

ISSN 0105-8517

EPSILON
A SYSTEM DESCRIPTION LANGUAGE

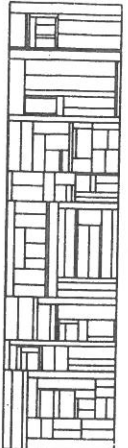
Kurt Jensen
Morten Kyng

DAIMI PB-150
October 1982

PB-150

K. Jensen & M. Kyng: EPSILON

TRYK: DAIMI/RECAU



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

ABSTRACT

This paper presents the Epsilon language and defines its formal syntax and semantics. Epsilon is a language for the description of systems, which contain concurrent components, some of these being edp-equipment or by other means representing highly structured information handling. The actions consist of continuous changes described by equations, of communication between the components and of normal algorithmic actions.

Epsilon may be used for the description of computer systems together with their environments, e.g. production equipment and human operators. Parts of such a description may serve as the system specification from which computer programs are developed. Epsilon is not itself an implementable language.

This paper defines the semantics of Epsilon by means of a model based on high-level Petri nets, i.e. a model founded on the notion of concurrency. The model also uses denotational semantics and equation systems.

Keywords and phrases: system description, concurrency, high-level Petri nets, denotational semantics.

CONTENTS

1.	System description	3
2.	Specification of attributes	19
3.	Specification of actions	39
4.	Syntax and static semantics	85
5.	Semantic model: Equation nets	99
6.	Semantics	121
	References	147

Chapter 1

SYSTEM DESCRIPTION

1.1	Systems	5
1.2	Background and intended use of Epsilon	8
1.3	Some Epsilon concepts and an example	10
1.4	Equation nets as part of the language (extension)	15
1.5	Directions of future work	16

1.1 SYSTEMS

As the societies evolve human beings have to cope with phenomena of growing complexity. In many situations a system concept is used to identify and delimit a set of phenomena, which someone wants to describe or analyse. This includes social, economical, ecological and technical systems. In a number of areas conceptual frameworks are developed to assist people in understanding and changing systems. These frameworks differ widely from one another as do the kinds of systems.

Below we sketch three different systems to illustrate variations with respect to:

- the scale of the system and the resources involved.
- the timespan over which the system exists.
- the construction of the system (is it from scratch, does the system evolve slowly and is it reasonable to think of it as "constructed" at all?).
- the stability of the system (how fixed are the rules governing the processes in the system?).

Some systems, such as the political system in Denmark, evolve over hundreds of years by the actions of millions of people. Radical changes are few. But some happen, and over the years significant changes accumulate, e.g. in the role of the royal institution.

Other systems are designed and implemented over a few years, e.g. the process control system of a plant. Are in operation for some years and then disposed of. The changes constituting the construction and destruction of a system is usually not considered

as being part of the system. In the present case different degrees of destruction of the plant may or may not be considered as invalidating our conception of the process control system.

Still other systems exist only for a brief period of time and are the result of the actions of a few people, e.g. a system consisting of two chess players. In this case any deviation from the rules of chess could be considered as being outside our conception of the system.

As indicated above each system is a part of the world. It is possible to treat it in a number of different ways and by different conceptual frameworks. The actual choice depends on the purpose for which the system is considered. Obvious choices in the three cases above would be conceptual frameworks from political science, engineering and chess.

In all three cases system description play an important role in understanding the considered system: the process of making a system description often adds considerably to ones understanding, as do the reading and discussion of a system description. In some cases system descriptions may be more or less formally analysed to deduce properties of the system [Kyng, 82]. For many consciously constructed systems the construction is based on system descriptions.

When direct analysis of the system description is complicated, computer simulation may often be useful. For example simulation has been used to predict voter-response to different kinds of campaign strategies. For the process control system, testing of the involved programs may be viewed as simulation. And in the chess example simulation may be used to create chess-robots.

A simulation program may also be viewed as a system description, and Epsilon is heavily influenced by the description tradition and the conceptual framework of Simula [Dahl, Myhrhaug & Ny-

gaard, 70]. It was observed that the process of making and discussing a simulation program often is more important in understanding a system than the results from running the program. This, and the inadequacy of existing descriptive tools, led to the development of Delta [Holbæk-Hanssen, Håndlykken & Nygaard, 75], which is a language aimed solely at system description, and not itself implementable on a computer system. Delta incorporates a number of the main concepts from Simula, but tries to avoid the restrictions imposed to make Simula executable on computer systems. Delta may be used to describe a large variety of systems containing human beings, edp-equipment, physical/chemical production processes etc.

A system description language imposes by its construction (conceptual framework) a set of restrictions on any description which can be made in the language. This limits the kinds of systems, which can sensibly be described. Furthermore, when making a description, it is in general not possible (or desirable) to include all aspects of a system: a limited set of properties must be selected and described, while the rest is ignored. When the same part of the world is described with different purposes, the obtained descriptions may differ widely, but still each of them may be very adequate for its purpose (and inadequate for the other). The remarks above indicate, that each system description should be evaluated in accordance with the purpose of making the description. Thus a simulation program describing voter-response can be evaluated pragmatically as an aid in allocating resources in a political campaign or scientifically as an aid in exploring a sociological theory. In the latter case an important part in the evaluation is the relation between the sociological theory and the simulation program: How easy is it to express the central concepts of the sociological theory in the programming language? How much is needed to explain the program to people familiar with sociological theories as the one considered? Et cetera.

1.2 BACKGROUND AND INTENDED USE OF EPSILON

The Epsilon language is based on concepts from computer science and it is primarily intended to be used in this area. We have used and developed concepts from Simula [Dahl, Myhrhaug & Nygaard, 70] and Delta [Holbæk-Hanssen, Håndlykken & Nygaard, 75]. Most notably the classes of objects and interruptable continuous state-transformations. Our semantic model is based on high-level Petri nets [Genrich & Lautenbach, 81], [Jensen, 81a, 81b, 82]. The conceptual framework of nets, especially the notion of concurrency [Petri, 75, 76], has influenced the development of Epsilon. Finally we have tried to use the insight recently gained in the area of concurrent programming [Hansen, 78], [Hoare, 78], [Ichbiah et. al., 80] and [Kristensen, Madsen, Møller-Pedersen & Nygaard, 81].

As indicated by the examples on simulation, a language like Epsilon may be used to describe a large variety of systems. The primary candidates to be considered for description in Epsilon are, however, systems containing concurrent components, some of these being edp-equipment or by other means representing highly structured information handling. Parts of such a description may serve as the system specification from which computer programs are developed. Epsilon is not itself an implementable language. As a typical example consider the process control system mentioned earlier. To develop the edp-components of such a system one needs descriptions of the production equipment, the daily actions of the workers, demands from management etc., as well as descriptions of the edp-components themselves. Systems analysts, workers and management will need different descriptions of the edp-components and their interaction with the other components of the system. Epsilon is intended to contribute to the development of tools to be used in such cases.

The degree of technical knowledge required of users of Epsilon depends on how they are supposed to use the language. People who

are to make comprehensive descriptions themselves or to study formal specifications in detail need a thorough knowledge of the language. This is the case for people who are to make an edp-implementation or perform detailed analysis. Those who are to read and discuss descriptions should need to know only the basic concepts.

In this paper we emphasize the description of object-structure, control-flow, synchronization and the distinction between interruptable, continuous actions and non-interruptable, instantaneous actions. In contrast, we only sketch how to describe predicates, partial-events, parameters, expressions, and types. For these syntactic categories a user of Epsilon is free to choose any suitable notation. If Epsilon is used in the design of an edp-system, it may be convenient to use the corresponding constructs from the intended programming language.

We have kept the language relatively small, and a number of additional concepts are needed in order to make Epsilon a full-fledged system description language, capable of dealing with large systems in a manageable way. These additional concepts are, except for a "virtual" concept, discussed in the sections on extensions. For an introduction to the virtual concept see [Holbæk-Hanssen, Håndlykken & Nygaard, 75]. A preliminary version of Epsilon has been described in [Jensen, Kyng & Madsen, 79b] and in [Jensen & Kyng, 80].

1.3 SOME EPSILON CONCEPTS AND AN EXAMPLE

A system described in Epsilon consists of a number of objects, which each has a set of attributes and executes a sequence of actions. The attributes of an object may be constants, variables, types, functions, procedures, tasks and internal objects. The objects may synchronize their actions via variables or by direct communication. The actions of an object consists of an alternating sequence of equational-actions and event-actions.

Most of the time objects execute equational-actions. Such an action is specified by an equation consisting of a list of changeable variables and a predicate to be kept unceasingly satisfied during the execution of the action. When a number of objects concurrently execute equational-actions the conjunction of their predicates constitutes the effective predicate, while the union of their variable-lists constitutes the set of changeable variables. Informally the semantics of a set of equational-actions concurrently executed by different objects can be described as follows: the effective predicate is kept unceasingly satisfied by changing the changeable variables. In other words, the effective predicate defines an equation system, which is solved with respect to the set of changeable variables and these are updated accordingly. If several solutions exist one is chosen non-deterministically.

Although some systems cannot be adequately described using a classical time concept, there are many situations where the existence of a global, totally-ordered and continuous time-scale shortens the description and enhances its clarity. Thus we allow each Epsilon object to contain an implicitly defined data-attribute TIME, which represents model-time for that object. Each TIME variable can only be used by its own object, and it can only be read, not updated by the object. Instead each TIME variable is implicitly updated by the transition rules of the semantic model. For systems with a classical time concept all TIME var-

iables are continuously increased in a synchronous way, i.e. they all have equal values in any system state. Systems with non-classical time concepts can be described in Epsilon, either by avoiding use of the implicitly defined TIME variables, or by defining non-standard rules for the way the TIME variables are updated.

The predicate of an equational-action may use a TIME variable and in this way describe a continuous state transformation executed over an interval of TIME values. The actual transformation may depend on the actions executed concurrently with the one considered, but only within the limits specified by its predicate. Also the duration of an equational-action may depend on the actions of other objects.

Event-actions are instantaneous (with respect to the TIME variables) and specified by sequences of algorithmic commands. Each event-action constitutes one indivisible state transformation. The effect of an event-action does not depend on concurrently executed actions. However in some cases two or more objects jointly execute a single event-action, this is illustrated by the example below:

We consider four identical balls following a circular orbit, cf. fig. 1.1. The balls may move in both directions or stand still. Elastic collisions between the balls may occur. Two balls which collide will exchange their velocities (speed and direction). It is assumed that no external forces influence the system, i.e. no friction, no gravitation, and no loss of energy.

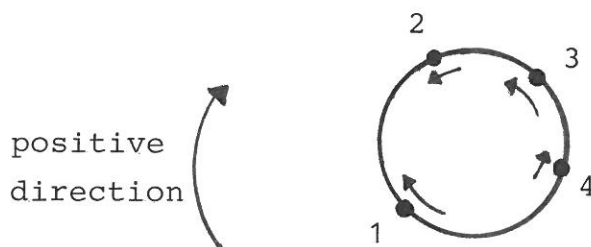


Figure 1.1

This system can be described in a number of different ways, depending on e.g. the purpose of the description and the language used. One description has just been presented using normal English and fig. 1.1. In fig. 1.2 and 1.3 we present more formal descriptions using Epsilon. The purpose is to illustrate a variety of Epsilon constructs, and we do not discuss the specific choices made for these two descriptions.

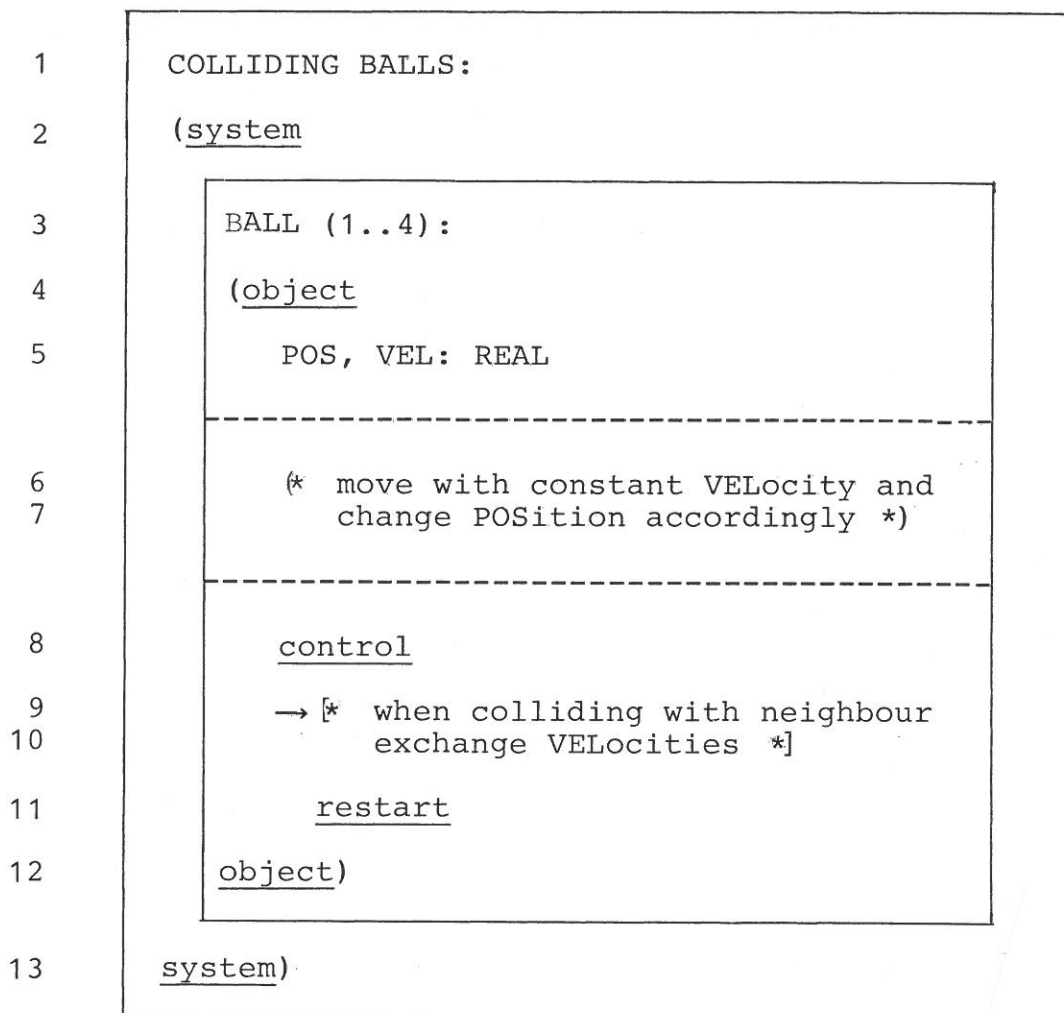


Figure 1.2

The system contains four BALL objects with identical specification lines (3-12). A BALL has two variables, POS and VEL, of type REAL, which describe POSition and VELOCITY. They are assumed to be initialized, but their initial values are not specified. Each BALL executes an equational-statement (6-7), which describes the continuous movement with constant VELOCITY between two collisions. The duration of the equational-action is determined by a control-section (8-11). When a collision occurs, the event-statement (9-10) is executed jointly with the matching event-statement of the colliding neighbour BALL. The matching pair of event-statements together describe the effect of the collision, i.e. the exchange of VELOCities. After the collision both BALLs restart execution of the equational-statement, which describes continuous movement.

The thin lines surrounding some of the language constructs in fig. 1.2 indicate the syntactic structure of the description. They can be omitted and have no influence on the semantics.

The description in fig. 1.2 is still rather informal, especially the statements. In fig. 1.3 we have made a more formal description of the system. The structure of the description is the same, except that we introduce a procedure, GETVEL, to be used in the communication between colliding BALLs. GETVEL has an input parameter V, which is call by value. Each BALL contains an equational-statement (8) and a control-section (9-19). The equational-statement specifies a variable, POS, to be changed by the execution of the statement and a predicate, which determines the value of POS, while the BALL is moving continuously without collisions. The barred names in (8) denote the values of the corresponding variables when execution of the statement began. The control-section determines when to stop execution of the equational-statement. This happens when the BALL instantaneously collides with its left or its right neighbour.

A collision with the left neighbour is represented by the event-statement (10-13). The do-clause describes that the object itself

```

1  COLLIDING BALLS:
2  (system
3
4      BALL (ID:1..4):
5      (object
6          POS, VEL: REAL
7
8          GETVEL (V: in REAL):
9          (procedure  VEL := V procedure)
10
11          -----
12          (* POS | POS =  $\overline{\text{POS}}$  + VEL * (TIME -  $\overline{\text{TIME}}$ ) *)
13
14          -----
15          control
16          → [* match BALL(ID ⊕ 1).GETVEL(VEL)
17             when  POS ≈ BALL(ID ⊕ 1).POS,
18                VEL < BALL(ID ⊕ 1).VEL
19             do GETVEL  *]
20
21          restart
22
23          .....
24
25          → [* match BALL(ID ⊕ 1).GETVEL(VEL)
26             when  POS ≈ BALL(ID ⊕ 1).POS,
27                VEL > BALL(ID ⊕ 1).VEL
28             do GETVEL  *]
29
30          restart
31
32          object)
33
34  system)

```

Figure 1.3

executes GETVEL. The match-clause describes that this must be done jointly with the left neighbour executing its GETVEL procedure. The when-clause describes the conditions for a collision: the two BALLs must be adjacently POSitioned (11) and the VELOCITY of left neighbour must be greater than that of the BALL itself (12). The parameter for each of the GETVEL procedures is supplied in the match-clause, i.e. by the other BALL, and the result of the joint execution of the two GETVEL procedures is that the two colliding BALLs exchange their VELOCities. A collision with the right neighbour is described analogously (15 - 18). In both cases execution of the equational-statement restarts after the collision, with new values for the barred variables.

1.4 EQUATION NETS AS PART OF THE LANGUAGE (EXTENSION)

The Epsilon language is essentially one-dimensional, in the sense that each system description consists of a sequence of characters. However, two-dimensional graphical tools are often valuable in system description [Brauer, 80], [Bødker & Hammerskov, 82]. A simple example is fig. 1.1 and another is given in chapter 2, where it is used to illustrate the object-structure in a system, fig. 2.2.

Our semantic model, called Equation nets, is based on high-level Petri nets, where colours, i.e. information, is attached to the individual tokens [Genrich & Lautenbach, 81], [Jensen, 81a, 81b, 82]. Equation nets are themselves a powerful two-dimensional graphical tool. In a later version we want to take advantage of this by extending Epsilon in such a way that system descriptions may directly include Equation nets. It will then be possible to describe a system by a mixture of net elements and sequential Epsilon text. To get the final net representing the semantics of a system description the text part will be translated into Equation nets, while the net part will be included directly (i.e. viewed as being already translated).

The extended Epsilon will be defined in such a way that the Equation nets representing the semantics of extended Epsilon descriptions have all the essential properties of the nets corresponding to the present version of Epsilon. An obvious choice will be to define the extended Epsilon in such a way that the two sets of properties are identical. It is however not necessarily the best choice, since some of the restrictions placed on the present version of Epsilon, in order to get well-structured textual descriptions, may be unnecessary for the graphical descriptions to be included in the extended Epsilon.

1.5 DIRECTIONS OF FUTURE WORK

The language presented in this paper is the result of a development effort initiated by our analysis of the Delta language [Jensen, Kyng & Madsen, 79a]. The language is syntactically and semantically simpler than Simula and Delta (and the earlier versions of Epsilon). Two major reasons for this are: the improved treatment of concurrency and non-determinism obtained from net theory and the substitution of the Delta interrupt by the more structured control/select-sections. Inclusion of additional concepts needed to make Epsilon a full-fledged system description language seems to present only few problems. Thus this paper ends the first phase of language development and definition. Our work in the nearest future will focus on the use of the language.

In [Kyng, 82], it is investigated how to specify and verify a small distributed system in Epsilon. It is demonstrated that equational-statements are well-suited for the specification of partial correctness by means of invariants. We intend to continue this work by considering the use of temporal logics for the direct specification of total correctness and by further developing the top-down specification and verification method of that paper.

Furthermore we are going to use Epsilon in a realistic system development process. Experience with Delta already indicate that the

use of such a language can be a valuable help [Håndlykken & Nygaard, 80]. We want to investigate the potentials of Epsilon when a relatively large number of people with different backgrounds are communicating in a system development process. Especially we want to experiment with the use of Epsilon extended with Equation nets, (cf. section 1.4).

The remaining chapters of this paper are organised as follows: Chapter 2 treats the object structure of Epsilon systems and the specification of attributes. Chapter 3 describes the specification of actions. The two chapters contain an informal description of their subjects and a discussion of some possible extensions. Chapter 4 defines the syntax and static semantics of Epsilon. Chapter 5 introduces our semantic model, and chapter 6 defines the semantics of Epsilon.

Chapter 2

SPECIFICATION OF ATTRIBUTES

2.1	Hierarchical systems of objects	21
2.2	Attribute declarations	25
2.3	Scope rules	28
2.4	Dynamic object generation and de- struction (extension)	31
2.5	Templates (extension)	32

2.1 HIERARCHICAL SYSTEMS OF OBJECTS

An Epsilon system consists of a hierarchy of objects. One of these objects, the system object corresponds to the delimitation of the system and contains all other objects as parts. The description of the system object textually encloses the description of the other objects. Each object is characterised by a selected set of attributes, such as variables, types, procedures and objects, and by a sequence of actions, which may involve the object itself and other objects. Consider the following description, fig. 2.1, where a number of objects are named and the hierarchy described. No other kinds of attributes and no actions are described.

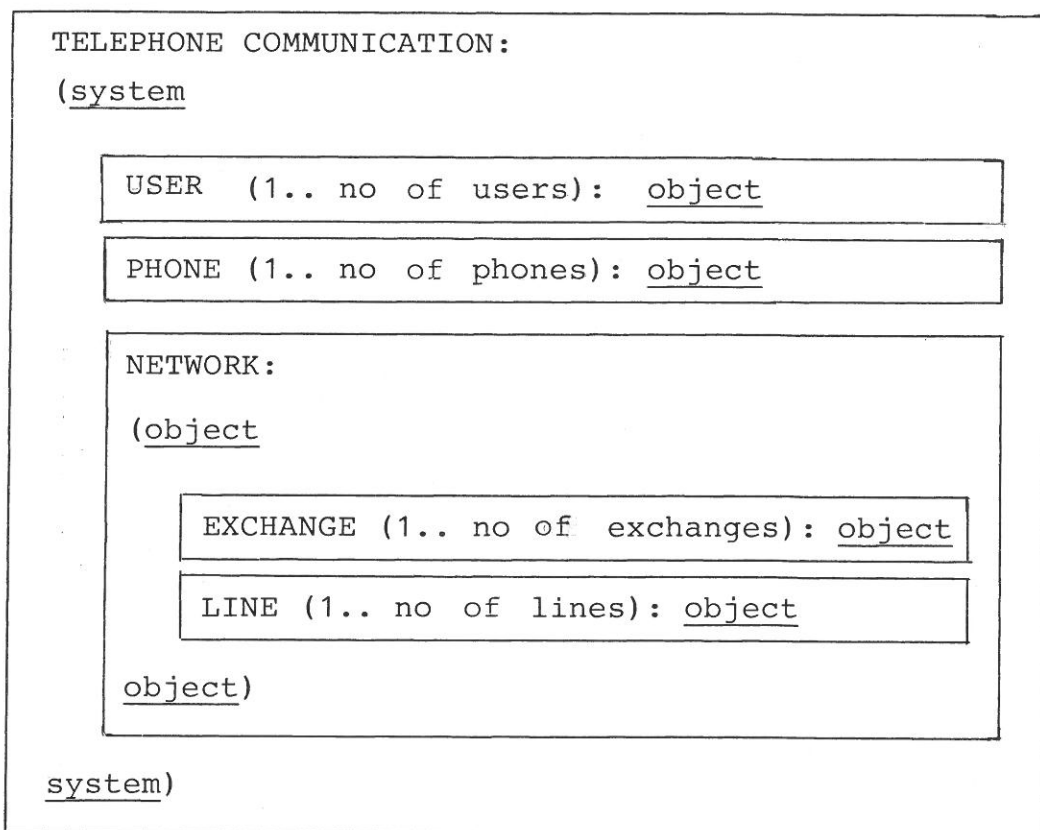


Figure 2.1

In this example the system object encloses three different kinds of objects: PHONES, USERS and a NETWORK. This last object is itself the encloser of two kinds of objects: EXCHANGES and LINES.

The system object is the direct encloser of PHONES, USERS and NETWORK, while it is the second level encloser of EXCHANGES and LINES. Each object has at most one encloser, at each level. In general any object may enclose other objects (have them as attributes), and thus an Epsilon system is a nested structure of objects.

We may depict the object hierarchy of fig. 2.1 as shown below, where each object is represented by a box. A pile of boxes indicate a family of objects, i.e. a set of objects originating from the same object-declaration and thus having identical attribute- and action-parts.

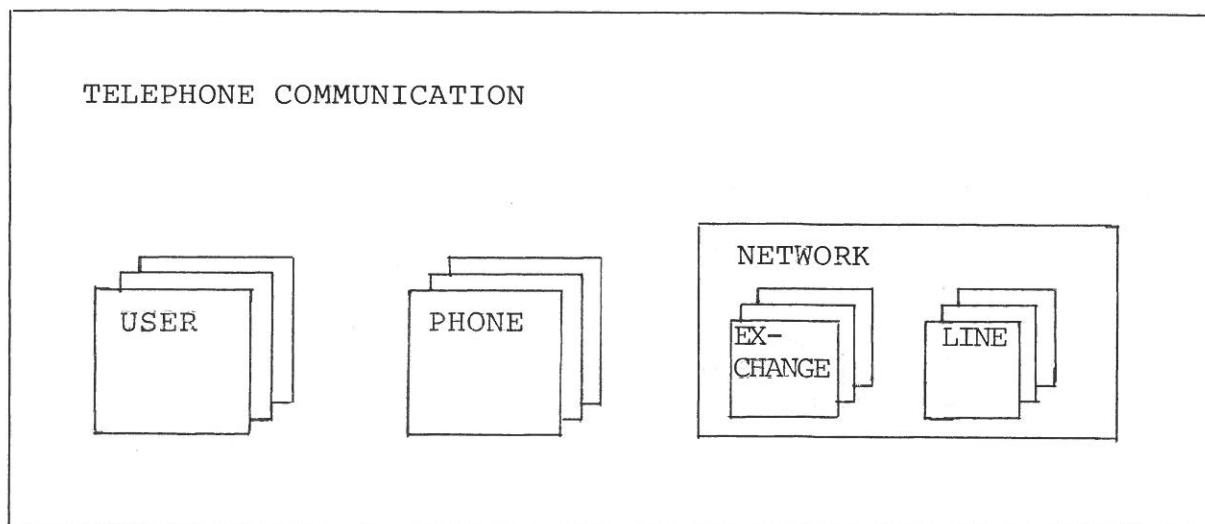


Figure 2.2

When an object INT is the attribute of another object ENC it is INT as a whole, which is an attribute of ENC. In particular this means that the individual attributes of INT are not attributes of ENC.

We have in fig. 2.1 used:

NAME: object

for object-declarations, where we do not want to describe the inter-

nal structure (attributes and actions) at this stage, i.e. as an abbreviation of the declaration:

NAME: (object object).

Similar abbreviations will be used for the other kinds of attributes: constants, variables, types, functions, procedures and tasks.

We note that the following conventions have been used in fig. 2.1:

The formal elements are:

- declared names, written in upper case letters, e.g. "PHONE".
- keywords, written in underlined, lower case letters, e.g. "system".
- special symbols, e.g. ":".

The informal elements are:

- undeclared entities, written in lower case letters, e.g. the integer constant "no of phones".

In general informal elements are described by lower case letters, and such elements may be attributes as well as actions, as illustrated by fig. 2.3 below. The dashed line separates the descriptions of attributes and actions.

