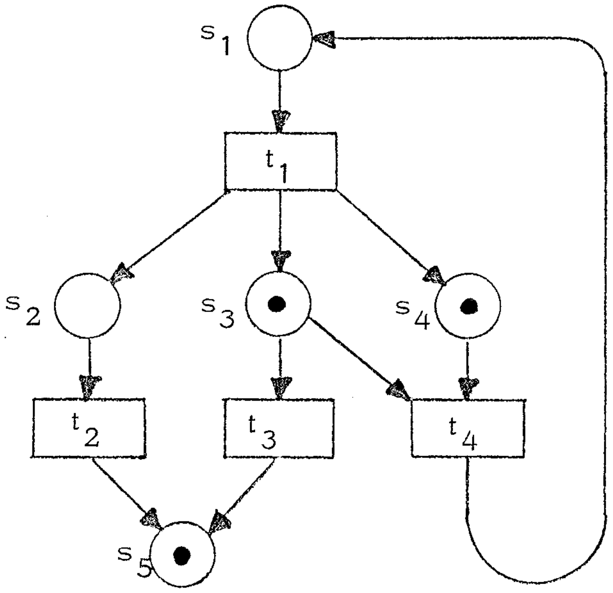
The only independent set with more than one transition is $T_{24} = \{t_2, t_4\}$
In m_2 (see [3.2]) T_{24} has concession.
If it fires m_2 is replaced by m_3:

[3.3]

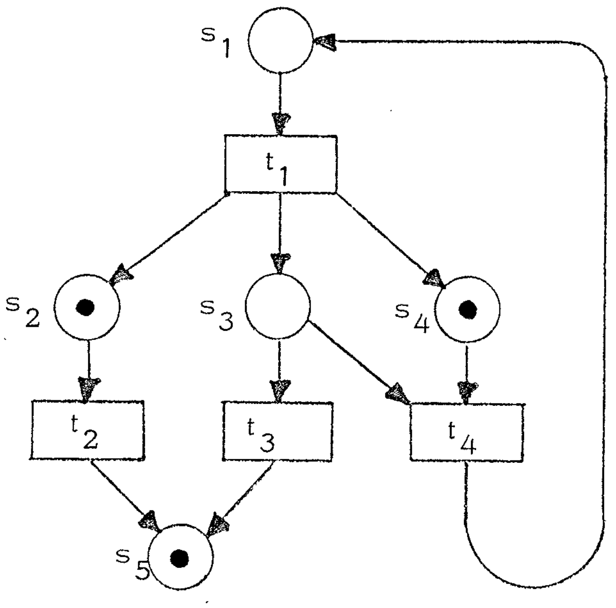


Beyond T_{24} three different transitions t_2, t_3 and t_4 each has concession in m_2 and may fire thereby replacing m_2 by m_4, m_5 and m_6 respectively:

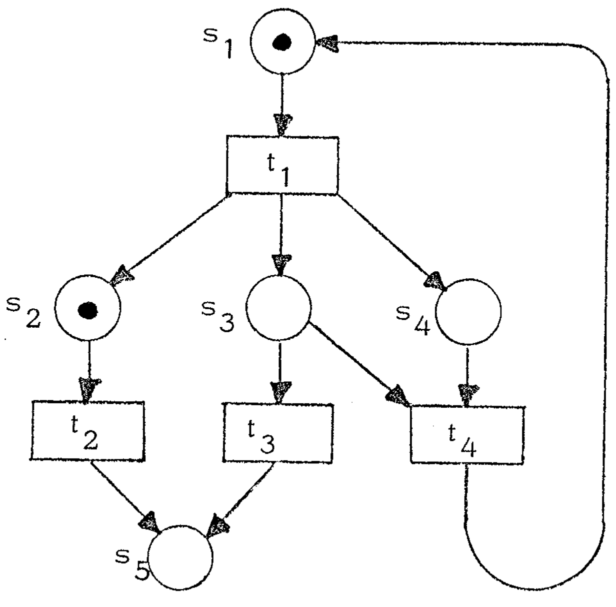
[3.4]



[3.5]



[3.6]



□

Let $T_1 \; T_2 \; \ldots . \; T_n$ be a sequence of nonempty subsets of transitions in a Petri net, pn, with markings m and m'.

$$m \xrightarrow[pn]{T_1 T_2 \ldots\ldots T_n} m'$$

$\Updownarrow$

There exists a sequence of markings for pn $m_o, m_1, \ldots\ldots m_n$ with $n \geq 0$ such that

(i) $m_o = m$

(ii) $m_{i-1} \xrightarrow[pn]{T_i} m_i \qquad (1 \leq i \leq n)$

(iii) $m_n = m'$

m' is reachable from m $(m \xrightarrow[pn]{}{}^{*} m')$ iff there exists a sequence $T_1 \; T_2 \; \ldots\ldots \; T_n$ such that

$$m \xrightarrow[pn]{T_1 \; T_2 \; \ldots \; T_n} m'$$

This defines the binary relation $\xrightarrow[pn]{}{}^{*}$ as the reflexive and transitive closure of $\xrightarrow[pn]{}$.

PN is the set of all Petri nets and PS is the set of all Petri snapshots.

The set of binary relations $\{ \xrightarrow[pn]{} \mid pn \in PN \}$ induces a binary relation on PSxPS:

$$(pn_1, m_1) \xrightarrow[P]{} (pn_2, m_2)$$

$\Updownarrow$

$$pn_1 = pn_2 \wedge m_1 \xrightarrow[pn_1]{} m_2$$

Analogously " $\xrightarrow[P]{}{}^{*}$ " is induced as a binary relation from

$$\{ \xrightarrow[pn]{}{}^{*} \mid pn \in PN \}$$

For each Petri snapshot, ps $\in$ PS, we are interested in the set of all Petri snapshots rechable from ps.

Thus we define a function

$$\mathcal{R}_p : \quad PS \longrightarrow \wp(PS)$$

by

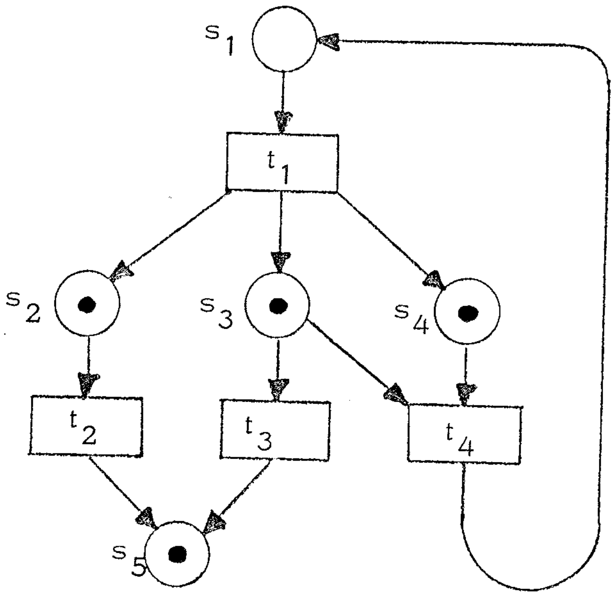
$$\mathcal{R}_p(ps) = \{ps' \in PS \mid ps \xrightarrow[P]{}{}^{*} ps'\}$$

($\wp$ denotes powersets).

Example 1. (continued)

In m_3 only t_1 has concession. If it fires we get a new marking, m_7:

[3.7]



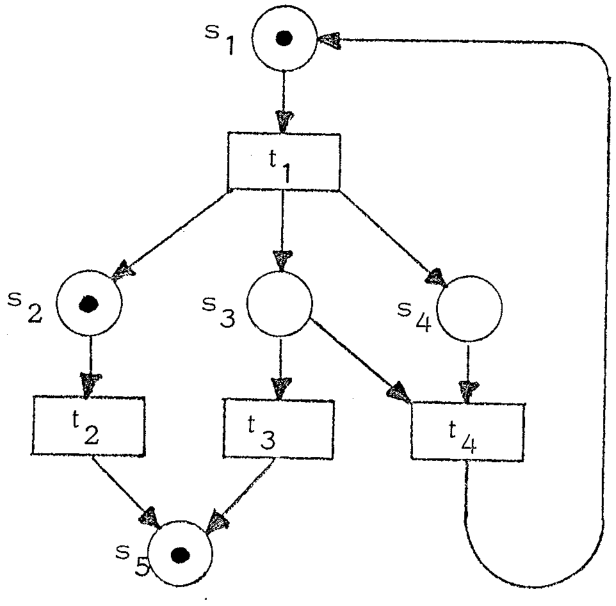
In m_4 only t_4 has concession. If it fires m_4 is replaced by m_3.

In m_5 no transition has concession.

In m_6 only t_2 has concession. If it fires we again get m_3.

In m_7 only t_4 has concession. If it fires we get a new marking, m_8:

[3.8]



In m_8 none of the four transitions have concession.

From the discussion above it follows that

$$\mathcal{R}_p \; (ps_1) = \{ \; ps_i \mid 1 \le i \le 8 \; \}$$

where ps_i is the Petri snapshot consisting of the Petri net pn and the marking m_i.

□

Let pn = (S, T, PRE, POST) and pn' =(S', T', PRE', POST') be Petri nets. pn is a subnet of pn' iff

(i) $S \subseteq S'$
(ii) $T \subseteq T'$
(iii) $PRE = PRE' \cap S \times T$
(iv) $POST = POST' \cap S \times T$

Let ps = (pn, m) and ps' = (pn,'m') be Petri snapshots with pn and pn' as defined above. ps is a subsnapshot of ps' iff

(i) pn is a subnet of pn'
(ii) $\forall s \in S\ [m(s) = m'(s)]$

In the following we will not always make a sharp distinction between a Petri snapshot ps = (pn, m) and the corresponding marking m. For instance we will say that a place, s, is marked in ps iff it is marked in m.

We will also use notation such as ps(s), where we really mean m(s) and write

$$ps \xrightarrow[P]{T_1\ T_2\ \dots . T_n} ps'$$

with ps' = (pn, m')
where we mean

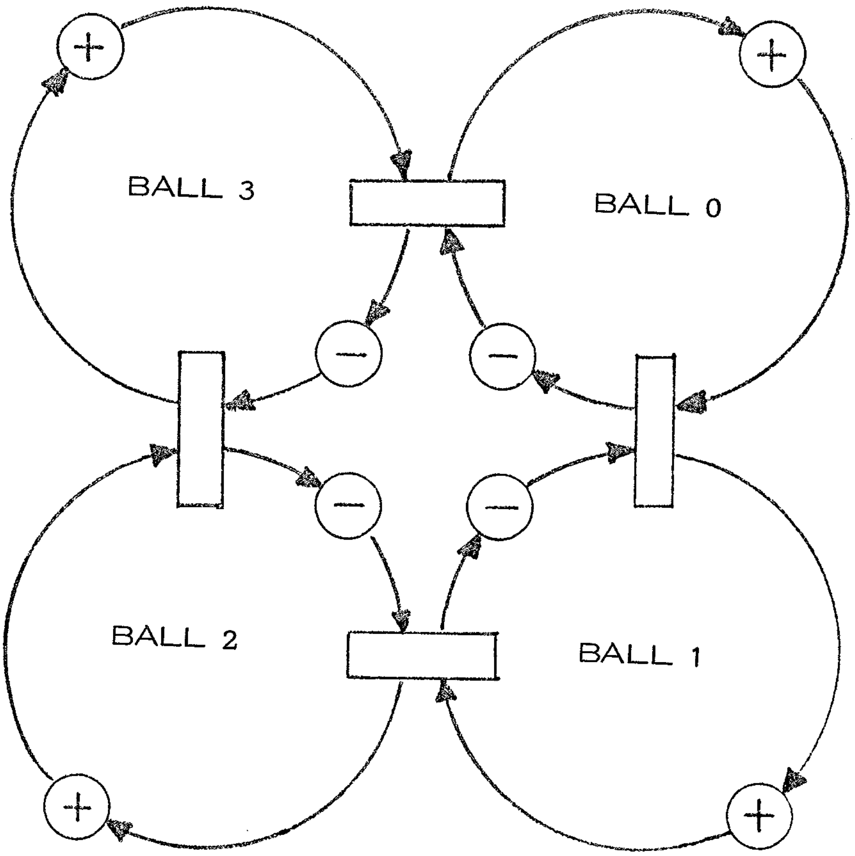
$$m \xrightarrow[pn]{T_1, T_2, \dots . T_n} m'$$

Example 2.

In example 1 the set of possible firings was finite.

As an example of a Petri net, which may perform an infinite sequence of firings we introduce the following net, which represents the system,described in chapter 2, consisting of four elastic balls moving in a circular orbit without loss of energy (see page 9).

[3.9]



Each ball moves either in positive direction ("+" marked) or in negative direction ("-" marked).

Two neighbouring balls moving against each other may collide thereby exchanging their direction of movement. This is represented by a firing of one of the four transitions.

Two neighbouring balls moving in the same direction may also collide. This changes their speed, but not their direction of movement. Thus this kind of collisions is not represented by the firing of a transition.

□

3.2 Extended Petri nets

In Petri nets, as they were defined in the last section, it is possible to add and remove tokens. It is however not possible to test, in a single transition, whether a place is marked or not without destroying the current marking.

Nondestructive tests must be performed using two transitions. The first performs the test while the second reestablishes the marking. It is necessary to make sure that the second transition actually does fire before other transitions "uses" the place.

When defining a semantics for Delta using Petri nets it turned out, that we often needed such nondestructive tests.

Hence we now introduce two new relations and define an extended Petri net to be a 6-tuple epn = (S, T, PRE, POST, TESTM, TESTU) where

1. S, T are sets (possibly infinite)
2. PRE, POST, TESTM, TESTU are relations on SxT (possibly infinite)
3. $S \cup T \neq \emptyset$
4. $S \cap T = \emptyset$
5. PRE, POST, TESTM and TESTU are mutually disjoint
6. range $(PRE \cup POST \cup TESTM \cup TESTU) = T$

As before S is a set of places and T a set of transitions.

Let EPN be the set of all extended Petri nets and define marking, marked, unmarked, extended snapshot ($es \in ES$), precondition, postcondition, testmarked condition, testunmarked condition, condition, independent, directly reachable ($\xrightarrow[E]{}$), reachable ($\xrightarrow[E]{}^*$), $\mathbb{R}_E$: $ES \longrightarrow \wp(ES)$, subnet, and subsnapshot in analogy with the corresponding definitions for Petri nets.

A transition, t, has concession in a marking, m, iff all preconditions and testmarked conditions are marked and all postconditions and testunmarked conditions are unmarked:

$$\forall s \in S \begin{bmatrix} (s,t) \in PRE \cup TESTM & \Rightarrow & m(s) = 1 \\ (s,t) \in POST \cup TESTU & \Rightarrow & m(s) = 0 \end{bmatrix}$$

Let m and m' be markings for an extended Petri net epn = (S, T, PRE, POST, TESTM, TESTU).

$$m \xrightarrow[\text{epn}]{T_1} m'$$

$$\Updownarrow$$

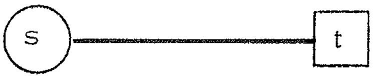
$$\begin{cases} \forall s \in S \left[\begin{array}{lll} s \in PRE_{T_1} & \Rightarrow & m(s) = 1 \ \wedge \ m'(s) = 0 \\ s \in POST_{T_1} & \Rightarrow & m(s) = 0 \ \wedge \ m'(s) = 1 \\ s \in TESTM_{T_1} & \Rightarrow & m(s) = 1 \ \wedge \ m'(s) = 1 \\ s \in TESTU_{T_1} & \Rightarrow & m(s) = 0 \ \wedge \ m'(s) = 0 \\ s \notin COND_{T_1} & \Rightarrow & m(s) = m'(s) \end{array} \right] \\ \\ \forall t_1, t_2 \in T_1 \ [t_1 \neq t_2 \Rightarrow COND_{t_1} \cap COND_{t_2} = \emptyset] \\ \\ T_1 \neq \emptyset \end{cases}$$

It should be remarked here, that similar extensions to Petri nets have been defined by several authors (see [Peterson 77] page 246).

Some applications of Petri nets define a place to be a sidecondition for a transition, t, if it is both a precondition and a postcondition for t. Then the rules for concession and firing are defined such that sideconditions are treated in the same manner as we treat testmarked conditions.

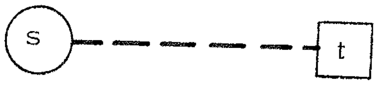
Extended snapshots are represented graphically in the same way as Petri snapshots with the following additions:

There is an arc



iff (s, t) ∈ TESTM

There is an arc



iff (s, t) ∈ TESTU

From the definitions it is obvious that everything, which can be modelled using Petri nets can also be modelled using extended Petri nets.

In [Jensen 78] it is shown that the converse is also true: For each extended Petri net epn a corresponding Petri net pn can be constructed such that all information about epn and its markings can be derived using information only about pn and its markings.

However it should be stressed that extended Petri nets is a valuable formalism in itself. Extended Petri nets can be defined, interpreted, understood, and manipulated as an independent selfcontained formalism.

3.3 Further extensions of Petri nets

In the last two sections we have defined Petri nets (with preconditions and postconditions) and extended Petri nets (which in addition have testmarked and testunmarked conditions).

When defining a semantics for Delta using extended Petri nets it turned out that there were situations in which we wanted to use transitions with more complicated kinds of conditions.

Instead of merely introducing extensions ad hoc we will now make a systematic investigation into the possible kinds of conditions which can be defined in nets with binary markings ($S \longrightarrow \{0, 1\}$).

The rules defined for extended Petri nets regarding concession and firing can be summarized as follows:

$(s, t) \in \text{PRE}$

$$\Downarrow\begin{cases}\text{s unmarked} & \Rightarrow \quad \text{t cannot fire}\\ \text{s marked} & \Rightarrow \quad \text{If t fires, s becomes unmarked}\end{cases}$$

$(s, t) \in \text{POST}$

$$\Downarrow\begin{cases}\text{s unmarked} & \Rightarrow \quad \text{If t fires, s becomes marked}\\ \text{s marked} & \Rightarrow \quad \text{t cannot fire}\end{cases}$$

$(s, t) \in \text{TESTM}$

$$\Downarrow\begin{cases}\text{s unmarked} & \Rightarrow \quad \text{t cannot fire}\\ \text{s marked} & \Rightarrow \quad \text{If t fires, s remains marked}\end{cases}$$

$(s, t) \in \text{TESTU}$

$$\Downarrow\begin{cases}\text{s unmarked} & \Rightarrow \quad \text{If t fires, s remains unmarked}\\ \text{s marked} & \Rightarrow \quad \text{t cannot fire}\end{cases}$$

This can also be shown in more compact form as follows:

[3.10]

		UNMARKED		
		CANNOT FIRE	REMAINS UNMARKED	BECOMES MARKED
MARKED	CANNOT FIRE		TEST UNMARKED	POST
	REMAINS MARKED	TEST MARKED		
	BECOMES UNMARKED	PRE		

What about the five empty boxes in this table? Have they a meaningful interpretation?

The answer is yes:

[3.11]

		UNMARKED		
		CANNOT FIRE	REMAINS UNMARKED	BECOMES MARKED
MARKED	CANNOT FIRE	BLOCKED	TEST UNMARKED	POST
	REMAINS MARKED	TEST MARKED	DUMMY	SET MARKED
	BECOMES UNMARKED	PRE	SET UNMARKED	CHANGE

BLOCKED: The transition cannot fire whatever the marking of the place is. This corresponds to the situation where the place is both a precondition and a postcondition for the given transition.

DUMMY: The transition can fire whatever the marking of the place is. The marking of the place is not changed by a firing.

CHANGE: The transition can fire whatever the marking of the place is. The marking of the place is changed by a firing.

SETMARKED: The transition can fire whatever the marking of the place is. After a firing the place is marked.

SETUNMARKED: The transition can fire whatever the marking of the place is. After a firing the place is unmarked.

These nine different kinds of conditions exhaust all possible kinds of conditions for deterministic transitions in binary nets.

If however we allow nondeterministic transitions, then the table looks as follows:

[3.12]

		UNMARKED			
		CANNOT FIRE	REMAINS UNMARKED	BECOMES MARKED	UNKNOWN MARKING
MARKED	CANNOT FIRE	BLOCKED	TEST UNMARKED	POST	-
	REMAINS MARKED	TEST MARKED	DUMMY	SET MARKED	INCREASE
	BECOMES UNMARKED	PRE	SET UNMARKED	CHANGE	-
	UNKNOWN MARKING	-	DECREASE	-	RANDOM

INCREASE The transition can fire whatever the marking of the place is. By a firing the marking of the place is increased or unchanged.

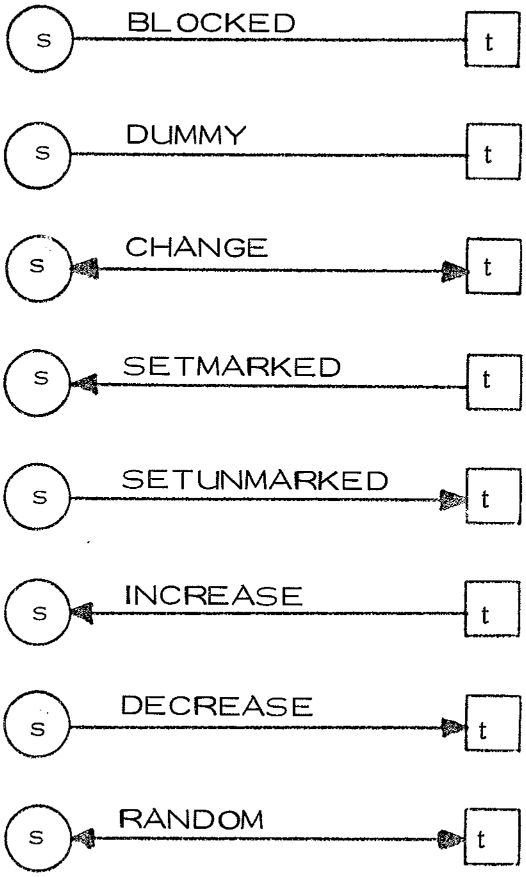
DECREASE : The transition can fire whatever the marking of the place is. By a firing the marking of the place is decreased or unchanged.

RANDOM: The transition can fire whatever the marking of the place is. After a firing nothing is known about the marking of the place.

It is difficult to give a natural interpretation of the four remaining kinds of conditions defined in [3. 12].

Instead of inventing a set of fancy arrows for the graphical representation of the different kinds of conditions in [3. 12] we propose to label the arrows with a text:

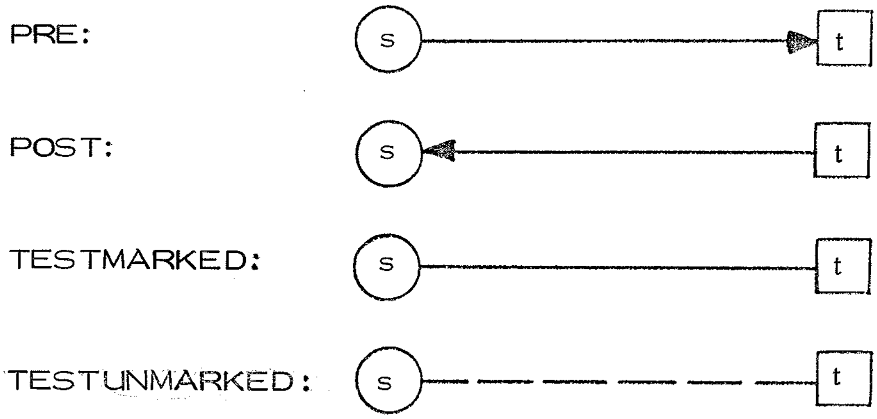
[3. 13]



An arrow-head at the end of a place (transition) indicates that tokens may be added (removed) by a firing.