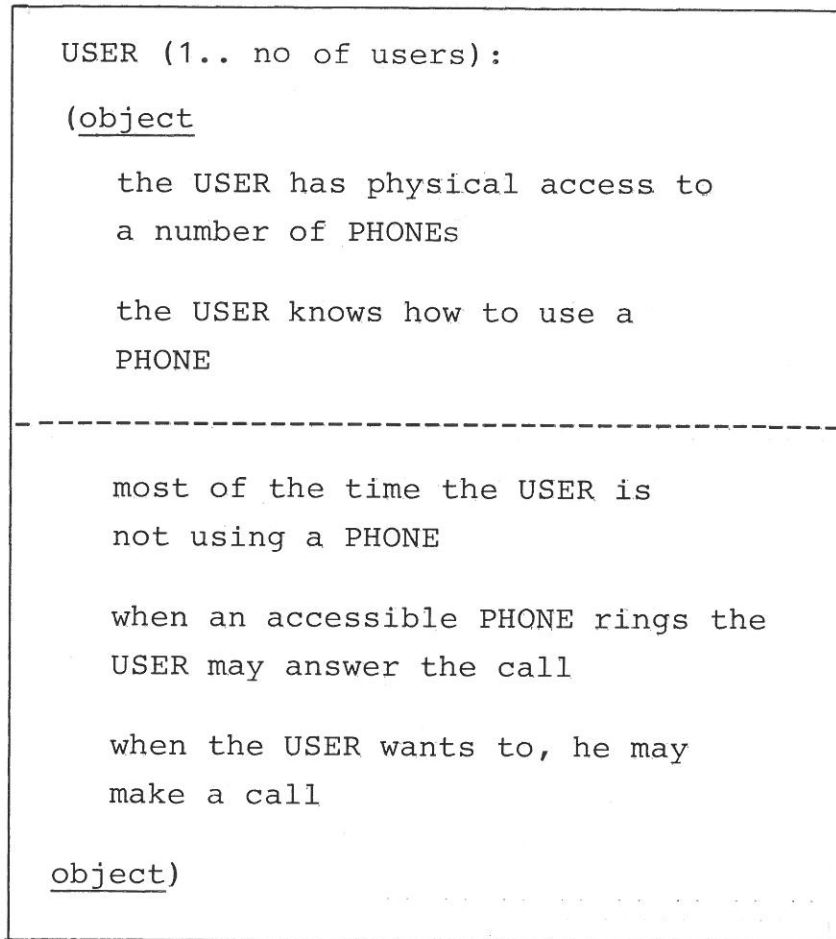


Figure 2.3

2.2 ATTRIBUTE DECLARATIONS

In Epsilon attributes are described by declarations. In section 2.1 we encountered the object-declaration. In fig. 2.1 the main purpose of the object-declarations was to introduce formal names such as USER(I) and PHONE(I), by which we can reference and distinguish the different objects. Furthermore the encloser relation was defined. In fig. 2.3 the object-declaration was used to give an informal description of the USER objects.

The ability of an object to execute a specific action pattern containing continuous, non-instantaneous actions may be described by a task-declaration in that object. In fig. 2.4 the main purpose of the task-declarations is to introduce formal names by which we can call the two action patterns.

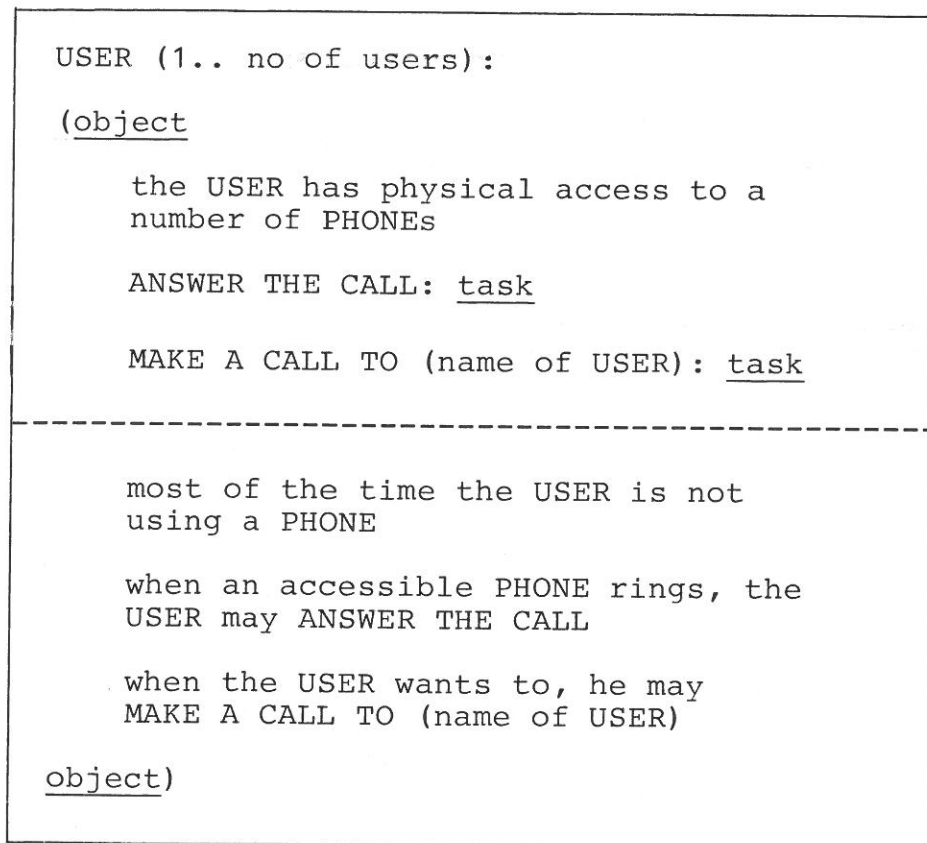


Figure 2.4

Now we elaborate the descriptions of the tasks ANSWER THE CALL and MAKE A CALL TO. We do so by introducing the action patterns LIFT,

DIAL and REPLACE. In all three cases we choose to regard the corresponding action as instantaneous and non-interruptable. Such action patterns are described by procedure-declarations. The DIAL procedure is considered to be local to the task MAKE A CALL TO, whereas the other two are not, since they are used in both tasks. The procedure-calls are enclosed in square, starred brackets. This distinguishes them from task-calls.

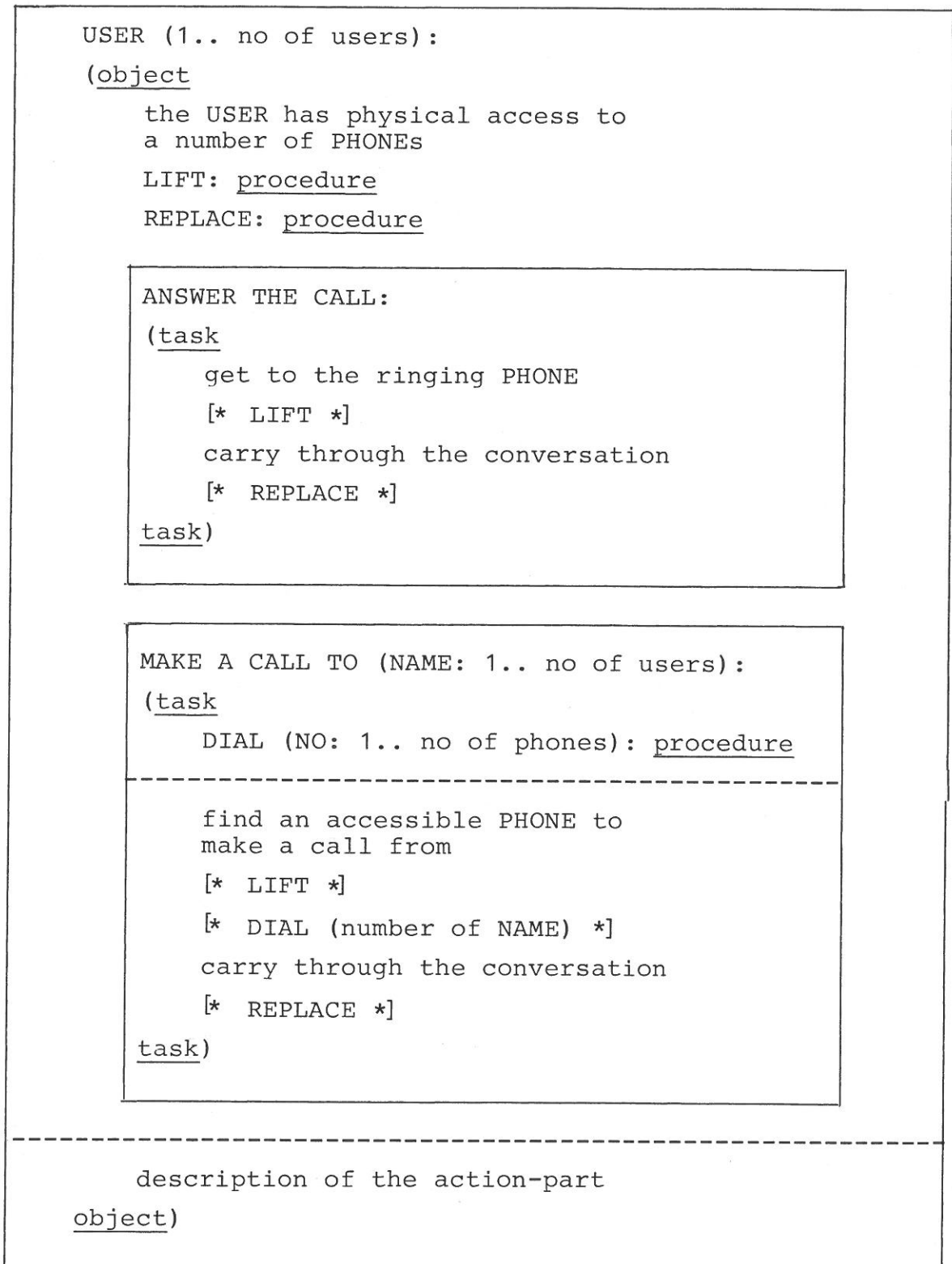


Figure 2.5

In fig. 2.5 we have included additional formal and informal elements. For each parameter we have specified a formal name and an informal type. In the procedure-call:

DIAL (number of NAME)

"number of" may be considered as an undeclared function converting a username to a phone number. It can be declared by the following function-declaration:

NUMBER OF (NAME: USER NAME): function return PHONE NO

where USER NAME and PHONE NO are declared by the following type-declarations:

USER NAME: type 1.. no of users
PHONE NO: type 1.. no of phones

The type-names are written in upper case letters to indicate that they are declared, while the two integer constants "no of users" and "no of phones" still are written in lower case letters to indicate that they are informal, undeclared language elements. In this paper we use the types known from PASCAL (integer, real, array, enumeration, etc.), but with a slightly changed syntax. The two integer constants, "no of users" and "no of phones", can be formally defined by the following data-declarations:

NO OF USERS, NO OF PHONES: const INTEGER

Data-attributes are either constants or variables. In the latter case the keyword "var" is optional. All data-attributes are initialized, but in the case above the initial values are unspecified.

We have now encountered six different kinds of declarations and they introduce six different kinds of attributes: objects, tasks, procedures, functions, types and data. Each kind of declaration is allowed to contain declarations of its own kind and of all succeeding kinds (referring to the list above). As an example procedures are allowed to contain procedures, functions, types and data, but not objects or tasks.

2.3 SCOPE RULES

The scope rules of declared names are based on the static structure of Epsilon descriptions: Each attribute is directly observable (and can be denoted by its name) in the entire syntactic entity in which it is declared, except for those inner syntactic entities which declare an attribute with the same name (redeclaration). When an object D is directly observable, each attribute A in D is indirectly observable and can be denoted by the notation "D.A" known from Simula. Indirect observability is used for three different purposes:

- a) To observe attributes in objects, which are not enclosers, but in the object hierarchy can be reached by ascending (outwards) through the enclosers to some level and then descend (inwards) exactly one step. Such objects are called lateral objects, and their attributes are not directly observable. As an example consider the object hierarchy of figure 2.2. The attributes of a PHONE are indirectly observable from the LINES, but not vice versa.
- b) To observe attributes, which cannot be directly observed, because they are hidden by a redeclaration.
- c) To emphasize that a directly observable attribute belongs to a certain object. This will always be done when we use procedures or functions declared in other objects.

Above we have, for all parts of an Epsilon description, defined the set of attributes, which are observable, directly or indirectly. It is, however, not all observable attributes, which can be used to describe the different kinds of actions and attributes, and thus we now define the set of useable attributes as a subset of the observable. To do this we distinguish between internal attributes belonging to the considered object and external attributes belonging to other objects.

All internal attributes are useable whenever they are observable. For external attributes, there are the following restrictions: External tasks are never useable. External types are only useable when they are directly observable. External procedures are only useable in the match-clauses of external event-statements, which are to be executed jointly with the procedures own object (c.f. section 3.5). External data and functions are only useable in the predicate of equational-statements (c.f. section 3.2), in the when-clauses of external event-statements, which are to be executed jointly with the attributes own object, and in initialisation-clauses of data-declarations (where they have to be directly observable). In all three cases the type of data and the parameter types and return type of functions have to be useable.

As an illustration of the rules for observability and useability of functions, types and data we consider fig. 2.6, which describes some attributes of a TELEPHONE COMMUNICATION system. The object hierarchy is as described in fig. 2.2.

The types declared in the system object are useable by all objects in the system. One of these are used in each PHONE object "NO: PHONE NO" and one of them in each USER object "NAME: USER NAME". The function RINGING of PHONE (I) is useable by the USERS and the other PHONES by means of "PHONE(I). RINGING". The STATUS variable of PHONE(I) is observable, but not useable since the type of this variable is internal to PHONE(I) and thus non-useable for the other objects.

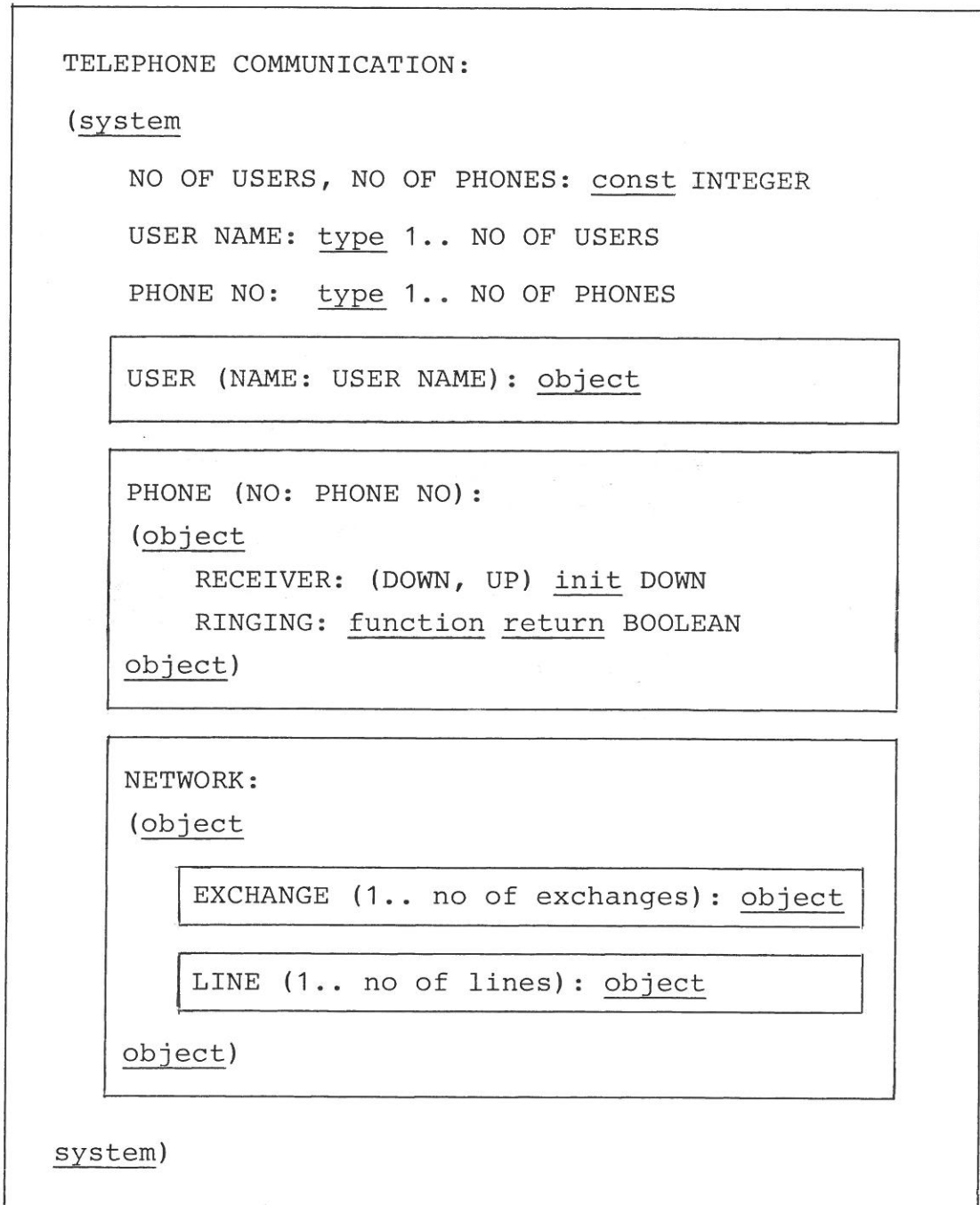


Figure 2.6

2.4 DYNAMIC OBJECT GENERATION AND DESTRUCTION (EXTENSION)

In the present version of Epsilon the object hierarchy is static in the sense that all objects of a system exist during the entire period which the system description covers. In a later version of the language we will include the possibility of dynamic creation and dynamic destruction of objects, to describe that objects pass the system boundary. Dynamic generation of objects is well-known from Simula and is also included in Delta. In [Jensen, Kyng & Madsen, 79a] we discuss dynamic generation and dynamic destruction of objects and proposes to include both explicitly in system description languages. When we include dynamic object generation an "object pattern" declaration is needed. Following Simula we name this a class-declaration, c.f. the next section. Class-declarations must be allowed to include parameter-clauses, and it must be possible to declare references as known from Simula and Delta.

We do not want to specify an upper limit on the number of objects of a certain class. In condition/event nets this presents a problem, and in [Jensen, Kyng & Madsen, 79a] we use infinite nets (but only finite markings). In coloured Petri nets the situation is easily handled by allowing the component of the token colour, which represents object identity, to have an infinite range.

2.5 TEMPLATES (EXTENSION)

In a later version of Epsilon we will introduce templates, based on the prefix concept of Simula. Then object-declarations will have the following form, where the class CLA is a template for the object O_A being declared:

Form A: O_A : object
 (CLA
 declarations
 action-part
 CLA)

The object O_A gets all the properties of CLA together with the properties described by itself: The attributes of O_A is the attributes of CLA combined by disjoint union with the attributes declared by O_A itself. The actions of O_A is the actions of CLA combined with the actions described by O_A itself. In section 3.9 we shall explain how to describe the merging of the two action-parts.

For the general form of object-declarations, shown above, we shall allow three different shorthands:

Form B: The declaration

O_B : (object
 declarations
 action-part
 object)

is a shorthand for the following object-declaration, where CLASS is a language-defined class having those properties, which are shared by all objects:

O_B : object
 (CLASS
 declarations
 action-part
 CLASS)

Form C: The declaration

O_C : object CLA

is used to declare an object O_C , which have exactly the properties of class CLA, i.e. as a shorthand for the following object-declaration:

O_C : object (CLA CLA)

Form D: The declaration

O_D : object

is used to declare an object O_D for which we do not want to describe attributes or actions at this stage, i.e. as a shorthand for the object-declaration:

O_D : object (CLASS CLASS)

Object-declarations of form B and D are used in the rest of this paper, while A and C are not (due to the absence of templates and class-declarations).

Classes can be used as templates to define objects, but also to define new classes, and in this way we can describe tree-structured hierarchies of templates. Each template inherits all the properties of the classes in its template sequence, i.e. the templates which lie between the root and the template in question. An object declared by form A is said to be an instance of its direct template CLA, but also to be an instance of each template in the template sequence of CLA. As shown in [Dahl, Dijkstra & Hoare, 72] templates are a strong and convenient tool for the structuring of large system descriptions. They allow a combination of top-down and bottom-up development.

Above we have considered templates for objects and classes. Analogously it is possible to introduce templates for all the other kinds of attributes and for blocks, event-blocks and evaluations (c.f. section

3.6). We then introduce, for each kind of entity, a main form A and three abbreviations B, C and D (corresponding to the four forms of object-declarations shown above). The notation, introduced for attribute-declarations in sections 2.1 - 2.2 and for blocks, event-blocks and evaluations in section 3.6, is however only a subset of these forms, since we in the main sections of the paper do not consider templates and abstract data types. The following table, fig. 2.7, gives an overview of the use of templates. For each kind of entity the table gives:

- the kind of template used
- the name of the default template, i.e. the template having those properties, which are shared by all entities of the kind in question
- the forms of definitions used throughout the main sections of this paper

Kind of entity	Kind of template	Default template	Forms used in this paper
class	class	CLASS	NONE
object	class	CLASS	B, D
task	task	TASK	B, D
block	task	TASK	B, D
procedure	procedure	PROCEDURE	B, D
event-block	procedure	PROCEDURE	B, D
function	function	FUNCTION	B, D
evaluation	function	FUNCTION	B, D
type	type	TYPE	C, D
data	type	TYPE	C, D

Figure 2.7

Objects, blocks, event-blocks, evaluations and data use classes, tasks, proce-

As an example of class- and object-templates consider the following description, where an object OBJ, with template CLA, declares two classes CLA_1 and CLA_2 , an object OBJ_1 , and a family of objects OBJ_2 :

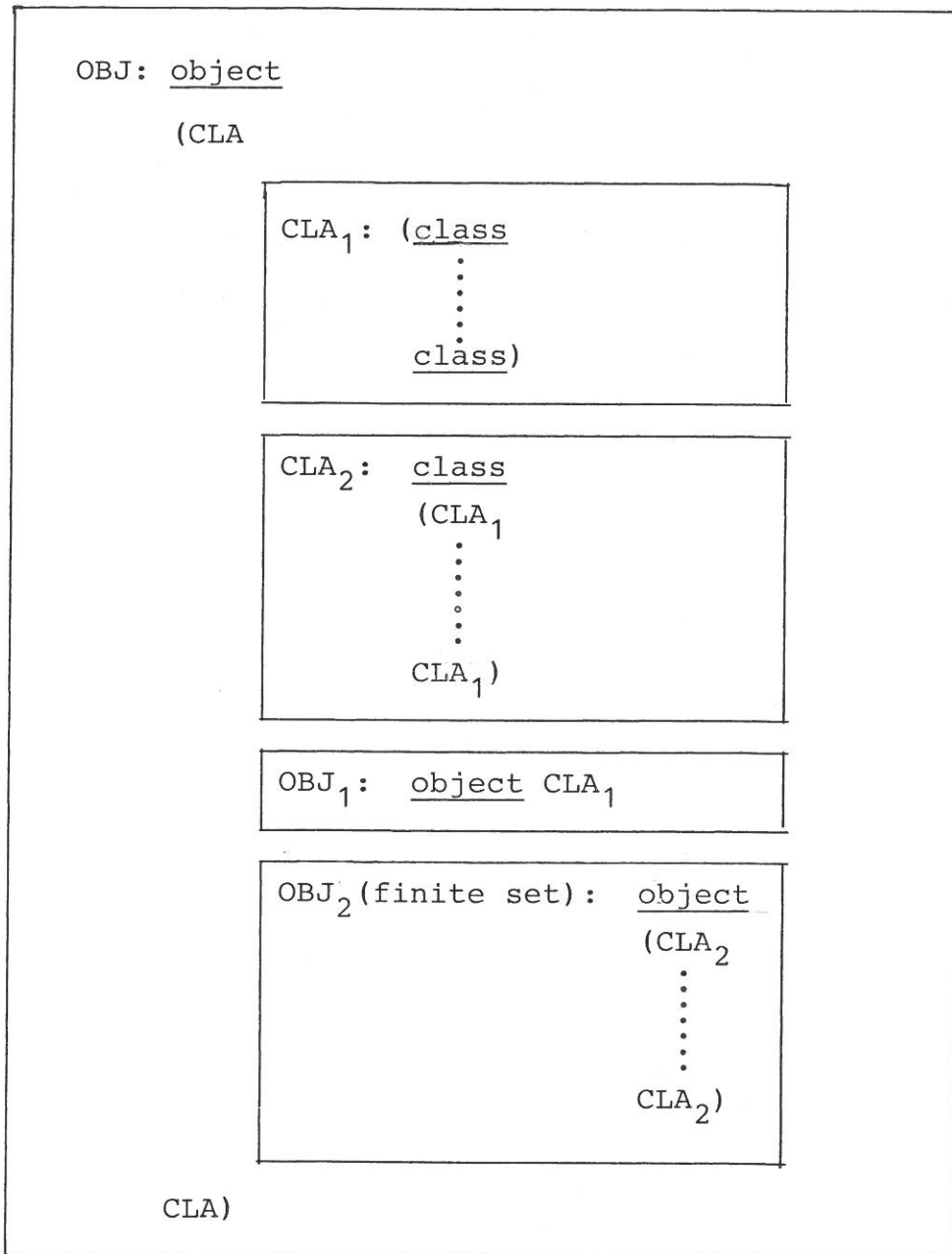


Figure 2.8

The object hierarchy of fig. 2.8 is as shown in fig. 2.9, where we have omitted CLASS from all template sequences

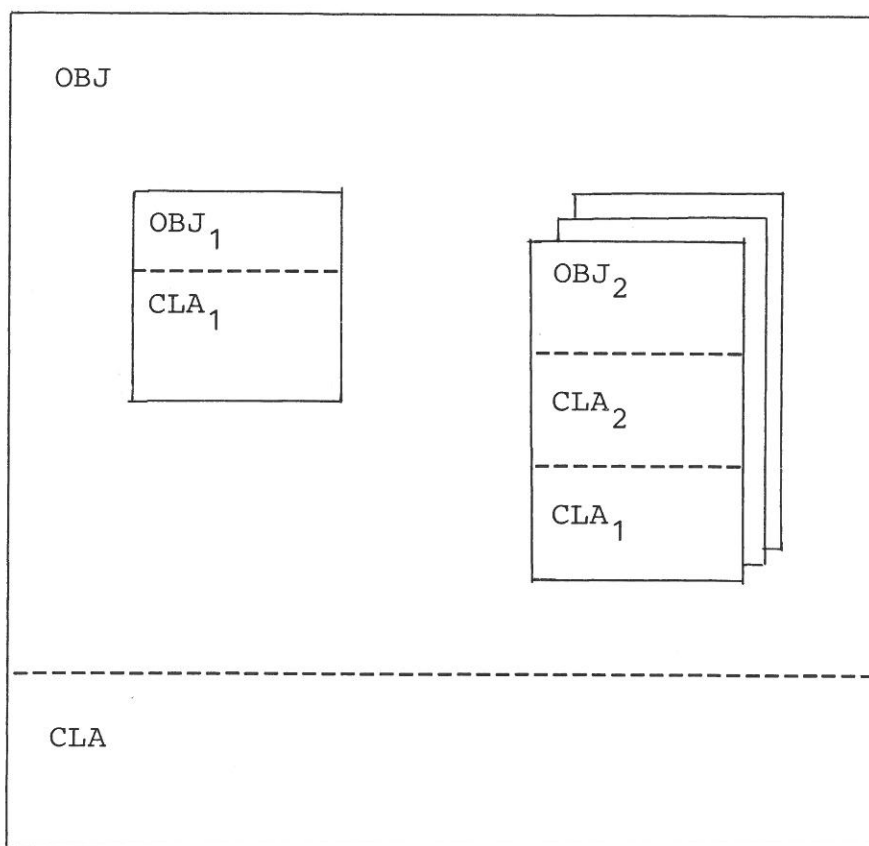


Figure 2.9

Chapter 3

SPECIFICATION OF ACTIONS

3.1	Continuous and instantaneous mode	41
3.2	Equational-statements	42
3.3	Internal event-statements	46
3.4	Statement sequences	47
3.5	External event-statements	52
3.6	Blocks and action patterns	56
3.7	Control-sections	61
3.8	Select-sections	71
3.9	Templates (extension)	75

3.1 CONTINUOUS AND INSTANTANEOUS MODE

The dynamic properties of an Epsilon system can be described by means of its possible behaviours. Each behaviour alternates between continuous mode and instantaneous mode. It can be viewed as a relation between model-time and the corresponding system states.

In continuous mode the state transformation is implicitly described by means of an equation system, which is unceasingly solved as model-time increases continuously over a closed interval. In instantaneous mode the state transformation is explicitly described by means of algorithms and model-time is constant. A more detailed description of the two modes is given in the following sections and in section 6.8.

Each object in the system may have an implicitly defined data-attribute TIME, which represents model-time for that object. The TIME variables are only useable by their own object, and they can only be read, not updated by the object. Instead they are implicitly updated, by the transition rules of the semantic model, when the system is in continuous mode. For systems with a classical time concept all TIME variables are continuously increased in a synchronous way; i.e. they all have equal values in any system state. Systems with non-classical time concepts can be described in Epsilon, either by avoiding use of the TIME variables, or by defining non-standard rules for the way the TIME variables are updated.

The basic actions of continuous mode and instantaneous mode are described by equational-statements and by event-statements respectively. Groups, interruptions, loops and branches are described by blocks, tasks, control-sections and select-sections.

3.2 EQUATIONAL-STATEMENTS

An equational-statement may have the following form:

$$(* v_1, v_2, \dots, v_n \mid \text{predicate} *)$$

The curved parentheses indicate that an equational-action is described. The list v_1, v_2, \dots, v_n consists of those variables which may be changed by the action. They must all be internal to the object executing the action. The predicate defines a function from states into truth-values. It may contain external data-attributes. The predicate together with the variable-list is referred to as the equation describing the action.

An equational-statement is normally executed over a non-instantaneous period of model-time, represented by the TIME variable taking all values in a closed interval $[t_1, t_2]$, where $t_1 < t_2$. During that period the evolving system states are implicitly described by the predicate, which is kept unceasingly satisfied by changing the variables from the list as TIME is continuously increased.

Normally a number of objects concurrently execute an equational-action each. Then the conjunction of their predicates constitutes the effective predicate, while the union of their variable-lists constitutes the set of changeable variables. The effective predicate is kept satisfied unceasingly by assignments to the changeable variables. This can also be viewed as the continuing solving of an equation system, where the changeable variables are the unknowns, while the effective predicate is the set of equations to be solved. If, at some value of TIME, more than one solution exists, one of them is chosen non-deterministically. If no solution exists the description is erroneous.

This kind of state transformation is called continuous mode. In the semantic model execution of an equational-action is represented by a place, which holds a token. During continuous-mode no tokens are moved, but the colour attached to tokens may change, representing the updating of TIME and of the changeable variables.

As an example of an equational-statement consider the following, which describes the changing position of a falling object:

$$(* \text{ POS} \mid \text{POS} = \overline{\text{POS}} - \frac{1}{2} \times G \times (\text{TIME} - \overline{\text{TIME}})^2 *)$$

POS is a variable of the object and describes its position. POS is listed to the left of the vertical bar indicating that POS is changed during the execution in such a way that the predicate to the right of the bar is kept unceasingly satisfied. G is a constant of the system object describing the strength of the gravitation.

In the predicate of equational-statements a variable-name with a horizontal bar denotes the value, which the variable had when execution of the equational-statement began. Barred variables may also be used in the predicate of event-statements (cf. section 3.4). There they denote the value, which the variable had when the immediately preceding equational-action began.

Next consider a system containing a TRAIN and some PASSENGERS:

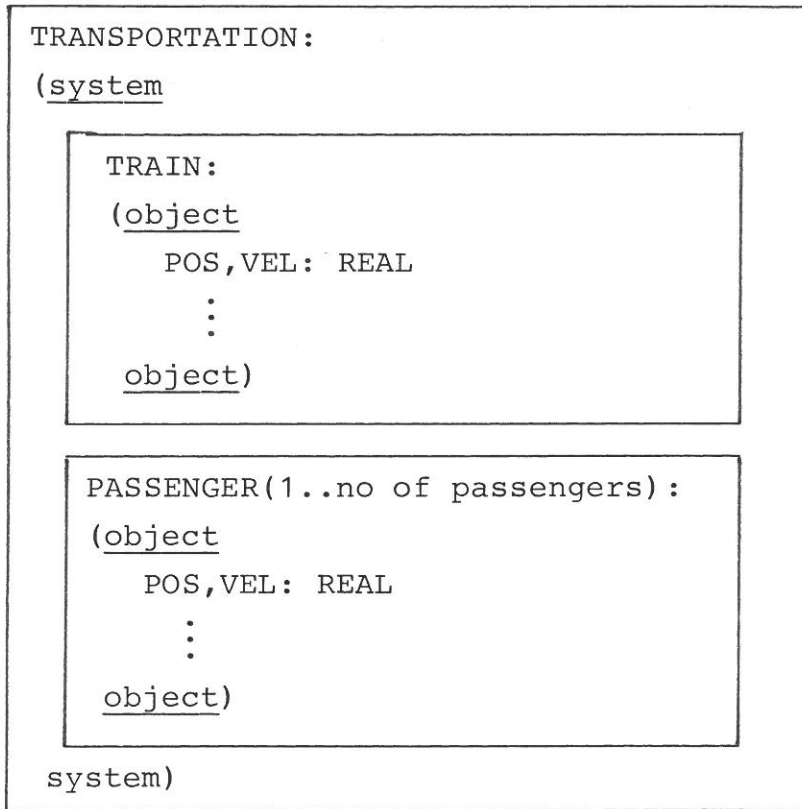


Figure 3.1

The action of the TRAIN is described by the following equational-statement:

$$(* \text{ POS} \mid \text{POS} = \overline{\text{POS}} + \int_{\text{TIME}}^{\text{TIME}} \text{VEL} \, dt *)$$

A PASSENGER walking in the TRAIN may be described by the following equational-statement, where VEL means a PASSENGER's "relative" velocity, and POS is his "absolute" position:

$$(* \text{ POS} \mid \text{POS} = \overline{\text{POS}} + \int_{\text{TIME}}^{\text{TIME}} \text{VEL} \, dt + (\text{TRAIN.POS} - \overline{\text{TRAIN.POS}}) *)$$

The first equational-statement above is executed by the TRAIN, while each PASSENGER concurrently executes a version of the

second equational-statement. All predicates of the equational-statements are included in the effective predicate, which determines the values of the changeable variables TRAIN.POS and PASSENGER(1).POS, ..., PASSENGER(no of passengers).POS, as a function of TIME.

In continuous mode the variables of an object can only be changed by equational-statements executed by the object itself. This means that in continuous mode the variables of an object cannot be changed, unless they are mentioned in the variable-list of the equational-statement currently being executed by the object. If the variable list of an equational-statement is omitted no variables of the object executing the statement are changed during that period of continuous mode. If the predicate is omitted execution of the statement imposes no restriction on the system state. This corresponds to the predicate, which is always satisfied. Thus the equational-action described by "(* *)" has no effect on the system state. It is called the neutral equational-action.

3.3 INTERNAL EVENT-STATEMENTS

An event-statement may have the following form:

[* when predicate do partial-events *]

The square brackets indicate that an event-action is described. The predicate defines a function from states into truth-values. Partial-events is a sequence of assignments, procedure-calls, event-blocks, alternative-, and repetitive-constructs which describes a sequential algorithm. The event-statement can only be executed if the predicate is satisfied. The total sequence of partial-events defines a single state transformation, which is executed as an indivisible and instantaneous action (i.e. without changing TIME).

The event-statement described above is internal in the sense that it describes an event-action, which is executed by a single object. Internal event-statements are only allowed to use attributes, which are internal to the object executing the statement. In section 3.5 we consider event-statements, which are external in the sense that they, together with external event-statements of other objects, describe a single event-action, which is executed jointly by all the involved objects.

Normally a number of event-actions are executed concurrently. However, concurrent event-actions are executed by disjoint sets of objects, and thus the above rules guarantee independency between them, in the sense that disjoint sets of data-attributes are used. This kind of state transformation is part of instantaneous mode. In the semantic model execution of an event-action is represented by a transition firing. Concurrent event-actions form one step in the firing sequence.

As an example of an internal event-statement consider the following, which describes the elastic collision between the

falling object from section 3.2 and the ground:

```
[* when POS = 0 do VEL := -VEL *]
```

To describe this collision as an instantaneous action is of course an abstraction. In reality the falling object interacts with the ground for some period of time during which they slightly deform each other. For a number of purposes, however, it is convenient to consider this complicated interaction as an instantaneous, indivisible action, where the total effect is to negate the velocity of the falling object.

If the predicate is omitted in an internal event-statement, the statement can always be executed. If the partial-events are omitted no state transformation is performed. Thus the event-action described by "[* *]" can always be executed. It has no effect on the system state and is called the neutral event-action.

3.4 STATEMENT SEQUENCES

Each object executes a strictly alternating sequence of equational-actions and event-actions. Execution of an equational-action normally continues over a period of model-time, and this period ends when the subsequent event-action becomes executable. An event-action is instantaneous and immediately followed by the subsequent equational-action.

As a very simple example, consider the following sequence of two statements, which also shows that equational-statements and event-statements can be informally described by means of lower

case text strings which describe the changeable variables, predicates and partial-events.

```
(* talking on a phone *)
[* when conversation is over
  do replace receiver *]
```

Execution of the equational-statement is ended when the event-statement becomes executable, i.e. when the predicate of the event-statement becomes satisfied.

Assuming that there has been paid for a certain PERIOD of time we may write:

```
(* talking on a phone *)
[* when TIME -  $\overline{\text{TIME}} \geq \text{PERIOD}$ 
  do replace receiver *]
```

The barred variable refers to the value of TIME when execution of the preceding equational-statement began, and thus the predicate becomes true when the conversation has been going on for a PERIOD of time.

As another example consider the following sequence of statements, which describes the melting and subsequent hardening of an alloy:

```
(* TEMP | TEMP = increase(TIME- $\overline{\text{TIME}}$ ,  $\overline{\text{TEMP}}$ , ENERGYSUPPLY *)
[* when TEMP = MELTED do ENERGYSUPPLY := 0 *]
(* TEMP | TEMP = decrease(TIME- $\overline{\text{TIME}}$ ,  $\overline{\text{TEMP}}$ ) *)
```

The first equational-statement describes how the temperature variable TEMP increases as an undeclared function of elapsed time, initial temperature and ENERGYSUPPLY. When TEMP reaches MELTED, which is a temperature slightly higher than the melting point, the event-statement becomes executable and the ENERGYSUPPLY is cut off. Next the second equational-statement is executed. It describes how TEMP decreases as an undeclared function of elapsed time and the temperature when the congealment began.

For this to be a reasonable description, the two undeclared functions, increase and decrease, must have a number of properties, e.g. continuity. Especially the value of TEMP, at the moment of TIME when ENERGYSUPPLY is cut off, must equal MELTED. This is guaranteed if the function decrease fulfills the following: $\forall t \geq 0: \text{decrease}(0,t) = t$.

This accordance between an event-action and the succeeding equational-action is something which we demand all descriptions to fulfill:

- an event-action must guarantee that the predicate of the succeeding equational-action is initially true; furthermore
- an event-action must not destroy the predicates of concurrent equational-actions.

For most descriptions it is trivial to convince oneself that the above restrictions are fulfilled. We believe that this is the case for all descriptions considered in this paper. In some cases it is, however, necessary with a proof to be certain that a description fulfills the restrictions. This problem is treated in [Kynng, 82].

Another approach to the question of accordance between actions is to impose a number of "syntactic" restrictions on the use of variables. However, before imposing such restrictions on all descriptions we want to gain more experience by using the Epsilon language in different areas.

If sequencing and control structure of an object demand two equational-statements to be executed immediately after each other, the two corresponding equational-actions are intervened by a neutral event-action. Analogously, if two event-statements immediately follow each other their execution is intervened by a neutral equational-action. This convention, where neutral

event- and equational-actions are implicitly inserted, guarantees that all objects strictly alternate between event- and equational-actions.

Next consider the situation where the alloy is transferred to some forms before congealment. This can be described in a number of different ways, obtained by substituting the event-statement from the description above by one of the following four statement sequences:

- (a) `[* when TEMP = MELTED do ENERGYSUPPLY := 0
TRANSFER ALLOY TO FORMS *]`
- (b) `[* when TEMP = MELTED do ENERGYSUPPLY := 0 *]
[* TRANSFER ALLOY TO FORMS *]`
- (c) `[* when TEMP = MELTED do ENERGYSUPPLY := 0 *]
[* transferring alloy to forms *]
[* when forms full *]`
- (d) `[* when TEMP = MELTED do ENERGYSUPPLY := 0 *]
[* transferring alloy to forms until full *]`

In the first two cases TRANSFER ALLOY TO FORMS is a procedure (and thus part of an event-statement to be executed instantaneously). In case (a) the procedure is executed together with the cut off of ENERGYSUPPLY. In case (b) the procedure is executed as an individual event-action following the cut off of ENERGYSUPPLY. Between the two event-actions a neutral equational-action is executed. The execution of this neutral equational-action is ended immediately, since the event-statement describing the succeeding event-action has no predicate and thus can be executed immediately. This means that the two event-statements in case (b) are executed in sequence, but at the same moment of model-time (i.e. the same value of TIME).

In the last two cases transferring alloy to forms is an (informally described) equational-action. In case (c) the action is described by an equational-statement, whose execution continues until the predicate of the succeeding event-statement becomes satisfied. Assuming that there is enough alloy, the predicate becomes satis-

fied after some finite, non-zero period of model-time. The effect of the event-action is to end the equational-action at that moment of model-time. The event-action itself has no effect on the system state, since the event-statement contains no partial-events. Case (d) is equivalent to case (c), except that we have introduced a shorthand notation, which allows us to describe an equational-action and an event-action by a single statement. This kind of notation can also be used, when one of the actions is to be described formally or when the event-action precedes the equational-action. The need for such shorthands, and their precise definition, is determined by the kind of the system, which we describe. In the description of some systems it might be a help to use:

[* when predicate do a,b := exp₁,exp₂ *)

to combine an assignment with the maintenance of the obtained equations, i.e. as a shorthand for:

[* when predicate do a,b := exp₁,exp₂ *]
 (* a,b | a = exp₁, b = exp₂ *)

In other kinds of systems we might use other shorthands.

Now we return to the two modes of behaviour: continuous and instantaneous. In continuous mode all objects execute equational-actions and model-time increases continuously. In this mode there is a one to one correspondence between system states and model-time. When one or more objects are ready to execute an event-action the system switches to instantaneous mode. In this mode an object may execute a sequence of event- and equational-actions. Model-time is not increased, and there is a many to one correspondence between system states and model-time. The instantaneous mode continues as long as at least one object is to execute an event-action. When all objects executes equational-actions and no object is to execute an event-action, the system switches to continuous mode and model-time increases again. The details of continuous and instantaneous mode are more formally defined in section 6.8.

3.5 EXTERNAL EVENT-STATEMENTS

In section 3.3 we considered internal event-statements, where only internal attributes of the object can be used. In this section we consider external event-statements, which are allowed to use external attributes too. An external event-statement is matched with external event-statements in one or more other objects and these event-statements together describe a single event-action, which is executed jointly by all the involved objects. In an external event-statement an object may read attributes from all of the involved objects, but is only allowed to update its own variables.

An external event-statement may have the following form:

```
[* match OBJ1.PROC1(....)
   when predicate
   do PROC2(....) *]
```

The match-clause specifies that the event-statement must be executed jointly with the procedure PROC₁, declared and executed by object OBJ₁. In general an external event-statement may be synchronized with any number of different objects. Then the match-clause contains a list of object/procedure specifications. The when-clause and the do-clause play similar roles as for internal event-statements, except that the when-clause can use data and functions belonging to any of the involved objects. The do-clause can only use internal attributes.

If an event-statement in its match-clause specifies an object and a procedure, the statement must be matched with an event-statement, which is in the specified object and which has the specified procedure in its do-clause. If an event-statement in its match-clause specifies an object, but no procedure the statement must be matched with an event-statement, which is in the specified

object and has no do-clause. In other words: there has to be a total match between the specifications in the match/do-clauses of matching event-statements. If more than two event-statements are involved, this total match has to be fulfilled for each pair of statements, where the match-clause of one specifies the other object and vice versa. The match of two event-statements may be indirect, in the sense that both are matched with a third event-statement as shown in fig. 3.2, where all three event-statements are to be executed jointly:

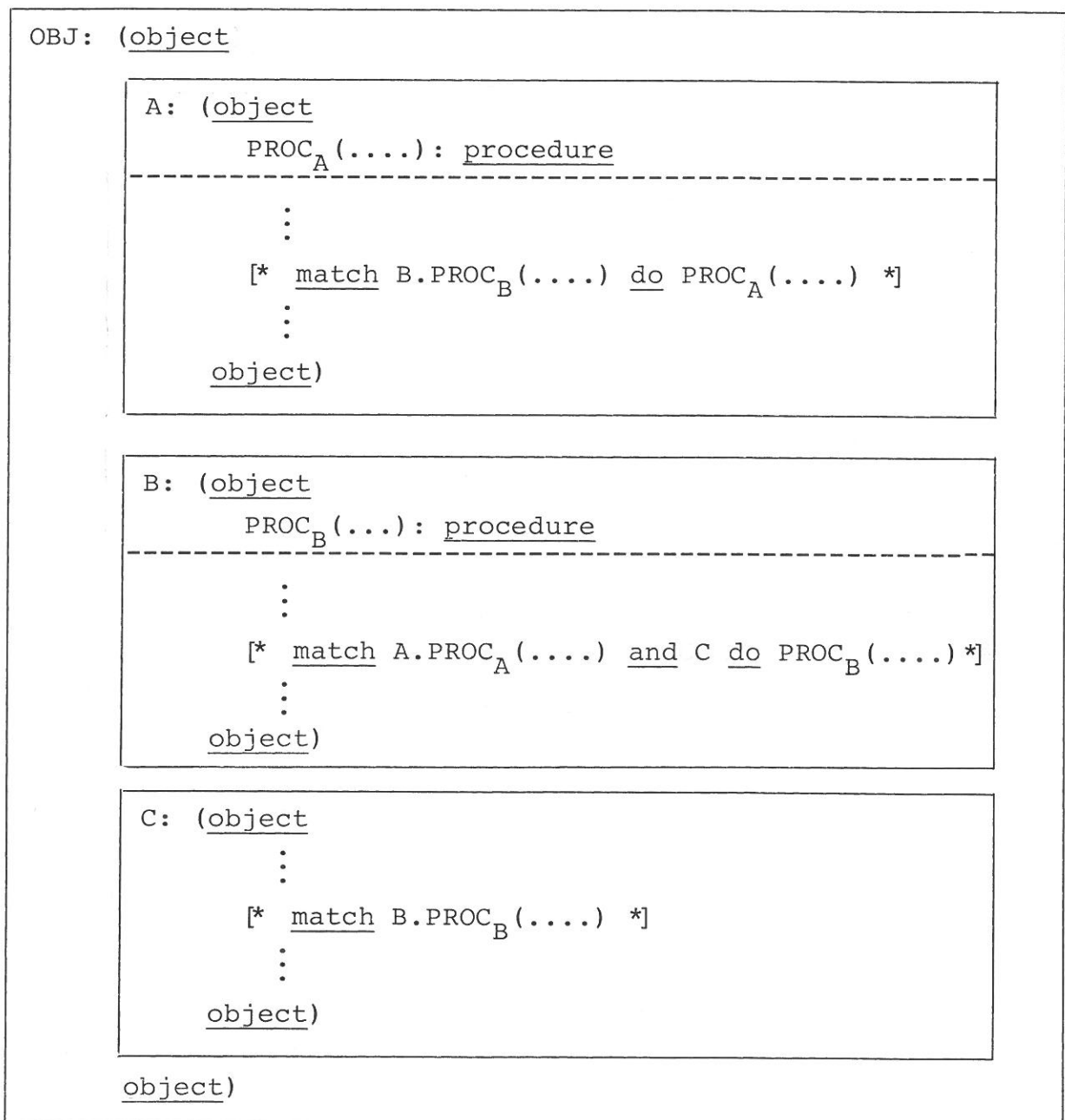


Figure 3.2

As indicated above, input-parameters may be specified in match-clauses as well as in do-clauses. When parameters of an external procedure are specified in a match-clause, the corresponding parameter types must be useable. Output-parameters can only be specified in do-clauses.

To sum up, the three possible clauses in an external event-statement have the following purposes:

- the match-clauses specify the event-statements to be synchronized with. This match is static in the sense that it does not depend on the present system state.
- the when-clauses specify the conditions under which the joint event-action can be executed. The predicates of all involved event-statements must be fulfilled. This test is dynamic in the sense that it depends on the present system state.
- the do-clauses specify the joint event-action to be executed. The joint state transformation is obtained by composition of the state transformations of the individual event-statements.

In section 3.9 we show how procedure templates may be used to relax the demand of total match in a safe way. That allows us to model e.g. "entries" as found in the "rendezvous" concept of ADA, [Ichbiah et al, 80], in a simple way.

As an example of matching external event-statements, consider the following description of parts of PHONES and USERS in a TELEPHONE COMMUNICATION system, fig. 3.3:

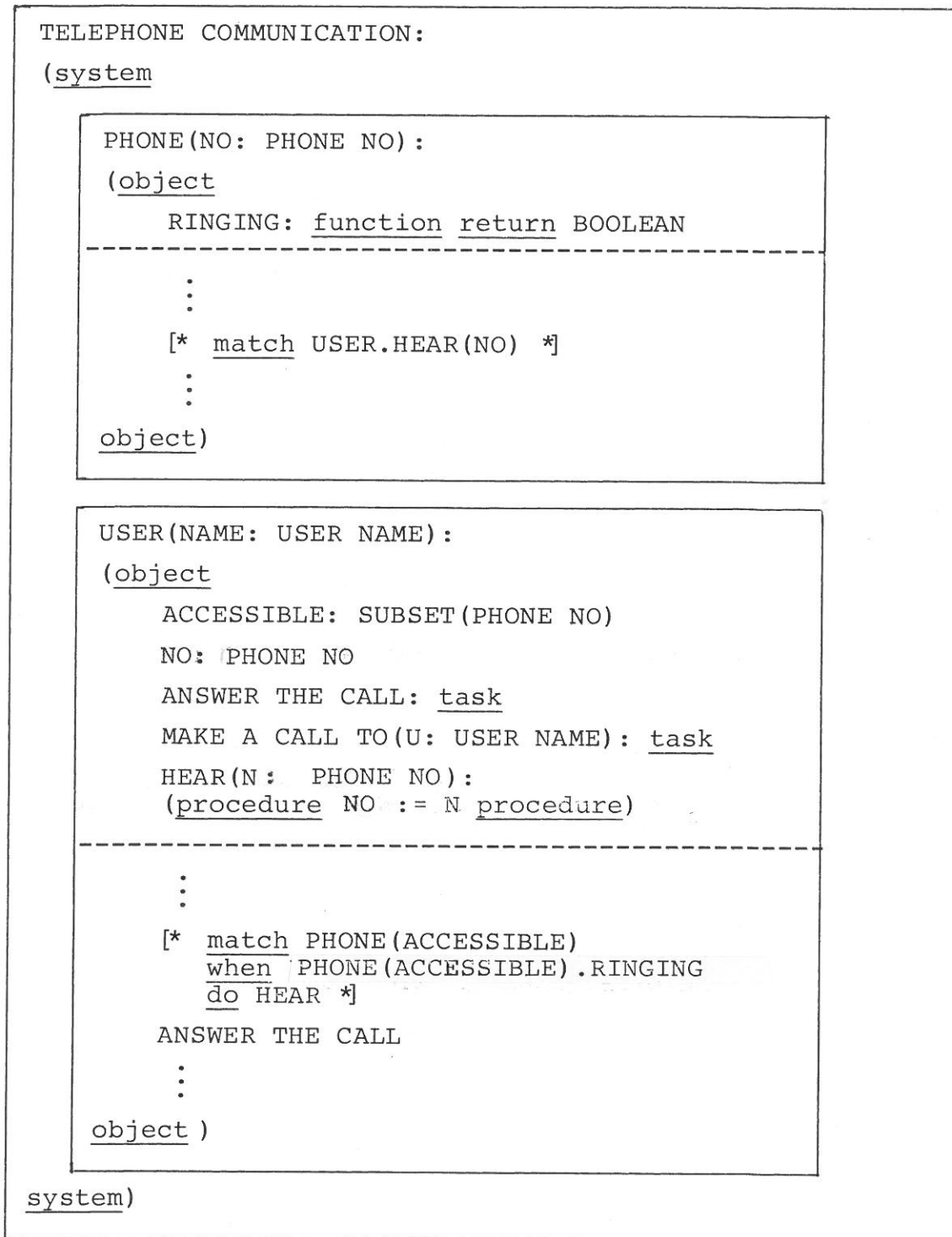


Figure 3.3

Each PHONE has a boolean function RINGING and an external event-statement: [* match USER.HEAR(NO) *]. This statement must be executed jointly with a USER executing its HEAR procedure and the actual-parameter is the NO of the PHONE.

Each USER has a variable ACCESSIBLE describing the subset of PHONES, which he at present can use. Moreover he has an external event-statement describing how to HEAR a RINGING PHONE. The match-clause and when-clause demand joint execution with a RINGING PHONE in the ACCESSIBLE subset. The HEAR procedure in the do-clause assigns the NO of the matching PHONE to the NO variable in the USER. When a USER has HEARD a RINGING PHONE he goes on by ANSWERING THE CALL.

For convenience we have in the match-clause of the USERS, included some state dependent information: the value of ACCESSIBLE. It is easy to move this information to the when-clause, and semantically the external event-statement in question is identical to

```
[* match PHONE
  when  PHONE.NO ∈ ACCESSIBLE,
        PHONE.RINGING
  do HEAR *].
```

3.6 BLOCKS AND ACTION PATTERNS

In the first five sections of this chapter we have seen how to describe equational-statements and event-statements. These two kinds of statements constitute the atoms from which more complex statements can be built. In this section we shall discuss how to group sequences of statements and declarations into suitable units by means of blocks and tasks. We shall also discuss how to group sequences of partial-events and declarations by means of event-blocks, procedures, evaluations and functions. The ideas behind the six structuring concepts in this section are well-known in most programming languages, and thus we shall only give a brief description.

Block-statements

The block-statement serves two different purposes: It may introduce a set of new attributes with a scope which is limited to the block, and thus help to modularise the description. Moreover it groups a sequence of statements to a single statement, which then can be used to built more complex statements. A block-statement may have the following form:

```
(block
  declarations
  action-part
block)
```

Task-statements

A task-statement specifies the call of a task. The task concept of Epsilon is similar to the procedure concept in most programming languages. In Epsilon tasks are used to group statements, while procedures and functions are used to group partial-events. A task-declaration may have the following form:

```
task-name (formal-parameters):
(task
  declarations
  action-part
task)
```

while a task-statement (task call) may have the form:

```
task-name (actual-parameters)
```

Task-declarations are not allowed to contain object-declarations. Tasks are similar to blocks, except that they define a pattern of actions, which are not immediately executed when declared, but only when explicitly called by the object which contain the task-declaration. In addition a task-declaration may define a

set of formal-parameters, which in each call is associated with a set of actual-parameters, and this enables the actions executed at two different calls of the same task to be slightly different.

Input parameters are call by value while output parameters are call by result (see [Gordon, 79]). The choice between these parameter mechanisms is in the task-declaration indicated by the keywords "in" and "out". Call by value is default. In this paper we shall use ADA-notation, [Ichbiah et al, 80] for formal- and actual-parameters. A user of Epsilon is free to choose any suitable parameter notation.

Execution of a task-statement consists of three parts, of which the first and third may be empty:

- (a) If the task has input parameters execution of the task-statement starts with an event-action which updates the corresponding formal-parameters with the values specified by the actual-parameters.
- (b) Execution of the statements contained in the task-declaration.
- (c) If the task has output paramters execution of the task-statement ends with an event-action which updates the corresponding actual-parameters with the values specified by the formal-parameters.

Event-blocks

An event-block is similar to a block-statement, except that it is executed as part of a single event-action and thus its action-part is only allowed to contain partial-events. An event-block may have the following form:

```
(event-block
  declarations
  partial-events
event-block)
```

Event-blocks are not allowed to contain object- or task-declarations.

Procedure-calls

A procedure is similar to a task, except that it is executed as part of a single event-action, and thus its action-part is only allowed to contain partial-events. A procedure-declaration may have the following form:

```

proc-name (formal-parameters):
  (procedure
   declarations
   partial-events
   procedure)

```

while a procedure-call may have the form:

```

proc-name (actual-parameters)

```

Procedure-declarations are not allowed to contain object- or task-declarations. A procedure-call can occur either in an internal event-statement or in an external event-statement executed jointly with a set of other external event-statements. In the last case the actual-parameters may be specified jointly by the object which call the procedure and the objects matching the call, in the sense that some parameters are specified by one object and some by other objects. An object can only specify a parameter if the corresponding type is useable.

Evaluations

An evaluation produces a value. Evaluations are executed as parts of expressions or predicates, and thus their action-part is only allowed to contain partial-events. An evaluation may have the following form:

```

    (evaluation
     declarations
     partial-events
    evaluation) return type-name

```

Evaluations are not allowed to contain object-, task- or procedure-declarations. The return-clause specifies the type of the value produced. Execution of the partial-events must end with a return-construct which specifies the value to be returned. An evaluation has no side-effects, i.e. it can only update variables declared in the evaluation itself.

Function-calls

A function-call produces and returns a value. Functions are executed as parts of expressions or predicates, and thus their action-part is only allowed to contain partial-events. A function-declaration may have the following form:

```

func-name (formal-parameters):
  (function
   declarations
   partial-events
  function) return type-name

```

while a function-call may have the form:

```

func-name (actual-parameters)

```

Function-declarations are not allowed to contain object-, task- or procedure-declarations. The return-clause specifies the type of the value to be returned by the function. Execution of the partial-events must end with a return-construct which specifies the value to be returned. A function can only use internal data-attributes and it has no side-effects, i.e. it can only update variables which are declared in the function itself and it cannot have output parameters.