For the kinds of conditions already defined in extended Petri nets we will use the same graphical notation as hitherto:

[3. 14]

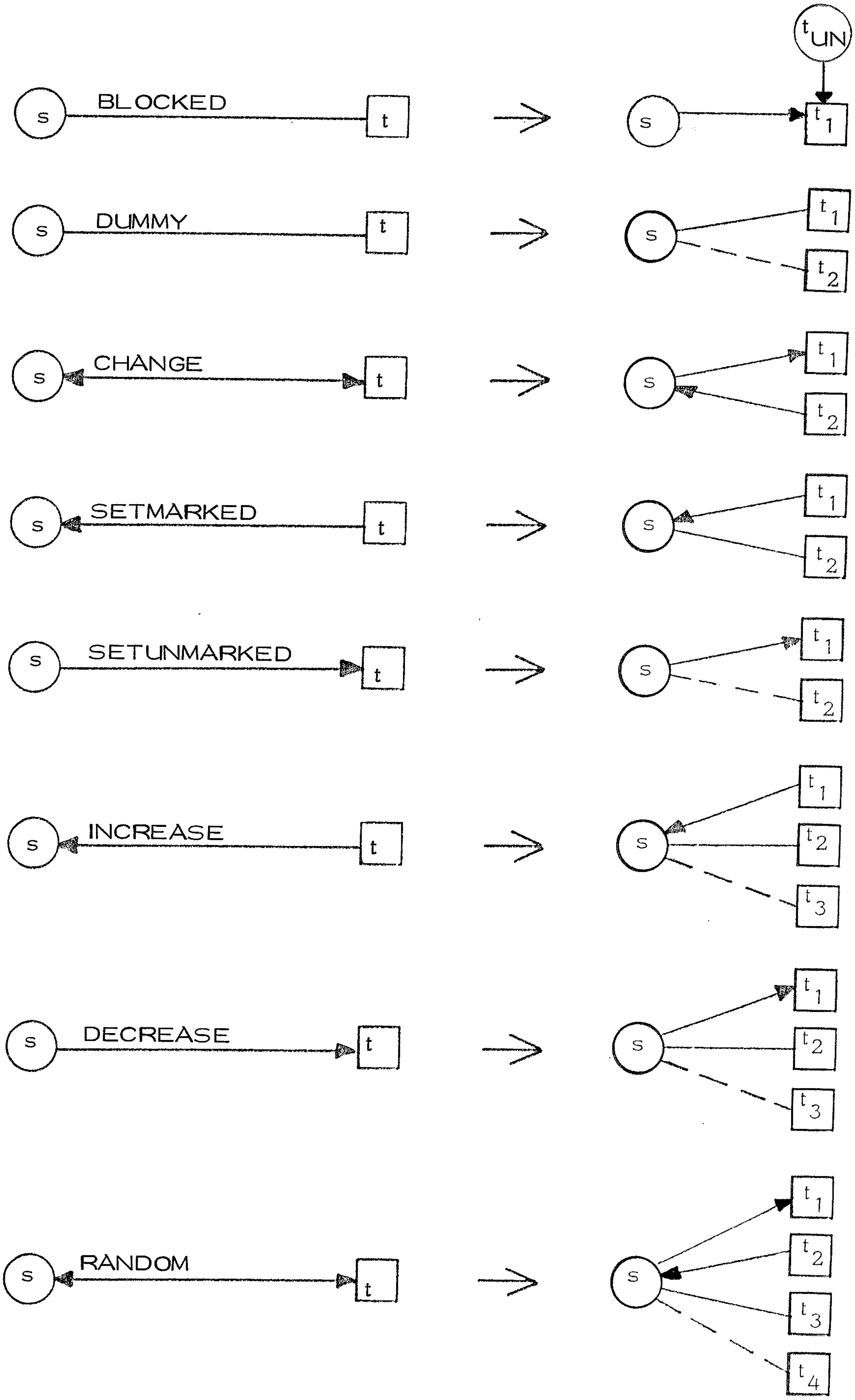


It is of course possible to give a formal definition of " superextended" Petri nets as a many-tuple containing a relation for each kind of conditions defined in [3. 12]. This will not be done.

Instead we merely regard a graph G containing arrows from [3. 13] and [3. 14] as a graphical abbreviation for the same extended Petri net as the graph, which

(i) Only contains arrows from [3. 14].

(ii) Is obtained from G by repeated application of the following set of substitutions.

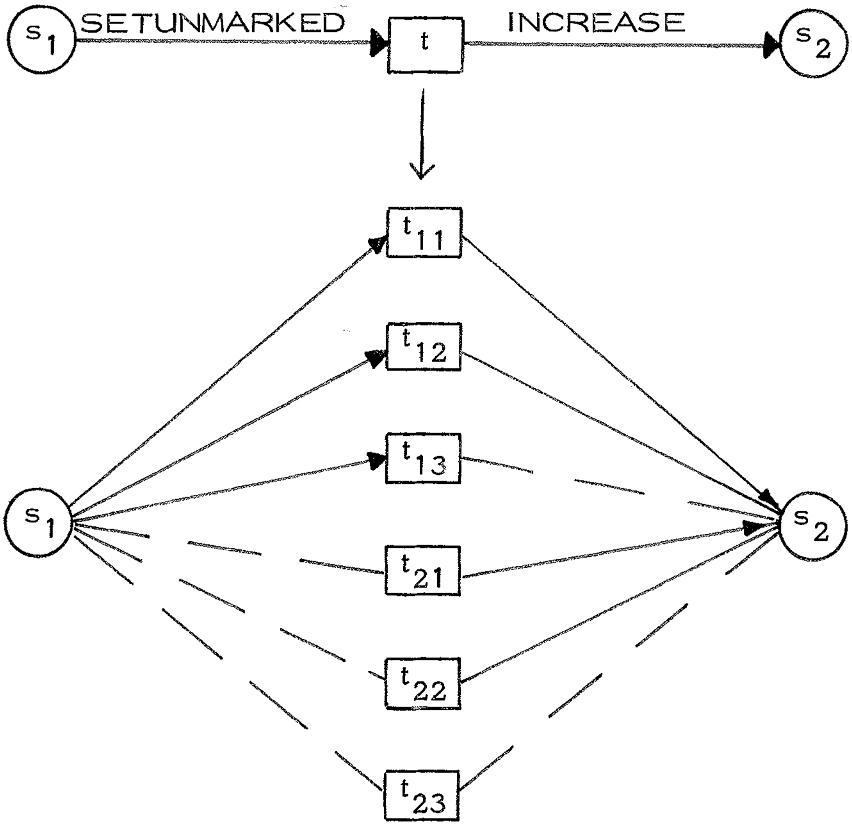

s
BLOCKED
t
t_{UN}
s
t_1
s
DUMMY
t
s
t_1
t_2
s
CHANGE
t
s
t_1
t_2
s
SETMARKED
t
s
t_1
t_2
s
SETUNMARKED
t
s
t_1
t_2
s
INCREASE
t
s
t_1
t_2
t_3
s
DECREASE
t
s
t_1
t_2
t_3
s
RANDOM
t
s
t_1
t_2
t_3
t_4

[3.15]

Each transition, t with a BLOCKED condition get a new unique place, t_{UN}, as precondition. This place will always unmarked.

When a transition t, has exatly one condition s, of a kind from [3. 13] (except BLOCKED) it is split into two, three or four transitions (se [3. 15]) which each inherits the precondition, postconditions, testmarked and testunmarked conditions of t. In addition each of the new transitions has s as condition of the kind shown in [3. 16].

When a transition, t, has more than one condition of a kind from [3. 13]. (except BLOCKED) it is split for each such condition, e. g:



[3. 16]

In particular this means that a transition with an infinite number of conditions may be split into an infinite number of transitions.

3.4 Finite markings and finite firings

We have made no restrictions at all concerning the size of our nets.

In particular it is possible that

(i) An infinite number of conditions may be marked concurrently.

(ii) An infinite number of transitions may fire concurrently.

However it turns out that most applications of Petri nets do not utilize (i) and (ii) in their full consequence.

Instead it is typical to have a situation, where the nets defined each contains an infinite number of places and transitions but satisfies the following two restrictions:

(i') Only a finite number of conditions may be marked concurrently.

(ii') Only an finite number of transitions may fire concurrently.

Analogous situations are known from many other fields of computer science:

> In an infinite virtual storage at any time only a finite number of bytes have been used.
>
> The possible "routes" during the execution of a WHILE-imperative or a recursive procedure can be described by an infinite tree, but each terminating execution only traverse a finite part of this tree.
>
> In a cellular automata the number of constituent automa is infinite, but at any time only a finite number is outside their neutral state.

In the rest of this paper we only consider markings and firings satisfying (i') and (ii') above. Such markings and firings are said to be finite.

For a given extended snapshot es = (epn, m) the following three restrictions are sufficient but not necessary to assure that all snapshots reachable from es and all firings in such snapshots are finite:

(i) m is finite

(ii) Each firing of a transition in epn adds at most a finite number of tokens to its conditions.

(iii) Only a finite number of transitions in epn can fire when all their conditions are unmarked.

All the extended Petri nets used in our definition of a semantics for Delta satisfy (i), (ii) and (iii) above.

Chapter 4

MINI-DELTA SEMANTICS

In [Delta 75] a semantics for Delta is defined in terms of transitions from one machine state to another in an abstract machine called IDSG (Idealized Delta System Generator). Each transition corresponds to the execution of a language element or to the concurrent execution of a set of language elements.

This kind of semantics is known as an interpretive semantics. For a discussion of different semantic approaches see [Hoare & Lauer 74].

In this chapter a semantics for Mini-Delta is defined in terms of extended Petri nets. Each language element is represented by a rather small extended Petri net. A Delta system described by a Delta-description, D, is then represented by an extended Petri net. This net is composed by a syntax directed translation, from small subnets representing the various language elements contained in D.

This approach can be viewed as an interpretive semantics, too. In this case the rules defining the behaviour of the abstract machine are the rules defining which transitions have concession and the rules defining how a new marking is derived when transitions fire.

In interpretive semantics it is common practice to separate each machine state into two disjoint parts:

- a memory state which defines the current value of all data items explicitly defined by the user (system reporter). In [Delta 75] these items are known as specified attributes.

- a control state which defines the current progress of execution. This can be viewed as the current values of a set of data items implicitly defined by the user (e. g. program pointers). In [Delta 75] these items are known as structural attributes.

The semantics defined in this chapter focusses upon changes in control states. It gives a detailed definition of the possible orders in which different actions can be performed.

Sequential execution correspondends to total orders, while concurrent execution corresponds to partial orders.

Concurrent execution of a set of action sequences implies that different actions can be performed at the same time. This is something much stronger than interleaved execution, where the different actions are performed in an unpredictable order but one by one.

The difference between concurrent and interleaved execution is analogous to the difference between a situation, where it is meaningless to ask whether $a < b$ or $b < a$ simply because a and b are unordered, and a situation, where we know that a and b are ordered, but not whether $a < b$ or $b < a$.

The two semantics ([Delta 75] and this chapter) are equivalent in the sense that for each Delta system they define the same set of possible partial orders.

The semantics defined in this chapter defines the effect of the changes which should be performed upon memory states, but it says nothing about how these changes are performed.

This chapter is divided into three sections. In section 1 three different phases for Delta-objects are defined. These merely describe whether an object possesses attributes and whether it performs actions. The different objects in a Delta system shifts between the three phases. In section 2 most of these phase-shifts are synchronized. Section 3 focusses on imperatives, executional modes, event registration and model-time. The remaining phase-shifts are synchronized.

All names for places and transitions will be written in capital letters.

4.1 Delta - Objects

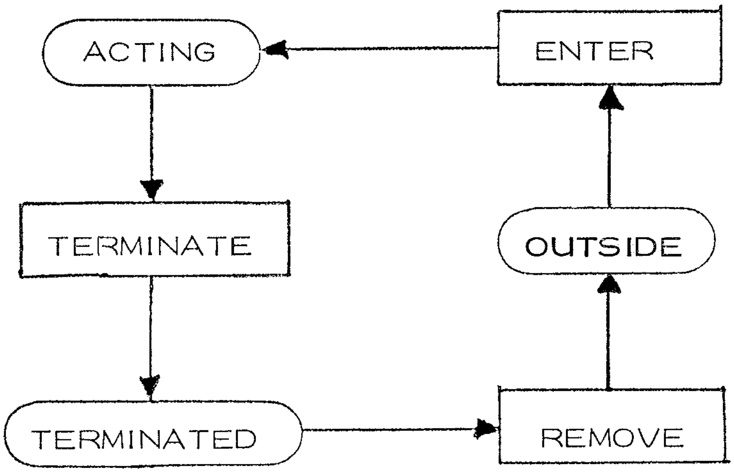
Delta-objects have two main properties:

(i) They may perform actions
(ii) They may possess attributes (data items or pattern declarations).

Each object is in one of three different phases. Initially it is OUTSIDE the considered system. Then it may ENTER the system either by generation of the system object or by execution of a NEW-imperative. Now the object is ACTING. It performs actions and it possesses attributes. An ACTING object may TERMINATE either explicitly by executing a TERMINATE imperative or implicitly because it has no more actions to perform or because its encloser TERMINATEs. Now the object is TERMINATED. It does not perform any actions, but it still possesses attributes. Finally the object may be REMOVEd from the considered system. Then it again becomes OUTSIDE.

By the discussion above it is straightforward to represent each Delta object by a subnet of the form:

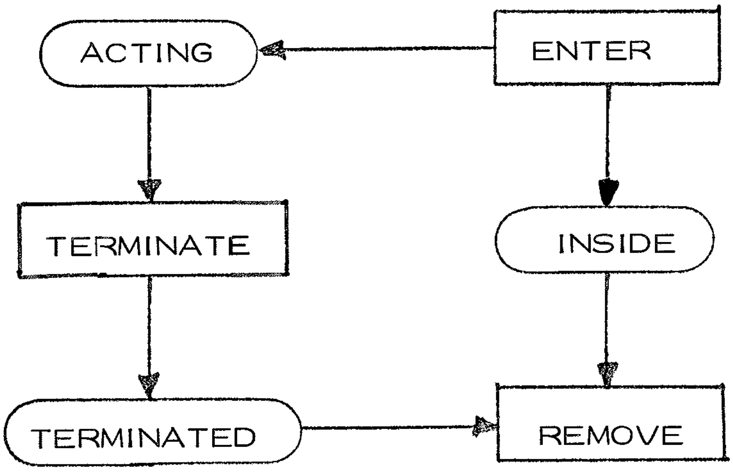
[4.1]



where initially only OUTSIDE is marked.

This will however not be done. Instead each Delta object is represented by a subnet of the form:

[4.2]



where all places are unmarked initially. INSIDE is the negation of OUTSIDE (i.e. INSIDE marked ⇔ OUTSIDE unmarked).

There are two reasons for choosing [4.2] in favour of [4.1]:

(i) In [4.1] at any time exactly one place is marked. Thus it is tempting to interpret the different firings of ENTER and REMOVE in a fixed subnet of form [4.1] as representing the <u>same</u> object being ENTERed and REMOVEd from the system. In Delta this is however meaningless. When an object is REMOVEd from the system, there are no means by which it is possible to reENTER the <u>same</u> object into the system.

(ii) For each class an unlimited number of objects may be INSIDE the considered system at the same time. Thus the net representing the entire system must contain for each class an infinite (but countable) number of subnets of form [4.1] or [4.2].
Using [4₀1] with all OUTSIDE places initially marked violates the demand made in chapter 3 to consider only finite markings.

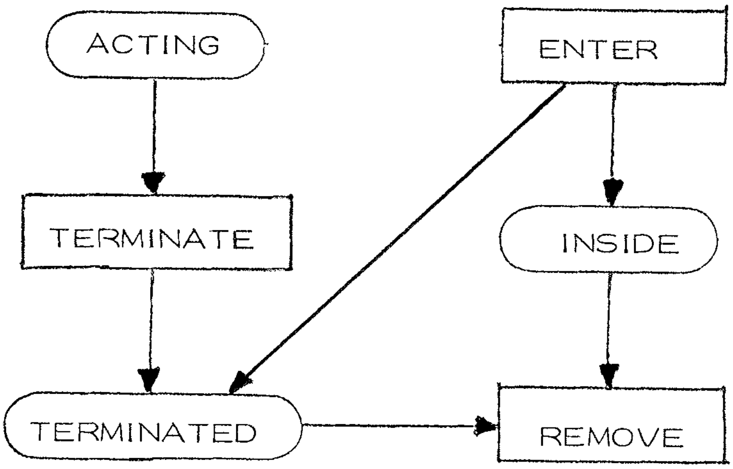
It should be noticed that the reuse of subnets of form [4.2] is analogous to the creation and destruction of activation records in an infinite virtual store:

(i) When an activation record is destroyed the storage used by it can be reused to create a new activation record. This second activation record may be "similar" to the first one but it always represents an execution (e.g. of a block, procedure or object) which is logically distinct from the execution represented by the first activation record.

(ii) At any time only a finite amount of storage is being, or has been, used.

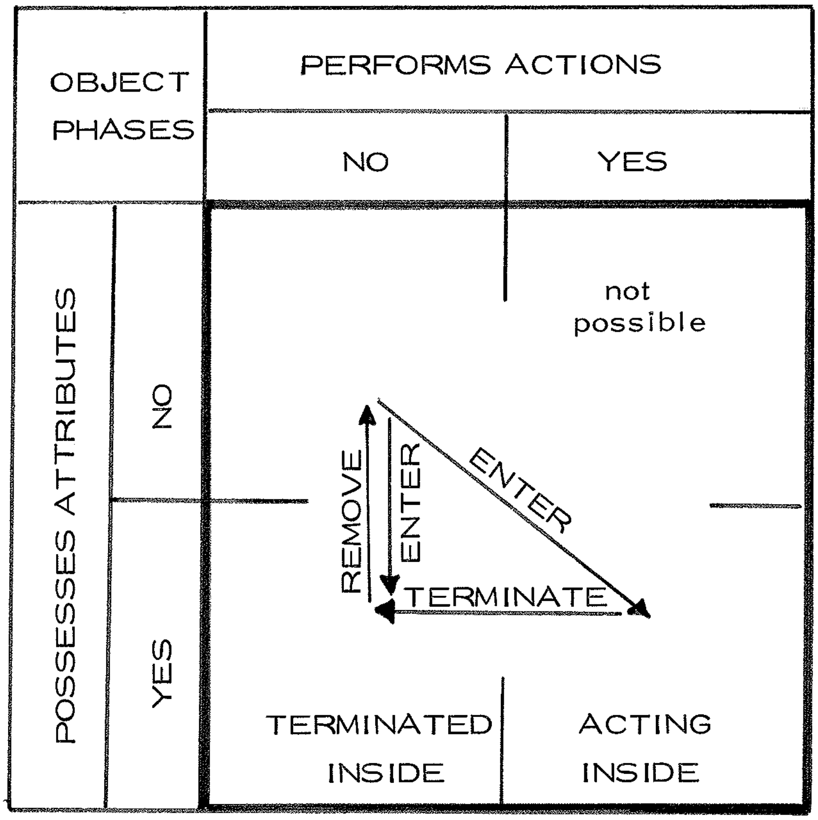
However [4.2] does <u>not</u> describe the behaviour of all Delta-objects. There are objects which never execute actions. When such an object ENTERs the system it immediately becomes TERMINATED. Thus this kind of object is described by:

[4.3]



The phase shifts included in [4.2] and [4.3] can be summarized in the following diagram:

[4. 4]



In the rest of this paper all subnets representing objects will be shown as [4. 2]. For objects without a prime task [4. 2] should be replaced by [4. 3].

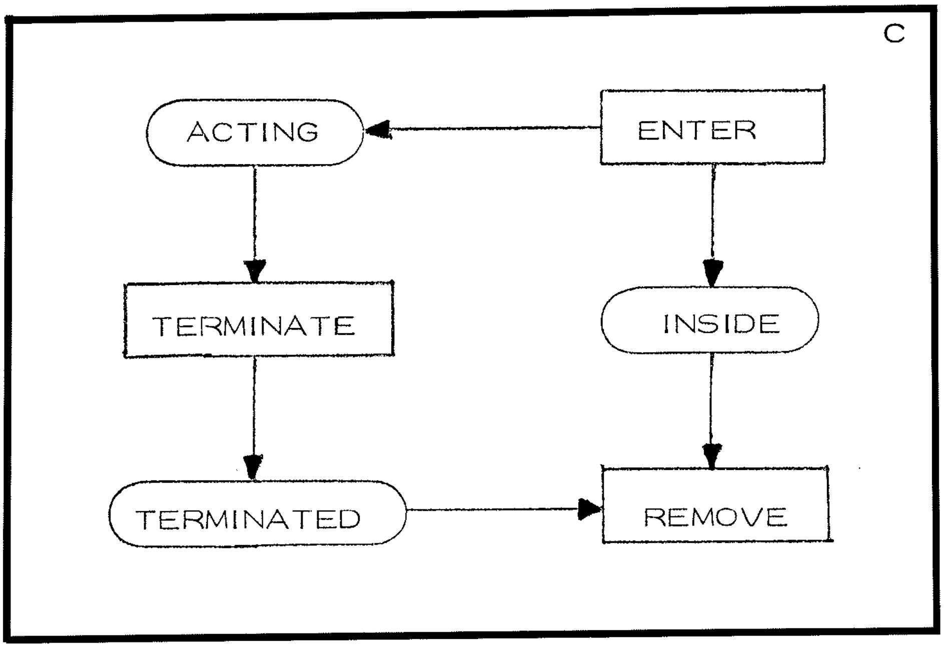
4.2 Synchronization between creation, termination and destruction of objects

In the last section we developed extended Petri nets, [4.2] and [4.3], representing single Delta objects. We now want to stick such smaller nets together to form a bigger net representing an entire Delta system.

UNSYNCHRONIZED NET

Our first attempt is a net containing a subnet of form [4.2] or [4.3] for each object, which may ENTER the system. These subnets are unsynchronized (i.e. unconnected).

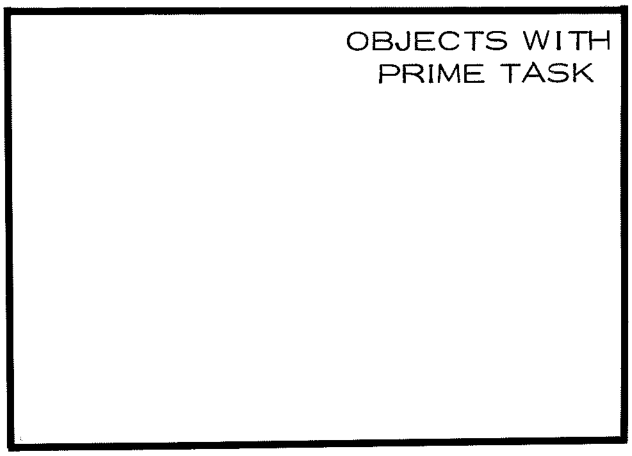
As mentioned earlier, for each class C an unlimited number of objects may be INSIDE the system at the same time. Thus C must be represented by an infinite but countable number of subnets of form [4.2] or [4.3]. This will be denoted by the following notation



[4.5]

This notation will be used when we have a finite or infinite set of identical subnets. The set of objects, tasks or imperatives represented by the set of subnets will be indicated by a text in uppermost right corner, e.g.:

[4.6]



SYNCHRONIZED NET

To represent a Delta system the various subnets representing individual objects must be synchronized. Recalling the definition of "litter" and "content" made in chapter 2 (page 17) we formulate the following synchronization constraints:

SYN1: All objects in a litter ENTER the system together.

SYN2: A litter can only ENTER the system if the primary object's encloser is INSIDE.

[4.7] SYN3: An object which is perceived (can be referenced) directly or indirectly by an ACTING object cannot be REMOVEd.

SYN4: When an object TERMINATEs all objects in its content TERMINATE at the same moment of model-time.

We now define more elaborated extended Petri nets. These nets satisfy SYN1, SYN2, and SYN3 but not SYN4. They are constructed from the unsynchronized nets on page 62 in three steps:

(SYN1') For each litter all ENTER transitions are mapped into a single transition (also called ENTER). Each place being a condition for one of the original ENTER transitions becomes a condition (of the same kind) for the new ENTER transition.

[4.8] (SYN2') For each litter this single ENTER transition is given the INSIDE place for the primary object's encloser as testmarked condition.

(SYN3') Each REMOVE transition gets a new place REMOVABLE as precondition. These places will never be marked. Thus REMOVE transitions never can fire.

(SYN1') and (SYN2') should be easy to comprehend and need no comments.

At a first glance (SYN3') may seem to be an unnecessarily crude method to achieve (SYN3). However, by a closer inspection this turns out not to be the case:

(i) When an object is TERMINATED and not perceived (directly or indirectly) by any ACTING object it cannot in any way influence the future behaviour of the system. Thus from a semantical viewpoint it is completely without interest, whether the object stays in the system or is REMOVEd.

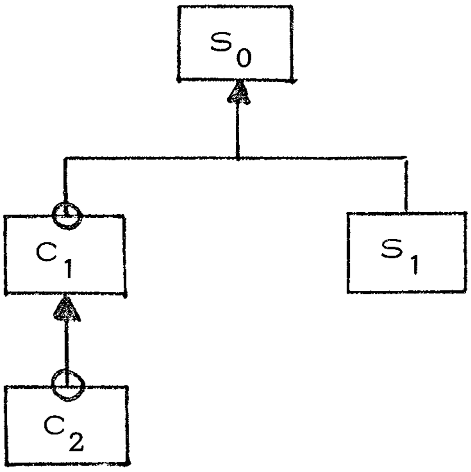
(ii) [Delta 75] states that objects which are TERMINATED and not perceived by any ACTING object may be REMOVEd not that they must be. We think that [Delta 75] at this point has been influenced by considerations about how to implement garbage collection in Simula. As stated above such considerations are irrelevant for the definition of a Delta semantics.

(iii) If someone should insist to model the situation described in [Delta 75] this can simply be done by adding a surrounding net, which marks REMOVABLE when the object is not perceived (directly or indirectly) by any ACTING object. When REMOVABLE and TERMINATED are marked REMOVE may fire but it is not obliged to do so.

EXAMPLE

Using the pictorial notation defined in [Delta 75] the following diagram

[4.9]

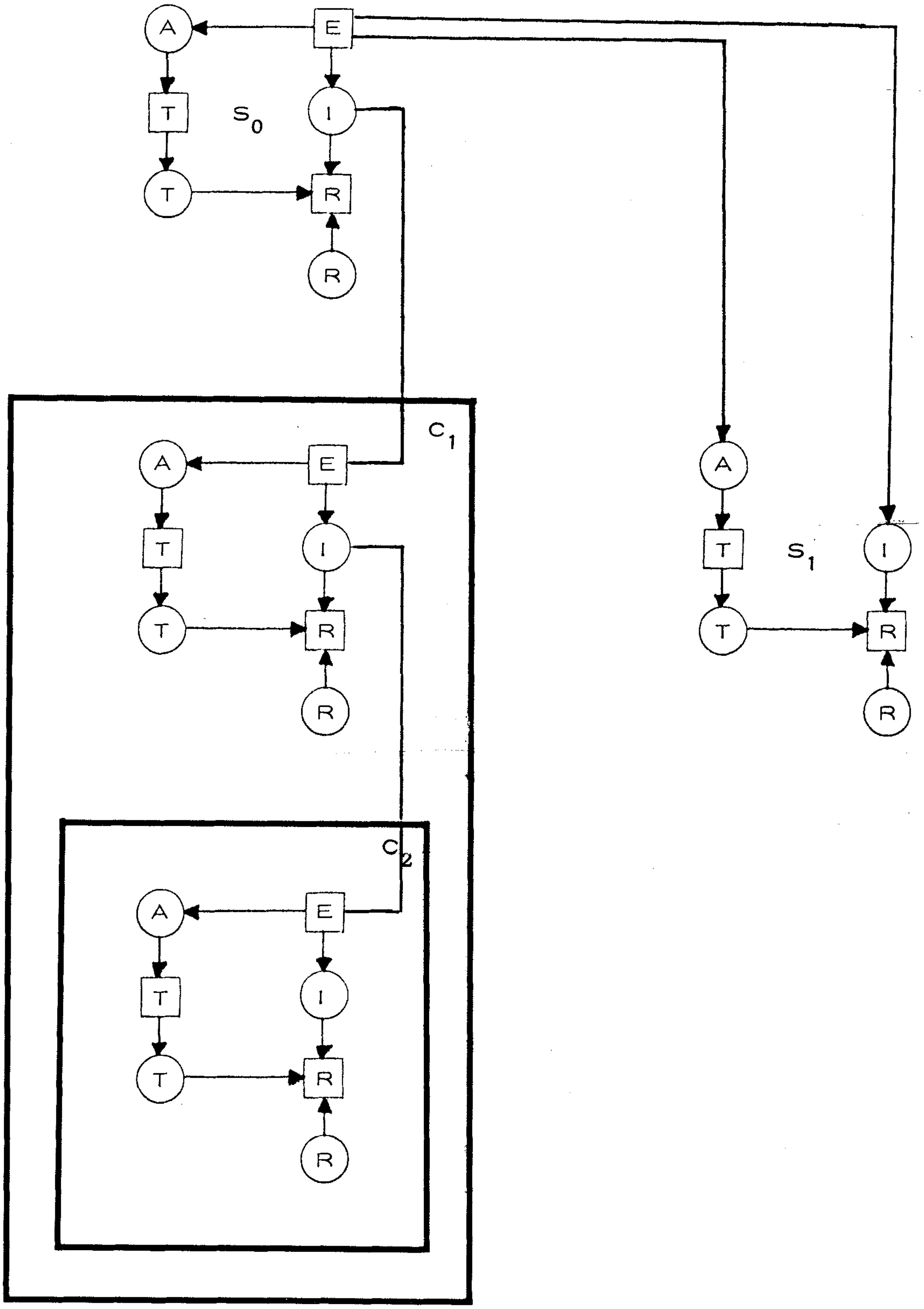


represents a Delta system where

(i) The system object S_0 is encloser for a singular object S_1 and a class of category defined objects C_1.

(ii) Each object in C_1 is encloser for a class of category defined objects C_2.

Using extended Petri nets constructed by [4.8] we get:



[4.10]

The arrow from C_1's ENTER to S_0's INSIDE indicates that each ENTER transition in a C_1-subnet has S_0's INSIDE place as testmarked condition.

Similar remarks apply to all other arrows "crossing boundaries" in this and following graphs.

Initially all places are unmarked. The only transition which may fire is ENTER belonging to S_0.

In [Delta 75, p. 84] it is said that "two objects C' and C" having the same class title A and identical class patterns, but which are directly enclosed by two different objects D' and D", are to be regarded as belonging to two different classes, even if D' and D" belong to the same class".

In [4.10] this is reflected by the nesting of C2-subnets with respect to C1-subnets .

□

4. 3 Synchronization between objects which execute concurrent actions

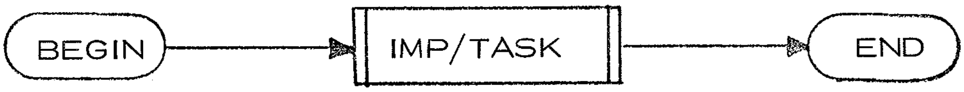
The subnets [4. 2] and [4. 3] defined in section 4. 1 and the 4 synchronizationconstraints formulated in section 4. 2 constitute a rather rudimentary view on the behaviour of Delta objects: Objects may alternate between different phases, but imperatives, attributes, and model-time are not represented.

In this section we will define extended Petri nets, built from nets constructed by [4. 8], by adding subnets which represent imperatives and model-time. These nets satisfy SYN1, SYN2, SYN3, and SYN4 (see page 63).

BASIC FORM OF NETS REPRESENTING IMPERATIVES AND TASKS

All imperatives and tasks will be represented by a net of the following form

[4. 11]



A transition (place) in a given subnet is a <u>border</u> node iff it has a condition (is condition for a transition) outside the subnet.

The notation $\boxed{t}$ indicates that the transition t will be replaced by a more elaborated subnet, where all border nodes are transitions (<u>closed</u> subnet).

Similarly the notation(s) will be used to indicate that the place s will be replaced by a more elaborated subnet, where all border nodes are places (<u>open</u> subnet).

ACTING AND OPERATING

For objects with a prime task [Delta 75] defines two closely related concepts: The object is ACTING from the moment it ENTERs the system until the moment it TERMINATEs, but it is only operating from the moment it executes an Initiation event (see p.69 and 77) until the moment it TERMINATEs. When an object becomes ACTING it also becomes operating (or TERMINATED) at the same moment of model-time.

It is very easy to confuse "ACTING" with "operating". In [Delta 75] this is done several times (e. g. page 252 and 470).

In chapter 5 we propose changes to the Delta semantics, such that "ACTING" and "operating" are identified and "Initiation events" disappear.

CONCURRENT MODE AND EVENT MODE

Delta imperatives are executed in two different modes.

In concurrent mode each operating object executes a concurrency imperative. This designates a set of property clauses; one for each operating object. The corresponding predicates are all concurrently imposed upon the system state.

In event mode only one object executes imperatives. Any other operating object still designates a property clause, but the corresponding predicates are not imposed upon the system state.

[Delta 75, p. 477] distinguishes between 4 different kinds of events:

[4.12]

(i) Initiation (INIT)

A new object ENTERed into the system has to take up its actions

(ii) Completion (COMP)

The execution of a concurrency imperative is completed

(iii) Interruption (INT)

The execution of a concurrency imperative is interrupted

(iv) Encloser termination (ENC)

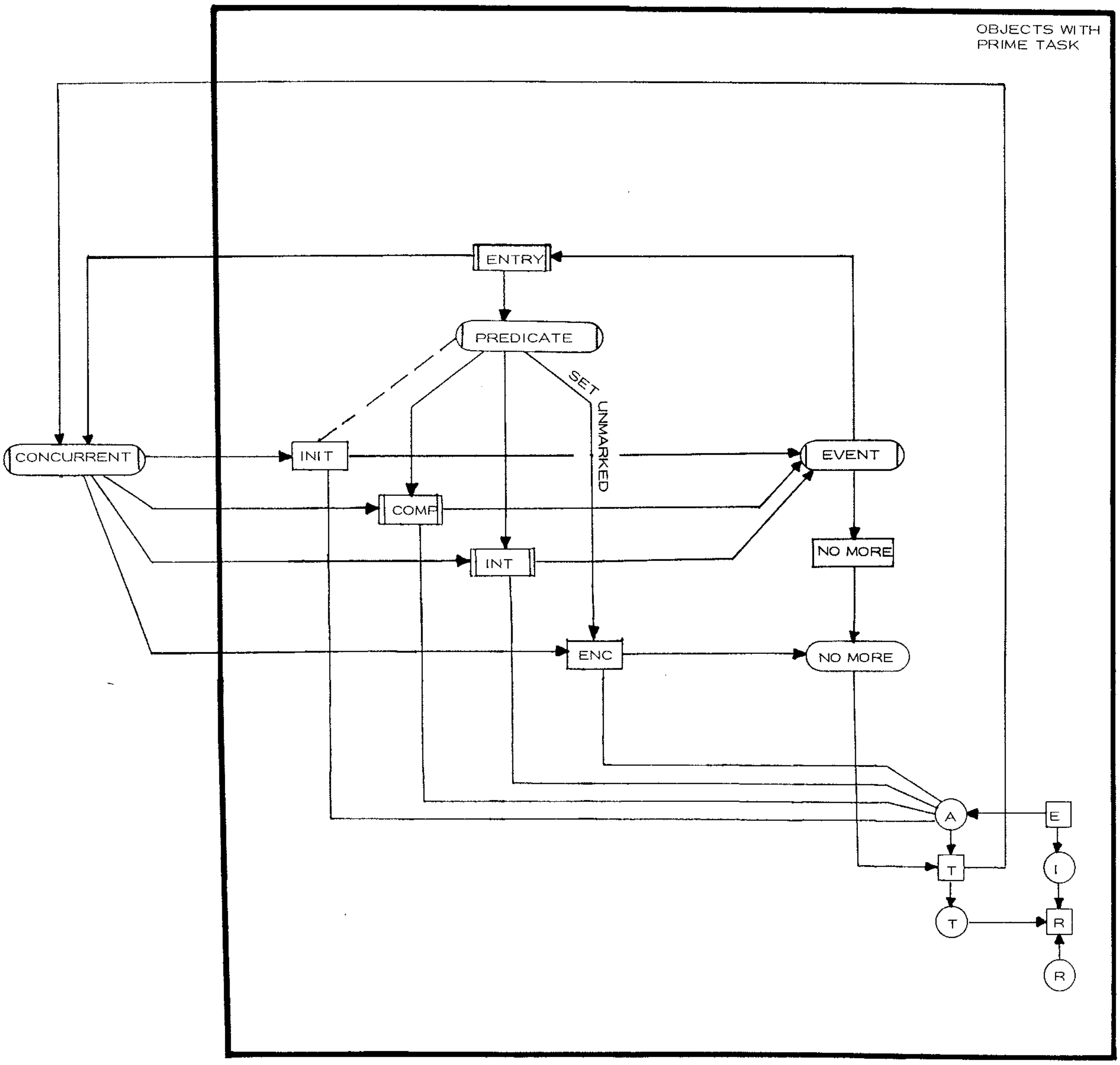
An object has to TERMINATE because the encloser of the object has TERMINATEd.

The alternation between concurrent mode and event mode is represented by the net [4.13].

Initially only CONCURRENT is marked.

During execution the system alternates between concurrent mode (CONCURRENT marked, all EVENT unmarked) and event mode (one EVENT marked, CONCURRENT and all other EVENT unmarked).

In concurrent mode each operating object executes a concurrency imperative with a given predicate. (PREDICATE marked for all operating objects).


OBJECTS WITH
PRIME TASK
ENTRY
PREDICATE
SET
UNMARKED
CONCURRENT
INIT
EVENT
COMP
INT
NO MORE
ENC
NO MORE
A
E
T
I
T
R
R

[4.13]

Each ACTING object OB may execute Delta events. Recalling the description on page 21 the execution of each event can be divided into three parts:

(i) The event starts by a firing of OB's, INIT, COMP, INT or ENC transition. If OB is operating the execution of a concurrency imperative stops.
(OB's PREDICATE becomes unmarked).

(ii) A specified sequence of event actions is executed
(OB's EVENT marked).

(iii) The event finishes by starting the execution of a new concurrency imperative (firing of OB's ENTRY marks OB's PREDICATE) or by making OB TERMINATED. In both cases the system returns to concurrent mode.

In event mode all operating objects except OB still designate a predicate (PREDICATE marked) but these predicates are <u>not</u> imposed upon the system state.

From [4.13] it immediately follows that Delta events are executed one by one. Between each pair of events the system returns to concurrent mode and a set of predicates is imposed upon the system state.

From [4.13] it also follows that the possible sequences for events executed by a fixed object can be described by

$$[\ \text{INT}\ \{\ \text{COMP}\ |\ \text{INT}\ \}\ ]\ \ [\ \text{ENC}]$$

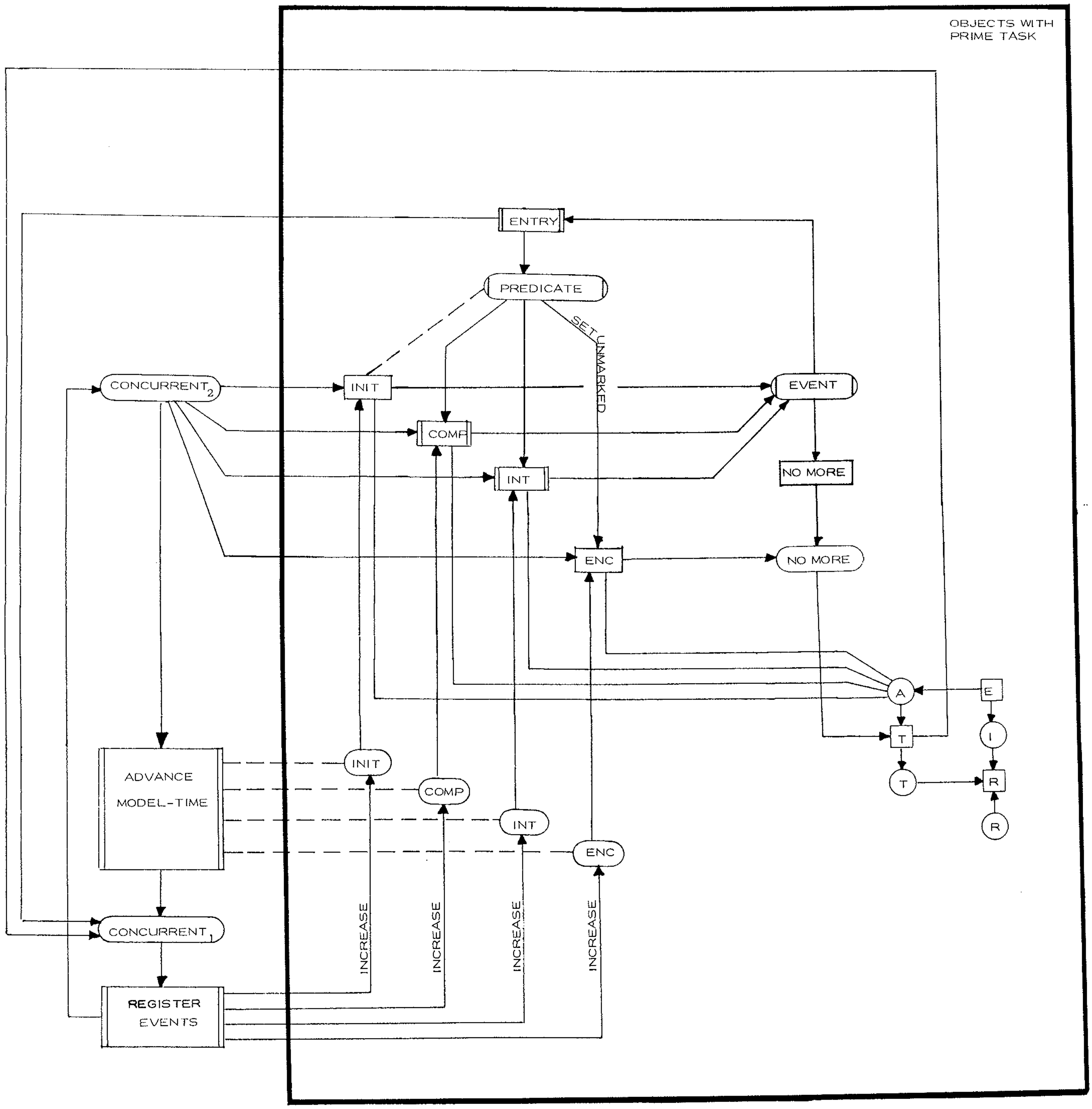
using the "Wirth-notation" introduced on page 24.

The net [4.13] is only a first step towards a model representing Delta systems executing imperatives in concurrent and event mode.

It is for instance not modelled under which circumstances a new object may be generated (firing of an ENTER transition).

REGISTRATION OF EVENTS

We now want to introduce registration of Delta events. To model this we make the following net:



[4.14]

Initially only CONCURRENT$_1$ is marked.

An initiation, completion, interruption or encloser termination event is registered iff the corresponding INIT, COMP, INT or ENC place is marked.

Assume that the system is in concurrent mode ($CONCURRENT_1$ or $CONCURRENT_2$ marked). As long as there are no registered events model-time may advance (ADVANCE MODEL-TIME fires).

When one or more events are registered model-time cannot advance (ADVANCE MODEL-TIME cannot fire since it has a marked testunmark condition).

The only thing which can be done is to execute one of the registered events (this is started by a firing of a INIT, COMP, INT or ENC transition). When the event execution finishes the system returns to concurrent mode ($CONCURRENT_1$ or $CONCURRENT_2$ marked).

As long as there are any registered events, model-time cannot advance. Between each pair of events the system returns to concurrent mode, a set of predicates is imposed upon the system state and events are registered (REGISTER EVENTS fires and may mark some INIT, COMP, INT or ENC places).

We have said nothing about the rules defining the situations in which Delta events should be registered. Moreover,according to the remarks on page 67 ,REGISTER EVENTS should be replaced by a more elaborated closed subnet.

This can indeed be done. It is possible to construct a subnet, which represents event registration as defined in [Delta 75]. Unfortunately this net is large and clumsy. It adds very little to the understanding of the verbal definition given in [Delta 75]. Thus the net will not be given in this paper, and we refer the reader to [Delta 75] p. 477-482.

Here we will only mention three things about the rules for event registration defined in [Delta 75]:

(i) Events can only be registered for ACTING objects.

(ii) At any time each object has at most one registered event.

(iii) Using these rules [4. 14] satisfies SYN4 (see page 62).