External functions are only useable in the predicates of external event-statements, which are to be executed jointly with the object containing the function, and in the predicates of equational-statements. In both cases the parameter types and the return type have to be useable. A function may simultaneously be called by several objects, in the when-clauses of a joint event-action or in the predicates of a set of concurrent equational-actions. This causes no problems due to the absence of side-effects.

3.7 CONTROL-SECTIONS

So far we have only considered simple sequences of equational- and event-statements. Often this is not sufficient, and thus, we show in this and the following section how to handle interruptions, loops and branches by means of control- and select-sections.

Control- and select-sections may be used to supervise the statement-list of an object, task or block. Then the statement-list is called a supervised-section. A control-section can interrupt any of the equational-actions in the supervised-section. A select-section can only interrupt the last equational-action in the supervised-section, and thus it determines what to do when execution of the supervised-section finishes by itself. In this section we describe control-sections, while select-sections are described in the following section.

A control-section may have the form shown in the lower rectangle of the fig. 3.4. It consists of the keyword "control" followed by a non-empty set of guarded-statement-lists. Each of these contains an arrow, an event-statement called a guard, a state-

ment-list, and a continuation description (One of the keywords: "continue", "leave" and "restart").

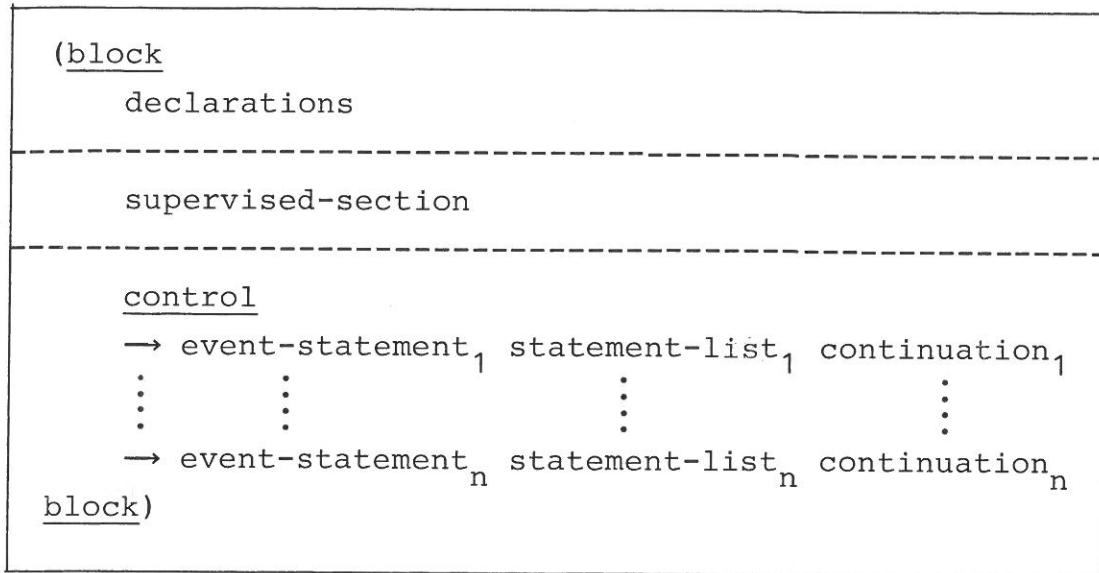


Figure 3.4

Execution of the block starts with the actions described by the supervised-section. During the execution of one of the equational-actions of the supervised-section, some of the guards in the control-section may become executable. If the guard is an internal event-statement, this simply means that the predicate becomes satisfied. For an external event-statement it must moreover be possible to match the event-statement with some set of external event-statements from other objects, in such a way that each involved object is ready to execute its event-statement, which implies that each has a satisfied predicate.

If some of the guards become executable, the ongoing equational-action is stopped and one of the guards together with its following statement-list is executed. Next, depending upon the continuation description, execution may either continue the block (at the stopped equational-action in the supervised-section), leave the block (in favour of the next statement) or restart the block (at the first statement in the supervised-section). When an equational-statement is restarted or continued, its barred variables are updated to the present values of the corresponding variables. An empty continuation description is for control-sections equivalent to a continue.

If several guards (all supervising the ongoing equational-action) becomes executable, they are said to be in conflict. Conflicts are resolved by non-deterministic choice and thus the order of the guarded-statement-lists is immaterial. It should be noted that event-actions are treated as instantaneous and indivisible; thus it is only the equational-actions of a supervised-section, which may be stopped. The supervised-sections will in this chapter be assumed always to end with an equational-action. This guarantees that there is an equational-action to be executed until one of the guards becomes executable. We return to this question in chapter 6, which defines the formal semantics.

As an example of a control-section we reconsider our telephone USERS once more. In fig. 2.4 a USER is most of the time not using a PHONE. The state of not using a PHONE may be changed either because the USER wants to make a call or because an accessible PHONE rings. These two kinds of possible interruptions are more formally described in fig. 3.5 by means of a control-section.

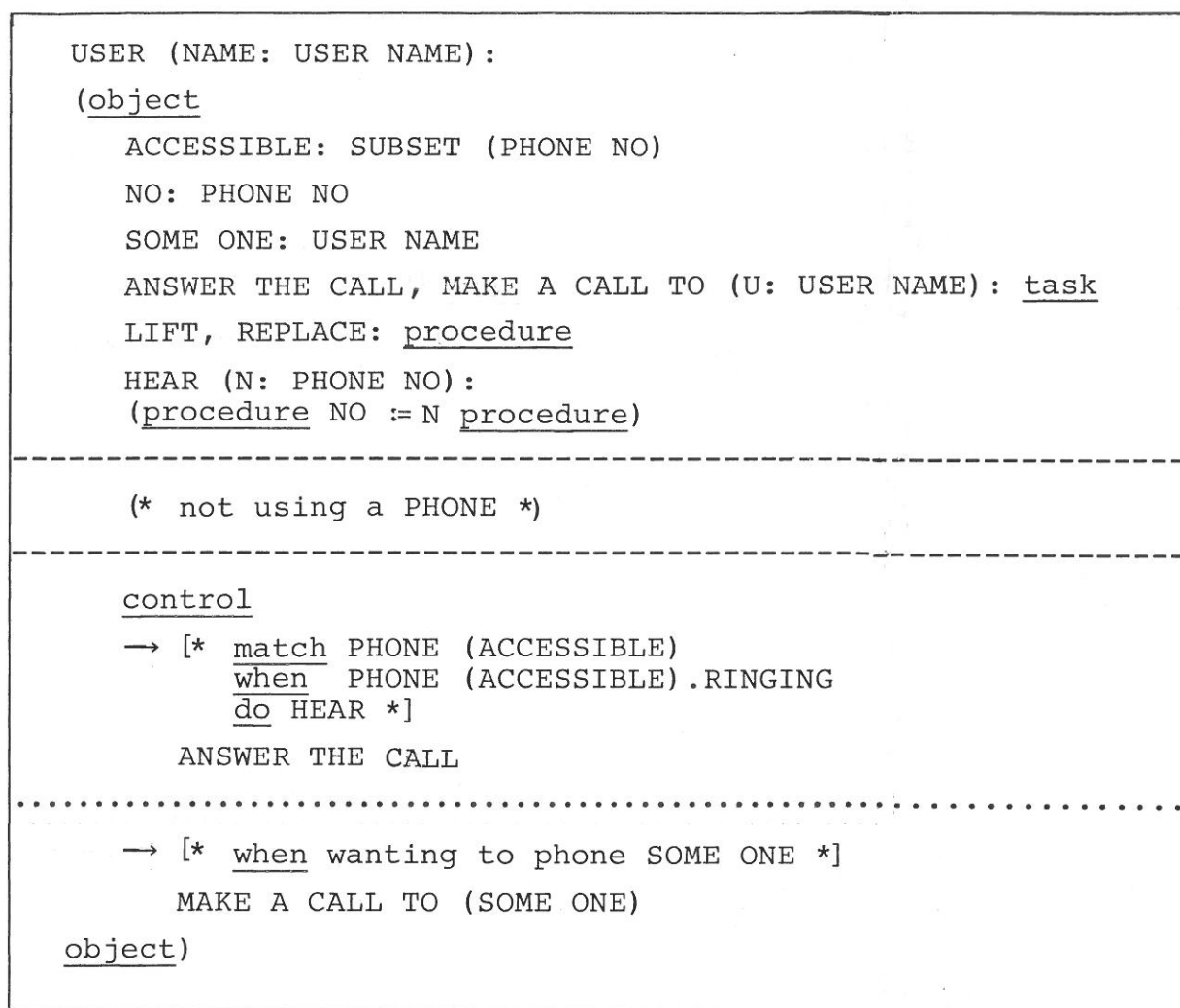


Figure 3.5

The equational-statement in the supervised-section is executed until one of the two guards becomes executable. Then the guard and the following task-statement are executed, and the USER continues execution of the equational-statement, which was stopped by the guard.

Next we elaborate in fig. 3.6 the description of how to ANSWER THE CALL. A USER begins to execute ANSWER THE CALL when he HEARS a PHONE RINGING. He walks towards the RINGING PHONE changing his POSITION as he walks. This equational-action is controlled by two

guards. Either the USER reaches the PHONE while it is still RINGING; then he LIFTS the receiver, leaves the block and starts talking. Or he is too late, in which case he leaves the task and continues not using a PHONE (as described in fig. 3.5).

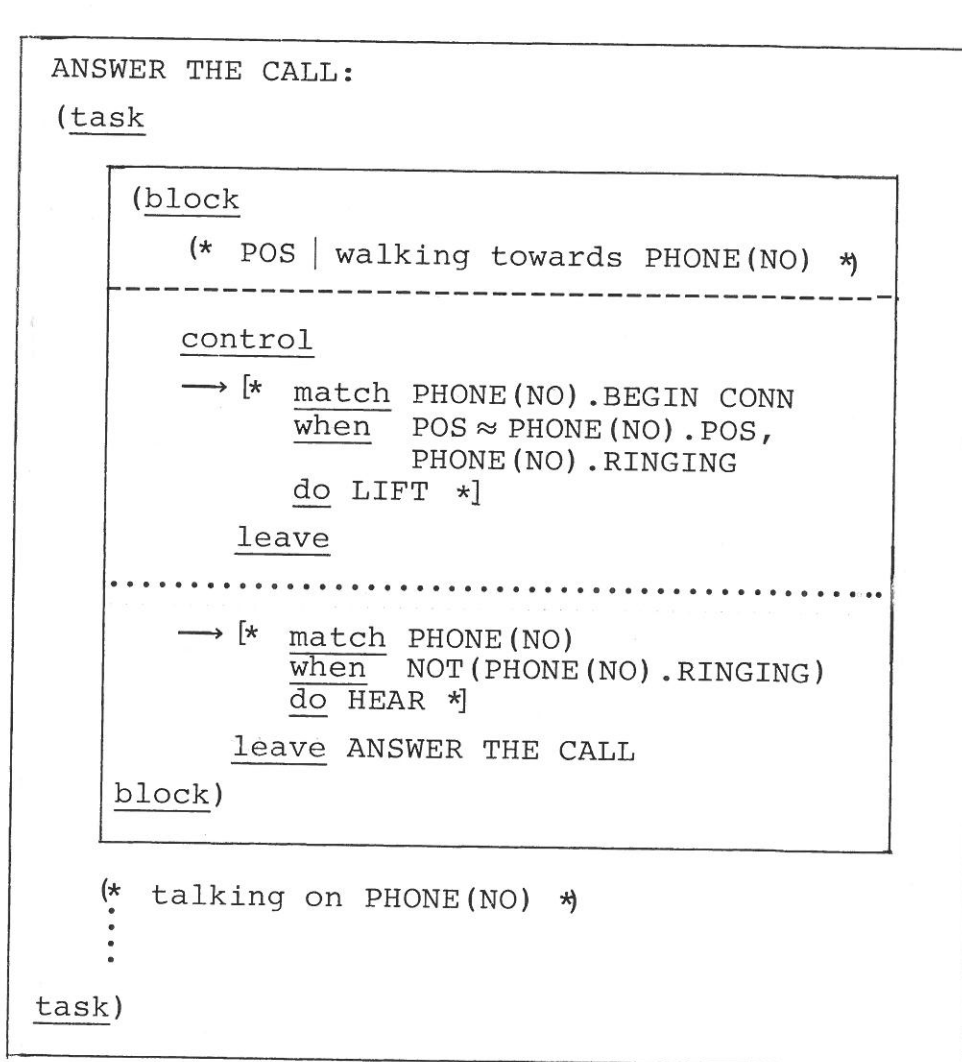


Figure 3.6

Normally a continuation description refers to the nearest block, task or object. As an example, the continuation description of the first guard in fig. 3.6 demands execution to leave the block. A continuation description may, however, also contain a continuation-name referring to an enclosing block, an enclosing task, or the entire object. If a continuation-name is present, the continuation description must be contained in the corresponding block, task or object. It is then the supervised-section of this which is continued, left or restarted. As an example, the continuation description of the second guard in fig. 3.6 demands execution to leave the task ANSWER THE CALL. Next we modify in fig. 3.7 the description of the PHONES in a way that corresponds to the description of USERS in fig. 3.5.

The STATUS of a PHONE may change from INACTIVE (with the RECEIVER DOWN) to the situation where the RECEIVER is UP and you hear a CONTInuous tone. Next to the situation, where a PHONE NO has been dialled and you hear NO TONE, until either you get a tone with SHORT intervals (indicating that the called PHONE already is engaged) or a tone with LONG intervals (indicating that the bell is RINGing at the called PHONE). In the latter situation the RECEIVER may be taken UP at the called PHONE and the two PHONES are CONNec-ted, until the calling PHONE returns to INACTIVE thereby making the called PHONE DISConnected.

Initially the PHONE is INACTIVE with the RECEIVER DOWN, as described by the equational-statement. This action is controlled by three guards. The first guard allows a USER to HEAR whether a PHONE is RINGing; it matches the second guard in ANSWER THE CALL of fig. 3.6 and the PHONE continues being INACTIVE with the RECEIVER DOWN. In fact it also matches the first guard of fig. 3.5, but an event-action corresponding to this match will never be executed, since the predicate of the latter guard is unsatisfied. The second guard is intended to match a calling PHONE, which identify itself via the actual-parameter of BEGIN RING. Notice that we have simplified the object hierarchy from chapter 2 and now assume that PHONES communicate directly and not via a NETWORK. Finally the third guard matches a USER, who LIFTs the RECEIVER as part of the task MAKE A CALL TO.

```

PHONE (NO: PHONE NO):
(object
  STATUS:      (INAC, CONT, NO TONE, SHORT,
                LONG, RING, CONN, DISC)
                init INAC
  RINGING:     (function return STATUS = RING
                function) return BOOLEAN
  RECEIVER:    (DOWN, UP) init DOWN
  CONNEXION:   PHONE NO
  BEGIN INAC:  (procedure STATUS := INAC
                RECEIVER := DOWN procedure)
  BEGIN RING(N: PHONE NO):
                (procedure STATUS := RING
                CONNEXION := N procedure)
  BEGIN CONN:  (procedure STATUS := CONN
                RECEIVER := UP procedure)
  BEGIN CONT, BEGIN NO TONE, BEGIN LONG,
  BEGIN SHORT, BEGIN DISC: procedure
  RECEIVE A CALL, CALLING: task
-----
  (* STATUS = INAC, RECEIVER = DOWN *)
-----
  control
  → [* match USER.HEAR(NO) *]
  .....
  → [* match PHONE.BEGIN LONG
     do   BEGIN RING *]
     RECEIVE A CALL
  .....
  → [* match USER.LIFT
     do   BEGIN CONT *]
     CALLING
object)

```

Figure 3.7

```

RECEIVE A CALL:
(task
  (* STATUS = RING, RECEIVER = DOWN *)

-----

  control
  → [* match USER.HEAR(NO) *]

.....

  → [* match USER.LIFT and
      PHONE(CONNEXION).BEGIN CONN
      do BEGIN CONN *]
      .
      .
      .
      leave

.....

  → [* match PHONE(CONNEXION).BEGIN INAC
      do BEGIN INAC *]
      leave
task)

```

Figure 3.8

Next we elaborate in fig. 3.8 the description of the task RECEIVE A CALL of the PHONES:

When RECEIVE A CALL starts execution the PHONE is RINGing with the RECEIVER DOWN, as described by the equational-statement. This action is controlled by three guards. The first guard allows a USER to HEAR whether a PHONE is RINGing; it matches the first guard in fig. 3.5 and the PHONE continues RINGing. In fact it also matches the second guard in fig. 3.6, but an event-action corresponding to this match will never be executed, since the predicate of the latter guard is unsatisfied. The second guard allows a USER to LIFT the RECEIVER. This guard matches the first guard in fig. 3.6 and a guard in the CONNEXION (calling PHONE). The third guard matches the CONNEXION becoming INACTIVE with the RECEIVER DOWN.

As the last example of control-sections we improve our description of the PHONES. A PHONE may, in all possible states be inspected by a USER to find out whether it is RINGing. This means that all equational-statements in the PHONES must be controlled by a guard like the first in fig. 3.7 and the first in fig. 3.8. These identical guards are in fig. 3.9 below combined to a single guard, which then controls all other statements of the PHONES.

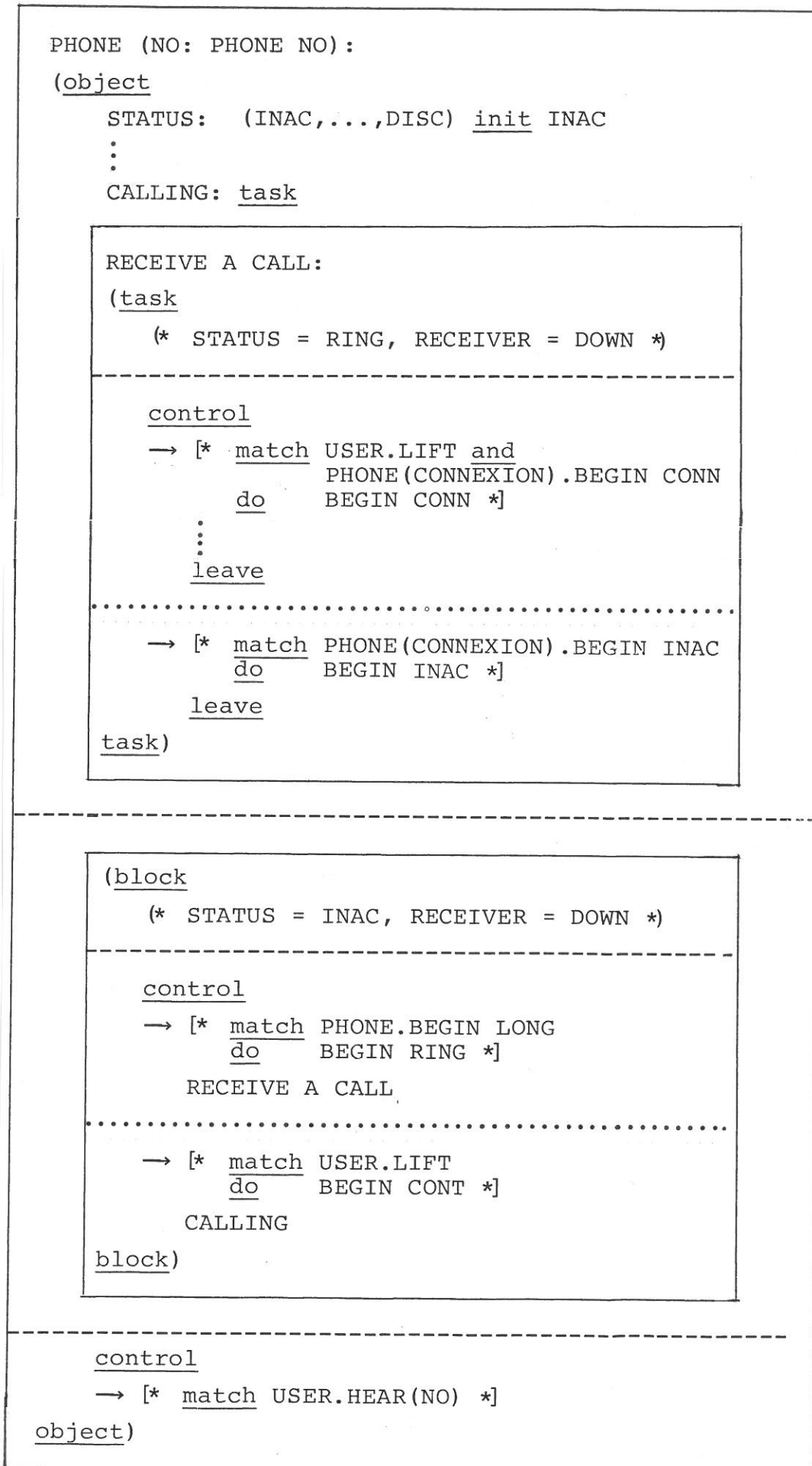


Figure 3.9

3.8 SELECT-SECTIONS

A select-section may have the form shown in the lower rectangle of fig. 3.10. It consists of the keyword "select" followed by a non-empty set of guarded-statement-lists, which have the same form as for control-sections.

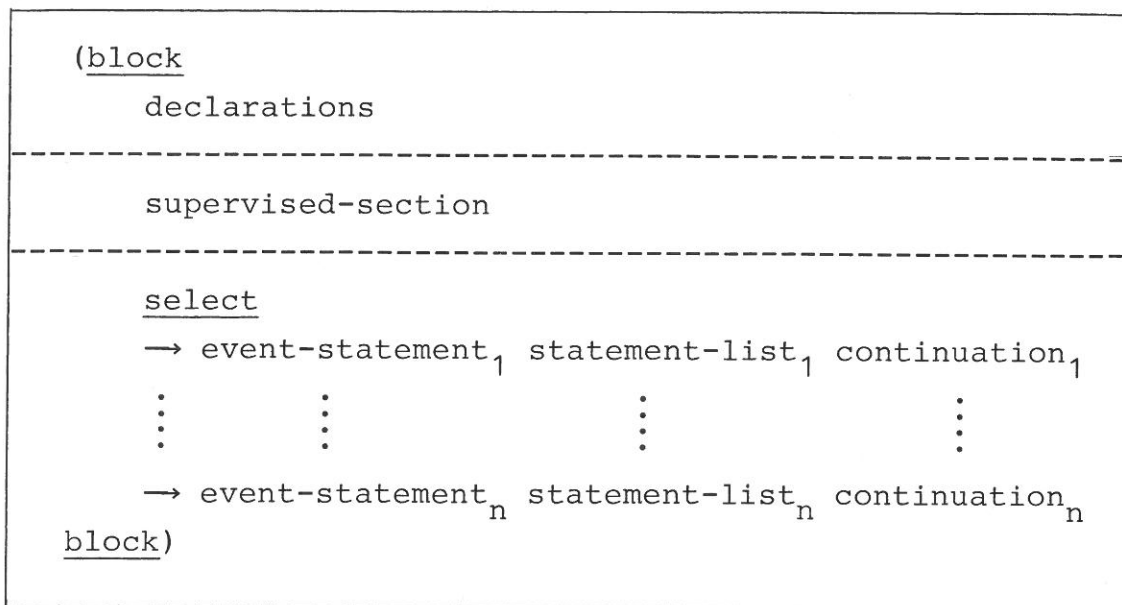


Figure 3.10

Execution of the block starts with the actions described by the supervised-section. In contrast to control-sections, the guards of a select-section can only stop execution of the last equational-action in the supervised-section, and thus the select-section determines what to do, when execution of the supervised-section finishes by itself. As for control-sections we assume in this chapter that the supervised-section ends with an equational-action. This guarantees that there is an equational-action to be executed until one of the guards becomes executable.

Equational-actions can only be continued, after they were stopped by a control-section: Thus a select-section is not allowed to con-

tinue its own supervised-section. But other supervised-sections may be continued, if the select-section is contained in their control-section. An empty continuation description is for select-sections equivalent to a leave.

When the supervised-section consists of a single equational-action, there is no difference between control and select. (In this case the continuations restart and continue are equivalent). It is thus possible to rewrite a number of the examples in the preceding section using select-sections. However, with more elaborate supervised-sections containing several equational-statements there is a profound difference between control and select.

As an example of a select-section we describe in fig. 3.11 the daycycle of a PERSON. This may be considered as an elaboration of the equational-statement "not using a PHONE".

A PERSON gets out of bed, takes a shower, eats breakfast and then is ready to do other things. Now a selection is made: If the current day is a workday the first guard in the select-section is executed and the PERSON starts working. If it is a day off the second guard is executed together with its statement-list. In both cases the PERSON finally leaves the block; then he goes to bed and starts sleeping. The sleep is ended when one of the guards in the lower select-section becomes executable.

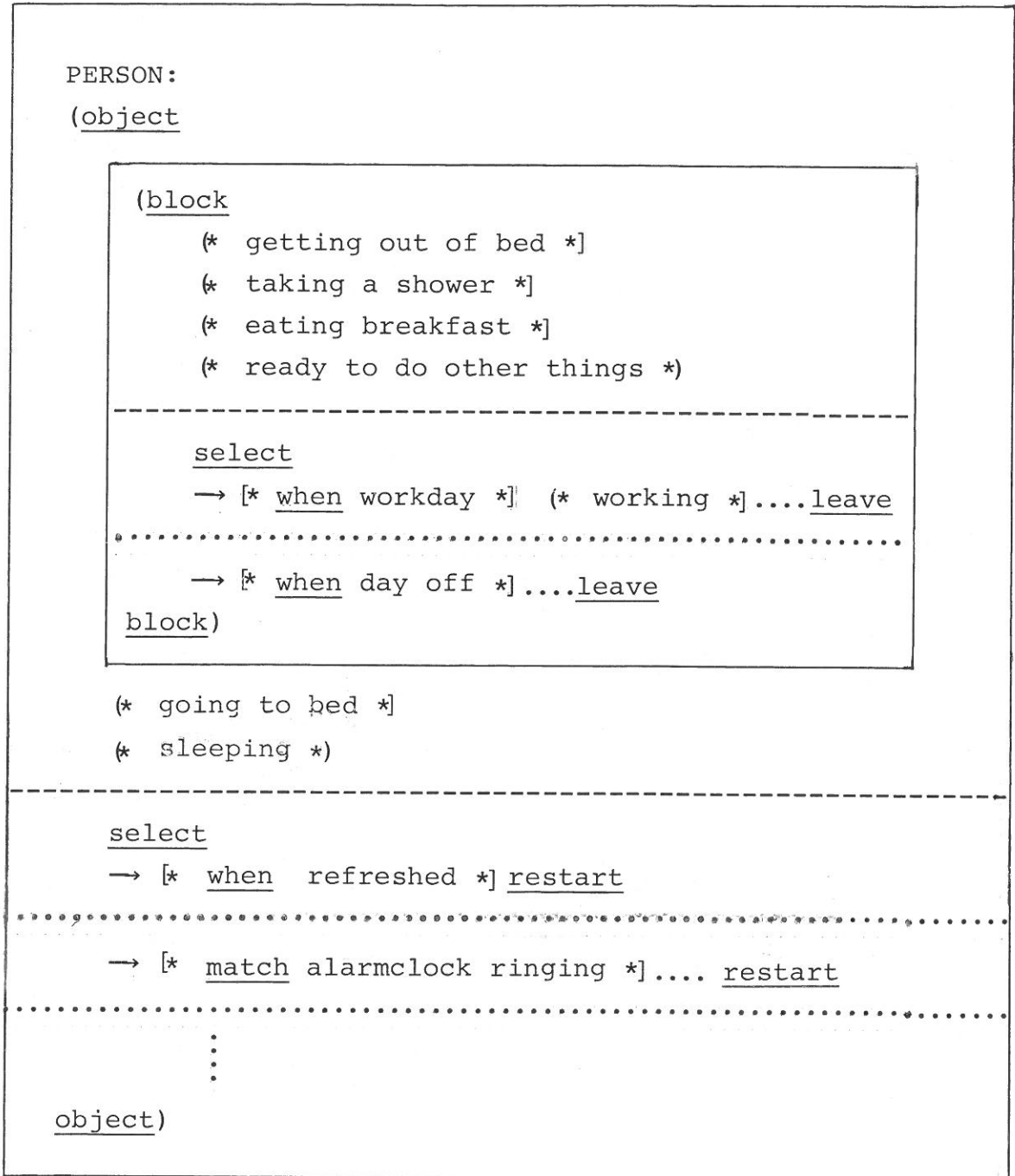


Figure 3.11

A block, task or object may have both a control-section and a select-section. As an example consider fig. 3.12, where we have combined the descriptions of fig. 3.5 and fig. 3.11 to sketch the daycycle of a PERSON, who among other things can use PHONES.