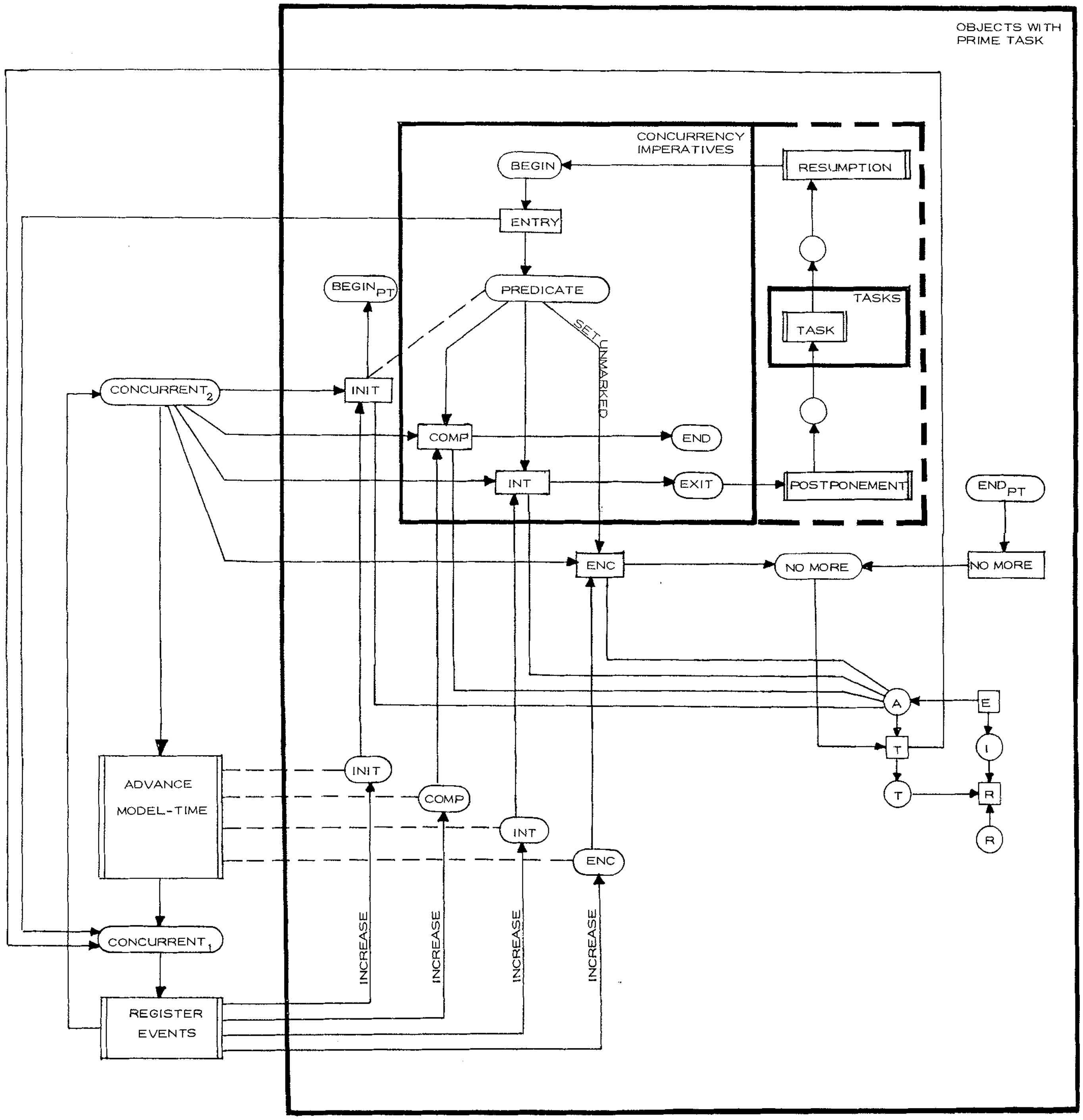
In chapter 5 we will propose changes to the Delta semantics such that the rules for event registration are drastically simplified.

A much more difficult and theoretically interesting problem is how to construct a closed subnet replacing ADVANCE MODEL-TIME. It is not possible to represent a continuous increase in model-time by a finite or countable sequence of transition firing. In chapter 6 we will return to this problem.

EXECUTION OF EVENTS

In [4. 14] we did not distinguish between the different concurrency imperatives executed by an object (each object had only one PREDICATE place).

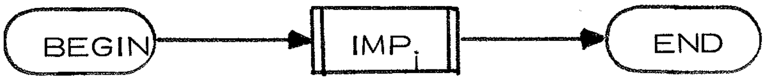
Now consider the following net, where each concurrency imperative is represented by its own subnet:


OBJECTS WITH PRIME TASK
CONCURRENCY IMPERATIVES
BEGIN
ENTRY
PREDICATE
SET UNMARKED
BEGIN$_{PT}$
CONCURRENT$_2$
INIT
COMP
END
INT
EXIT
RESUMPTION
TASKS
TASK
POSTPONEMENT
END$_{PT}$
ENC
NO MORE
NO MORE
A
E
T
I
T
R
R
ADVANCE MODEL-TIME
INIT
COMP
INT
ENC
INCREASE
INCREASE
INCREASE
INCREASE
CONCURRENT$_1$
REGISTER EVENTS

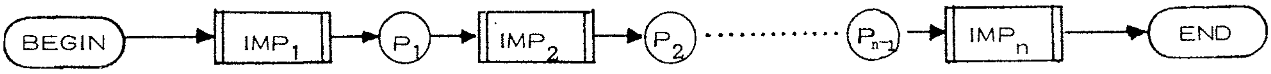
[4. 15]

To understand [4. 15] and the following nets, it is important to remember that the different subnets representing imperatives executed by an object are combined (by a syntax directed translation) into a big net, representing all actions which can be executed by the object.

Let T be a task consisting of a sequence of imperatives IMP_1; IMP_2; ... ; IMP_n, where each imperative IMP_i is represented by a net of the form

[4. 16] 

Then T is represented by the following net

[4. 17] 

where each place P_i is END place for IMP_i and BEGIN place for IMP_{i+1}

Thus in [4. 15] the BEGIN place for one concurrency imperative may be identical to the END place of another.

Similar remarks apply for the composition of subnets representing postponement clauses,interrupting tasks and resumption clauses. We will return to this on page 78.

Initially only $CONCURRENCY_1$ is marked.

Assume that the system is in concurrent mode. If one or more events are registered model-time cannot advance. The only thing which can be done is to execute one of the registered events.

Let OB be a given ACTING object. If OB is operating it is executing a concurrency imperative, IMP.

The relationship between OB and the event chosen for execution can be of five different types, all of which are described below using the three steps introduced on pages 21 and 71.

1. Initiation event for OB

Then OB is not operating, and this is the first event executed by OB.

(i) OB's INIT transition fires. This marks the BEGIN place for OB's prime task $(\text{BEGIN}_{\text{PT}})$.

(ii) A sequence of event actions is executed. It starts with the first action in OB's prime task and continues until a concurrency action or the end of OB's prime task is reached. If the first imperative encountered is a concurrency imperative the sequence may be empty.

(iii) The event finishes by firing of the ENTRY transition for the concurrency imperative reached or by firing of OB's NO MORE followed by a firing of OB's TERMINATE. In both cases the system returns to concurrent mode.

2. Completion event for OB

Then OB is operating

(i) IMP's COMP transition fires. This completes the execution of IMP, and its predicate will no longer be imposed upon the system state (IMP's PREDICATE place becomes unmarked).

(ii) As in 1, except that the sequence starts with the action immediately following IMP.

(iii) As in 1.

3. Interruption event for OB

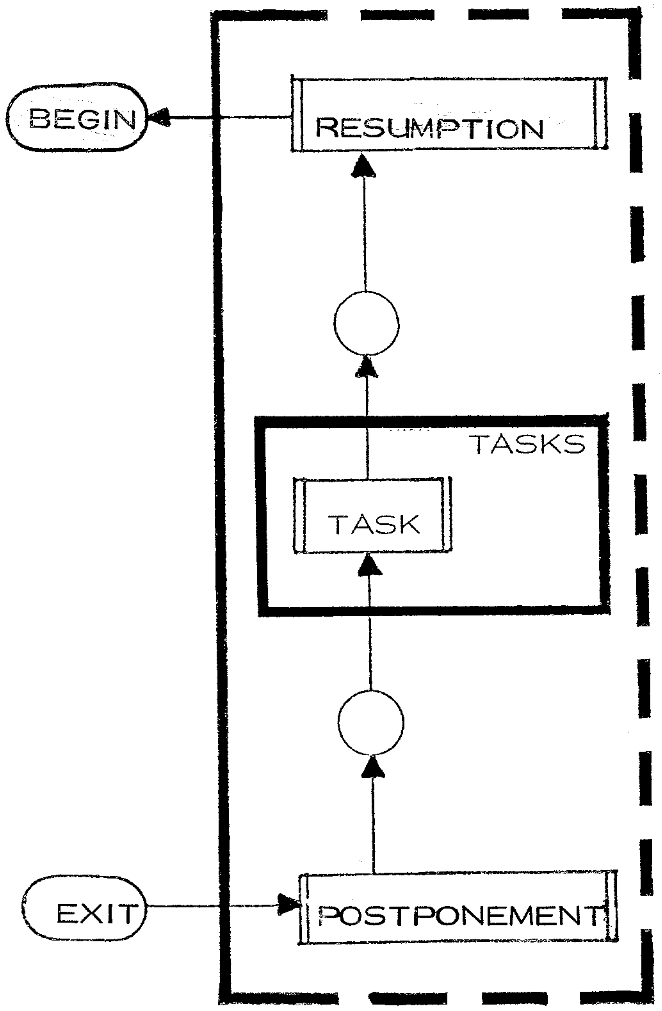
Then OB is operating.

(i) IMP's INT transition fires. This postpones the execution of IMP, and its predicate will no longer be imposed upon the system state (IMP's PREDICATE place becomes unmarked).

(ii) As in 1, except that the sequence starts with the first action in IMP's postponement clause.

(iii) As in 1.

POSTPONEMENT, TASK and RESUMPTION may contain subnets representing concurrency imperatives and thus already contained in [4.15]. The subnet



[4.18]

should be understood as merely indicating:

(i) The EXIT place for a given concurrency imperative is identical to to the BEGIN place for its postponement clause.

(ii) The END place for a postponement clause in a given concurrency imperative is identical to the BEGIN places in subnets representing all tasks, which may be used to interrupt the imperative.

It should be stressed that each concurrency imperative has its <u>own</u> subnets representing interrupting tasks. These interrupting tasks may again be interrupted. This implies that the net representing a given task may be infinite.

In [4.15] it is not modelled how to select the subnet representing the actual interrupting task among the subnets representing all possible interrupting tasks.

(iii) The END places for all interrupting tasks are identical to the BEGIN place for the resumption clause in the interrupted concurrency imperative.

(iv) The END place of the resumption clause in a given concurrency imperative is identical to imperatives BEGIN place.

If a concurrency imperative cannot be interrupted, has no postponement clause,or no resumption clause,[4.18] may be simplified accordingly.

4. Encloser termination event for OB

Then OB may be operating or not, and this will be the last event executed by OB

(i) OB's ENC transition fires. If OB is operating the execution of IMP is finished (IMP's PREDICATE place becomes unmarked).

(ii) Empty.

(iii) The event finishes by a firing of OB's TERMINATE. The system returns to concurrent mode.

5. The event is executed by an object different from OB

Then OB may be operating or not.

If OB is operating IMP's PREDICATE place remains marked during the execution of the event; but the corresponding predicate will not be imposed upon the system state.

When the event finishes and the system returns to concurrent mode IMP's predicate will again be imposed upon the system state.

The reader should notice that the description above interprets a marked PREDICATE place as follows: it indicates that the corresponding predicate is imposed upon the system state, if and only if the system is in concurrent mode (CONCURRENT$_1$ or CONCURRENT$_2$ marked).

CONCURRENCY IMPERATIVES

In [4.15] all concurrency imperatives were modelled by the same form of subnets. This is not strictly correct.

There are seven different kinds of concurrency imperatives distinguished by seven different kinds of duration clauses/designational clauses.

"PASSINGLY", "PAUSE" and "EMPTY"

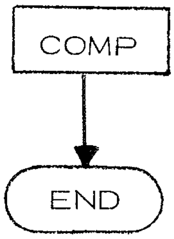
These imperatives are all modelled by the subnet from [4.15] without modifications.

The rules defining event registration excludes interruption events for "PAUSE" and completion events for "EMPTY". Thus the places and transitions corresponding to such events can be omitted without any change in the defined semantics.

"ADVANCE", "CONCLUDE" and "TERMINATE"

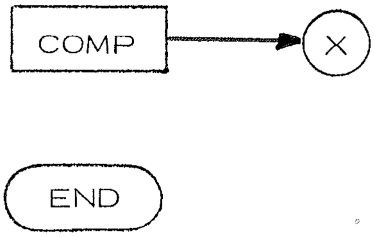
These imperatives are all modelled by the subnet from [4.15] with

[4.19]



replaced by

[4.20]



where the place X is the place defined by:

"ADVANCE" : END place for the concurency imperative containing the "ADVANCE" imperative in its postponement clause or resumption clause.

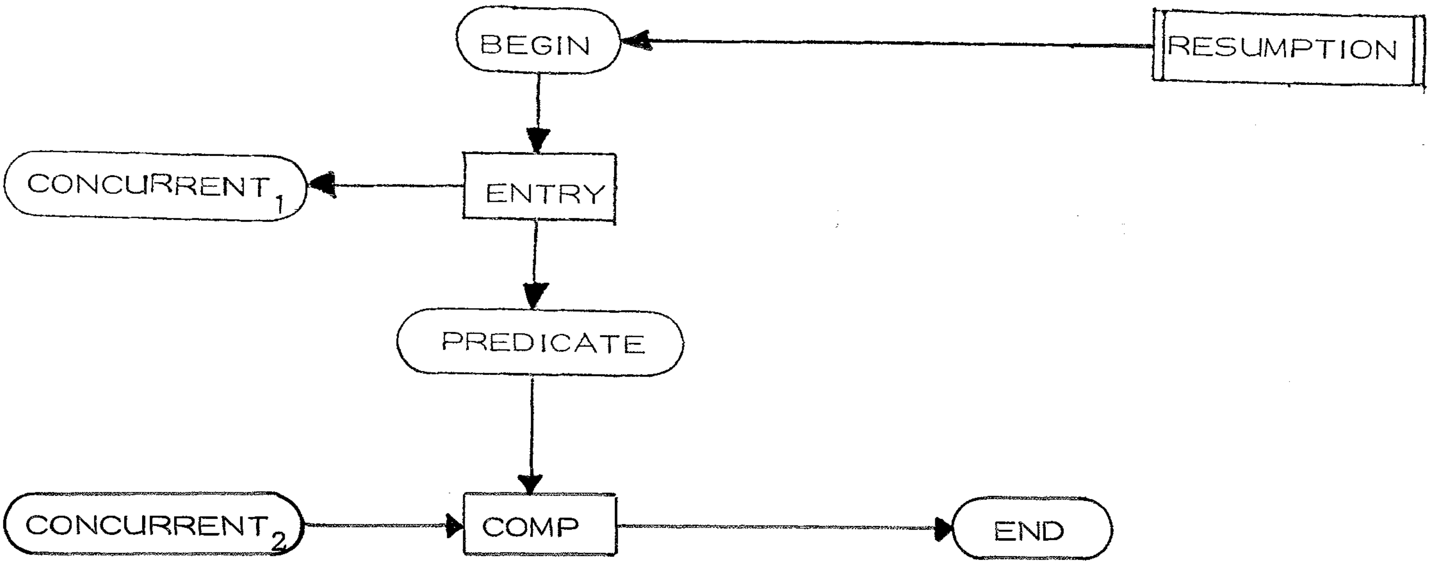
"CONCLUDE": END place of the activity containing the "CONCLUDE" imperative.

"TERMINATE":END place of the object's prime task.

The rules defining event registration excludes interruption events for "ADVANCE", "CONCLUDE" and "TERMINATE". Thus the places and transitions corresponding to such events can be omitted without any change in the defined semantics.

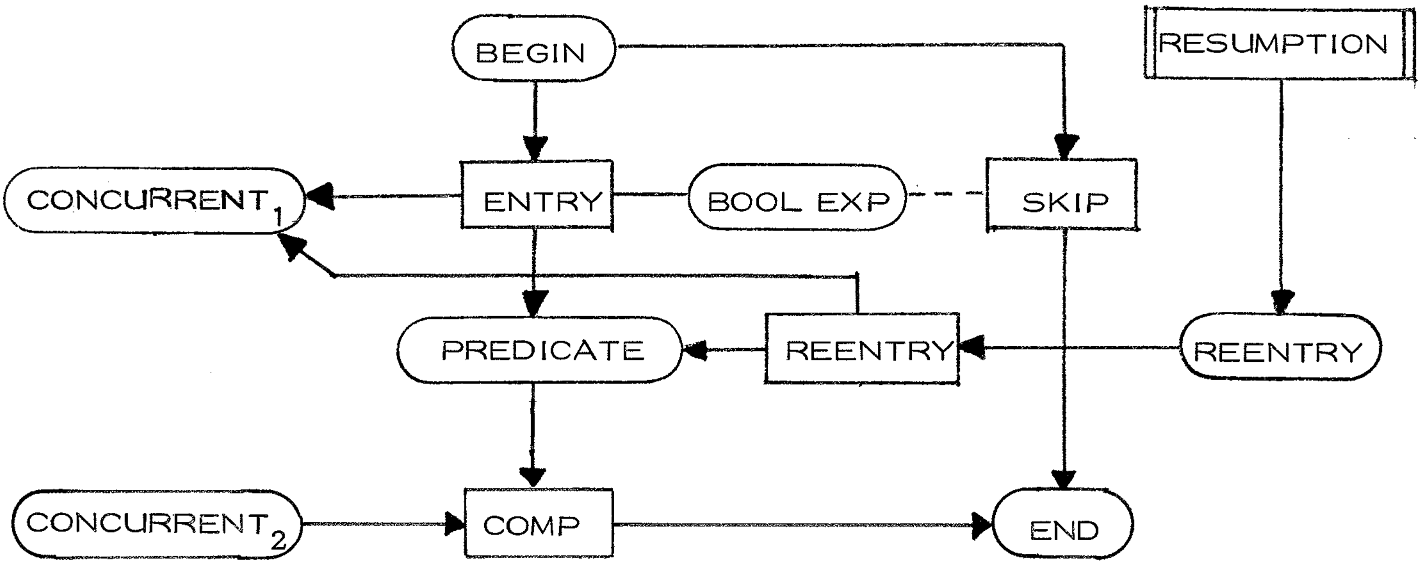
"WHILE"

The imperative "WHILE B" where B is a boolean expression is modelled by the subnet from [4.15] with



[4.21]

replaced by



[4.22]

The place BOOL EXP is marked in a system state S iff

a) B is true in S

b) The object excuting IMP is INSIDE in S

(Property b is requested in order to avoid infinite markings, see p. 53).

When execution of the WHILE-imperative, IMP, starts it is tested whether the boolean expression B in the duration clause is true. This test is an event action. If B is false, the system remains in event mode and the execution of IMP is SKIPped. If B is true, the concurrent action defined by IMP's property clause is executed. When the execution of IMP is resumed after an interrupt, B is not tested in an event action.

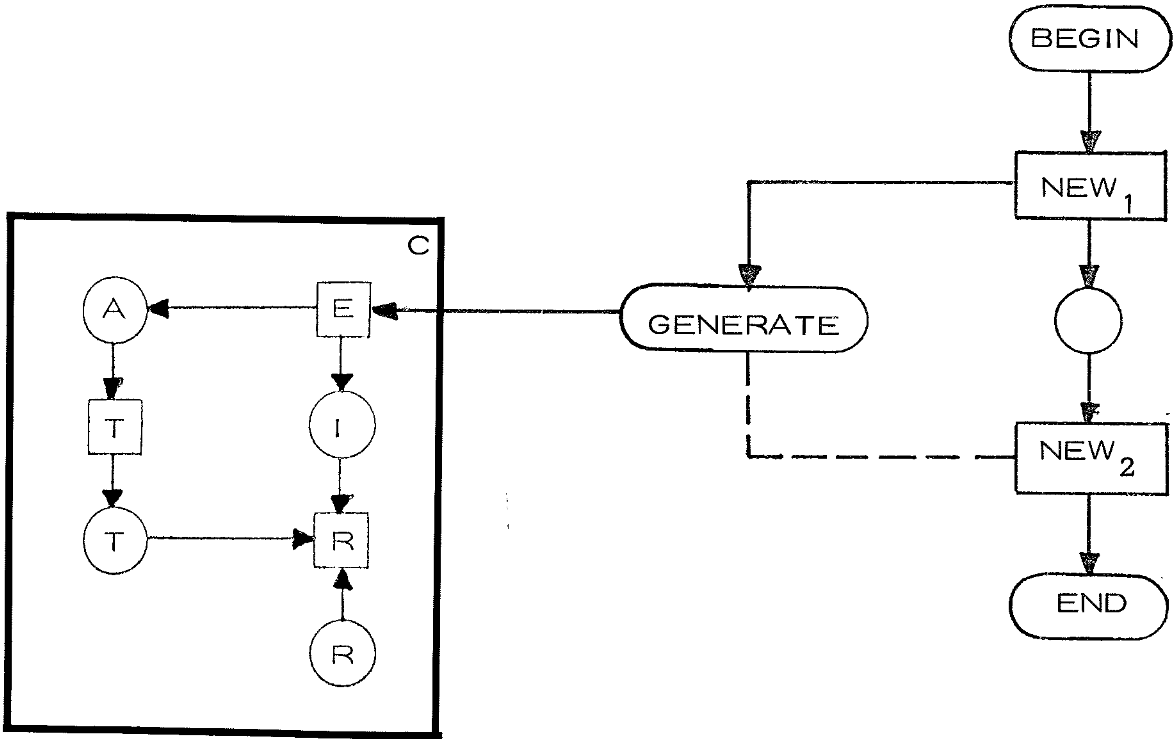
In this paper it is not modelled how the correct marking of BOOL EXP is maintained during concurrent execution.

We think that the above semantics for "WHILE" is unnecessarily complicated and unfit for verification purposes. Thus we suggest that "WHILE" should be modelled by the subnet from [4.15] without modifications. We will return to this in chapter 5.

EVENT IMPERATIVES

Mini-Delta has two different kinds of event imperatives: "NEW" and "INTERRUPT".

"NEW":

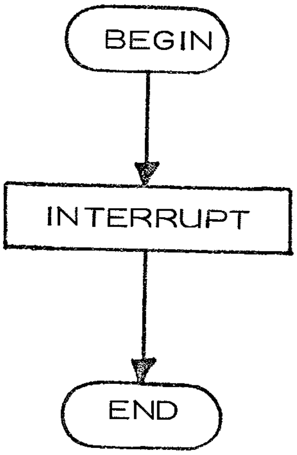


[4.23]

For each class C all "NEW" imperatives in the system generating objects of class C share the same GENERATE place. Thus there is exactly one GENERATE place for each class.

"INTERRUPT":

[4.24]

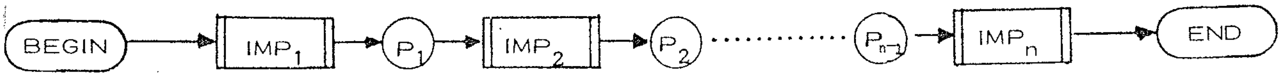


This looks trivial: The reason is that when an interrupt is sent the only thing which is done is to indicate that a given task has been sent to interrupt a given object. This is done merely by updating a complicated datastructure- the agenda of the interrupted object. (See page 15).

COMPOSITE IMPERATIVES

There are six different kinds of composite imperatives in Mini-Delta:

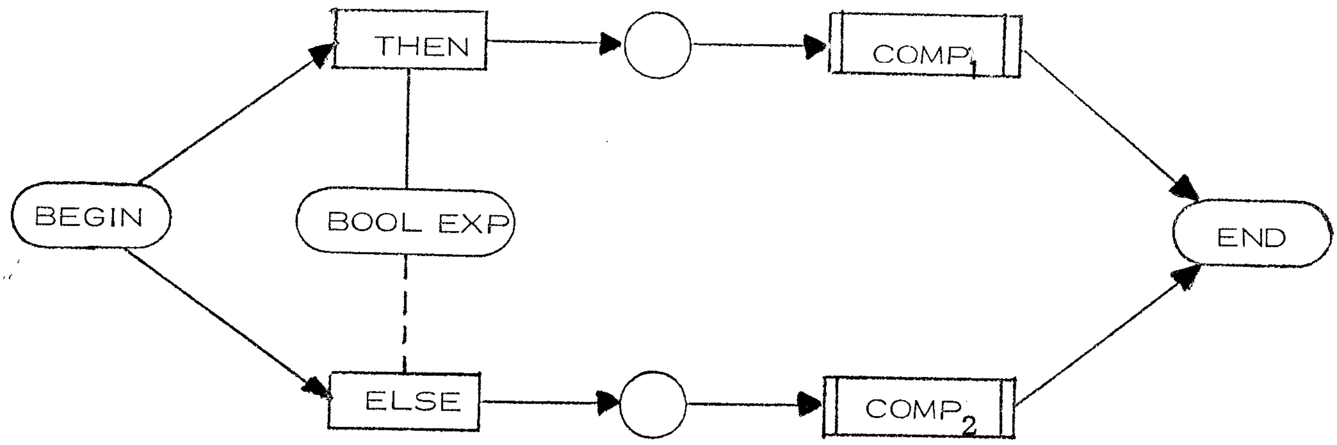
(* IMP_1 ; IMP_2 ; .... ; IMP_n *):



[4.25]

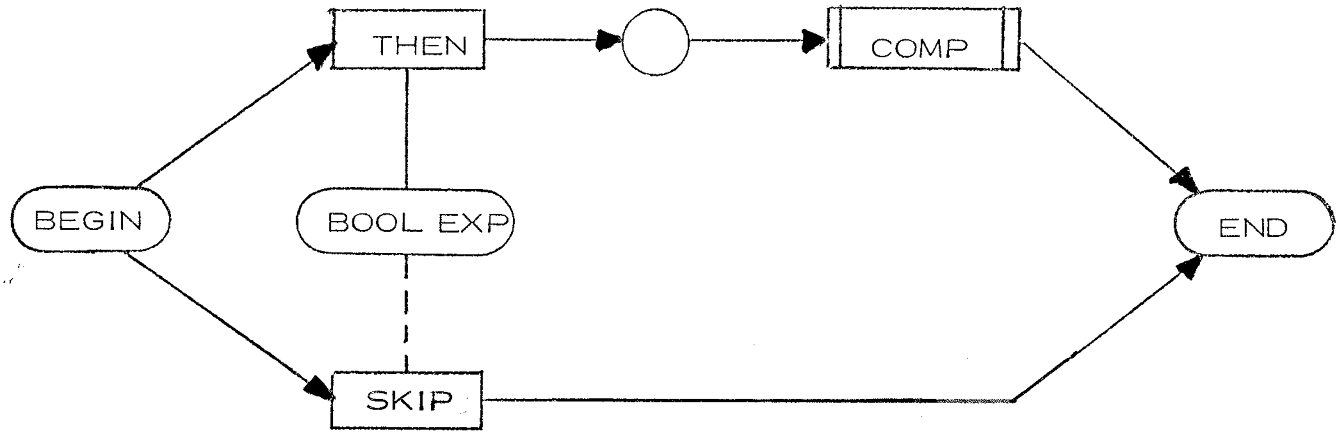
P_i is END place for IMP_i and BEGIN place for IMP_{i+1}. Similar remarks apply for [4.26] - [4.30].

"IF BOOL EXP THEN COMP$_1$ ELSE COMP$_2$" :



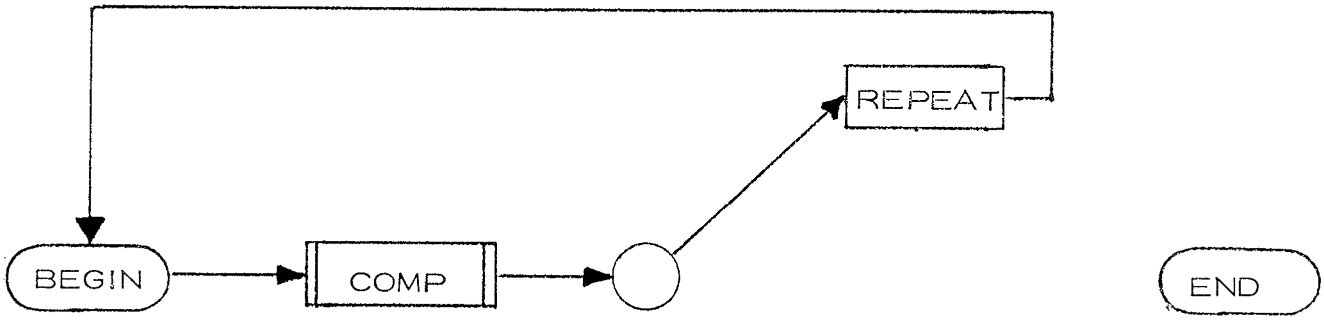
[4. 26]

"IF BOOL EXP THEN COMP " :



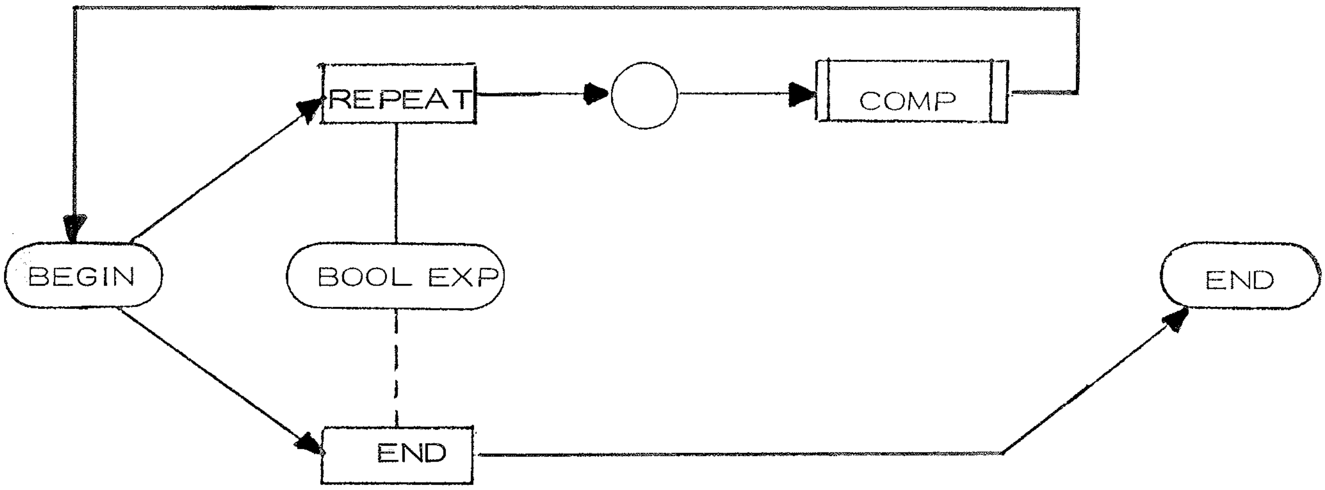
[427]

"REPEAT COMP":



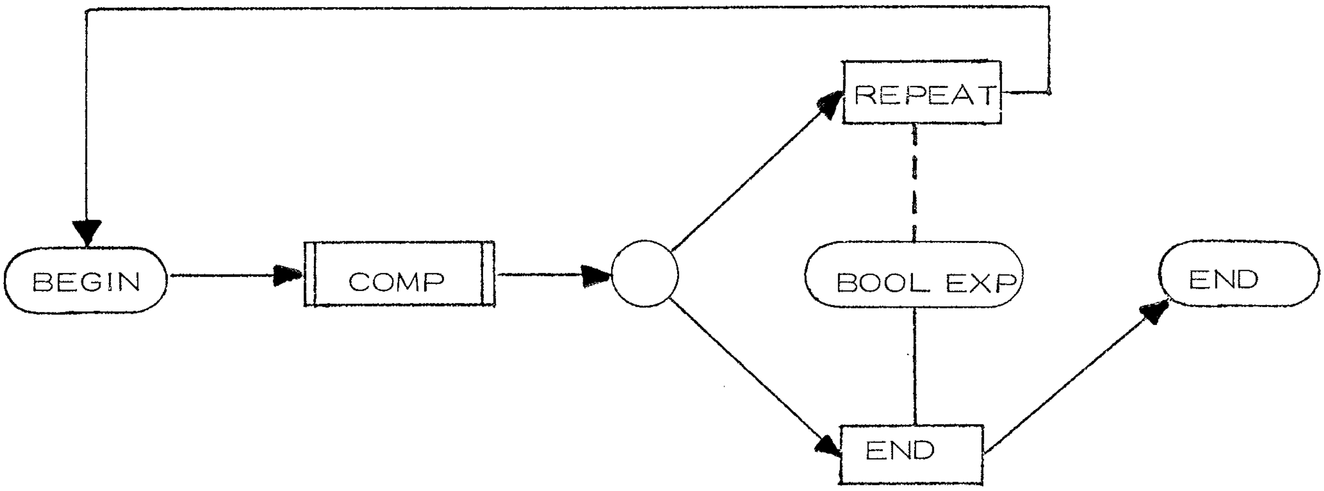
[4. 28]

"WHILE BOOL EXP REPEAT COMP":



[4. 29]

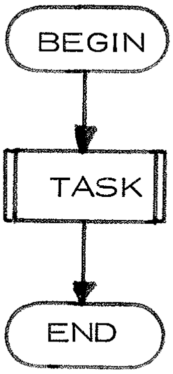
"REPEAT COMP UNTIL BOOL EXP"



[4.30]

Each composite imperative with a boolean expression has a BOOL EXP place. These places are analogous to the BOOL EXP place for the concurrency imperative "WHILE" (See page 83) and the remarks there also apply for composite imperatives.

TASK IMPERATIVES



[4.31]

Task imperatives generate a singular or category defined task. In both cases a subnet representing the generated task is "copied" directly into the spot of the task imperative.

This corresponds to what is known in compiler theory as an open procedure call.

For category defined recursive tasks this implies that the net representing the task imperative becomes infinite.

DELTA SYSTEMS

This closes our definition of Mini-Delta with its current semantics.

Once more we want to remind the reader that to get an extended Petri net representing an entire Delta system, a syntax directed translation is used to combine the various subnets representing model-time, event registration, executional modes and the objects with their phases and imperatives.

In chapter 5 we propose changes to the semantics defined in this chapter.

Chapter 5

PROPOSALS FOR CHANGES TO THE DELTA SEMANTICS

In this chapter we discuss the parts of the semantics (defined in chapter 4 and [Delta 75]) which concerns the concurrency imperatives, creation and destruction of objects, and registration and execution of events. Based on the discussion we propose a number of changes to the semantics. Syntax is not discussed.

We believe that Delta would gain from a thorough syntactic revision(e. g. it is confusing to use WHILE as a keyword in two conceptually very different types of imperatives). Such a revision is, however, outside the scope of this paper, and we find isolated changes confusing and unnecessary for our present discussion of the semantics.

5.1 Imperatives

In this section we discuss the semantics of concurrency imperatives. We change (and simplify) the semantics of "WHILE" in such a way that the other concurrency imperatives (except for "ADVANCE", "CONCLUDE" and "TERMINATE" - the so-called "structured jumps") become syntactically sugared versions of simple forms of the WHILE - imperative. We argue that of the structured jumps, "ADVANCE" can be dispensed with and we change the other two into event imperatives. The net effect of this is that now "WHILE" is the only concurrency imperative.

CONCURRENCY IMPERATIVES

[Delta 75] defines two categories of imperatives:

- event imperatives and
- concurrency imperatives.

> "An event imperative describes an event action. That is an action
> which is event internal [i. e. executed when the system is in event
> mode]. This implies that no other object may execute any actions
> [5. 1] (changing the system state) between the completion of the action
> preceding the event action, and the initiation of the action following the event action.
> A concurrency imperative describes a concurrency action. That is, an action which is executed in concurrency with actions executed by other objects [i. e. executed when the system is in concurrent mode]". [Delta 75, p. 339].

Both types of imperatives can be composite such as for example the composite event imperative:

```
WHILE BOO REPEAT   (* BOO: = f(BOO) *)
```

and the composite concurrency imperative

```
WHILE BOO REPEAT
   (*  BOO: = f (BOO);
       WHILE TIME < ft (BOO)  LET  {TRUE} *)
```

In both these examples all assignments and tests on the value of BOO are executed in event mode and the composite concurrency imperative in the last example does thus not describe a concurrency action (cf. [5. 1]).

In order to make the semantics of imperatives correspond to their names we shall (as we did in connection with Mini-Delta in chapter 2) classify Delta-imperatives as

- event imperatives,
- concurrency imperatives,
- composite imperatives,
- task impertives.

Thus all parts of composite and task imperatives which are described by concurrency imperatives (i. e. by "WHILE" and its syntactically sugared versions) are executed in concurrency mode and all other imperatives in event mode.

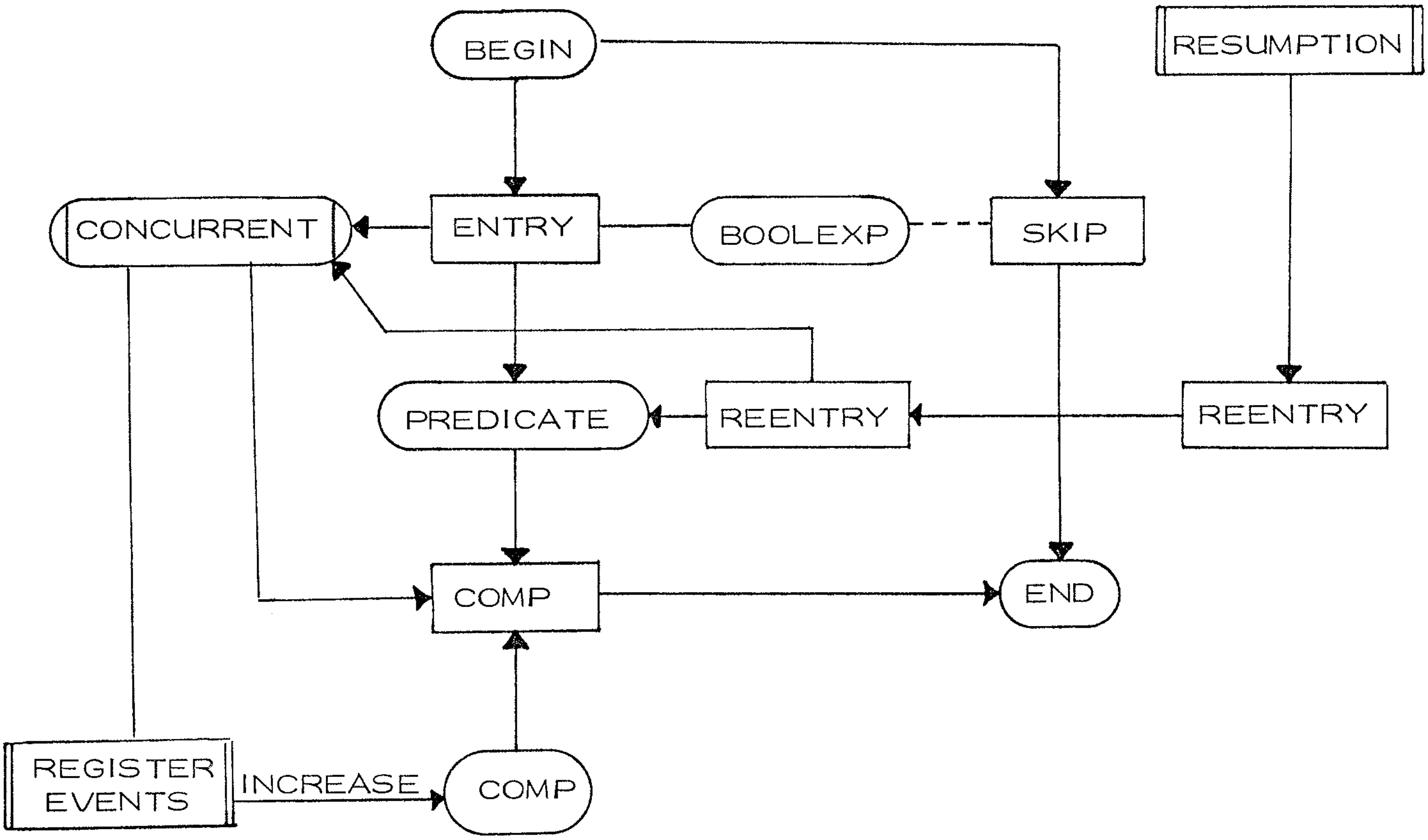
The first step towards achieving this is to change the semantics of "WHILE", since according to [Delta 75] this imperative is executed in both event and concurrency mode.

THE WHILE IMPERATIVE

For the sake of simplicity we shall consider only the following simple form of "WHILE":

[5.2] WHILE BOO LET {PRED}

(it follows from the discussion below how the different versions of "WHILE" are affected). [5.2] is essentially represented by the following subnet of the net [4.15] (p.75, cf. [4.21] and [4.22] p.82 and p.83):

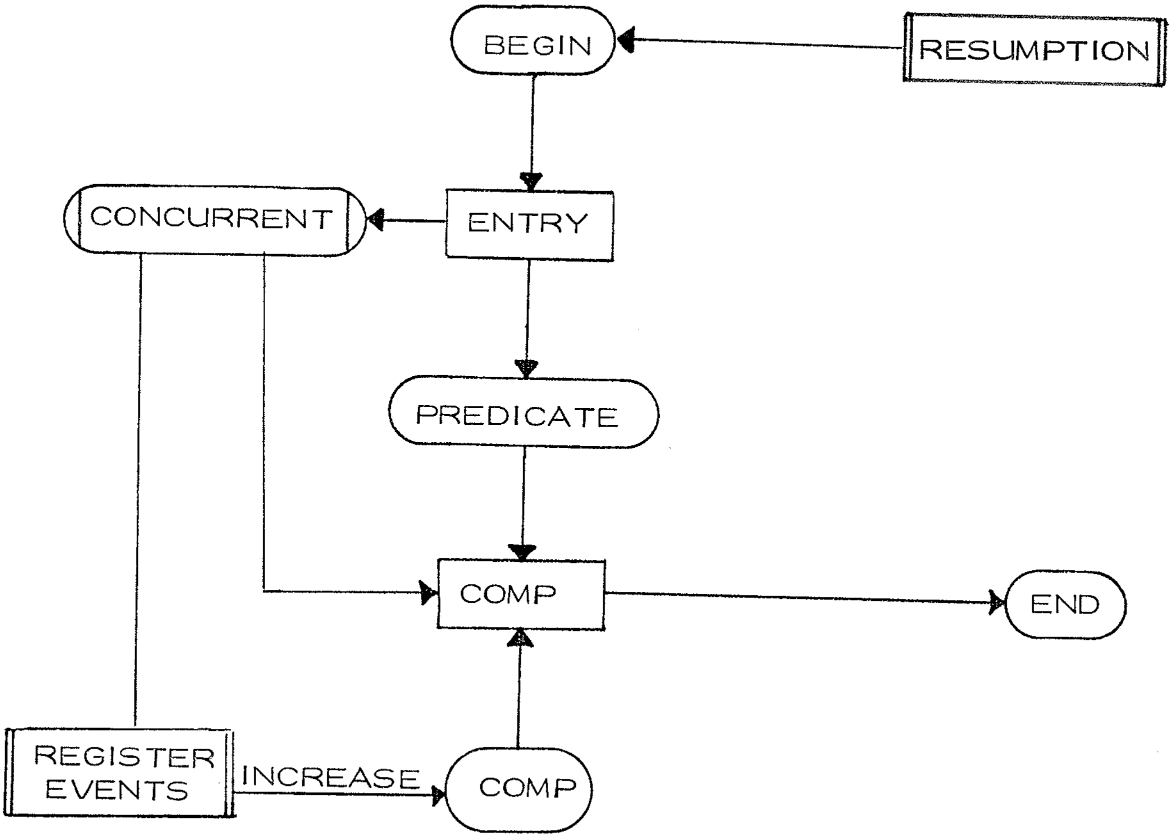


[5.3]

Note that the place CONCURRENT in [5.3] replaces CONCURRENT$_1$, CONCURRENT$_2$ and ADVANCE MODEL-TIME in [4.15].

It follows from the net that when the WHILE-imperative begins executing, BOO is tested in event mode (BEGIN is marked, hence CONCURRENT is unmarked). Once PREDICATE has been marked BOO is only tested in concurrent mode (this test is not modelled explicitly. It is a part of REGISTER EVENTS which only fires when CONCURRENT is marked).

In accordance with our wish to make "WHILE" the basic concurrency imperative, we now change the semantics by removing the initial test on BOO in event mode, i.e. we now represent [5.2] by the following net:



[5.4]

To sum up, this means that "WHILE" is now modelled by the subnet from [4.15] without modifications, just as "EMPTY", "PASSINGLY" and "PAUSE".

EQUIVALENCE

We close our discussion of the WHILE-imperative by considering the relation between the old semantics ([5.3]) and the new ([5.4]).

When the WHILE-imperative [5.2] is part of an entire Delta description, subnets representing the imperatives and subnets representing the objects etc. are combined into an extended Petri net, which together with the initial marking, forms a resultant snapshot which represents the entire Delta description.

We will show that the two semantics have the same expressive power. By this we mean that given a Delta description and one of the two semantics, it is always possible to construct a second Delta description, which (with respect to the other semantics) has a resultant snapshot being equivalent (see below) to the resultant snapshot of the first Delta description (with respect to the first semantics).

We will not consider equivalence of extended snapshots in general. We will only consider equivalence in connection with resultant snapshots obtained from Delta descriptions, which have exactly the same object structure. Moreover in the considered Delta descriptions each object will contain exactly the same set of attributes (except for auxilliary variables). Imperatives in the prime task and in the attributes may be different in the two descriptions, but they must use exactly the same set of predicates and boolean expressions (except for auxilliary variables).

This means that the resultant snapshots are identical except that some closed subnets (representing sequences of imperatives) are substituted by other closed subnets, which then contain exactly the same set of PREDICATE places and BOOL EXP places.

The essential marking of a snapshot ps = (pn, m) is denoted EM(ps) and defined as the restriction of m to places named by PREDICATE, BOOL EXP, CONCURRENT$_i$, INIT, COMP, INT, ENC, GENERATE, ACTING, TERMINATED, INSIDE, REMOVABLE and NO MORE.

Intuitively this means that we "forget" about places which are unnamed or named by BEGIN, END and REENTRY.