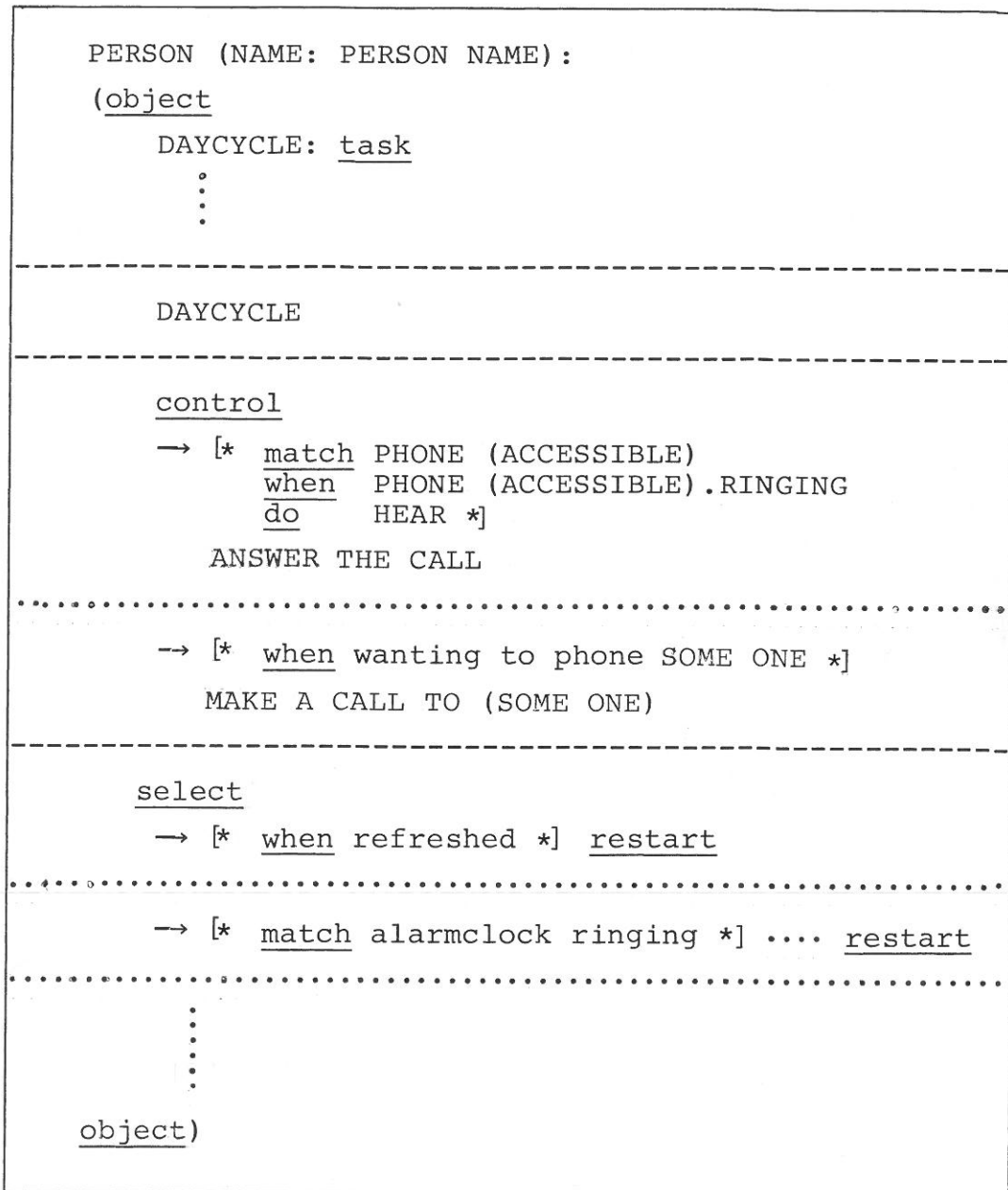


Figure 3.12

DAYCYCLE is a task containing the actions described in the upper part of fig. 3.11. Its execution is supervised by a control-section and by a select-section. The control-section may stop any of the equational-actions in DAYCYCLE, while the select-section may stop only the last equational-action, i.e. sleeping.

It is easy to verify that all "normal" control structures, such as if-statements and while-statements, can be realized, in a simple way, by means of control- and select-sections.

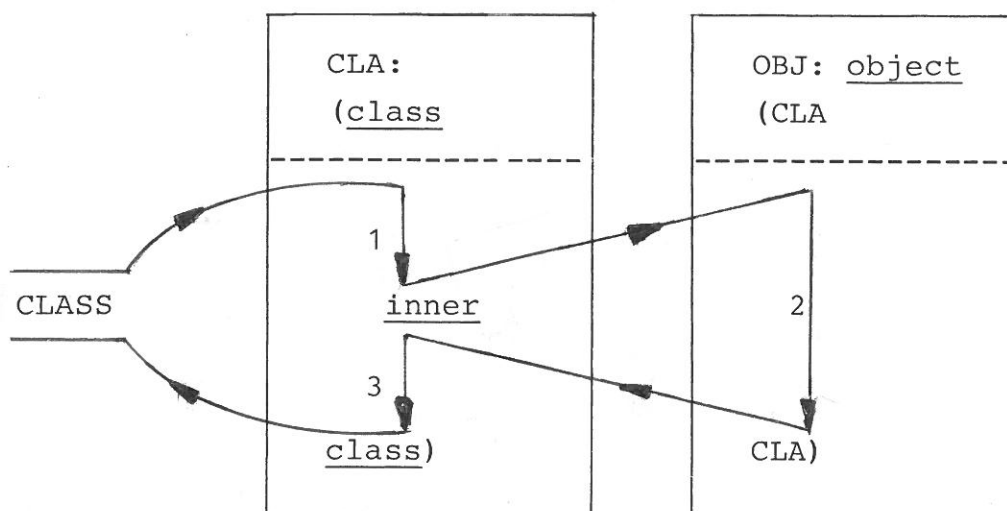
3.9 TEMPLATES (EXTENSION)

In this section we return to the subject of templates i.e. the use and extension of patterns. First we show how action-parts are combined when templates are involved. Then we consider templates for blocks and event-blocks. We illustrate how templates can be used to relax, in a safe way, the demand for a total match between match/do-clauses of external event-statements. Finally we show how to describe "entries" as found in ADA.

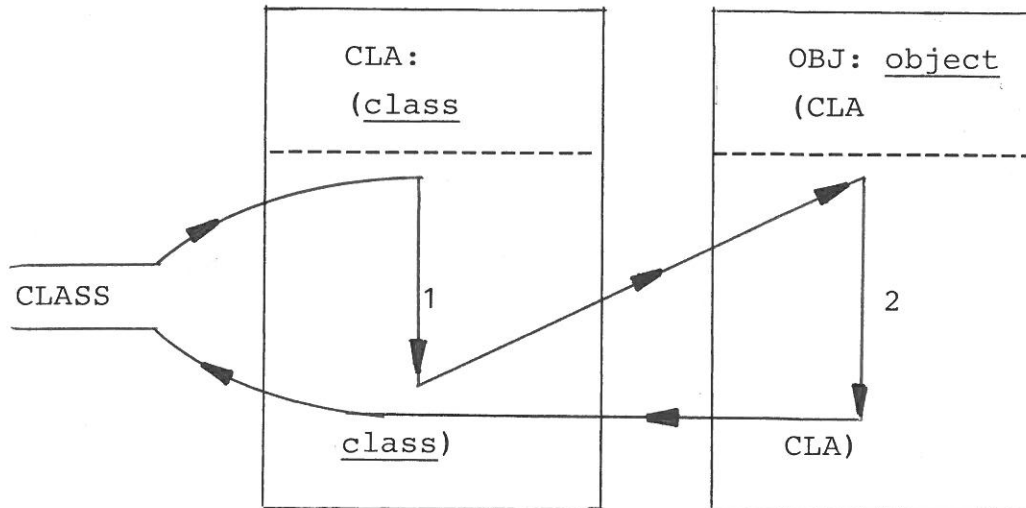
Combination of action-parts

In section 2.5 we described how to combine the attribute-parts by means of disjoint union. Here we describe how to combine the action-parts. We shall describe the rules by means of classes and objects, but they work for the other kinds of attributes as well.

Let OBJ be an object with template CLA. In Simula execution of OBJ normally starts with the actions of CLA followed by those described in OBJ itself. This can be changed by the keyword "inner" which in CLA specify a call of the actions described in OBJ itself:



The default case, where all actions in CLA are executed before the actions described in OBJ itself, corresponds to an implicit "inner" placed at the end of each action-part:



An "inner" always refers to the nearest enclosing declaration. When this is a class- or task-declaration an "inner" is allowed where a statement is. (This is also the case in Delta, but not in Simula). When the nearest enclosing declaration is a procedure- or function-declaration an "inner" is allowed where a partial-event is.

Templates for blocks and event-blocks

Tasks and procedures can be used as templates to describe abstract control structures. In fig. 3.13 we declare a task CLOCK, which can evoke a time-out for a block having it as template. During the execution of such a block COUNT is increased whenever a "special event" occurs and the block is left when COUNT reaches the value specified by the actual-parameter. We have in fig. 3.13 omitted the keyword "block" in front of (CLOCK CLOCK). This may always be done when we have a block or an event-block specified by form A (cf. section 2.5).

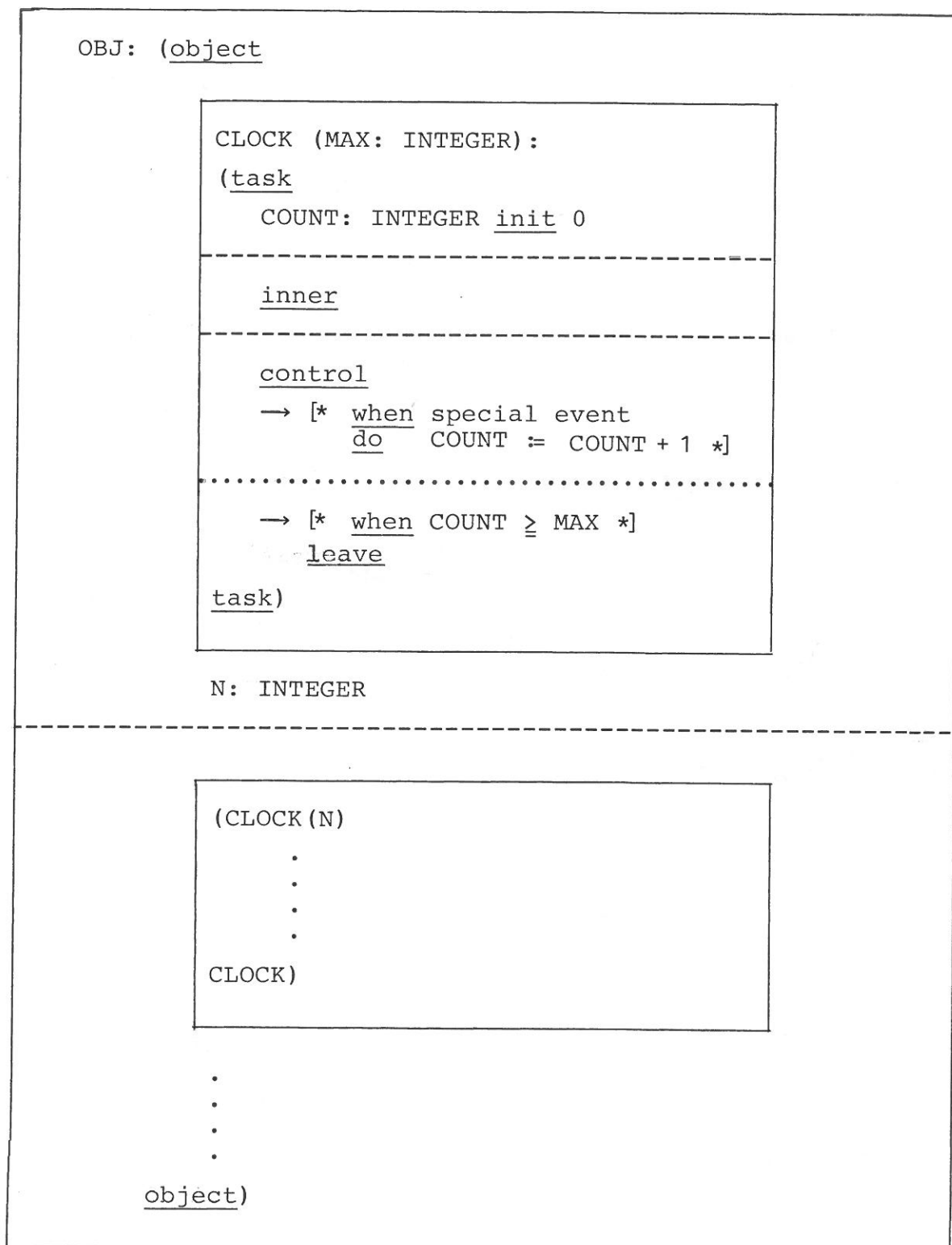


Figure 3.13

In fig. 3.14 we declare a class containing a compound data structure and a procedure which SEARCH through the data structure one element at a time. We also declare an object, where SEARCH is used to compute the sum of the elements in the data structure.

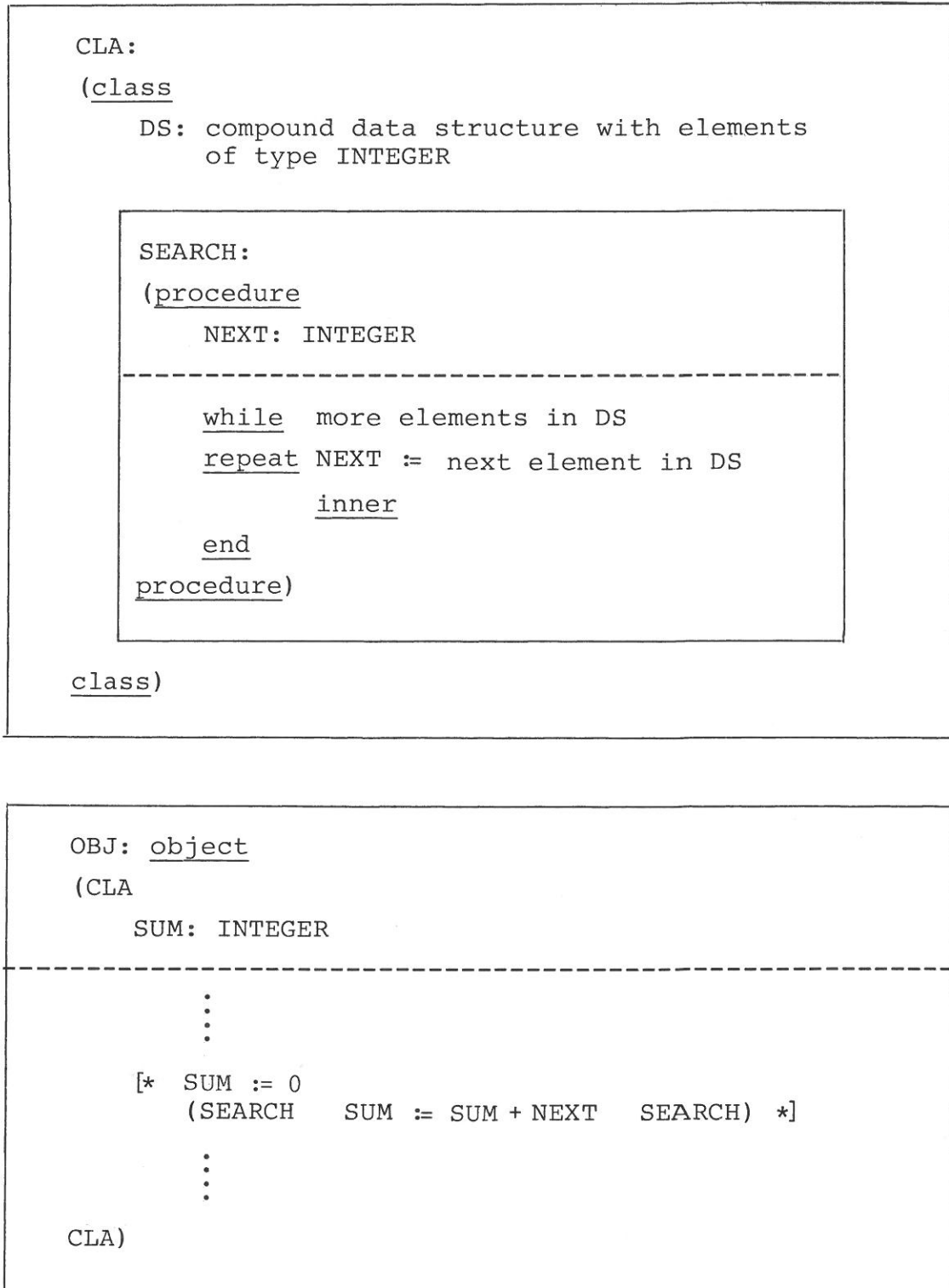


Figure 3.14

Template match

According to the rules given in section 3.5, there must be a total match between the match/do-clauses of external event-statements. In this subsection we illustrate how procedure templates may be used to achieve a more flexible and yet secure communication between objects.

In fig. 3.15 we consider two objects A and B with a number of external event-statements. We only have a match for the two pairs of event-statements, which are indicated by the two upper unbroken lines.

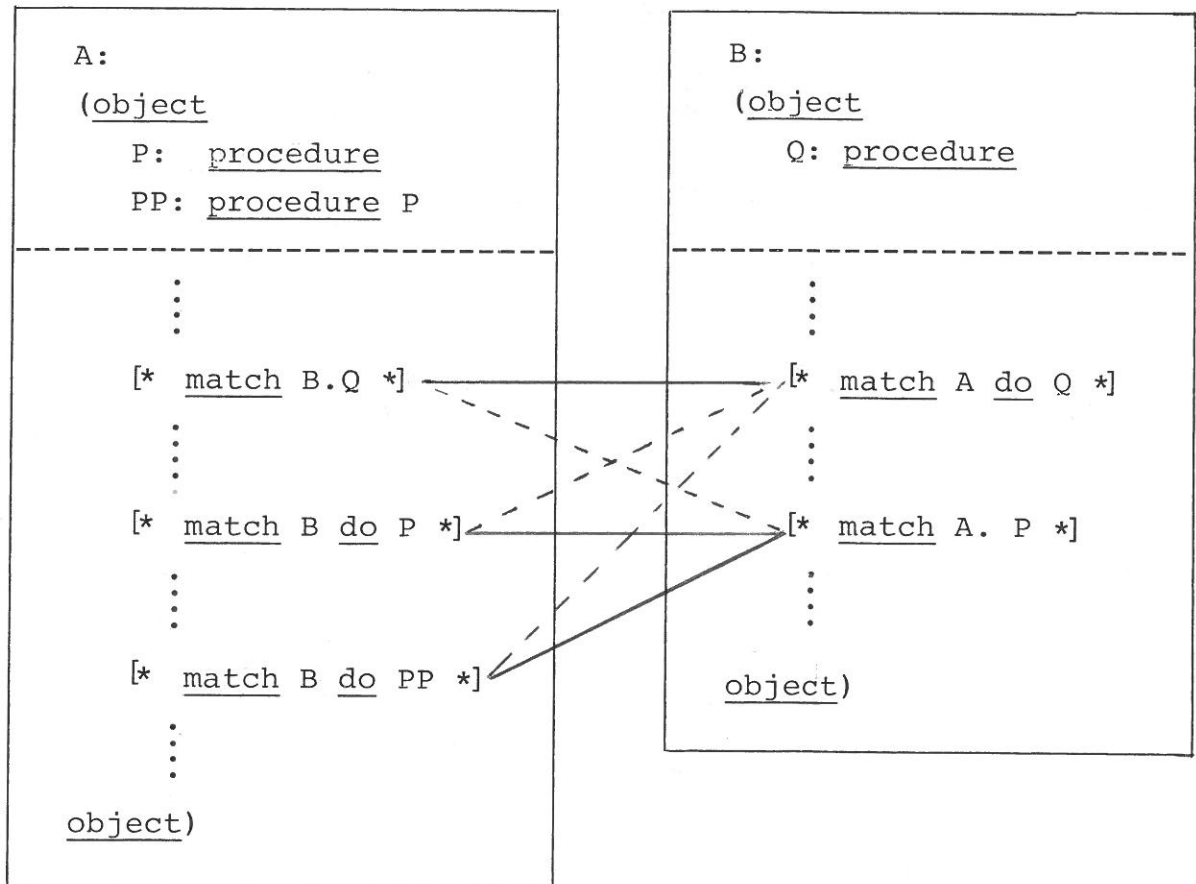


Figure 3.15

We now replace the rule of total match with the rule of template match, which states that a match-clause specifying a procedure P, matches exactly those do-clauses which have a procedure or event-block with P as template. Template match retains security, in the sense that a large number of undesired matches still are excluded. In the example above template match excludes all the pairs connected by dashed lines. Only the pair indicated by the lower unbroken line is added by changing the match rule.

Having introduced the rule of template match we can allow the do-clause of external event-statements to have exactly the same form as for internal event-statements (i.e. be a sequence of partial-events). The do-clause is then either:

- a single procedure-call
- a single event-block
- someother non-empty sequence of partial-events, which we shall consider a shorthand for the event-block containing the sequence of partial-events (and having PROCEDURE as template)

or the do-clause is omitted.

The match-clause then contains, for each matching object A, either:

- a procedure-call: A.PROC_A
- no procedure-call: A

The possible matches between match/do-clauses of the forms above are, in fig. 3.16, indicated by unbroken lines.

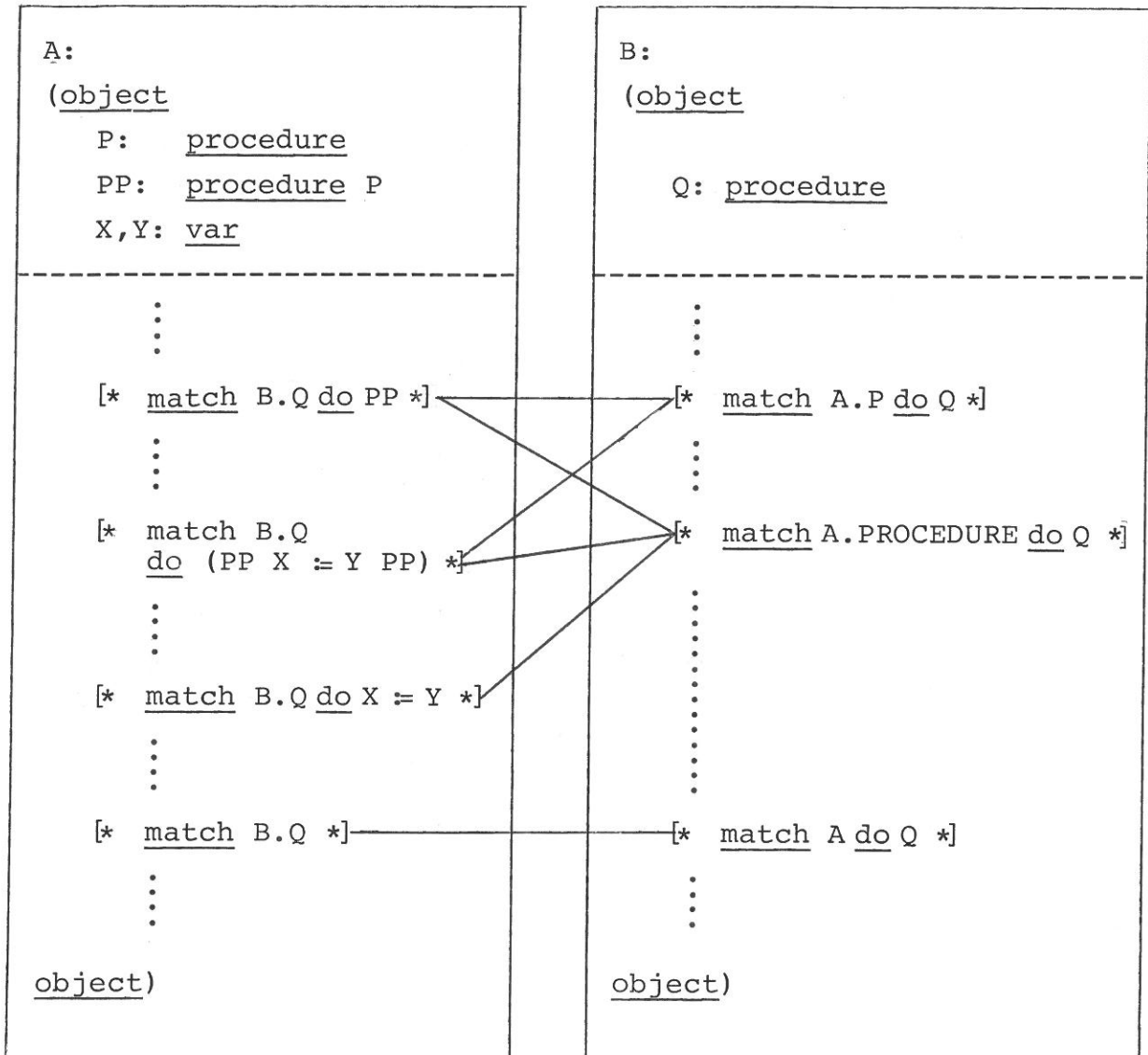


Figure 3.16

We shall also allow template match for the object specifications in match-clauses. Then a match-clause may specify a class CLA, and this means that the event-statement must be matched with an object having CLA as template.

ADA-entries

To illustrate the flexibility added by the template match, we show how to model "entries" as found in the "rendezvous" concept of ADA. Fig. 3.17 shows an ADA program, which is taken from [Ichbiah et al, 80] p. 9.2. It describes a task with a PROTECTED ARRAY. The array can only be accessed via the entries READ and WRITE, and the entry mechanism guarantees mutual exclusion of all accesses to the array.

```

task PROTECTED_ARRAY is
  -- INDEX and ELEM are global types
  entry READ (N : in INDEX; V : out ELEM);
  entry WRITE (N : in INDEX; E : in ELEM);
end;

task body PROTECTED_ARRAY is
  TABLE : array(INDEX) of ELEM := (INDEX => 0);
begin
  loop
    select
      accept READ (N : in INDEX; V : out ELEM) do
        V := TABLE(N);
      end READ;
    or
      accept WRITE(N : in INDEX; E : in ELEM) do
        TABLE(N) := E;
      end WRITE
    end select;
  end loop;
end PROTECTED_ARRAY;

```

Figure 3.17

A corresponding Epsilon description looks as shown in fig. 3.18.