Two resultant snapshots (which have an identical structure in the manner described above) are equivalent iff for each sequence of extended snapshots reachable from one of them

$$ps_{11} \rightarrow ps_{12} \rightarrow \cdots\cdots \rightarrow ps_{1n} \qquad (n \geq 0)$$

there exists a sequence of extended snapshots reachable from the other resultant snapshot

$$ps_{21} \rightarrow ps_{22} \rightarrow \cdots\cdots \rightarrow ps_{2m} \qquad (m \geq 0)$$

such that the two sequences

$$\left\{EM(ps_{1i})\right\}_{1 \leq i \leq n} \quad \text{and} \quad \left\{EM(ps_{2i})\right\}_{1 \leq i \leq m}$$

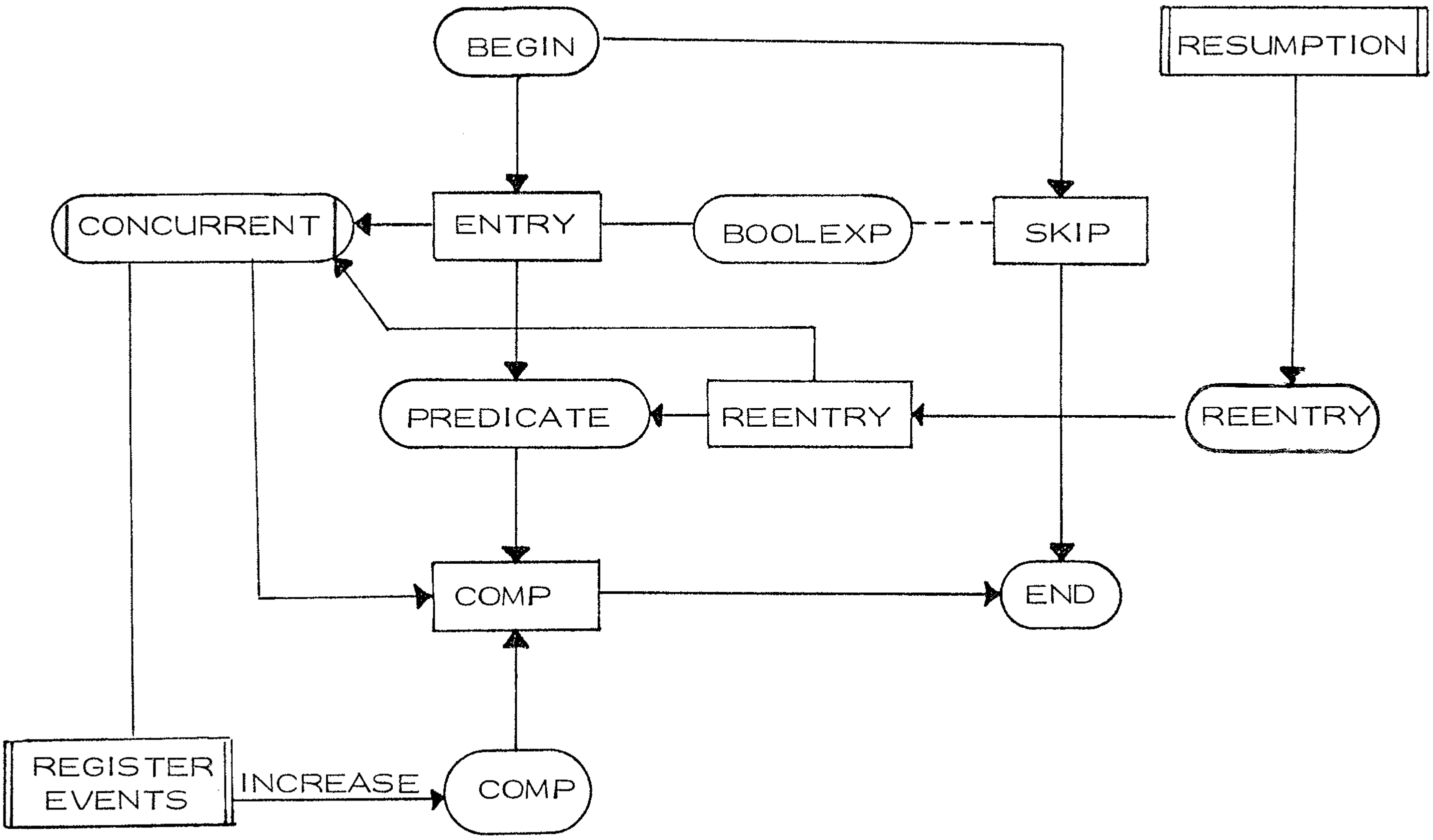
contains exactly the same set of essential markings, in the same order but possibly with different multiplicity.

MODELLING THE OLD BY THE NEW

It is possible to model the old semantics of the WHILE-imperative [5.2] with the new, simply by prefixing with the deleted test in event mode.

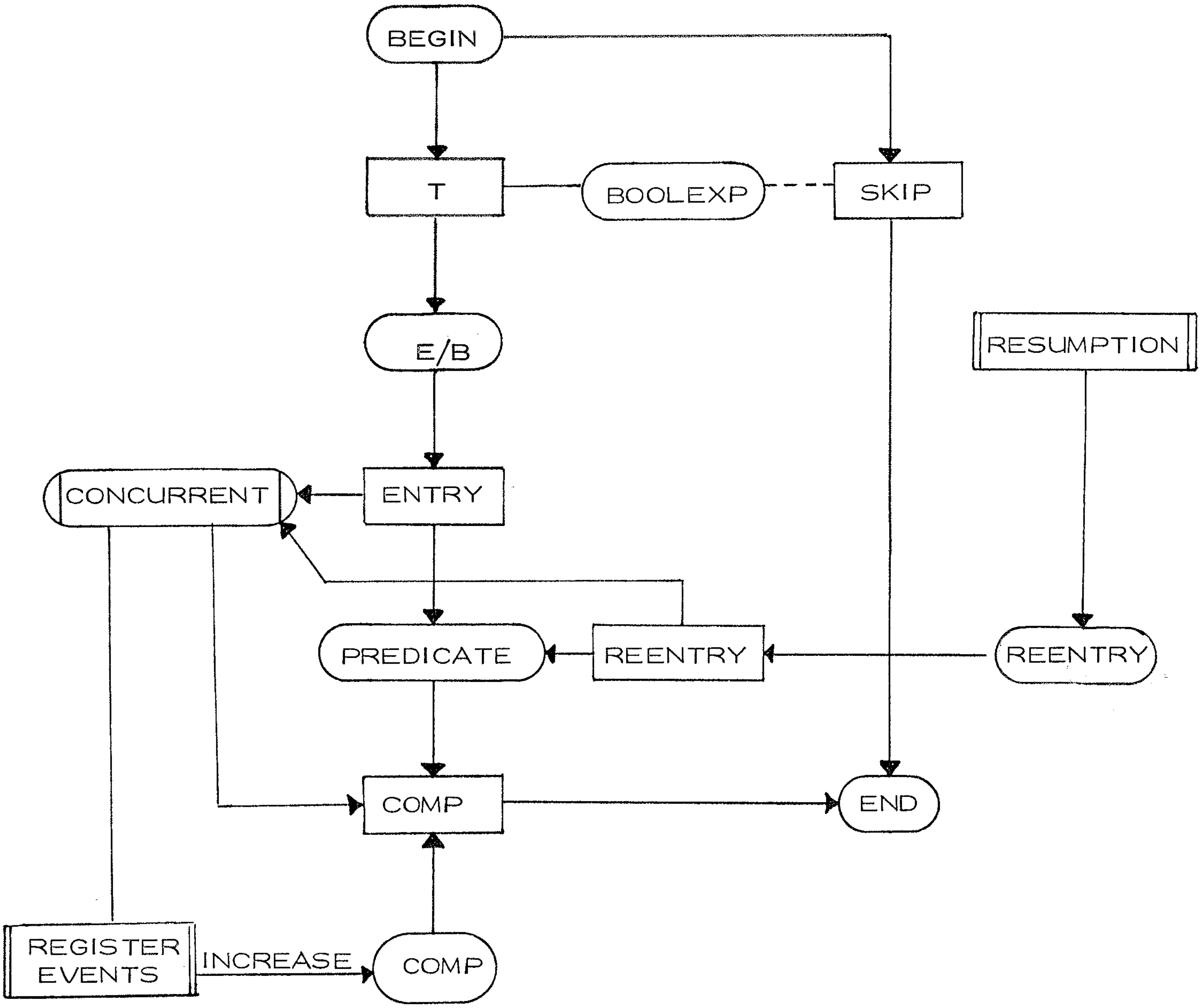
Below we indicate how to transform subnets like [5. 3] (i. e. the old semantics) into subnets representing the new semantics (i. e. [5. 4]) plus a test in event mode (i. e. an "IF-THEN") in such a way that the two resultant snapshots are equivalent.

Consider the subnet [5. 3] once more:



([5. 3])

Next consider the net below

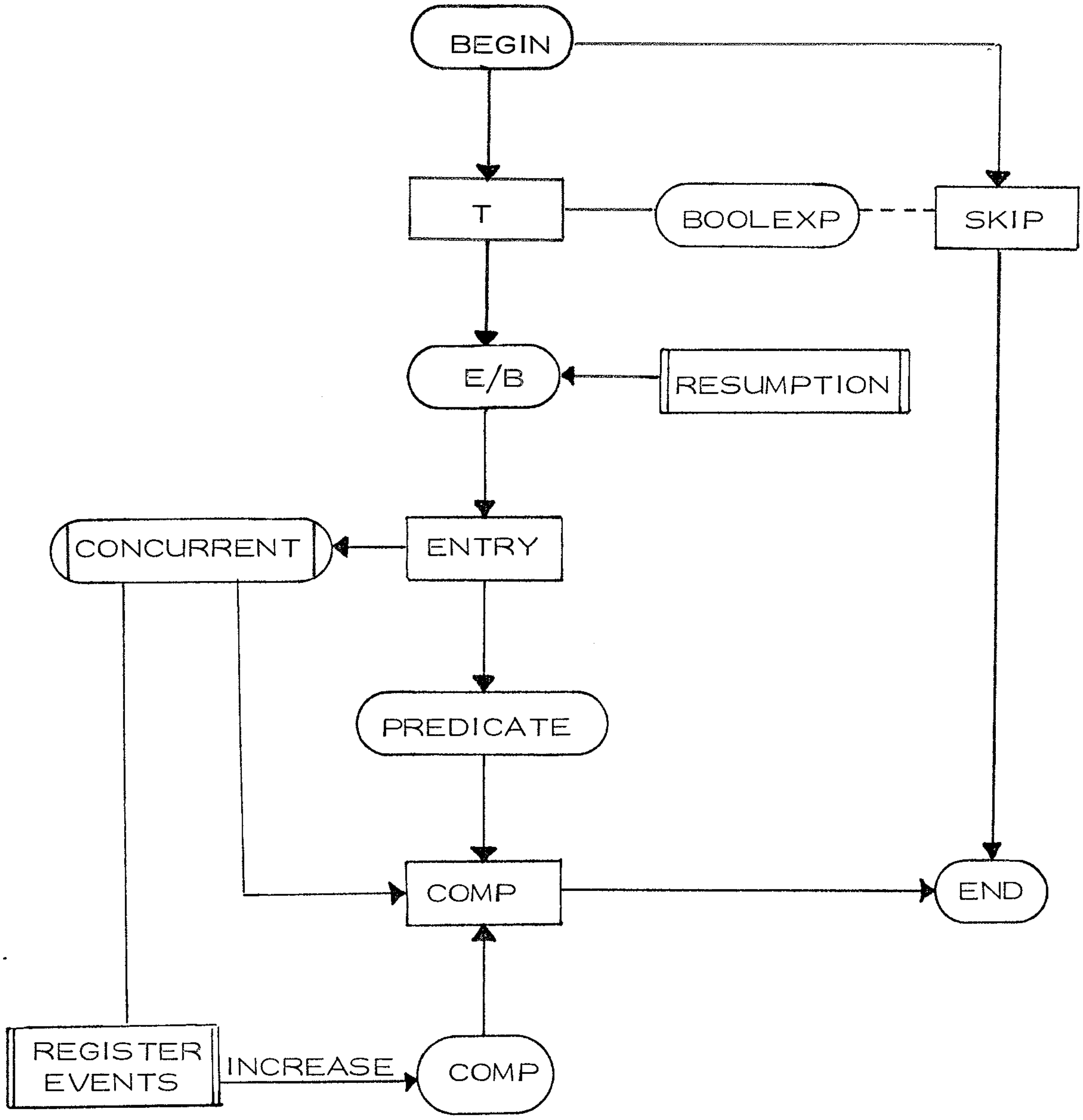


[5. 5]

The resultant snapshot of [5. 3] is equivalent to the resultant snapshot of [5. 5].

This equivalence relies on the fact that Delta events are mutually exclusive, (cf.[4.15]), thus no other transitions can fire concurrently with or between the firings of T and ENTRY in [5.5].

By a similar reasoning it follows that the resultant snapshot of [5.5] is equivalent to the resultant snapshot of [5.6] below.



[5.6]

The subnet [5.6] in turn represents the new semantics of

```
          IF BOO THEN
          (* WHILE BOO LET { PRED} *)
```

(cf. [5.4] and [4.27]).

MODELLING THE NEW BY THE OLD

We are not able to model the new semantics by the old, without introducing an auxiliary boolean variable. The problem is that we now have to eliminate the effect of the initial test in event mode, instead of adding it.

The new semantics of the WHILE-imperative [5.2] is equivalent to the following imperatives in the old semantics

```
          A: = TRUE;
[5.7]
          WHILE A ∨ BOO LET {PRED ∧ A = FALSE}  DEFINE A
```

Drawing the nets is left as an exercise for the reader.

"WHILE" AND ITS RELATION TO "EMPTY" "PASSINGLY" AND "PAUSE":

"EMPTY" is syntactic sugar for "WHILE TRUE".

With the new semantics of the WHILE-imperative we may regard the time concurrency imperatives "EMPTY" and "PASSINGLY" and the instant concurrency imperative "PAUSE" as special cases of "WHILE".

"EMPTY" is syntactic sugar for "WHILE TRUE".

"PAUSE" is syntactic sugar for "WHILE FALSE".

Finally "PASSINGLY" is syntactic sugar for "WHILE PENETRATING INTERRUPT".

PENETRATING INTERRUPT is introduced as a new language defined boolean function; it has the value true, iff the object executing the function has a penetrating interrupt on its agenda.

With the old semantics of the WHILE-imperative there were no such simple relations between these imperatives.

"EMPTY" was syntactic sugar for "WHILE TRUE", but "PAUSE" could only be modelled by means of an auxillary boolean variable and "PASSINGLY" could not be modelled by means of the WHILE-imperative or any other imperative.

VERIFICATION RULES AND STRUCTURED JUMPS

Below we first discuss a rudimentary form of a verification rule for the WHILE-imperative [5.2]. Then we discuss how the rule is affected when we consider forms of the WHILE-imperative which includes EXIT and REENTRY clauses.

With the old semantics of the WHILE-imperative [5.2] we may formulate the following rule:

[5.8] When the place END is marked the predicate
PRED or non BOO holds.

The rule is easily deduced from the net [5.3].

With the new semantics we get the following improved rule (cf. [5.4]):

[5.9] When the place END is marked the predicate
PRED holds.

Note that the semantics (both the old and the new) is defined in such a way that PRED is imposed after each interrupt.

Next we consider the situation, when the WHILE-imperative has the following form:

```
            WHILE BOO LET  {PRED}
[5.10]      EXIT (*  IMP 1  *)
            REENTRY (*  IMP 2  *)
```

If IMP1 or IMP2 involves the execution of a structured jump ("ADVANCE", "CONCLUDE" or "TERMINATE") then the predicate PRED is not imposed after the interruption of the action. Instead a predicate, denoted PRED_s, associated with the structured jump, is imposed and the execution of [5.10] completed. Let us consider the places which are marked after the imposition of PRED_s in the three different cases:

"ADVANCE": The END place of [5.10] is marked.

"CONCLUDE": The END place of the activity containing [5.10] is marked.

"TERMINATE": The END place of the prime task of the object executing [5.10] is marked.

Thus in the case of "ADVANCE" [5.9] is invalidated, whereas in the last two cases it is not. Based on this and our experience with Delta we propose simply to dispense with "ADVANCE". People who want some further motivation may read the following "excurses", others may skip them.

EXCURS 1: "ADVANCE" VERSUS "CONCLUDE" AND "TERMINATE"

The problem [Delta 75] tries to solve by introducing "ADVANCE" may be formulated in the following way

[5.11] "after the return from an interrupt it may not be meaningful to impose the predicate of the interrupted action, instead we should continue with the execution of the imperative following the interrupted action".

We have not been able to find examples of the kind described by [5.11], and we do not believe that they exist. Consider for instance the following example:

```
        pick up knife and fork;
        WHILE hungry LET {FOOD = f (F_o, TIME-T_o)}
[5.12]  EXIT (* put down knife and fork *)
```

```
REENTRY (* IF meaningless to impose PRED
           THEN (* ADVANCE LET { $PRED_A$ } *)
           ELSE (* pick up knife and fork *)
clean the table;
do the dishes.
```

If it is "meaningless to impose PRED" due to the fact that the interrupt from which we returned consisted of the burning down of the house, then we should certainly not continue with the action "clean the table". Rather the whole activity should be CONCLUDED.

In fact all the examples we know of on the use of "ADVANCE", in situations where it is not meaningful to impose the predicate of the interrupted action, are of the same kind as [5.12] above, i.e. they gain from being rewritten using "CONCLUDE", (or "TERMINATE").

EXCURS 2: OTHER USES OF "ADVANCE".

During our search for an example on a reasonable use of "ADVANCE" we found that the major part of the examples didn't concern situations where it was essential to avoid the imposition of the interrupted action regardles of the value of the boolean expression in its duration clause. This is illustrated by the following example from [Delta 75, p. 468]:

```
        "LET {TRUE}
        REENTRY (* ADVANCE *);
[5.13]  I 1;
        I 2;
        .
        .
        .
```

[which] describes that the object will wait, (doing nothing) until it is interrupted. When the interrupting task has been executed, the execution will continue with the imperative I 1". (This is a formal description of "await" used in chapter 2. p.11).

We find it reasonable to keep the "duration information" inside the duration clause, instead of distributing it throughout three different clauses as invited to by the ADVANCE-imperative. That is, we propose to rewrite examples that use "ADVANCE" in the same way as in [5.13] such that the boolean expression of the duration clause includes information about whether the action has been interrupted or not.

[5.13] may be rewritten in the following way :

```
.
.
.
INTERRUPT: = FALSE;
WHILE NOT INTERRUPT LET { TRUE }
REENTRY (* INTERRUPT: = TRUE* );
.
.
.
```

(End of excurses).

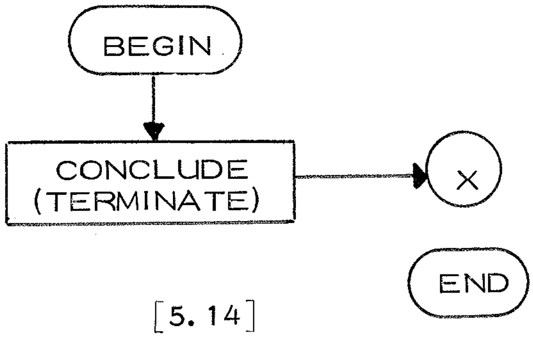
STRUCTURED JUMPS AND PREDICATES.

Except for syntactic sugar we have three different concurrency imperatives left: "WHILE", "CONCLUDE" and "TERMINATE".

The last two combine the imposition of a predicate, i.e. the establishing of a concurrent state, with a structured jump.

We propose to separate these two concepts. This is done by changing "CONCLUDE" and "TERMINATE" to event imperatives describing structured jumps. "WHILE" is then the only concurrency imperative.

The new semantics of the event imperative "CONCLUDE" ("TERMINATE") is represented by:



[5.14]

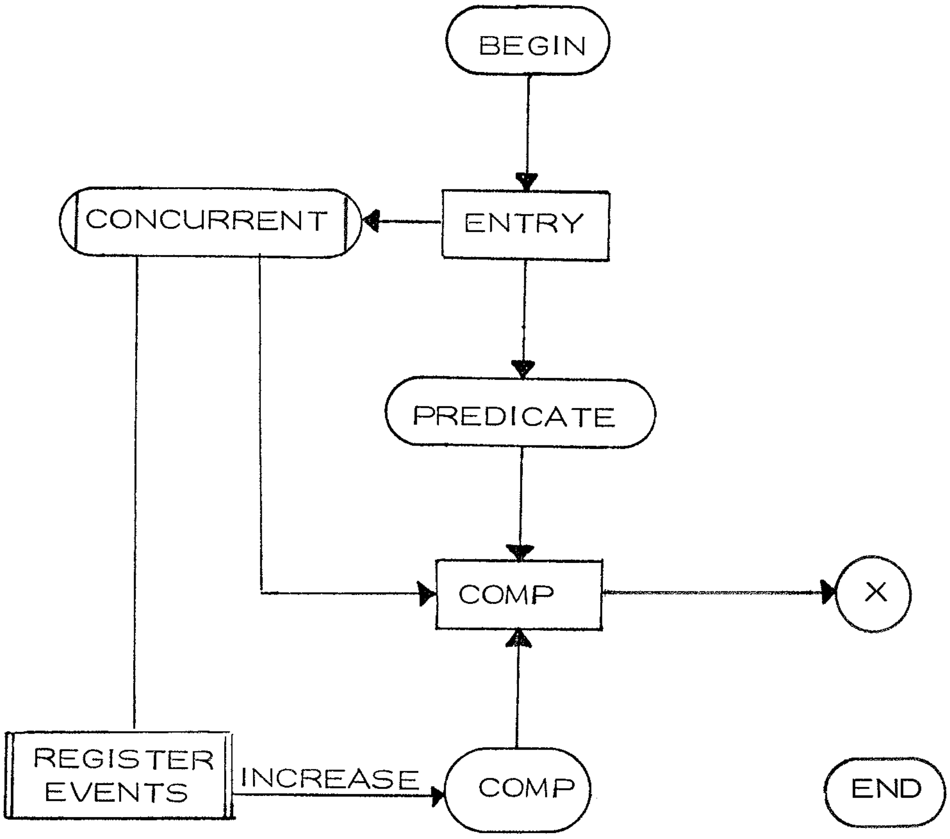
where X is END place of the activity containing the CONCLUDE-imperative (END place of the object's prime task).

We cannot model the new semantics of "CONCLUDE" ("TERMINATE") with the old since with the old, we have to establish a concurrent state. Depending on the actual nesting in the activity in question it may or may not be possible to model the new semantics by means of a "GOTO".