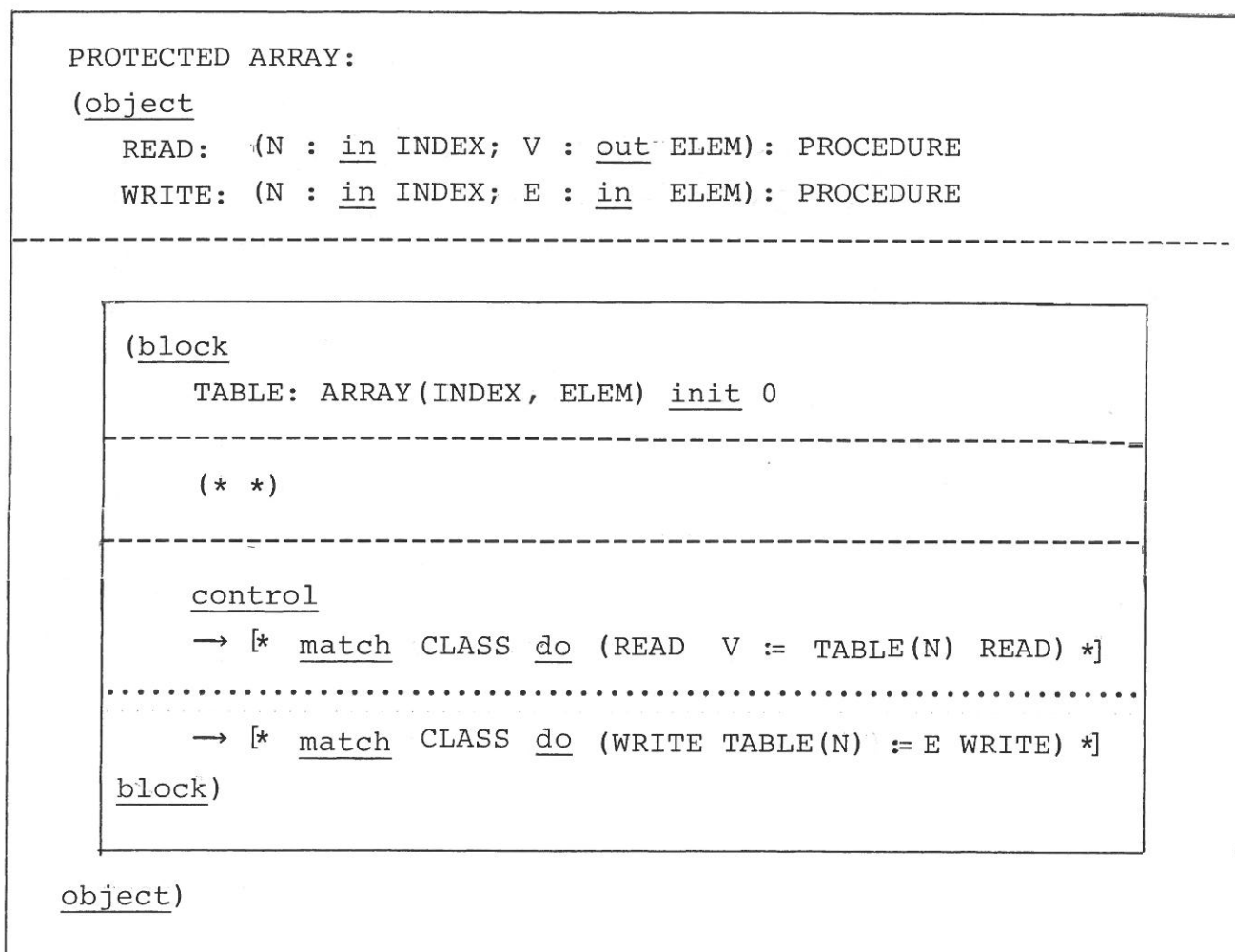


Figure 3.18

The ADA designers argue that it may be inconvenient to be forced to know the communication partner (e.g. for program libraries). However, often we intend procedures to be involved only in synchronization with certain other objects. It is then appropriate to be able to state this restriction explicitly in the program. In the present case we allow, as the ADA program, READ and WRITE to be involved in synchronization with any object. If we want to restrict access to the array, we substitute "CLASS" by the appropriate class- and/or object-names.

Chapter 4

SYNTAX AND STATIC SEMANTICS

4.1	Extended BNF	87
4.2	Syntax of declarations	88
4.3	Syntax of statements	92
4.4	Syntax of partial-events	95

4.1 EXTENDED BNF

This chapter contains a commented syntax of Epsilon: The context free part is defined formally, whereas the context sensitive part (static semantics) is only informally described. In addition some explanatory text concerning the semantics is included. The Epsilon constructs described as extensions in section 1.4, 2.4, 2.5 and 3.9 are neither included in the formal syntax of this chapter nor in the formal semantics in chapter 6.

The syntax of Epsilon is defined using an extended BNF. Non-terminals are sequences of letters, possibly containing a hyphen (-). Terminals are keywords (underlined sequences of letters) or special symbols such as "(", "→", ":", etc. The symbol ::= has the usual BNF-meaning. The righthand side of a production may include regular expressions: the bar | separates alternatives, the brackets {...} denote that the corresponding clause is optional, i.e. may appear zero or one times, {*...*} denote repetition zero or more times, while {+...+} denote repetition one or more times. The BNF-symbols (including the special brackets) do not appear in terminals and thus we need not surround terminals by quotes.

Subscripts on non-terminals indicate the number of the syntax rule in which the corresponding non-terminal is defined. Non-terminals without subscripts (on the righthand side of a production) are not formally defined in this paper. The use of separators in lists is not described in the formal syntax; comma is used to separate names, expressions and predicates.

4.2 SYNTAX OF DECLARATIONS

```
(1) system-description ::=
    name:
    (system
     { * declaration2 * }
     { action-part9 }
    system)
```

This describes the system object. It is allowed to contain all kinds of attributes (declarations).

```
(2) declaration ::=
    object-declaration3
    || task-declaration4
    || procedure-declaration5
    || function-declaration6
    || type-declaration7
    || data-declaration8
```

This introduces six different kinds of declarations. Each declaration is allowed to contain declarations of its own kind and of all succeeding kinds (referring to the list above). As an example procedures are allowed to contain procedures, functions, types and data, but not objects or tasks. Each declaration defines one or more names, and when we in some of the following syntax rules use the non-terminal "object-name" this indicates a name already declared as an object. Analogously for the other kinds of declared names.

(3) object-declaration ::=
 name { ({name:} type-name₇ {(actual-parameters)}) }:
 (object
 {* declaration₂ *}
 { action-part₉ }
object)

This describes either a singular object or a family of objects with identical attribute- and action-parts. If the type is omitted a single object is declared. If present the type must specify a finite set of values and the cardinality of this set determines the number of objects declared. If present the second name defines a constant data-attribute initialised to that value in the finite set which corresponds to the object. This means that inside an object-declaration beginning with "OBJ(ID:....):" the expression "OBJ(ID)" refers to the object itself. Object-declarations are allowed to contain all kinds of attributes.

(4) task-declaration ::=
 name {(formal-parameters)}:
 (task
 {* declaration₂ *}
 { action-part₉ }
task)

This describes an action pattern, which may contain equational-statements and thus be non-instantaneous. The task-declaration is not allowed to contain object-declarations.

(5) procedure-declaration ::=
 name {(formal-parameters)}:
 (procedure
 {* declaration₂ *}
 {* partial-event₂₂ *}
procedure)

This describes an instantaneous action pattern. It is to be executed as part of a single event-action and thus its action-part is only allowed to contain partial-events. The procedure can only use internal data-attributes. The procedure-declaration is not allowed to contain object- or task-declarations.

```
(6) function-declaration ::=
    name {(formal-parameters)}:
    (function
     { * declaration2 * }
     { * partial-event22 * }
    function) return type-name7
```

This describes an instantaneous action pattern and thus its action-part is only allowed to contain partial-events. It produces a value of the type specified by type-name. The function can only use internal data-attributes and it has no side-effects, i.e. it may only update variables which are local to itself and it may not have output parameters. Execution of the function must terminate with a partial-event of the form "return expression" and this determines the value to be returned. The function declaration is not allowed to contain object-, task- or procedure-declarations.

```
(7) type-declaration ::=
    name: type type-name7 {(actual-parameters)}
```

This describes a data type. In this paper we use the types known from PASCAL. The standard types such as INTEGER, REAL, ARRAY, ENUMERATION, etc., are assumed to be implicitly declared in the system object.

(8) data-declaration ::=
 name: {var || const} type-name₇ {(actual-parameters)}
 {init initialisation}

This describes a data-attribute which can take the values defined by the type. Data-attributes are either variables or constants. Default is variable. All data-attributes are assumed to have an initial value when generated and the init-clause may be used to specify this initial value.

4.3 SYNTAX OF STATEMENTS

```
(9) action-part ::=
      { * statement10      * }
      { control-section17 }
      { select-section18 }
```

The control-section may interrupt any equational-action in the statement-list. The select-section can only interrupt the last equational-action in the statement-list, and thus it determines what to do when execution of the statement-list finishes by itself.

```
(10) statement ::=
      equational-statement11
      || event-statement13
      || task-statement15
      || block-statement16
```

There are four different kinds of statements.

```
(11) equational-statement ::=
      (* equation12 *)
```

This describes an equational-action.

```
(12) equation ::=
      { * data-name8 * } { | } { predicate }
```

All data-attributes in the list must be internal variables, while the data-attributes in the predicate may be any of those

which are useable according to the scope rules in chapter 2. Predicates have no side-effects. We do not specify a syntax of predicates, but allow any formal or informal description which maps useable data-attributes into truth-values. If either the variable-list or the predicate is omitted the vertical bar can be omitted too.

(13) event-statement ::=

```

    [* {when predicate} {do} {* partial-event22 *} *]
    [] [* match object-proc14 {when predicate} {do procedure-call24} *]

```

The first form describes an internal event-action and all data-attributes in predicate and partial-events must be internal to the object containing the statement. If either the when-clause or the partial-events is omitted the keyword "do" may be omitted too. The second form describes part of an external event-action and all data-attributes in predicate must belong to the object itself or one of the matching objects.

(14) object-proc ::=

```

    object-name3 {(expression)}
    { . procedure-name5 {(actual-parameters)}}

    [] {({} object-proc14 or object-proc14 {})}
    [] {({} object-proc14 and object-proc14 {})}

```

The first form specifies either a unique object to be matched with, or a subset of a family from which any object can be chosen for matching. If present the expression must yield a value from (or a subset of) the finite set in the definition of object-name. In the last two forms parenthesis are used to avoid ambiguity. If present the procedure-name must correspond to the name in the do-clause of the matching object.

(15) task-statement ::=

```

    task-name4 {(actual-parameters)}

```

Tasks can only be called by the object in which they are declared. In this paper we do not define a syntax of parameters,

but allow any suitable notation.

(16) block-statement ::=

```
{name :}
  (block
   { * declaration2 * }
   { action-part9 }
  block)
```

The block is not allowed to contain object-declarations.

(17) control-section ::=

```
control
  {+ guarded-statement-list19 +}
```

(18) select-section ::=

```
select
  {+ guarded-statement-list19 +}
```

(19) guarded-statement-list ::=

```
→ event-statement13 { * statement10 * } { continuation20 }
```

If no continuation is specified, continue is default for control-sections, while leave is default for select-sections.

(20) continuation ::=

```
restart { continuation-name21 }
[] leave   { continuation-name21 }
[] continue { continuation-name21 }
```

(21) continuation-name ::=
 object-name₃
 [] task-name₄
 [] block-name₁₆

The continuation must be contained in the object, task or block, specified by the continuation-name. If no continuation-name is specified the continuation refers to the nearest enclosing block, task or object. It is only possible to continue a block, task or object from its control-section.

4.4 SYNTAX OF PARTIAL-EVENTS

(22) partial-event ::=
 assignment₂₃
 [] procedure-call₂₄
 [] alternative-construct₂₅
 [] repetitive-construct₂₆
 [] event-block₂₇
 [] return-construct₂₈

(23) assignment ::=
 {+ data-name₈ +} := {+ expression +}

All data-attributes in the assignment must be internal. Those on the lefthand side must be variables and their number and types must match the number and types of the expression-list. The individual assignments are executed from left to right and expressions do not have side-effects. We shall not specify a syntax for expressions, but allow any formal or informal description, which map internal data-attributes into a value of the correct type.

(24) procedure-call ::=
 procedure-name₅ {(actual-parameters)}

Procedures can only be called (executed) by the object in which they are declared.

(25) alternative-construct ::=
 if predicate then {+ partial-event₂₂ +}
 { else {+ partial-event₂₂ +} } end

(26) repetitive-construct ::=
 while predicate repeat {+ partial-event₂₂ +} end

(27) event-block ::=
 (event-block
 {* declaration₂ *}
 {* partial-event₂₂ *}
 event-block)

The event-block is not allowed to contain object- or task-declarations.

(28) return-construct ::=
 return expression

This can only be used in the partial-events of an evaluation or a function-declaration. It describes the value to be returned. The expression must have the type specified in the return-clause of the evaluation-or function-declaration.

(29) function-call ::=
 function-name₆ {(actual-parameters) }

Functions have no side-effects. They can only be called in expressions and predicates.

(30) evaluation ::=
 (evaluation
 {* declaration₂ *}
 {* partial-event₂₂ *}
 evaluation) return type-name₇

Evaluations have no side-effects. They can only occur in expressions and predicates. The execution of an evaluation must terminate with a partial-event of the form "return expression". An evaluation is not allowed to contain object-, task- or procedure-declarations.

Chapter 5

SEMANTIC MODEL: EQUATION NETS

5.1	A new semantic approach	101
5.2	Denotational semantics	104
5.3	High-level Petri nets	105
5.4	Combination of subnets	107
5.5	Combination of external transitions	115

5.1 A NEW SEMANTIC APPROACH

In chapter 6 we define a formal semantics for the Epsilon language. Analogously to the use of formal syntax, formal semantics enhances our possibilities to give a complete and unambiguous language definition. This supports consistent use of the language, i.e. language dissemination. Moreover the use of formal syntax and formal semantics promotes language design, in the sense that the required formality discloses inconsistencies and indicates where related concepts can be unified.

Another major reason for the use of formal semantics is the ability to perform rigorous analysis of the created descriptions. This is only possible if the language has a foundation in terms of a formal model or theory.

We shall define the semantics of the Epsilon language by means of a syntax-directed translation from Epsilon descriptions into Equation nets, which are high-level Petri nets with equations attached to the places. Our semantic approach draws on three different sources:

- denotational semantics, [Gordon, 79], which are used to describe declarations, expressions, predicates and partial-events.
- high-level Petri nets, [Jensen, 82], which are used to describe control-flow at the statement-level, including synchronization between objects.
- equation-systems, which are used to describe the effect of equational-actions only. (In the definition of algorithmic programming languages like Simula, Pascal and ADA equation-systems are not necessary).

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

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

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

- syntactic domains (such as declarations and partial-events).
- syntactic clauses (the syntax in chapter 4).
- semantic domains (such as stores, environments and Equation nets).
- semantic functions (defining a syntax-directed mapping from syntactic domains into semantic domains).

The main difference between our approach and normal denotational semantics is the inclusion of Equation nets as a semantic domain. This enables us to benefit from the use of nets in describing flow of control, and we avoid the use of continuations. Nets can be bound into environments and the semantic functions of programs and commands map into nets. The inscriptions of the nets (on places, transitions and arcs) are obtained by means of other semantic functions. The marking of a net contains a token for each sequential process, i.e. for each object. The position of the token acts as a program counter indicating the current action, while the colour attached to the token represents the current store and environment of the object.

Our approach is intended mainly to contribute to the specification of concurrent languages, although it can be used for sequential languages too. The approach has also been presented in [Hansen & Madsen, 82] where it was used to define the semantics of two small languages representing the key-issues of Pascal and Concurrent Pascal, respectively.

It should be remarked, that this is still one of the very first attempts to use the new approach. It must be evaluated remembering the long time, and the many efforts, used to get denotational semantics to its current state of art. If our approach turns out to be valuable, much work remains to be done. On the practical side notation and auxiliary functions have to be improved, while on the theoretical side the well-definedness of the recursive functions into a net domain have to be established. Some analysis methods are proposed in [Kyng, 82].

Our semantic model has been developed gradually. In [Jensen, Kyng & Madsen, 79a] we define the control-flow of Delta (with minor restrictions) by means of a semantic model, called Extended Petri nets. This model is built on condition/event-nets, but transitions are allowed to test the marking/unmarking of places without changing it.

In [Jensen, Kyng & Madsen, 79b] we define a preliminary version of Epsilon by means of a semantic model, called Concurrent systems. Following the ideas of [Mazurkiewicz, 77] and [Keller, 76] the nets are augmented by a separate, global data part and each transition has attached an expression, which can test and modify the data variables. To model equational-actions we also attach an expression to each place. This expression describes an equation-system, i.e. a list of changeable variables and a predicate to be kept unceasingly satisfied while the place is marked.

In the present paper we define Epsilon by means of Equation nets, which are high-level Petri nets with equations attached to the places. We have replaced the separate, global data part of Concurrent systems by colours (attached to the individual tokens) each representing the attributes (environment and store) of an object. Furthermore we use notation from denotational semantics to make our semantics more precise and more complete.

5.2 DENOTATIONAL SEMANTICS

In denotational semantics the meaning of a language is defined by means of functions connecting syntactic domains (such as declarations, expressions and statements) with semantic domains (such as stores, environments and locations). Denotational semantics has a solid mathematical basis in the theory of Scott-domains, which involve partial ordered sets and fixpoint-equations. We shall not describe denotational semantics, but refer the reader to [Tennent, 76], [Stoy, 77] and [Gordon, 79].

Each Epsilon object has its own environment and its own store. Jumps and control-structures (at the statement level) are handled by loops and branches in the constructed Equation nets, and this allows us to avoid the use of continuations, which is one of the

more difficult concepts of denotational semantics. In this paper we will, as far as possible, use the notation and definitions from [Gordon, 79] which we shall assume the reader to be familiar with. In particular we shall use the $*$ -operator to compose semantic functions. The composition $f * g$ is equivalent to $g \circ f$, except that

a) Errors are propagated: Suppose

$$f: D_1 \rightarrow [D_2 + \{\text{error}\}] \text{ and } g: D_2 \rightarrow [D_3 + \{\text{error}\}].$$

Then the composition

$$f * g: D_1 \rightarrow [D_3 + \{\text{error}\}]$$

is defined by

$$f * g = \lambda d_1. (f d_1 = \text{error}) \rightarrow \text{error}, g(f d_1)$$

b) g may be curried: Suppose

$$f: D_1 \rightarrow [[D_2 \times D_3] + \{\text{error}\}] \text{ and } g: D_2 \rightarrow D_3 \rightarrow [D_4 + \{\text{error}\}]$$

Then the composition

$$f * g: D_1 \rightarrow [D_4 + \{\text{error}\}]$$

is defined by

$$f * g = \lambda d_1. (f d_1 = \text{error}) \rightarrow \text{error}, (f d_1 = (d_2, d_3)) \rightarrow g d_2 d_3$$

5.3 HIGH-LEVEL PETRI NETS

The practical use of Petri nets, to describe concurrent systems, has shown a demand for more powerful net types, to describe complex systems in a manageable way. In place/transition-nets it is often necessary to have several identical subnets, because a

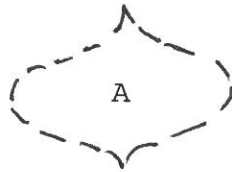
folding into a single subnet would destroy the possibility to distinguish between different processes. The development of high-level Petri nets is in this respect a significant improvement. Now information can be attached to each token as a token-colour and each transition can fire in several ways represented by different firing-colours. The relation between a firing-colour and the involved token-colours is defined by expressions attached to the arcs. Restrictions on the possible firing-colours are defined by predicates attached to the transitions. It is now possible to distinguish between different processes, even though their subnets have been folded into a single subnet. It shall be stressed that the "colour" attached to a token or a firing can be a complex information unit, such as the entire state of a process or the contents of a buffer area. New colours can be created by transition firings and there may be an infinite number of them. We shall not define high-level Petri nets, but refer the reader to [Genrich & Lautenbach, 81] and [Jensen, 81a, 81b, 82]. In this paper we shall, as far as possible use the notation from [Genrich & Lautenbach, 81] and [Jensen, 82], and we shall assume the reader to be familiar with at least one of these papers.

When we translate an Epsilon description into an Equation net, (which is a high-level Petri net with equations attached to places), we first translate each object-declaration into a finite state machine, i.e. a net where all transitions have exactly one input arc and one output arc (with weight one). Each object, specified by the declaration, is represented by a token. The position of the token acts as a program counter indicating the current action, while the colour of the token indicates the contents of the current environment and the current store. Initially the token is situated at the first equational-action to be executed and its colour represents the initial environment and store. It should be noted that a family of objects is represented by a single finite state machine containing a token for each of the objects.

5.4 COMBINATION OF SUBNETS

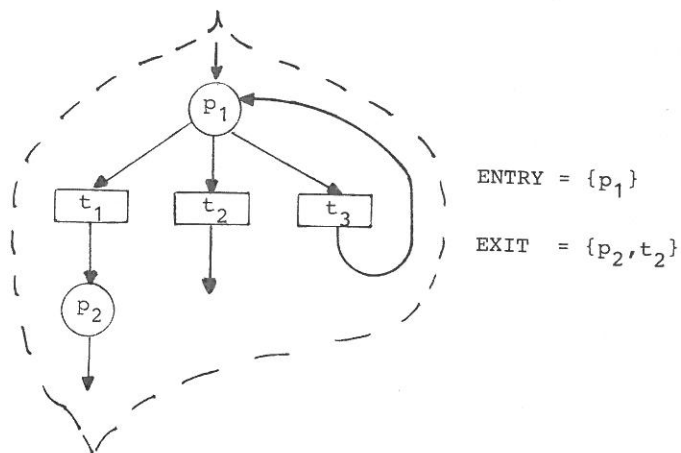
The translation from object-declarations into finite state machines is syntax-directed in the sense that we construct the subnet of an object, task, action-part, or statement by combining the subnets of its constituent parts. Thus we need to define a notation for subnets and for subnet composition.

We shall use the notation



to indicate a subnet A with a non-empty set of entry-nodes (having a dangling input arc) and a possible empty set of exit-nodes (having a dangling output arc).

Example 1



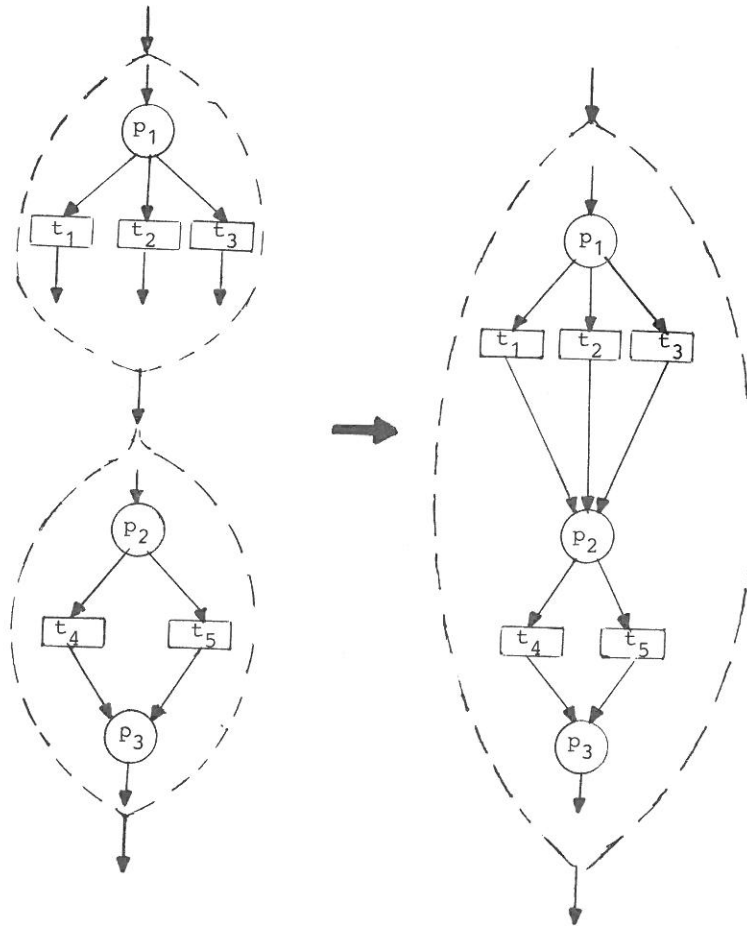
When a subnet consists of a single place or a single transition (being both entry and exit) we shall normally omit the surrounding dashed line. Next we describe four different operations to connect and modify subnets. None of the operations change the equations, predicates and expressions attached to places, transitions and arcs, and thus we can describe the operations in terms of the net structure only (assuming equations, predicates and expressions simply to be copied).

When we draw a directed arc from one subnet S_1 to another subnet S_2 this indicates normal sequencing, in the sense that the exit-nodes of S_1 are connected to the entry-nodes of S_2 . In doing this we may need to insert extra nodes (representing neutral actions) and we may need to split transitions. The operation is formalised in the following three steps:

- 1) The arc between the two subnets is replaced by a set of arcs connecting each exit-node of the source S_1 with each entry-node of the target S_2 .
- 2) If an arc connects two places (transitions) a transition (place) is inserted between them.
- 3) If a transition as a result of step 1 and 2, has x input-places and y output-places it is split into $x \times y$ identical transitions, each of them connecting one input place with one output place.

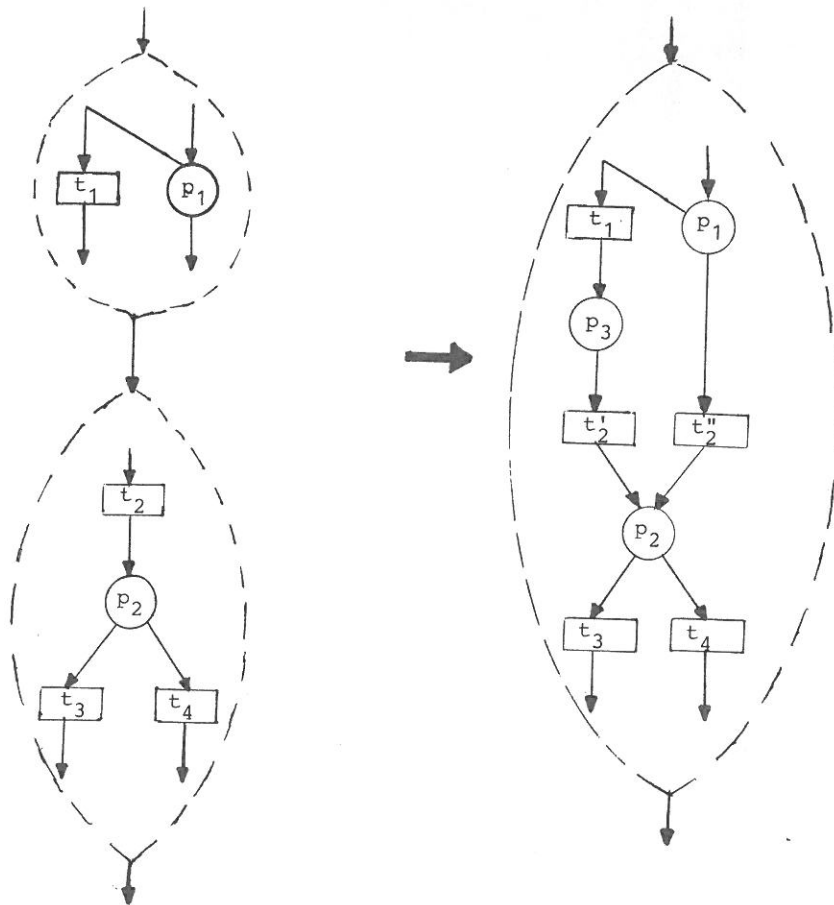
Example 2

In the following, very simple example, step. no. 2 and 3 do not change the net:



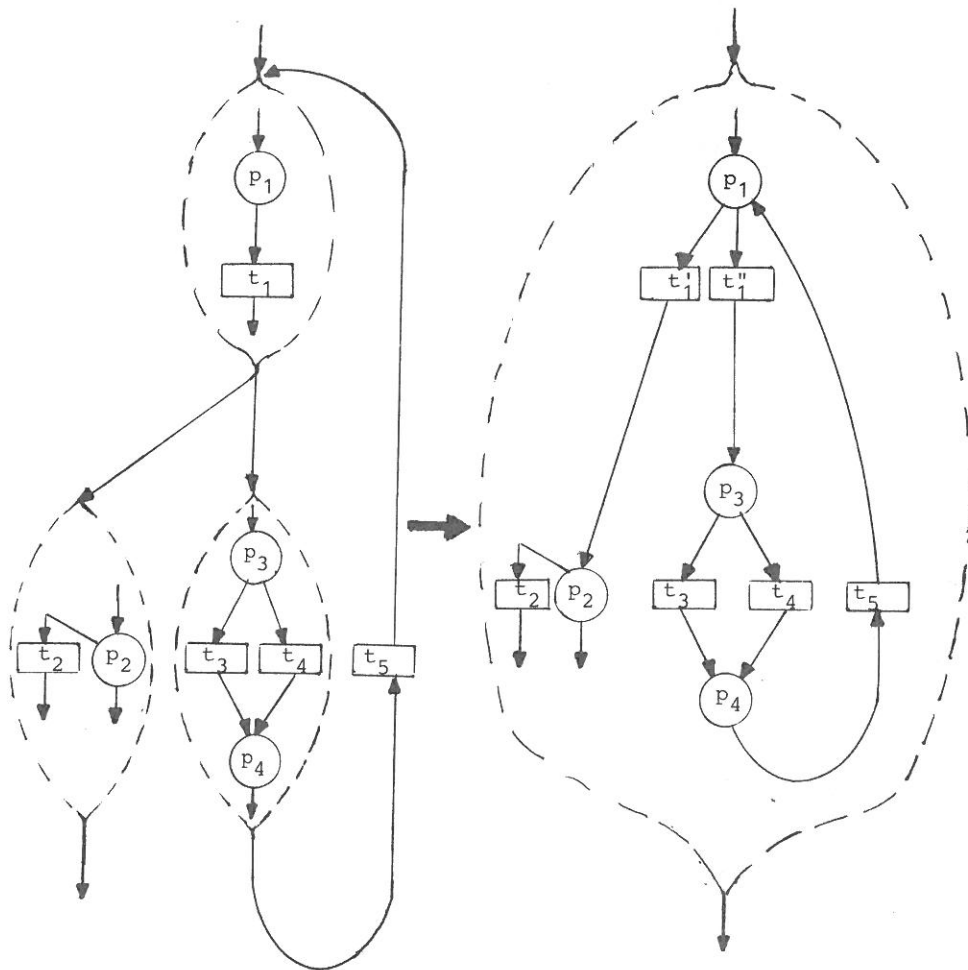
Example 3

In the following, slightly more complicated example, a new place p_3 is added, while t_2 is split into t_2' and t_2'' :

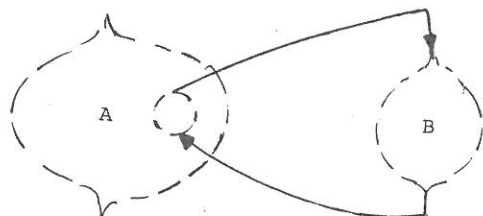


Example 4

When we construct a subnet (of an object, task, action-part or statement) from several constituent subnets (connected by directed arcs) we finish each of the three steps for all arcs, before we proceed to the next step.

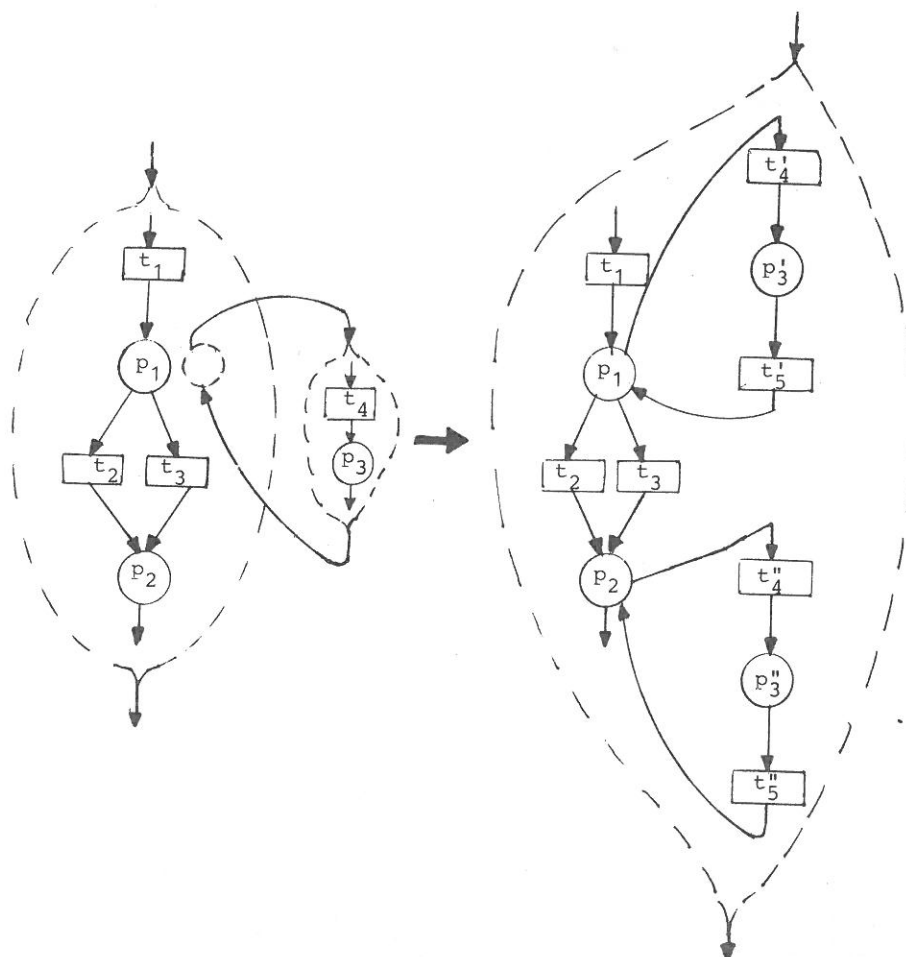


The notation

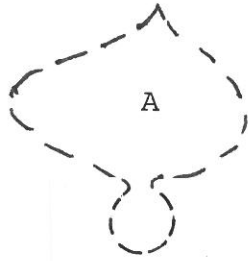


denotes an operation, which connects a copy of the subnet B to each place in the subnet A.

Example 5



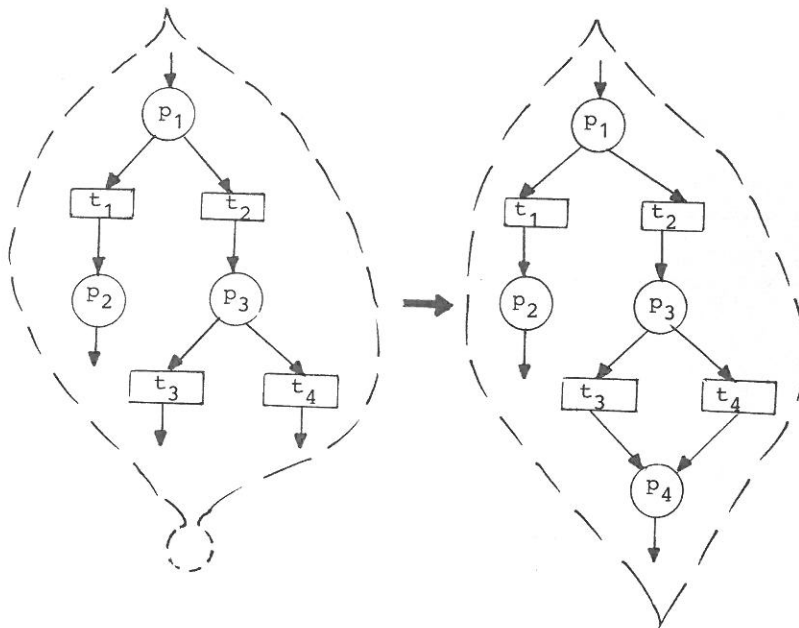
The notation



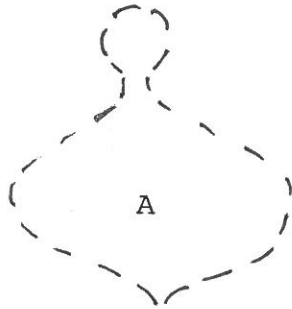
denotes an operation, which modifies a subnet A to a new subnet A_0 in such a way that all exit-nodes are places.

If all exit-nodes of A already are places nothing is done: $A_0 = A$. If one or more exit-nodes are transitions, a new place with a dangling output arc is added. The new place becomes an exit-node and all former exit-nodes being transitions are connected to it (and are no longer exit-nodes).

Example 6



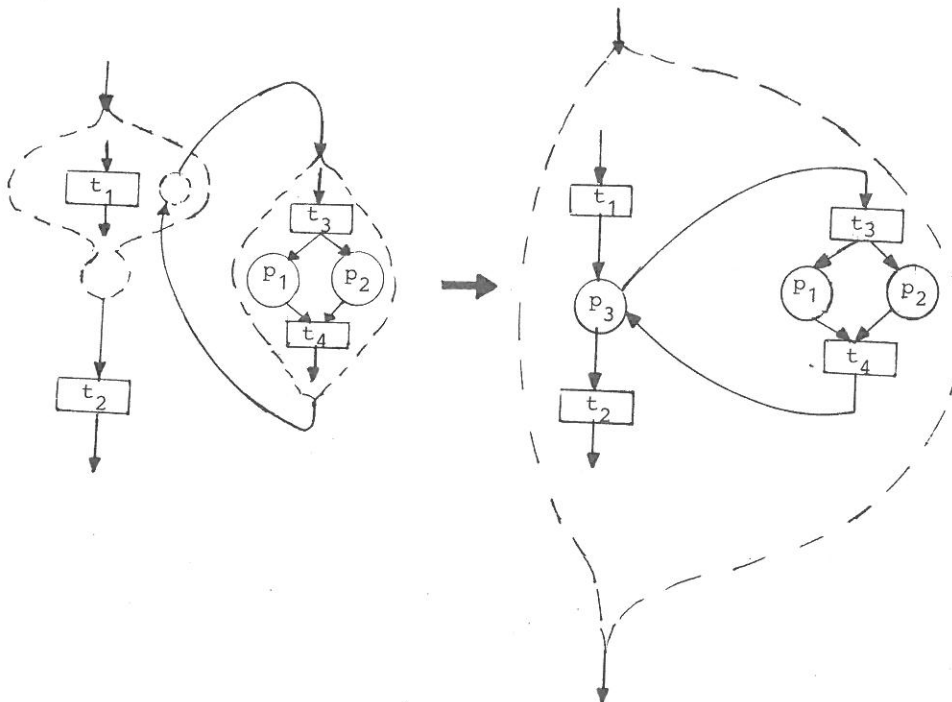
Analogously the notation



denotes an operation, which modifies a subnet A to a new subnet A^0 in such a way that all entry-nodes of A^0 are places.

Example 7

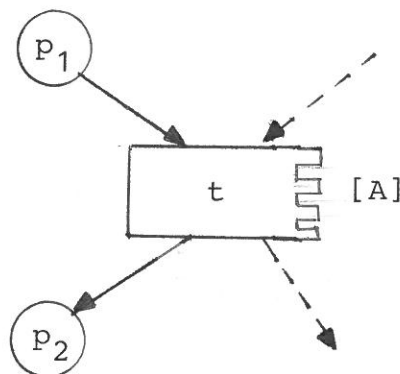
The order in which individual applications of the four subnet-operations are resolved is without importance, except that applications of the last two kinds of subnet-operations always are resolved prior to applications of the first two kinds:



5.5 COMBINATION OF EXTERNAL TRANSITIONS

The transitions of each finite state machine are divided into two disjoint groups: some of them are internal, in the sense that they represent internal event-statements. They are already in their final form. Others represent external event-statements. They are external, in the sense that they have to be combined with transitions representing matching event-statements. This means that the resulting Equation net, representing a full Epsilon description, consists of a set of superposed finite state machines, i.e. a net where each transition has exactly as many input arcs as it has output arcs (each with weight one). In such a net the number of tokens is constant, and in our case it equals the number of objects in the system.

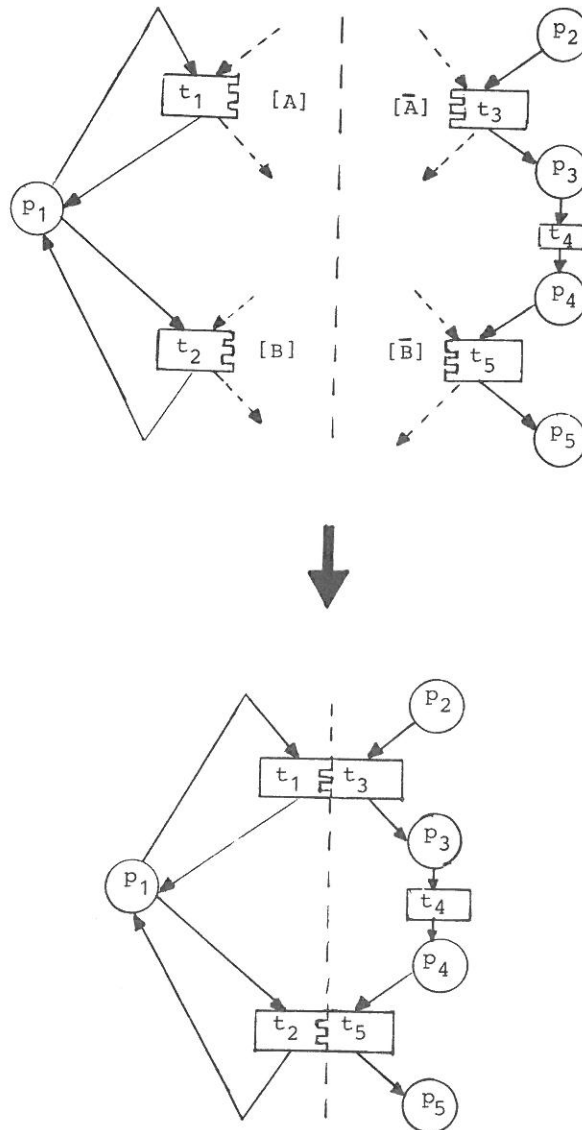
In our semantics, external transitions are drawn in the following way:



The label A describes the transitions with which t can be combined. We use barred and unbarred capitals, such as A and \bar{A} , to denote matching labels. The two dangling arcs are dashed to indicate that they are not part of the finite state machine to which t belongs; and they are not taken into account, when combining subnets to build the finite state machine corresponding to an object-declaration, as described in the preceding section.

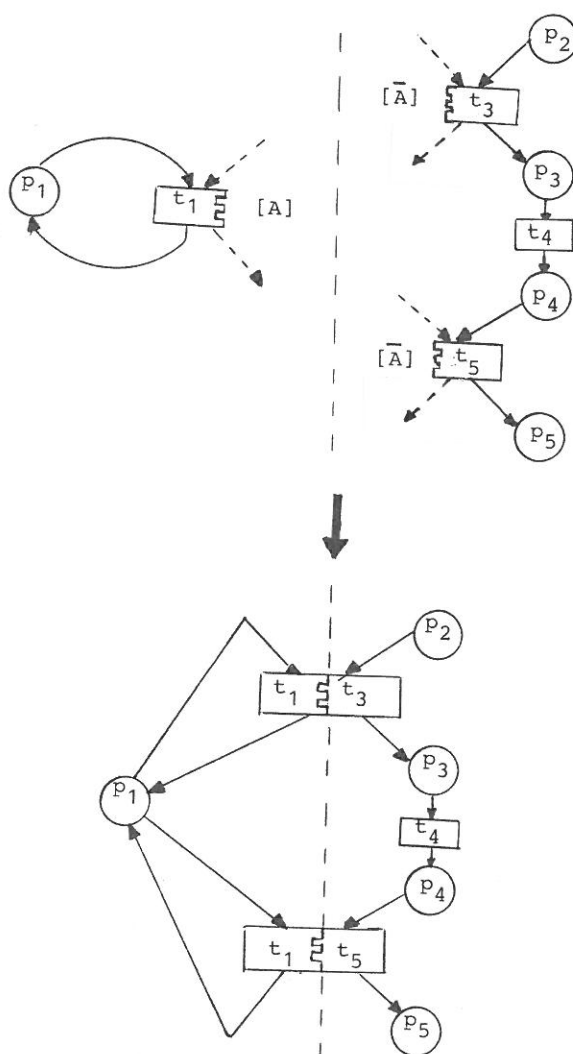
Example 8

In the following example we consider two different objects, each having a finite state machine with two external transitions.



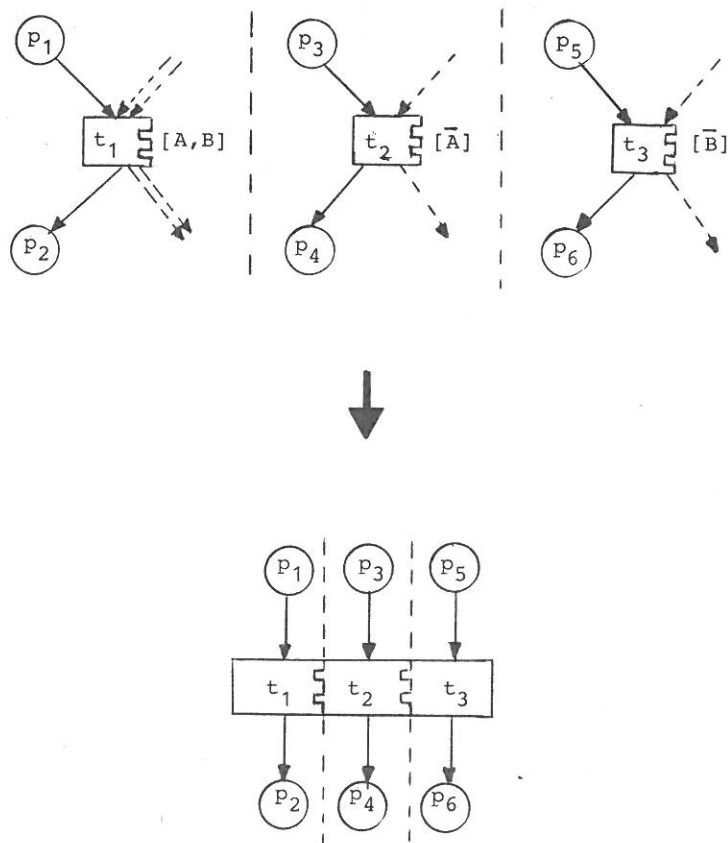
Example 9

If an external transition matches several other transitions it is duplicated as the transition t_1 below:



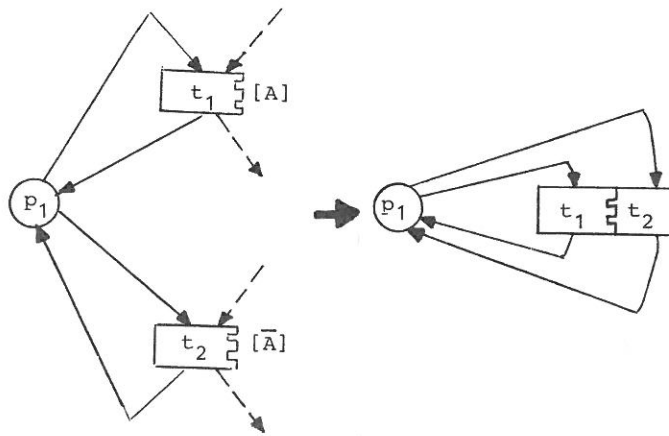
Example 10

An external transition may simultaneously be combined with more than one transition:



Example 11

An external transition may be combined with a transition in its own finite state machine. This can only happen when the finite state machine describes a family.



There is, however, one exception from the finite state machine property described above. Each block- and task-statement has a special place, used to save the old environment during execution. It is only when ignoring these environment places that Epsilon descriptions translate into superposed finite state machines. It is easy to recognize the environment places since they have Env as the set of token-colours. All other places in the Equation nets have $Env \times Store$ as the set of token-colours.

We have now described some concepts from denotational semantics and high-level Petri nets. The third constituent of our semantic model, equation-systems, will be treated in section 6.8.

Chapter 6

SEMANTICS

6.1	Domains and semantic functions	123
6.2	Semantic clauses for expressions and predicates	124
6.3	Semantic clauses for partial-events	126
6.4	Semantic clauses for declarations	128
6.5	Semantic clauses for statements	132
6.6	Semantic clauses for action-parts	137
6.7	Semantic clauses for system-descriptions	138
6.8	Dynamic behaviour of Equation nets	139
6.9	Evaluation of our semantic approach	145

6.1 DOMAINS AND SEMANTIC FUNCTIONS

In this section we define the syntactic domains, the semantic domains and the functionality of the semantic functions.

Primitive syntactic domains

Ide	The domain of identifiers	I
-----	---------------------------	---

Compound syntactic domains

Exp	The domain of expressions	E
Dec	The domain of declarations	D
Pev	The domain of partial-events	P
Sta	The domain of statements	S
Acp	The domain of action-parts	A
Sys	The domain of system-descriptions	Y

Primitive semantic domains

Loc	The domain of locations	l
Bv	The domain of basis values	v
Bool	The domain of booleans	b
Net	The domain of Equation nets	n

Compound semantic domains

Rv	= Bv + Bool	R-values	e
Dv	= Obj+Task+Proc+Fun+Loc+Rv+Env	denotable values	d
Env	= Ide \rightarrow [Dv+{unbound}]	environments	r
Store	= Loc \rightarrow [Rv+{unused}]	stores	s
Obj	= Net	objects	o
Task	= Exp* \rightarrow Ide* \rightarrow Net	tasks	t
Proc	= Rv* \rightarrow Store \rightarrow [Rv* \times Store]	procedures	p
Fun	= Rv* \rightarrow Store \rightarrow [Rv \times Store]	functions	f

SEMANTIC FUNCTIONS

E :	$\text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Dv} \times \text{Store}]$
R :	$\text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Rv} \times \text{Store}]$
T :	$\text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Bool}$
\mathcal{D} :	$\text{Dec}^* \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Env} \times \text{Store}]$
P :	$\text{Pev}^* \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Store}$
S :	$\text{Sta}^+ \rightarrow \text{Env} \rightarrow \text{Net}$
A :	$\text{Acp} \rightarrow \text{Env} \rightarrow \text{Net}$
\mathcal{V} :	$\text{Sys} \rightarrow \text{Net}$

To be strictly correct, all semantic functions and all auxiliary functions must have {error} or {error-net} added to their output domain (c.f. the definition of "*" in section 5.2). If a token-colour contains an "error-store" or "error-environment", the token can not be used in transition firings. If a net contains "error-net" as a subnet, its dynamic behaviour is undefined (c.f. section 6.8).

6.2 SEMANTIC CLAUSES FOR EXPRESSIONS AND PREDICATES

Syntax and semantics for expressions and predicates (boolean expressions) are not described in this paper. This allows the user to apply any suitable language-constructs for these syntactic categories, e.g. the constructs of a programming language to be used for implementation of the described system. In Epsilon, expressions are always evaluated as part of a single event-action, and thus their semantics are defined only by means of denotational semantics, without the use of nets. From the literature of denotational semantics it is wellknown how this can be done.

The semantics of expressions is defined by means of three semantic functions (of which we shall assume the first to be provided by the user):

$$\begin{array}{l}
 E: \quad \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Dv} \times \text{Store}] \\
 R: \quad \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Rv} \times \text{Store}] \\
 T: \quad \text{Exp} \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow \text{Bool}
 \end{array}$$

Even though expressions have no side-effects, it turns out to be convenient, for the semantic clauses, to include the resulting unaltered store in the result of E and R . E evaluates the expression without dereferencing (c.f. [Gordon, 79] pp. 74 - 75). This means that the resulting value may be any element in Dv . The two other functions are defined by

$$\begin{array}{l}
 R[E] \text{ r s} = E[E] \text{ r s} * \text{deref} * \text{Rv?} \\
 T[E] \text{ r s} = R[E] \text{ r s} * \text{Bool?} * \lambda b \text{ s} . b
 \end{array}$$

where the auxiliary functions

$$\begin{array}{l}
 \text{cont} : \text{Loc} \rightarrow \text{Store} \rightarrow [\text{Rv} \times \text{Store}] \\
 \text{deref} : \text{Dv} \rightarrow \text{Store} \rightarrow [\text{Dv} \times \text{Store}] \\
 \text{Rv?} : \text{Dv} \rightarrow \text{Store} \rightarrow [\text{Rv} \times \text{Store}] \\
 \text{Bool?} : \text{Dv} \rightarrow \text{Store} \rightarrow [\text{Bool} \times \text{Store}]
 \end{array}$$

are defined by

$$\begin{array}{l}
 \text{cont l s} = (\text{s l} \neq \text{unused}) \rightarrow (\text{s l}, \text{s}), \text{ error} \\
 \text{deref d s} = \text{isloc d} \rightarrow \text{cont d s}, (\text{d}, \text{s}) \\
 \text{Rv? d s} = \text{isRv d} \rightarrow (\text{d}, \text{s}), \text{ error} \\
 \text{Bool? d s} = \text{isBool d} \rightarrow (\text{d}, \text{s}), \text{ error}
 \end{array}$$

where we for an arbitrary domain A , for all $b \in \text{Bool}$ and all $a_1, a_2 \in A$, use the following notation (c.f. [Gordon, 79] pp. 41 - 42).

$$b \rightarrow (a_1, a_2) \stackrel{\text{DEF}}{=} \begin{cases} a_1 & \text{if } b = \text{true} \\ a_2 & \text{if } b = \text{false} \\ \text{error} & \text{if } b = \text{error} \end{cases}$$

In the remaining semantic clauses we shall, without further definition, use a number of domain-testing functions analogous to $Rv?$ and $Bool?$

6.3 SEMANTIC CLAUSES FOR PARTIAL-EVENTS

$P: Pev^* \rightarrow Env \rightarrow Store \rightarrow Store$

Partial-events define a sequential and deterministic store-transformation to be executed as part of a single event-action. Their semantics is defined by means of denotational semantics, without the use of nets.

Sequence of partial-events

$$P[PP^*]rs = P[P]rs * P[P^*]r$$

$$P[]rs = s$$

(23) Assignment

$$P[I_D := E]rs = E[I_D]rs * Loc? * \lambda l.R[E]r * update\ l$$

$$P[I_D I_D^* := EE^*]rs = P[I_D := E]rs * P[I_D^* := E^*]r$$

where the auxiliary function

$$update : Loc \rightarrow Rv \rightarrow Store \rightarrow Store$$

is defined by

$$update\ l\ e\ s = s[e/l]$$

The notation $s[e/l]$ represents the store, which is identical to s , except that the value e has been bound to location l . We shall use the same kind of notation for environments (c.f. [Gordon, 79] p. 43 and pp. 64 - 65).

(24) Procedure-call

Procedures are described by the semantic domain

$$\text{Proc} = \text{Rv}^* \rightarrow \text{Store} \rightarrow [\text{Rv}^* \times \text{Store}]$$

where the first Rv^* represents the input parameter values, while the second Rv^* represents the output parameter values. In this paper we give the semantics for procedures which have exactly one input parameter followed by exactly one output parameter. The generalization to zero or more than one input/output parameters is cumbersome in notation, but straightforward in idea (c.f. [Gordon, 79], pp. 98 - 99)

$$\begin{aligned} P[\underline{I}_p(E, I) \text{ r s}] \\ &= R[E] \text{ r s} * \lambda e. E[I] \text{ r} * \text{Loc?} * \lambda l. \\ &\quad E[\underline{I}_p] \text{ r} * \text{Proc?} * \lambda p. p e * \text{update l} \end{aligned}$$

(25) Alternative-construct

$$\begin{aligned} P[\underline{\text{if}} \ E \ \underline{\text{then}} \ P_1^+ \ \underline{\text{else}} \ P_2^+ \ \underline{\text{end}}] \text{ r s} \\ &= T[E] \text{ r s} \rightarrow P[P_1^+] \text{ r s}, P[P_2^+] \text{ r s} \end{aligned}$$

$$\begin{aligned} P[\underline{\text{if}} \ E \ \underline{\text{then}} \ P^+ \ \underline{\text{end}}] \text{ r s} \\ &= T[E] \text{ r s} \rightarrow P[P^+] \text{ r s}, \text{ s} \end{aligned}$$

(26) Repetitive-construct

$$\begin{aligned} P[\underline{\text{while}} \ E \ \underline{\text{repeat}} \ P^+ \ \underline{\text{end}}] \text{ r s} \\ &= T[E] \text{ r s} \rightarrow P[P^+] \text{ r s} * P[\underline{\text{while}} \ E \ \underline{\text{repeat}} \ P^+ \ \underline{\text{end}}] \text{ r}, \text{ s} \end{aligned}$$

This is a recursive definition.

(27) Event-block

$$\begin{aligned}
 P[(\text{evbl } D^* P^* \text{ evbl})] r s \\
 = \mathcal{D}[D^*] r s * P[P^*]
 \end{aligned}$$

We will not describe the semantics of functions and evaluations, since they are executed as part of expressions. Moreover they resemble procedures and event-blocks.

6.4 SEMANTIC CLAUSES FOR DECLARATIONS

$$\mathcal{D}: \text{Dec}^* \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Env} \times \text{Store}]$$

The output environment contains those bindings which are due to Dec^* together with those already contained in the input environment. We define \mathcal{D} by means of two auxiliary functions: \mathcal{D}' deals with data-declarations (which may update store) while \mathcal{D}'' deals with all other kinds of declarations (which may be mutually recursive).

$$\mathcal{D}': \text{Dec}^* \rightarrow \text{Env} \rightarrow \text{Store} \rightarrow [\text{Env} \times \text{Store}]$$

$$\mathcal{D}'': \text{Dec}^* \rightarrow \text{Env} \rightarrow \text{Env}$$

$$\mathcal{D}'[DD^*] r s = \mathcal{D}'[D] r s * \mathcal{D}'[D^*]$$

$$\mathcal{D}''[DD^*] r = \mathcal{D}''[D] r * \mathcal{D}''[D^*]$$

$$\mathcal{D}'[D] r s = \mathcal{D}'[] r s = (r, s) \text{ when } D \text{ is not a data-declaration}$$

$$\mathcal{D}''[D] r = \mathcal{D}''[] r = r \text{ when } D \text{ is a data-declaration}$$

$$\mathcal{D}[D^*] r s = \mathcal{D}'[D^*] r s * \lambda r' s'. (r'', s')$$

$$\text{whererec } r'' = \mathcal{D}''[D^*] r'[r'']$$

(8) Data-declaration

$$\begin{aligned} \mathcal{D}'[I_D: \underline{\text{var}} I_T \underline{\text{init}} E] r s \\ = R[E] r s * \text{ref} * \lambda l s'. (r[l/I_D], s') \end{aligned}$$

where the auxiliary functions

$$\begin{aligned} \text{new}: \text{Store} &\rightarrow [\text{Loc} \times \text{Store}] \\ \text{ref}: \text{Rv} &\rightarrow \text{Store} \rightarrow [\text{Loc} \times \text{Store}] \end{aligned}$$

are defined by

$$\begin{aligned} \text{news} &= (\exists l: s l = \text{unused}) \rightarrow (l, s), \text{ error} \\ \text{refes} &= \text{news} * \lambda l s. (l, \text{update } l \text{ es}) \end{aligned}$$

$$\begin{aligned} \mathcal{D}'[I_D: \underline{\text{const}} I_T \underline{\text{init}} E] r s \\ = R[E] r s * \lambda es. (r[e/I_D], s) \end{aligned}$$

In this paper we give the semantics for data-declarations which define a simple variable or constant. The definition of compound data-structures, such as arrays, records and files, can be found in [Gordon, 79] pp. 118 - 128. We shall not define the semantics of type-declarations. Type-checking is discussed in [Gordon, 79] pp. 142 - 146.