The old semantics of

[5.15] <u>CONCLUDE</u> (<u>TERMINATE</u>) <u>LET</u> {PRED}

is represented by

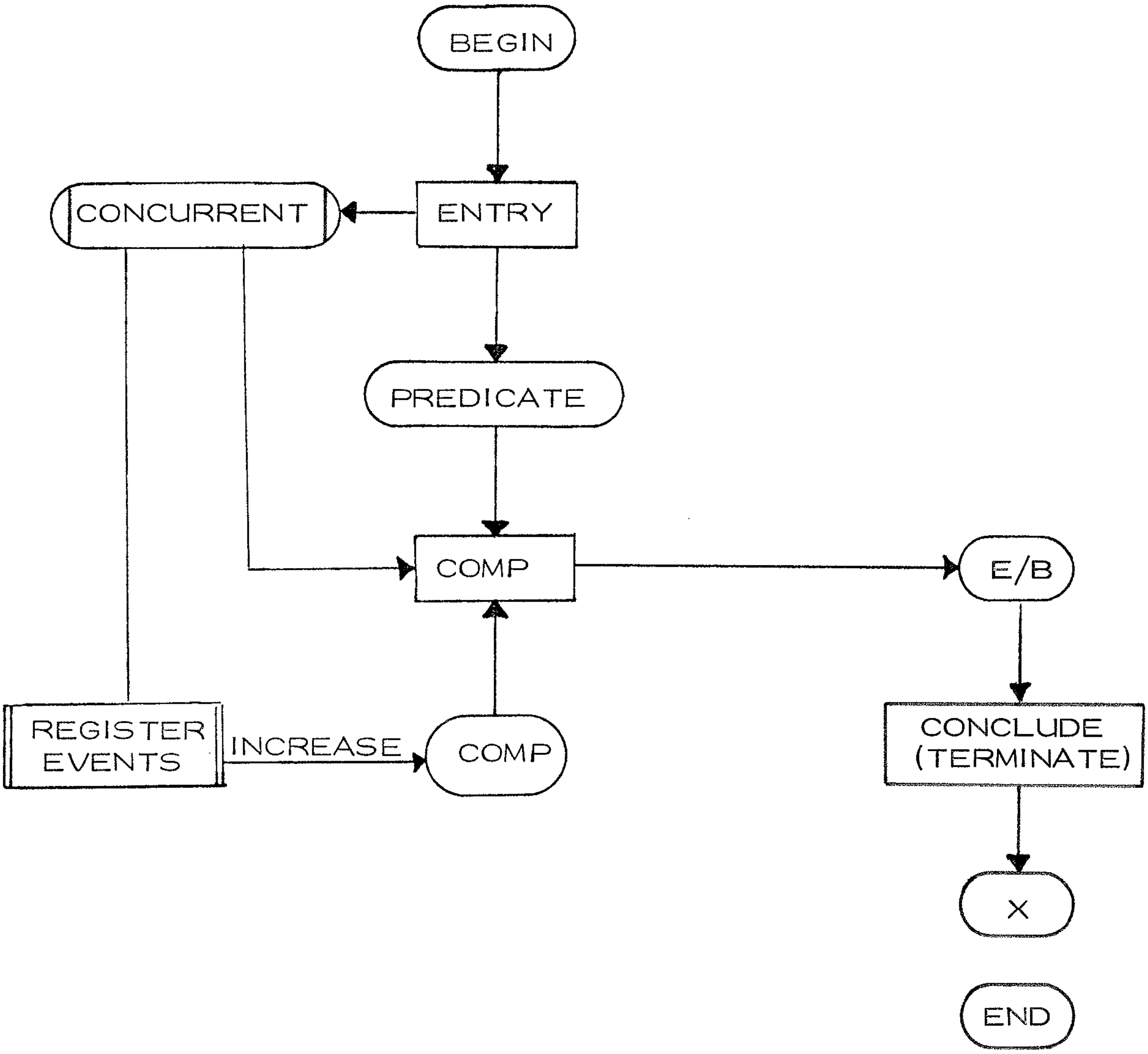


[5.16]

The new semantics of

[5.17] PAUSE LET {PRED} ;
CONCLUDE (TERMINATE)

is represented by:



[5. 18]

The resultant snapshot of [5. 16] is equivalent to the resultant snapshot of [5. 18].

CHANGES AND ACHIEVEMENTS.

Below we sum up the changes discussed in section 5.1:

1. Introduce four categories of imperatives (event, concurrency, composite, and task). This was already done in our introduction to Mini-Delta in chapter 2.

2. Simplify the semantics of "WHILE".

3. Skip one of the imperatives intended for "structured jumps" ("ADVANCE") and change the other two ("CONCLUDE and TERMINATE") to event imperatives.

4. Skip the subdivision of concurrency imperatives (time and instant).

By these changes we achieve the following:

1. A consistent classification of imperatives into welldefined disjoint categories.

2. A strong simplification of the relations between different kinds of concurrency imperatives (now all concurrency imperatives are syntactically sugered versions of "WHILE").

3. Improved verification rules (or stated differently,imperatives which are easier to understand).

5.2 Creation and destruction of objects

In this section we discuss a number of changes related to the part of the semantics dealing with creation and destruction of objects.

Our primary aim is to treat creation and destruction symmetrically. It turns out that it is possible to do this and at the same time simplify the semantics of these concepts considerably.

The main changes consist of

1. The introduction of a LEAVE-imperative describing explicit destruction of objects.

2. The abolition of initiation and encloser termination events.

Furthermore we show how to remove the division of the object phase INSIDE into ACTING and TERMINATED.

A SYSTEM DESCRIPTION EXAMPLE

Below we present a system description argument in favour of a symmetrical treatment of creation and destruction of objects by considering the post office system from [Delta 75, p. 39-51].

The post office contains three clerks and a varying number of customers needing service from the clerks.

A customer is informally described in the following way [Delta 75, p. 47]:

[5.19]

> Customer,
>
> when entering the post office he behaves as follows : While he needs more service he repeats these actions : he consults his list of yet not completed tasks, and then he selects a suitable queue, enters the queue and waits till he is at the front of the queue, then he participates in the completion of the desired service, and afterwards he leaves the queue. Then, the list being empty, he leaves the post office.

In order to formalize this description in the Delta language it is reasonable to demand that the language contains constructs which allow us to describe explicitly :

[5.20]

a) The creation of an object (corresponding to "when entering the post office").

b) The destruction of an object (corresponding to "Then, ..., he leaves the post office").

EXPLICIT CREATION AND IMPLICIT DESTRUCTION

In this subsection we discuss the Delta semantics as it is defined in [Delta 75].

An object is INSIDE a system from the moment it ENTERs and until it is REMOVEd.

An object ENTERs a system when the associated NEW imperative is executed (or when the system object is generated). This covers [5.20a].

The destruction of an object cannot be described explicitly in the present Delta language and thus [5.20b] is not covered.

[Delta 75, p.119] contains the following argument against explicit destruction:

[5.21] "We want to secure that actions do not become meaningless during their execution because the attributes used in their specification cease to exist".

This argument then leads to the following "retention rule" [Delta 75, p.120]:

[5.22] "a component is retained as long as it is perceived by at least one actor".

This was rephrased to our terminology as SYN 3 in [4.7]:

> An object which is perceived (can be referenced) directly or indirectly by an ACTING object cannot be REMOVEd.

[Delta 75] contains a detailed analysis of conditions under which objects may be removed without violating [5.22]. As discussed in connection with the formulation of SYN3' (cf [4.8], p.63) these considerations are truly within the realm of garbage collection and has no significance for the semantics.

We do not consider [5.21] as a valid argument against explicit destruction of objects. In fact we regard [5.21] as providing the wrong kind of security.

Consider the following example:

[5.23] A customer object A, which has completed all its tasks is TERMINATED in order to indicate that the simulated "real-world" customer leaves the post office (cf [5.1]). Due to an erroneous system description A is later referenced by the object B.

Obviously [5.23] is a logical error. But [5.21] prohibits it from being recognized as a semantical error.

THE DISTINCTION BETWEEN "ACTING" AND "TERMINATED"

When an object is INSIDE the system it may, in [Delta 75], be either ACTING or TERMINATED (cf [4.4], p.60).

When an object is ACTING it possesses attributes and performs actions When the object is TERMINATED it possesses attributes, but it cannot perform actions (and once an object is TERMINATED it cannot become ACTING again).

Objects are TERMINATED for two different reasons:

a) When a system reporter wants to destroy an object, the most he can do is to TERMINATE it, due to [5.21].

[5.24] b) When a system reporter wants to secure or emphasize that an object will execute no more actions, although its attributes may be used by other objects, then he may TERMINATE it.

A good example of an object TERMINATED as described in [5.24a] would be a formalized description of the customer objects described in [5.19]. The main example of an object TERMINATED as described in [5.24b] are objects with no prime task (i.e. "record-like" objects). They never become ACTING but are TERMINATED when ENTERed into the system.

Instead of TERMINATEing an object a system reporter could let it execute a neutral action, like LET {TRUE}. However, he must assure that the object will not accept interrupts. This may be achieved by means of priorities.

We introduce a new standard priority, TOP, which no interrupts can penetrate. ([Delta 75] defines only a "bottom-element" for the set of priority values; we denote this element by the keyword BOTTOM).

When a system reporter wants to secure [5.24b] he should let the object execute

```
[5.25]     LET{TRUE} PRIORITY TOP .
```

The execution of [5.36] will continue as long as the object exists, it cannot be interrupted and it does not change the state of the system.

We propose that objects with no explicitly described prime task by default get the prime task described by :

```
           TASK BEGIN
[5.26]          LET {TRUE} PRIORITY TOP
           END TASK
```

ENCLOSER TERMINATION

As the last subject, before presenting our proposals concerning creation and destruction of objects, we consider encloser termination events. They are introduced in order to satisfy the following rule [Delta 75, p. 121] :

[5.27] "a component terminates when a member of its lineage terminates".

This we rephrased to our terminology as SYN4 in [4.7] (on page 62):

> "When an object TERMINATEs all objects in its content TERMINATE at the same moment of model-time".

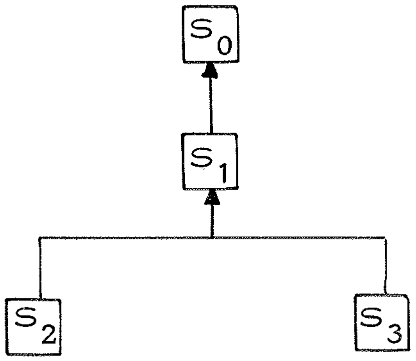
The reason for introducing [5.27] (SYN4) is that it should be possible for an object being the bound of a subsystem to "achieve by its own actions the termination of the subsystem" [Delta 75, p. 121].

We agree with this argument as presented in [Delta 75], and in the derived rule [5.27] but not in the technical solution reflected in the semantics defined in [Delta 75]. With that semantics it is in fact possible for an object to execute an unlimited number of events although it is contained in a subsystem whose bound is TERMINATED. This is illustrated by the following example (which may be skipped, if one is not interested in the technical details).

EXAMPLE

Consider a system represented by the diagram [5.28]:

[5.28]



Let us assume that the four singular objects are operating in the concurrent state, CS, at model-time t, and that there are registered

- a completion event for S_0
- completion or interruption events for S_i , i = 1, 2, 3.

Let the first event executed after CS be the completion event for S_0 and assume that it ends by S_0 being TERMINATED.

As stated in SYN4 this implies that all the objects in the (sub)system with S_0 as bound should TERMINATE at this moment of model-time, t. But S_1 has to TERMINATE before encloser termination events can be registered for S_2 and S_3. In order to TERMINATE, S_1 must at least execute one event, and before this happens S_2 and S_3 may execute an unlimited number of events as illustrated below:

Assume that S_2 is executing the compound imperative

```
          WHILE BOO REPEAT
[5.29]    (*   PAUSE LET {TRUE};
               INTERRUPT R S3 BY FAST
          *)
```

and that S_3 is executing

```
[5.30]    PASSINGLY LET {TRUE}
```

where R S3 is a reference to S_3 and FAST is a not time assuming task procedure.

The execution of [5.29] ends when BOO becomes false and the execution of [5.30] when there are no interrupts on the agenda of S_3, or – for both of them – when encloser termination events are registered and executed due to the fact that S_1 is TERMINATED.

OUR PROPOSAL : EXPLICIT CREATION AND EXPLICIT DESTRUCTION

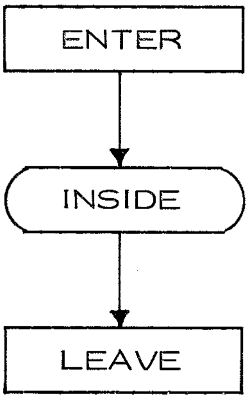
Following the discussion in the previous subsections we propose to introduce a new event imperative for the description of explicit destruction of objects.

This imperative has the syntax, LEAVE, and it replaces the TERMINATE-imperative.

The LEAVE-imperative covers [5.20b] and it allows [5.23] to be recognized as a semantical error.

We propose to abolish the division of objects INSIDE the system into ACTING and TERMINATED. When we do this all objects are represented by a subnet of the form:

[5.31]

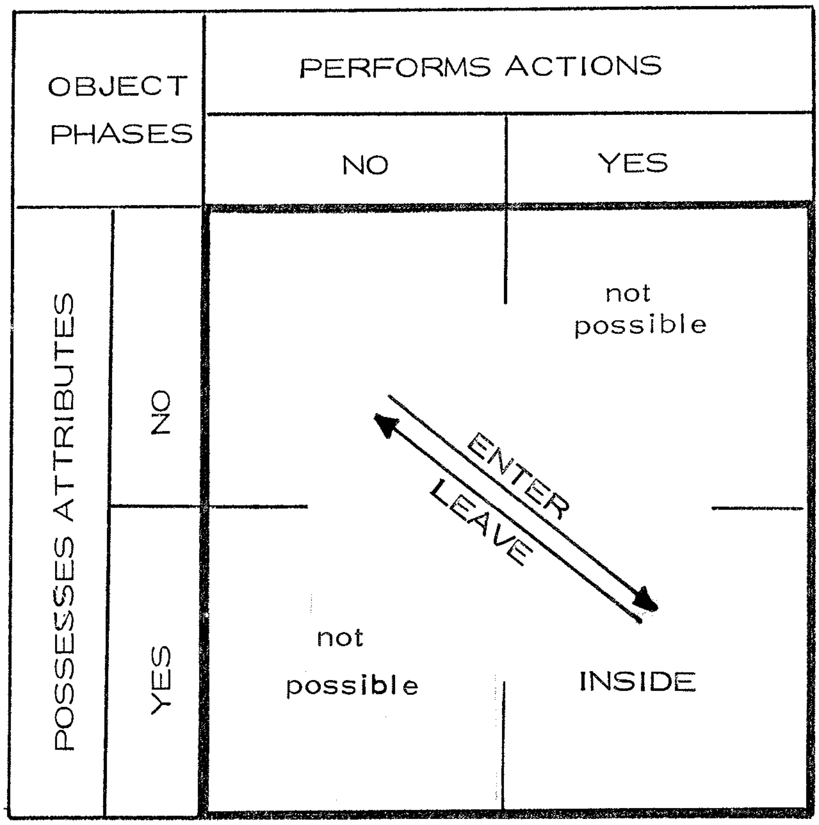


(cf [4.2] and [4.3] on p. 58 and 59).

LEAVE can be seen as the conjunction of the old TERMINATE and REMOVE transitions.

The phase shifts can now be summarized in the following simplified diagram (cf [4.4] on p. 60):

[5.32]



We can find no arguments in favour of the slow, step-wise TERMINATEion of a subsystem. We propose the following reformulation of SYN4 :

[5.33] When an object LEAVEs, as a part of the execution of the imperative, IMP, all objects in its content LEAVE as a part of the execution of IMP.

This implies that a whole subsystem is destroyed as a part of the execution of an event imperative ("LEAVE") in the same way as a whole litter is created as a part of the execution of an event imperative ("NEW"). I. e. we take an important step towards a symmetrical treatment of creation and destruction of objects, and we avoid the slow, step-wise TERMINATEion of a subsystem described in the example.

INITIATING THE PRIME TASK

In this subsection we consider the situation immediately after the creation of an object as it is defined in [Delta 75] and discuss how to simplify it.

When a NEW-imperative is executed as a part of an event, E, a litter consisting of one or more objects is generated, i. e. one or more objects ENTER the system.

The objects having prime tasks are from then on INSIDE, but not operating (cf. p. 68).

In the concurrent state, CS, following the execution of E, events are registered and among these initiation events. Each time an initiation event is executed one object becomes operating.

We consider this step-wise initiation as unnecessary and difficult to understand. We can find no arguments in favour of the sub-phase "INSIDE but not operating" and we propose to abolish it.

In order to do this we introduce the following convention :

we will by default insert
[5.34] "PAUSE LET {TRUE} PRIORITY TOP ; "
as the first imperative in an object's prime task.

We are now able to change the creation of an object in such a way that it immediately begins executing the first imperative in its prime task.

With this change initiation events are no longer needed. Apart from avoiding "INSIDE but not operating" objects, the dismissal of the initiation events is also one of the changes needed to achieve a symmetrical treatment of creation and destruction of objects.

TECHNICAL NOTE 1

"Default insertions" like the one in [5.34] and those discussed in [5.36] below should be described using prefixes. However, since we do not treat this mechanism in chapter 4 we will not use it here.

TECHNICAL NOTE 2

Objects with no prime task should, prior to the application of [5.34], be given the prime task described by [5.26] (p. 111).

As a consequence objects with no explicitly described prime task, will execute one event (their prime task will be described by

```
TASK BEGIN
    PAUSE LET {TRUE} PRIORITY TOP;
    LET {TRUE} PRIORITY TOP
END TASK
```

cf [5.26] and [5.34]).

If one considers this consequence as undesirable, it may be avoided by changing [5.34] to :

[5.35] If an object's prime task does not begin with a concurrency imperative, then we will by default insert

PAUSE LET {TRUE} PRIORITY TOP;

as the first imperative.

FINISHING THE PRIME TASK

When the end of an object's prime task is reached, there are at least four different possibilities. In all four cases the semantics can be described by means of existing constructs in the language, i.e. by (implicitly) inserting an imperative at the end of the prime task:

[5.36]

1. It may imply that the object is forced to LEAVE the system. This is described by inserting the imperative LEAVE.

2. It may constitute an error, i.e. the system reporter did not want the object to finish its prime task. This is described by inserting the imperative LET {FALSE}.

3. It may imply that the object no longer can change the state of the system, although its attributes may be used by other objects. This is described by inserting the imperative LET {TRUE} PRIORITY TOP.

4. It may imply that the object no longer can change the state of the system, by executing imperatives in its prime task, but only by executing tasks imposed upon it by interrupts from other objects. This is described by inserting the imperative LET {TRUE} PRIORITY BOTTOM.

Possibility 1 and 3 correspond to [5.24a] and [5.24b] respectively (see p. 110). Possibility 3 resembles the semantic choice made in [Delta 75].

One may use a security argument as the one presented in connection with [5.23](p. 109) in favour of possibility 2.

Finally possibility 4 is illustrated by data objects, which, after some initial actions, are ready to perform operations on themselves.

Irrespectively of the solution chosen, a system reporter may make his own choice simply by explicitly inserting the appropriate one of the four alternatives listed in [5.36] as the last imperative in the prime task.

The actual choice is thus of less importance. We prefer the second.

CHANGES AND ACHIEVEMENTS

Below we sum up the changes discussed in section 5.2 :

1. Introduce a LEAVE imperative describing explicit destruction.
2. Abolish initiation and encloser termination events.
3. Abolish the division of INSIDE into ACTING and TERMINATED.

By these changes we achieve the following :

1. A symmetrical and simplified treatment of creation and destruction of objects.
2. A reduction of the number of different kinds of events.
3. A reduction of the number of different kinds of object phases.

By these changes the four synchronization-constraints defined in [4.7] must be reformulated as follows:

SYN1* : All objects in a litter ENTER the system together (old SYN1).

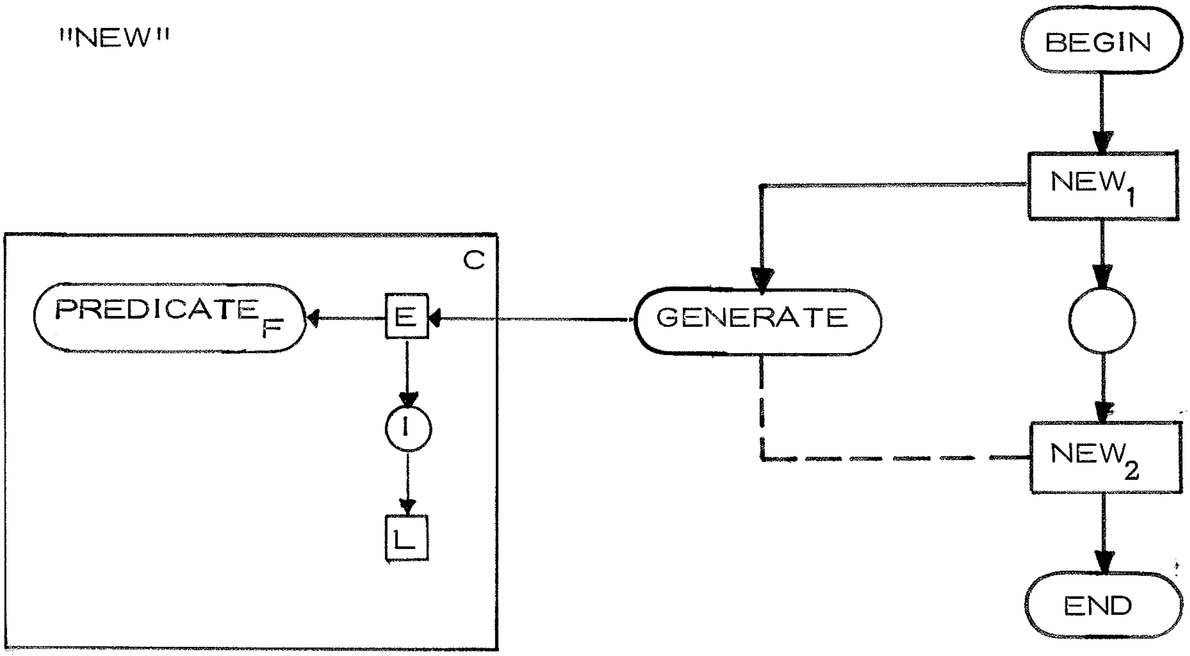
SYN2* : A litter can only ENTER the system if the primary object's encloser is INSIDE (old SYN2).

SYN3* : When an object LEAVEs, as a part of the execution of the imperative, IMP, all objects in its content LEAVE as a part of the execution of IMP ([5.33]).

In the following subsection we present the modified subnets corresponding to these changes

MODIFIED SUBNETS REPRESENTING "NEW", "LEAVE", EVENT REGISTRATION, AND EVENT EXECUTION

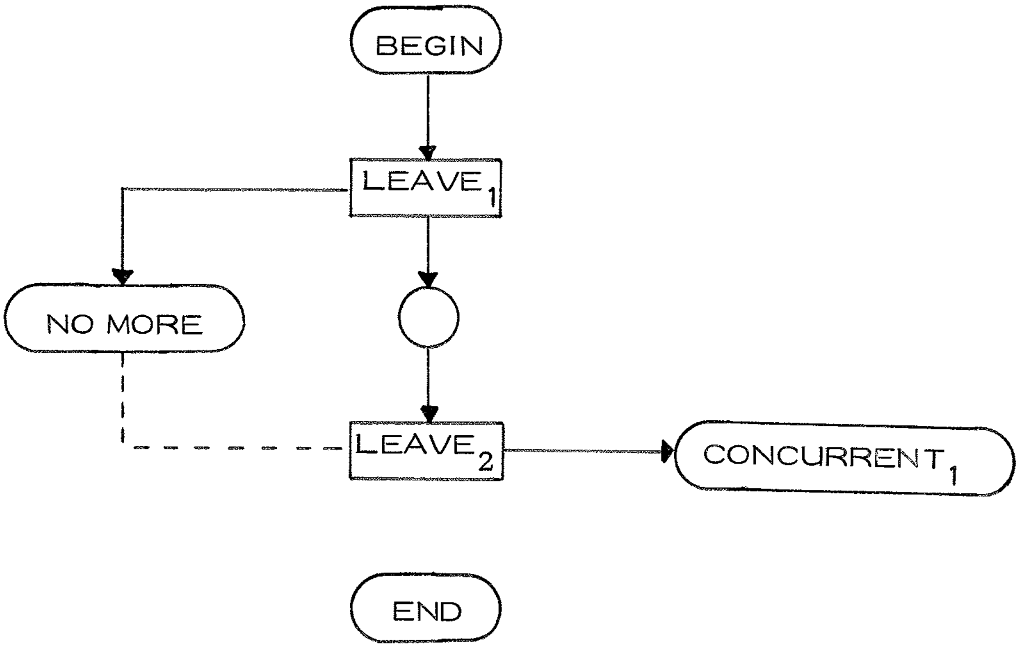
"NEW"



[5.37]

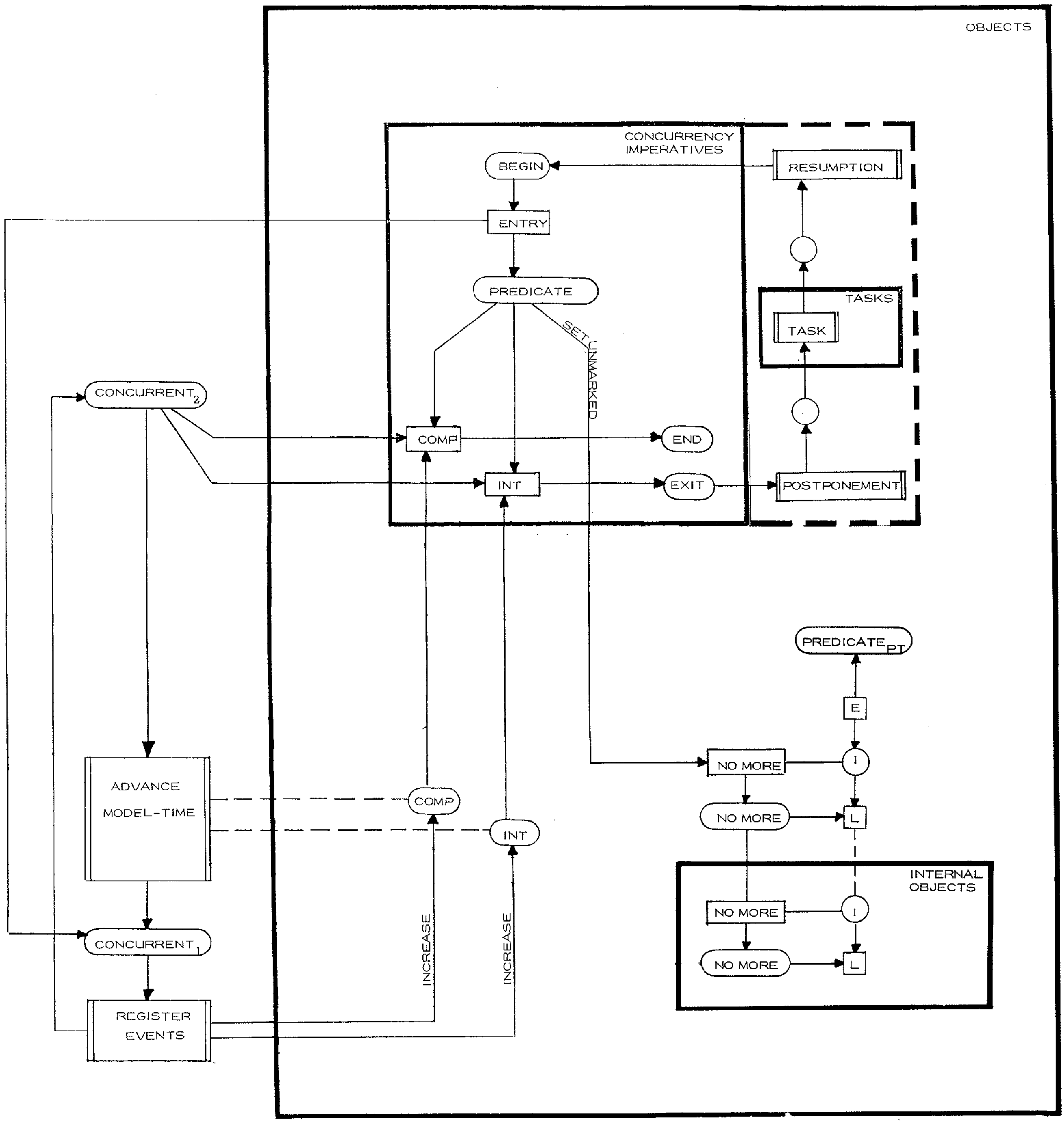
PREDICATE_F is the PREDICATE place belonging to the first imperative in the prime task of the generated object.

"LEAVE"



[5.38]

This net must be seen in connection with the modified version of [4.15] representing event registration and event execution :


OBJECTS
CONCURRENCY IMPERATIVES
BEGIN
RESUMPTION
ENTRY
PREDICATE
TASKS
TASK
SET UNMARKED
CONCURRENT$_2$
COMP
END
INT
EXIT
POSTPONEMENT
PREDICATE$_{PT}$
E
I
NO MORE
NO MORE
L
ADVANCE MODEL-TIME
COMP
INT
INTERNAL OBJECTS
NO MORE
I
NO MORE
L
CONCURRENT$_1$
INCREASE
INCREASE
REGISTER EVENTS

[5. 39]

The NO MORE place in [5.38] is identical to the NO MORE place in [5.39] belonging to the object executing the LEAVE-imperative.

It should be noticed that a LEAVE-imperative, IMP, executed by an object, OB_1, cannot be finished (firing of IMP's $LEAVE_2$) until all objects in OB_1's subsystem are outside (their INSIDE places unmarked). Thus SYN3* is satisfied.

For the system object, S_0 [5.39] contains a technical problem. S_0 has no encloser and thus S_0's NO MORE transition has concession whenever S_0 is INSIDE. This problem can be solved merely by entirely removing S_0's NO MORE transition(and all arcs connected to it) from the net representing the considered system. This works since S_0 only can LEAVE when it executes a LEAVE-imperative itself (and then all S_0's PREDICATE places are already unmarked).

5.3 Registration and execution of events.

In this section we first discuss event registration and execution as defined in [Delta 75]. Then based on the discussion, we propose to eliminate event registration. Our proposal is strongly influenced by the nondeterministic firing rules for Petri nets and resembles the ideas behind guarded commands defined in [Dijkstra 75]. The proposal constitutes a major simplification of the semantics, it deals symmetrically with completion and interruption events, and we obtain a stronger proof rule for the WHILE - imperative. For the discussion in this section we will assume that the changes proposed in the two preceding sections have been made.

REGISTRATION AND EXECUTION IN DELTA 75.

In order to describe the solution chosen in [Delta 75] we first introduce the concepts completion state and interruption state.

A concurrent state is a completion state for an object iff the object is executing a concurrency imperative with a boolean condition which is false.

A concurrent state is an interruption state for an object iff the object is executing a concurrency imperative with a resistance, which is penetrated by an interrupt on the agenda of the object.

A completion event is registered for an object in a given state iff

a) the state is a completion state for the object and
b) the object has no registered events.

An interruption event is registered for an object in a given state iff

a) the state is an interruption state for the object,
b) the state is not a completion state for the object, and
c) the object has no registered events.

These rules implies that each object may have at most one registered event in each system state. A registered event is executed before model-time advances. If two or more events are registered (for different objects) in the same system state a nondeterministic choice is made to select one of the registered events for execution. The other events remain registered.

ARGUMENTS AGAINST THE SOLUTION IN DELTA 75.

We have several arguments against the way in which events are registered and chosen for execution in [Delta 75].

First of all the solution in [Delta 75] is complicated, and it is possible to obtain a stronger proof rule by eliminating event registration (cf. the following subsection).

Secondly. When several different objects are ready to execute events a nondeterministic choice is made to select one for execution. When a state is both a completion state and an interruption state a deterministic rule is used always to register a completion event. In our opinion it is ad hoc to resolve conflicts between events in different objects by nondeterminism when conflicts betwen events in the same object are resolved by determinism. Moreover it is difficult to see why completion events should be favoured at the expense of interruption events. Indeed there are system descriptions in which this would be natural. But there are other system descriptions in which the opposite would be more natural. Thus such a choice between different kinds of events should be reflected in the actual system descriptions themselves. The semantics of a language should merely specify a nondeterministic choice.

Thirdly. In [Delta 75] each object may have at most one registered event in each system state. This is assured by favouring an already registered "old" event at the expense of "new" events, which could otherwise be registered for that object.
It seems ad hoc to favour the registration of "new" completion events at the expense of "new" interruption events, but not at the expense of "old" interruption events. In particular this means that the "deterministic " choice between completion and interruption events may depend heavily on the nondetermistic way in which registered events are chosen for execution, as illustrated by the following example:

Consider a system with at least 3 objects: OB_1, OB_2 and OB_3. The system is in a concurrent state where events are registered for OB_1 and OB_2, but not for OB_3. The event for OB_1 affects the system state in such a way that it becomes a completion state for OB_3. The event for OB_2 affects the system state in such a way that it becomes an interruption state for OB_3. Assume that no other than the four mentioned events can be registered at the considered moment of model-time.

Dependent on the order in which registered events are chosen for execution OB_3 may now execute either a completion event, an interruption event, or both.

ELIMINATION OF EVENT REGISTRATION.

Based on the discussion in the previous subsection we propose the following:

1) Eliminate event registration.

2) When a state is completion state or interruption state for more than one object, or when it is completion state and interruption state for the same object a nondeterministic choice is made to select one of the corresponding events for execution.

This proposal constitutes a major simplification of the semantics. Moreover we obtain a stronger proof rule for the WHILE-imperative

Consider a concurrent imperative WHILE BOO LET {PRED} and remember the corresponding proof rule [5. 9] (p. 100), which read:

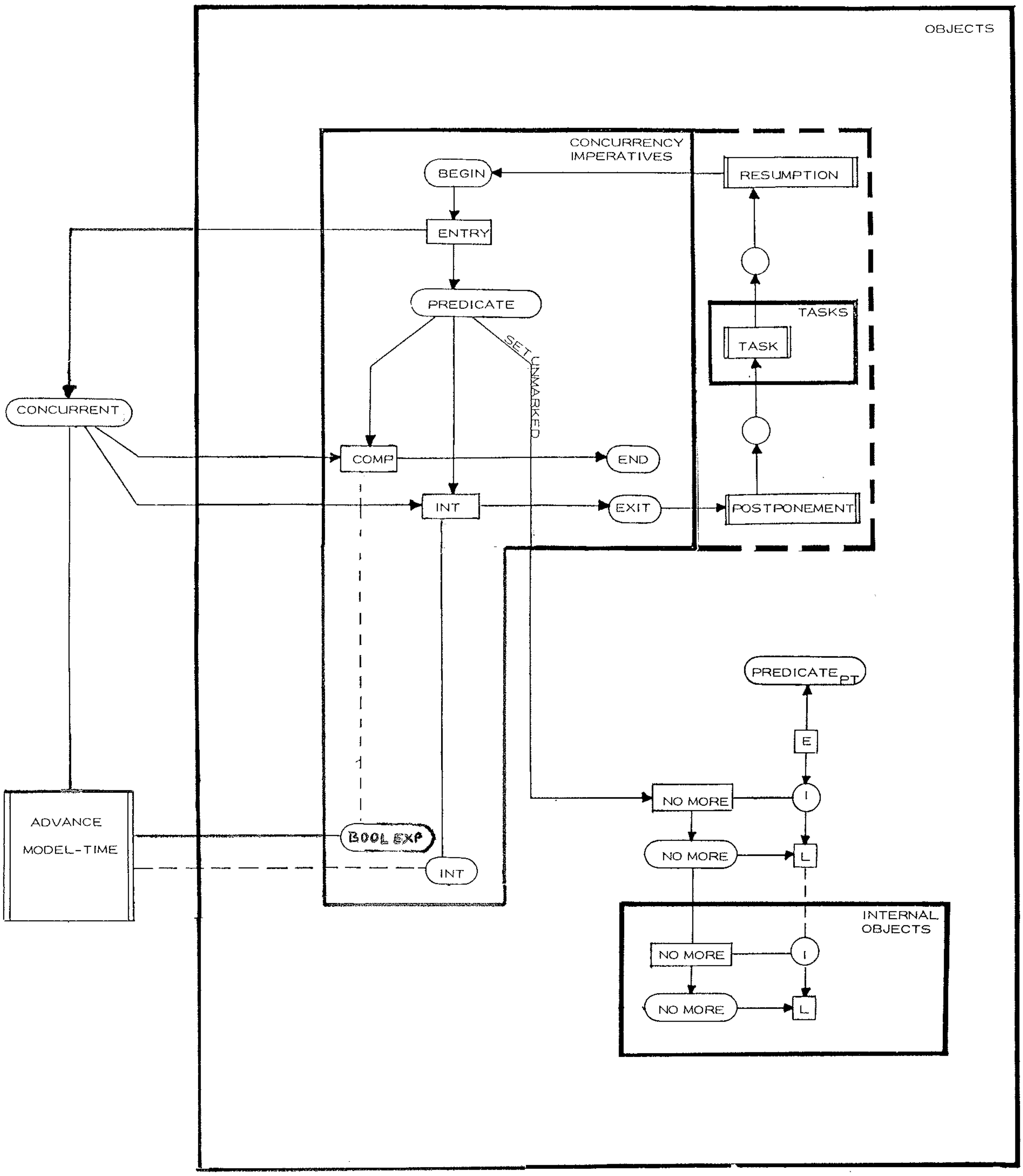
When the place END is marked the predicate PRED holds.

Now it is possible to replace this by

[5. 40] When the place END is marked the predicate PRED and non BOO holds.

Earlier this rule could not be achieved, because the execution of other events might reestablish BOO between the registration of a completion event and its execution.

MODIFIED SUBNET REPRESENTING SELECTION OF EVENTS FOR EXECUTION.



[5.41]

The place INT (BOOL EXP) is (un)marked in a concurrent state iff the state is an interruption (completion) state for the object.

In this paper it is not modelled how the correct marking of these places is maintained during system execution.

The BOOL EXP place is analogous to the BOOL EXP places used on page 83 and 86-88.

Chapter 6

CONCLUSION

In chapters 4 and 5 we have used extended Petri nets to describe, analyse, and design a semantics for Delta.

This kind of approach must be evaluated in at least two different ways :

1. By the value (i. e. readability, exactness, completeness etc.) of the semantic description in chapter 4 or a similar description containing the changes proposed in chapter 5.

2. By the value of the proposed changes themselves.

With respect to 1. it is primarily up to the users of the semantic descriptions to evaluate the degree of success or failure.

Our opinion is that Petri nets (with certain extensions) may be a valuable tool in the future definition of semantics. However, we certainly do not claim that our approach is optimal, complete or finally developed.

We think that some of the semantic changes proposed in chapter 5 are major and add considerably to the clarity, exactness and consistency of the Delta language.

This is confirmed by the fact that it always turned out to be possible to give "system description arguments" for the changes which originated by "net arguments".

We close this paper by pointing out a number of topics where further work has to be done.

PETRI NETS ARE STATIC

It is not possible to add or remove subnets dynamically. This implies that all places (transitions) which may be marked (fire) must be included in the net from the very beginning.

The relationship between static nets, infinite virtual storages, and open procedure calls has already been indicated (see pages 58 and 88).

[Keller 76] allows multiple markings and uses the same subnet to represent all instances of a set of processes belonging to the same category. Then each subnet may contain tokens belonging to different processes and each token carries a label identifying the process "owning" it.

In [Keller 76] transition firings merely copy the labels. This could be extended such that transition firings may create new labels.

Doing this it would for instance be possible to represent recursive tasks (see page 89) by finite subnets (label each token with a number defining the dynamic level of the corresponding call). It would also be possible to avoid representing each class of objects by an infinite number of subnets (see page 61).

MODELLING THE DATA PART

We have not modelled quantities explicitly in this paper. This can be done using the formalisms proposed in [Keller 76], and [Mazurkiewicz 77].

As mentioned in chapter 4 (see page 55) machine states can be partitioned into memory states and control states. Keller uses a set of program variables to model the memory states and a Petri net to model the control states (called place variables). Each transition, t, has attached to it an expression of the form

$$\underline{\text{when}}\ P_t\ (\xi)\ \underline{\text{do}}\ \xi \leftarrow F_t\ (\xi),$$

where ξ is the program variables, P_t a predicate and F_t a function. If t fires, the memory state is changed by assigning the value of $F_t\ (\xi)$ to ξ.

The control state is the marking of places. It may be changed by firing of transitions.

Quantities could be represented as program variables and the imperatives of the Delta objects as nets. Changes of the quantities caused by the usual assignment imperative can now be easily modelled. Predicates of concurrency imperatives do not fit into this. In a later subsection we consider this. First we consider a difference between the nets defined in [Keller 76] and Petri nets.

DISJOINT RESOURCES

In Petri nets transitions may be concurrent. In Keller's formalism they fire one by one and concurrent transitions would be a problem since expressions attached to transitions may refer to the same program variables.

It would, however, be possible to allow concurrent transitions if the expressions attached to these transitions use disjoint parts of the program variables. Such a requirement seems to be reasonable since concurrent transitions in Petri nets require disjoint conditions.

With this requirement it turns out that Keller's formalism may be viewed as a concurrent system as defined in [Mazurkiewicz 77]. A concurrent system consists of a net and a set of resources (which could be Keller's program variables). Here transitions may be concurrent only if they use disjoint resources.

Like Keller's transitions, Delta events are mutually exclusive. We would like to modify our semantics of Delta such that Delta events using disjoint parts of the system state can be concurrent. The work of Mazurkiewicz seems to be a reasonable starting point for this purpose.

PREDICATES ATTACHED TO PLACES

In our nets we have places corresponding to predicates of concurrency imperatives. When the system is in concurrent mode a marking of such a place is interpreted to mean that the system state (which we do not model explicitly) must satisfy the predicate.

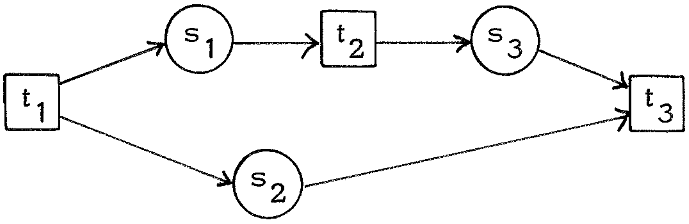
Keller's (and Mazurkiewicz's) formalism could be extended to have predicates over program variables attached to places in the same way as expressions are attached to transitions. This should mean that the predicates attached to marked places are imposed on the program variables.

The expressions attached to places would have the form

$$\underline{\text{let}}\ P_t(\xi)\ \underline{\text{def}}\ VL_t$$

where ξ is the program variables, P_t a predicate and VL_t a subset of the program variables. VL_t specifies the variables which may be altered in order to satisfy the predicate.

As an example of this consider the following net :



Let the transitions (t_i , i = 1, 2, 3) have attached the expressions

<u>when</u> B_i (ξ) <u>do</u> $\xi \leftarrow F_i$ (ξ) , i = 1, 2, 3.

and let the places (s_i , i = 1, 2, 3) have attached the expressions

<u>let</u> P_i (ξ) <u>define</u> VL_i , i = 1, 2, 3.

When t_1 fires the program variables are assigned the value of F_1 (ξ) and the predicates $P_1(\xi)$ and $P_2(\xi)$ are imposed. When t_2 fires the program variables are assigned the value of $F_2(\xi)$ and $P_2(\xi)$ and $P_3(\xi)$ are imposed. Etc.

The problem with disjoint resources must be reconsidered. An easy solution would be to require that concurrent places and transitions may only refer to disjoint parts of the program variables. In the above example this means that P_2 may not refer to variables mentioned in P_1, F_2, and P_3 and vice versa.

Such a solution is, however, not directly useable to model Delta. In Delta it is important that a number of objects together may influence the same part of the system state in concurrent mode. Such objects would then have places whose attached expressions use non-disjoint resources.

THE TIME CONCEPT

We have not been able to model the TIME concept in a satisfactory way.

In [Petri 76] the problem of modelling continuous time is discussed and a net for modelling time is given. This is, however, not useable in our semantic definition as it corresponds to a model system (or a program execution) rather than to a system description. However, it gives rise to an alternative way of describing model systems, by viewing them as nonsequential processes as defined in [Petri 76]. A discussion of this approach is outside the scope of this paper.

If we relax on the requirement to model continuous TIME and instead use discrete TIME it is easy to find a solution. Our nets could be synchronised by a clock which performs a stepwise increment of TIME. Using Keller's formalism and referring to the net [4. 15], TIME could be a program variable and the transition ADVANCE MODEL-TIME could have attached the expression

$\underline{\text{do}}$ TIME : = TIME $+\epsilon$,

where ϵ may be an arbitrarily small stepsize.

In [Kyng 76] the use of discrete TIME is discussed in connection with obtaining executable programs from DELTA descriptions.

References

[Delta 75] E. Holbæk-Hanssen, P. Håndlykken and K. Nygaard :
System Description and the Delta Language.
Norwegian Computing Center, Oslo, Norway, 1975.

[Dijkstra 75] E.W. Dijkstra:
Guarded Commands, Nondeterminacy and Formal
Derivation of Programs.
Comm. ACM 18, 8 (August 1975), 453-457.

[Genrich & Thiagarajan 78] H.J. Genrich and P.S. Thiagarajan :
Net Progress.
Computing Surveys Vol 10, No. 1 (March 1978), 84-85.

[Hoare & Lauer 74] C.A.R. Hoare and P.E. Lauer :
Consistent and Complementary Formal Theories
of the Semantics of Programming Languages.
Acta Informatica, Vol 3, Fasc 2, 1974,
Springer Verlag, 135-153.

[Holt & Commoner 70] A.W. Holt and F. Commoner :
Events and Conditions.
Applied Data Research N.Y. 1970 ; also in Record
Project MAC Conf. Concurrent Systems and
Parallel Computation, ACM, N.Y. 1970, 3-52.

[Jensen 78] Kurt Jensen :
Extended and Hyper Petri Nets.
DAIMI TR-5, August 1978.

[Keller 76] R.M. Keller :
Formal Verification of Parallel Programs.
Comm. ACM 19, 7 (July 1976), 371-384.

[Kyng 76] M. Kyng :
Implementation of the Delta Language Interrupt Concept within the Quasiparallel Environment of SIMULA.
DAIMI PB-58, August 1976.

[Lauer & Campbell 75] P. E. Lauer and R. H. Campbell :
Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.
Acta Informatica 6 (1975), 297-332.

[Mazurkiewicz 77] A. Mazurkiewicz :
Concurrent Program Schemes and their Interpretation.
DAIMI PB-78, July 1977.

[Nygaard 73] K. Nygaard :
On the Use of an Extended SIMULA in System Description.
Norwegian Computing Center, Oslo, Norway, 1973.

[Pearl 78] E. Wegner and C. Hopmann :
Semantics of a Language for Describing Systems and Processes.
IST Report 36. Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Mai 1977 (revised January 1978).

[Peterson 77] J.L. Peterson :
Petri Nets.
Computing Surveys Vol 9, No. 3, (September 1977), 223-252. (see also [Genrich & Thiagarajan 78]).

[Petri 62] C.A. Petri :
Kommunikation mit Automaten.
Schriften des Rheinisch-Westfalischen Institutes für Instrumentelle Mathematik an der Universität Bonn, Heft 2, Bonn W. Germany 1962 ; Translation : G.F. Greene, Supplement 1 to Tech. Rep. RADC-TR-65-337, Vol 1, Rome Air Development Center, Griffiss Air Force Base, N.Y. 1965.

[Petri 73] C.A. Petri :
Concepts of Net Theory.
Proc. Symp. Summer School on Mathematical Foundations of Computer Science, High Tatras, Sept. 3-8, 1973, Math. Inst. Slovak Academy of Science, 1973, 137-146.

[Petri 75] C.A. Petri :
Interpretations of Net Theory.
Interner Bericht 75-07, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, W. Germany, July 1975.

[Petri 76] C.A. Petri :
Nichtsequentielle Prozesse.
Interner Bericht 76-6, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, W. Germany, June 1976. (translated into English by P. Krause and J. Low).

[Simula 70] O.-J. Dahl, B. Myhrhaug, and K. Nygaard :
Common Base Language.
Norwegian Computing Center, Oslo, 1970.

[Wirth 77] N. Wirth :
What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions.
Comm. ACM 20, 11 (November 1977), 822-823.