(5) Procedure-declaration

We give the semantics for procedures which have exactly one input parameter followed by exactly one output parameter. Such a procedure is represented by an element of the semantic domain

$$\text{Proc}_{11} = \text{Rv} \rightarrow \text{Store} \rightarrow [\text{Rv} \times \text{Store}]$$

(c.f. the semantics of procedure-call in section 6.3).

$$\mathcal{D}''[I_P(I_I : \underline{\text{in}}, I_O : \underline{\text{out}}) : (\underline{\text{proc}} D^* P^* \underline{\text{proc}})] r = r[p/I_P]$$

whererec $p = \lambda e s . \text{ref } e s * \lambda l_I . \text{new} * \lambda l_O .$
 $\mathcal{D}[D^*]r[p, l_I, l_O / I_P, I_I, I_O] * P[P^*] * \text{cont } l_O$

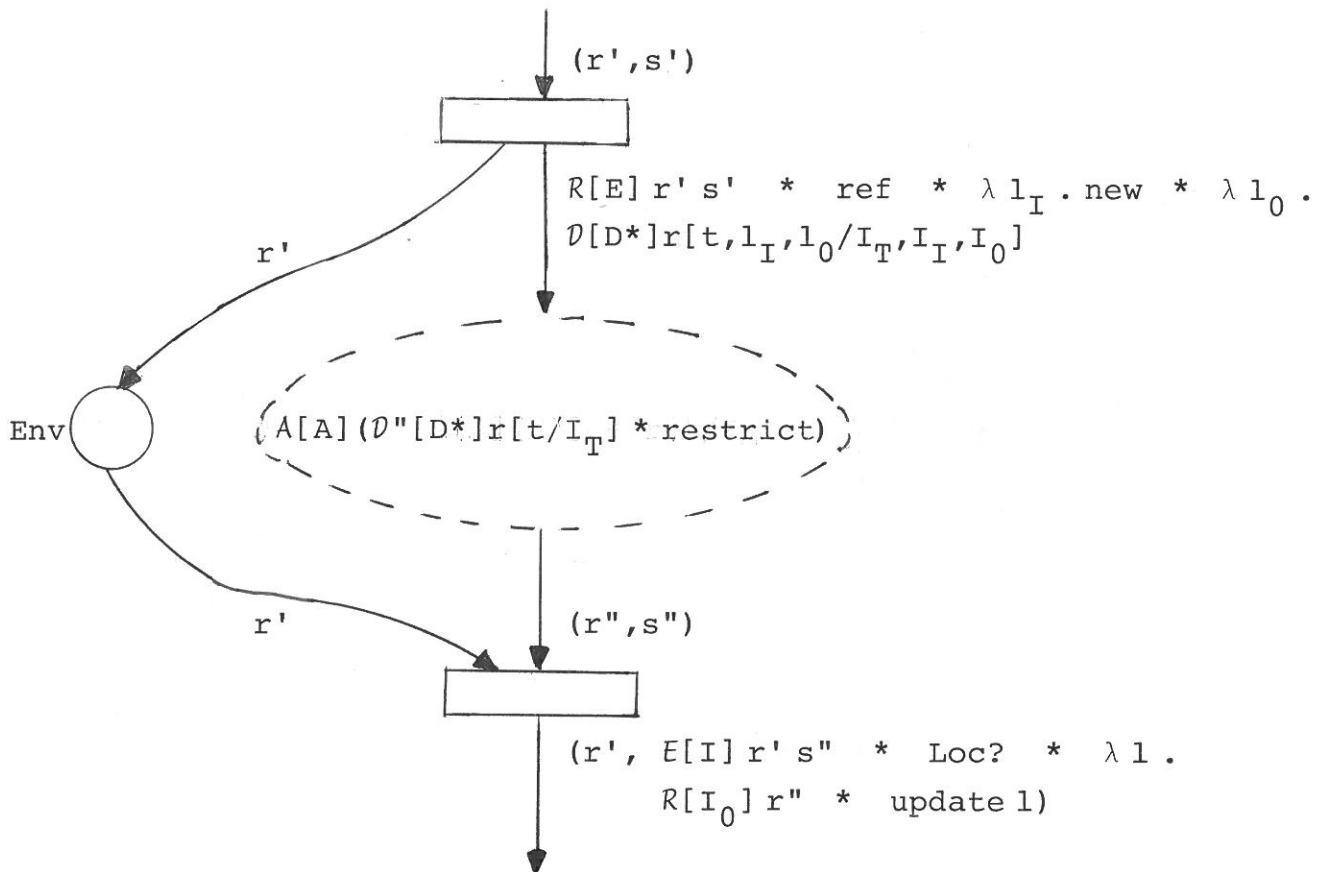
(4) Task-declaration

We give the semantics for tasks which have exactly one input parameter followed by exactly one output parameter. Such a task is represented by an element of the semantic domain

$$\text{Task}_{11} = \text{Exp} \rightarrow \text{Ide} \rightarrow \text{Net}$$

$$\mathcal{D}''[(I_T(I_I : \underline{\text{in}}, I_O : \underline{\text{out}}) : (\underline{\text{task}} D^* A \underline{\text{task}})] r = r[t/I_T]$$

whererec $t = \lambda E I . n$ where $n =$



The auxiliary function

$$\text{restrict}: \text{Env} \rightarrow \text{Env}$$

is defined by

$$\text{restrict } r = \lambda I. \text{ isTask}(rI) \rightarrow rI, \text{ unbound}$$

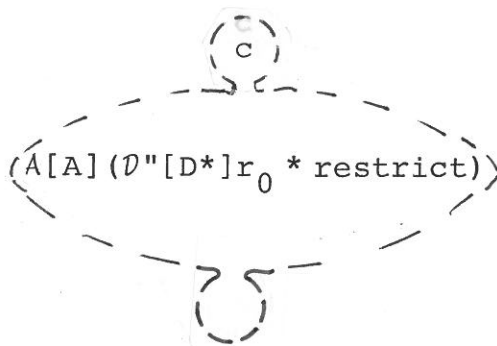
It should be noted that evaluation of the action-part, in order to insert the correct Equation nets for its task-statements, uses that part of the environment, which corresponds to task-declarations. This part of the environment is static, in the sense that it can be determined at "compile-time" and does not involve dynamic aspects, such as recursion-level.

(3) Object-declaration

We first define the semantics of singular objects

$$\mathcal{D}''[I_0 : (\underline{\text{obj}} \ D^* \ A \ \underline{\text{obj}})] r = r[n/I_0]$$

whererec $n =$



Initially the entry place of the net has a single token with the colour $c = (r_I, s_I) = \mathcal{D}[D^*]r_0[n/I_0]s_0$

The default environment r_0 and the default store s_0 may contain elements which correspond to predefined attributes, e.g. the TIME variables described in section 1.3. It is assumed that the set of locations in s_0 is disjoint to the sets of locations used for previous objects in the system.

Next we define the semantics for a family of objects. Let I_T be the name of an enumeration-type or subrange-type with elements $\{k_1, k_2, \dots, k_n\}$. The semantics of a family of objects declared by $I_0(I_D : I_T) : (\text{obj } D^* \text{ A obj})$ is defined by the Equation net above, except that the initial marking has a token for each object, i.e. for each element in type I_T . The token-colour corresponding to object $I_0(k_j)$, where $1 \leq j \leq n$, is defined by

$$c_j = \mathcal{D}[I_D : \underline{\text{const}} \text{ ENUMERATION } (k_1, k_2, \dots, k_n) \underline{\text{init}} k_j] r_I s_I$$

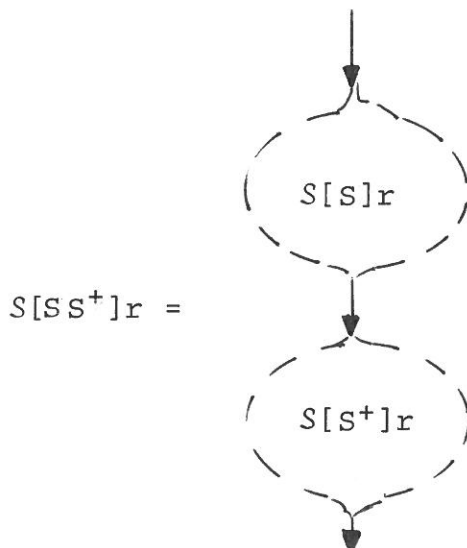
The objects in the family are assumed to have disjoint sets of locations.

6.5 SEMANTIC CLAUSES FOR STATEMENTS

$$S: \text{Sta}^+ \rightarrow \text{Env} \rightarrow \text{Net}$$

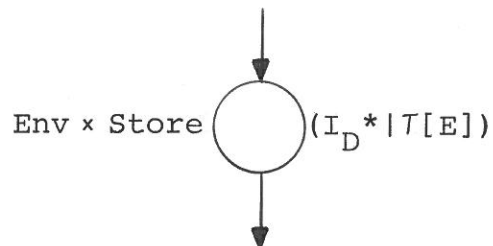
The environment contains only tasks and it is used for task-statements only. (c.f. the semantics of action-part).

Sequence of statements



(11+12) Equational-statement

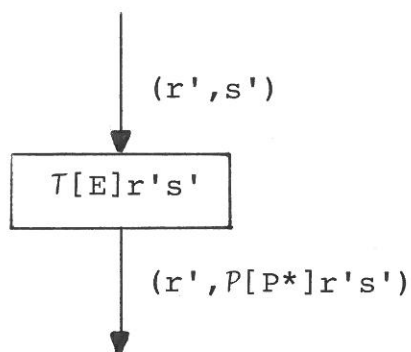
$$S[(\ast I_D \ast \mid E \ast)] r =$$



The dynamic behaviour described by the place-equation is defined in section 6.8. It should be noted that the arcs adjacent to the place have no arc-expressions, and this property guarantees that we never get an arc with two arc-expressions. If $I_D \ast$ or E is missing the corresponding part of the place-equation is omitted.

(13a) Internal event-statement

$$S[[\ast \text{when } E \text{ do } P \ast]] r =$$



A missing predicate corresponds, by default, to the always satisfied predicate TRUE:

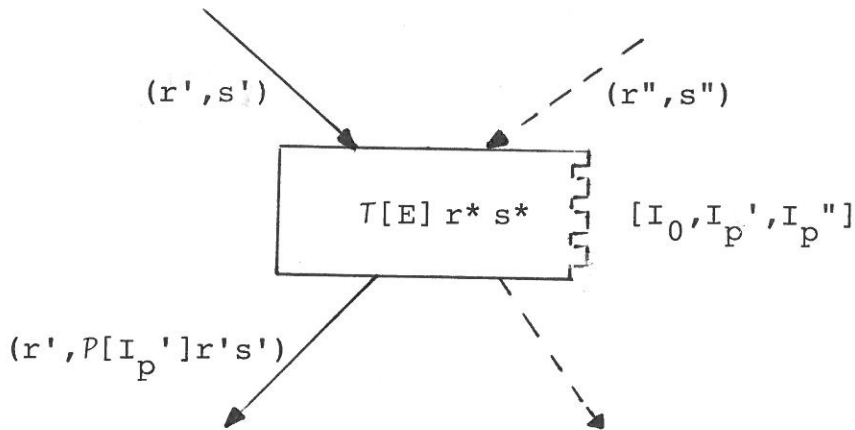
$$S[[\ast P \ast]] r = S[[\ast \text{when } \text{TRUE} \text{ do } P \ast]] r$$

(13b+14) External event-statement

We give the semantics for external event-statements matching a single object. Generalization is cumbersome in notation, but straightforward in idea.

We first define the semantics of external event-statements without procedure parameters

$$S[[* \text{ match } I_0 \cdot I_p'' \text{ when } E \text{ do } I_p' *]] r =$$



where $r^* = r'[r''/I_0]$

and $s^* = s' + s''$

where

$$(s' + s'')l = \begin{cases} s'l & \text{if } l \text{ is a location of } s' \\ s''l & \text{if } l \text{ is a location of } s'' \end{cases}$$

It should be remembered that s' and s'' have disjoint sets of locations (c.f. the semantics of object-declaration). If E is missing the transition inscription is omitted. If I_p' is missing the lower arc-expression becomes (r', s') . If I_p' or I_p'' is missing the corresponding component of the transition-label becomes "NONE".

E is allowed to use attributes from I_0 and the semantic function E (from which T is defined) is assumed to handle such external attributes by the semantic clause

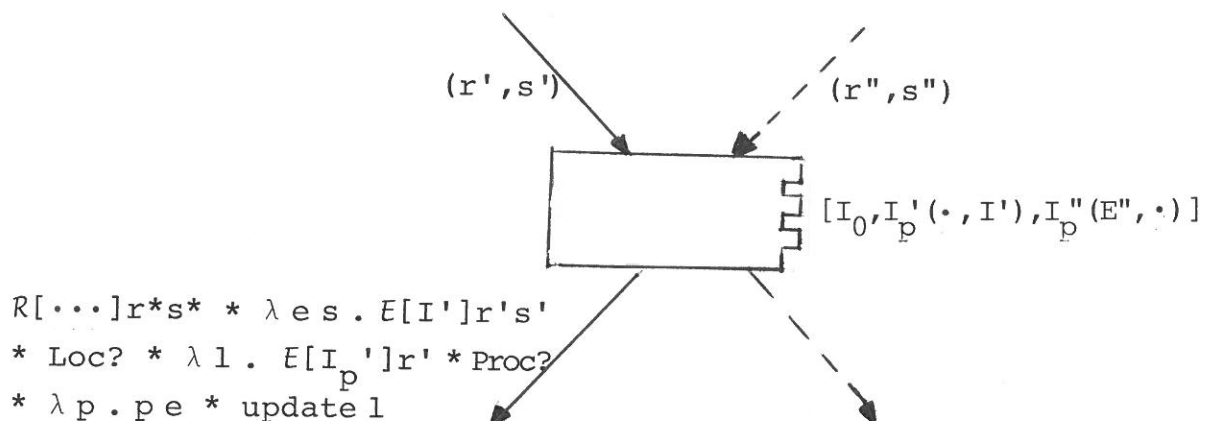
$$E[I_0 \cdot I_D] r s = E[I_0] r s * Env? * E[I_D]$$

The transition-label $[I_0, I_p', I_p'']$ specifies match-information to be used when the transition is combined with a transition in the net representing object I_0 . The rules determining whether two transition-labels make a match or not, are in detail described in section 3.5 and 3.9, and they will not be repeated in the present chapter.

For technical convenience we assume all object-names in the system to be unique (i.e. without repetitions and disjoint from the other attribute-names). This allows us to use the object-name, when we want to refer to the Equation net representing an object (c.f. the first component of the transition-label above). If object-names are not already unique, this can be achieved by replacing each object-name with the ordered list of object-names corresponding to the enclosers (c.f. section 2.1).

Next we define the semantics of external event-statements with procedure parameters. Input parameters for the procedures may be specified by any of the matching objects (to avoid ambiguity each parameter must be specified by exactly one object). This means that the arc-expression $P[I_p'] r' s'$ in the net above must be replaced by the right-hand side of the semantic clause for procedure-calls (c.f. section 6.3) with R being evaluated in (r^*, s^*) while the rest is evaluated in (r', s') .

We illustrate this by giving the semantics in the case where I_p' and I_p'' both have exactly one input parameter (to be specified by the other object), followed by exactly one output parameter.

$$S[[* \text{ match } I_0 . I_p''(E'', \cdot) \text{ do } I_p'(\cdot, I') *]] r =$$


where the argument of R is to be provided from the match-information, when the transition is combined with a transition in the net representing object I_0 .

(15) Task-statement

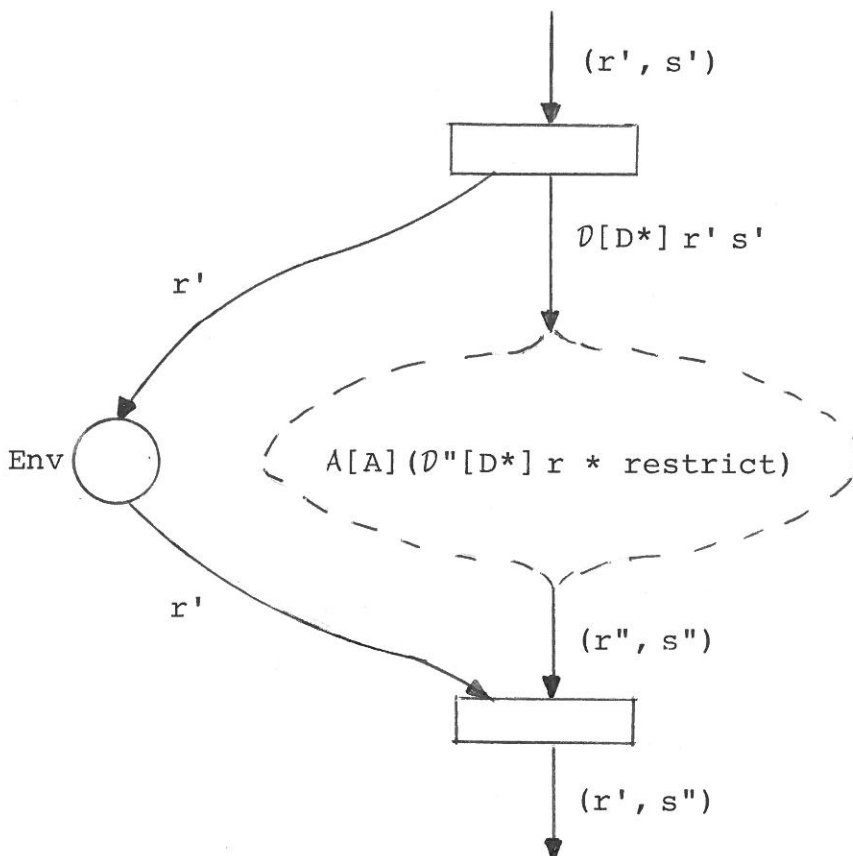
$$S[I_T(E, I)]r = \text{isTask}(r I_T) \rightarrow r I_T E I, \text{ error-net}$$

where error-net =



(16) Block-statement

$$S[(\text{block } D^* \text{ A block})]r =$$



The name of a block has only semantic relevance when used as a continuation-name (c.f. the semantics of action-part).

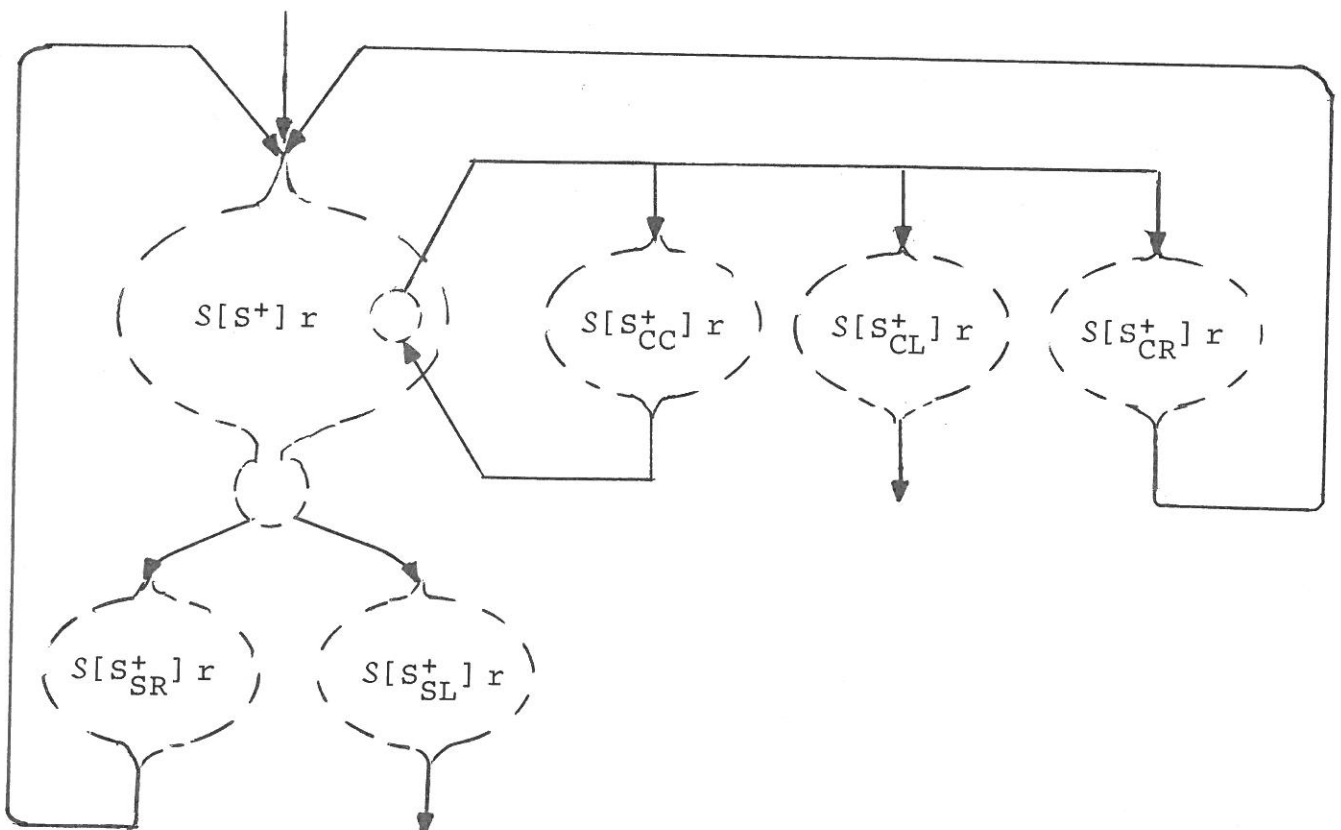
6.6 SEMANTIC CLAUSES FOR ACTION-PARTS

A: Acp \rightarrow Env \rightarrow Net

The environment contains only tasks and it is used for task-statements only (c.f. the semantics of task-declaration, object-declaration and block-statement).

(9 + 17-21) Action-part

We first define the semantics of action-parts, which do not contain continuation-names. Then the guarded-statement-lists can be divided into five groups (according to whether they appear in a control- or select-section, and according to whether they have restart, leave or continue as continuation). We give the semantics of an action-part, which has exactly one guarded-statement-list in each of the five groups. Generalization to other combinations of guarded-statement-lists is straightforward.

$$A[S^+ \text{ control} \rightarrow S_{CR}^+ \text{ restart} \rightarrow S_{CL}^+ \text{ leave} \rightarrow S_{CC}^+ \text{ continue} \\ \text{select} \rightarrow S_{SR}^+ \text{ restart} \rightarrow S_{SL}^+ \text{ leave}] r =$$


It should be noted that the subnets representing the guarded-statement-lists always have a single transition as entry-node. If the statement-list S^+ is missing the upper leftmost subnet consists of a single place only.

Next we define the semantics of action-parts containing continuation-names. When a continuation-name I_C is present, the outgoing arc of the corresponding guarded-statement-list has a label of the form $RESTART(I_C)$, $LEAVE(I_C)$ or $CONTINUE(I_C)$, and it is left dangling, without having a destination yet. Such dangling arcs are directed to the correct node, when the action-part of the continuation-name is constructed. If desired this can be made more formal by providing A with an extra input domain of type Ide , so that A knows the object-, task- or block-name of the action-part being evaluated:

$$A: Acp \rightarrow Ide \rightarrow Env \rightarrow Net$$

6.7 SEMANTIC CLAUSES FOR SYSTEM-DESCRIPTIONS

$$V: Sys \rightarrow Net$$

(1) System-description

$$\begin{aligned} & V[I_S: (\underline{sys} \ D^* \ A \ \underline{sys})] \\ & = \mathcal{D}''[I_S: (\underline{obj} \ D^* \ A \ \underline{obj})] r_0 \ * \ collect-nets \ * \ combine-nets \end{aligned}$$

where the auxiliary functions `collect-nets` and `combine-nets` are defined below. The default environment r_0 is described in the semantics of object-declaration (c.f. section 6.4).

Collect-nets

$$collect-nets: Env \rightarrow [Ide \times Net]^*$$

This function collects a list of named Equation nets, each representing an object or a family of objects.

$$\text{collect-nets } r = X \cup \left(\bigcup_{r' \in R} \text{collect-nets } r' \right)$$

where $X = \{(I, n) \mid r I = n \wedge \text{isObj } n\}$

and R contains all environments, which appear (as the environment-component) in token-colours (of the initial-marking) in a net contained in X .

To be strictly formal the definition above defines a function into sets of named Equation nets and not into sequences; but a set can always be transformed to a sequence by introducing some ordering-relation.

Combine-nets

$$\text{combine-nets: } [\text{Id} \times \text{Net}]^* \rightarrow \text{Net}$$

This function combines a sequence of named Equation nets into a single net by combining (gluing together) transitions, which have matching labels. If a transition matches several other transitions it is duplicated (c.f. section 5.3).

6.8 DYNAMIC BEHAVIOUR OF EQUATION NETS

In the previous sections we have defined a syntax-directed translation from Epsilon descriptions into Equation nets. We will, however, for an Epsilon description not only be interested in the corresponding net, but also in the dynamic behaviour of that net. In this section we define the dynamic behaviour for the kind of Equation nets used in our formal semantics. There are two different kinds of Epsilon descriptions. First we define the behaviour of those Equation nets, which involve model-time. Secondly, we define the behaviour of those which do not. At the end of this section we explain and motivate some of the important choices underlying our definition of dynamic behaviour.

Descriptions involving model-time

Each behaviour alternates between continuous and instantaneous mode (c.f. section 3.1). The behaviour defines a function from model-time into single markings (continuous mode) or into maximal firing sequences (instantaneous mode).

In instantaneous mode the model-time is constant and all changes in the marking of the net is due to transition firings according to the normal firing rules of high-level Petri nets. Each instantaneous mode corresponds to one moment of model-time and this moment is by the behaviour mapped into a maximal firing sequence, i.e. a firing sequence ending with a marking in which no transition can fire. This dead marking becomes the first marking of the succeeding continuous mode.

In continuous mode no transition can fire, but model-time is continuously increased over an interval. The positions of tokens are constant (since no transition fires), but the colours of tokens change as a function of model-time. Assume that we (disregarding environment places) have n tokens with colours $\{(r_i, s_i) \mid 1 \leq i \leq n\}$, and assume that the corresponding places have the place-equations $\{(I_i^* \mid F_i) \mid 1 \leq i \leq n\}$ where $F_i: \text{Env} \rightarrow \text{Store} \rightarrow \text{Bool}$ (some of the tokens, representing objects in the same family, may be at the same place and thus have identical place-equations). I_i^* are internal data-attributes, and thus their locations can be determined from the local environments r_i only. F_i may however involve external attributes and thus it must be given a compound environment r_i^* and a compound store s_i^* (analogous to the way we handled external event-statements in section 6.5):

$$r_i^* = r_i[r_1, \dots, r_{i-1}, r_{i+1}, \dots, r_n / I_{01}, \dots, I_{0i-1}, I_{0i+1}, \dots, I_{0n}]$$

$$s_i^* = s_1 + s_2 + \dots + s_{i-1} + s_i + s_{i+1} + \dots + s_n$$

where I_{0j} is the object-name corresponding to environment r_j . While model-time increases, the equation system is kept satisfied by changing the contents of the locations $\cup\{r_i(I_i^*) \mid 1 \leq i \leq n\}$ and

thus $\{s_i | 1 \leq i \leq n\}$ in such a way that $\{F_i r_i * s_i * | 1 \leq i \leq n\}$ always evaluate to true. In continuous mode there is a one-one correspondence between model-time and markings. Each continuous mode is minimal in the sense that it ends whenever a transition can fire. This non-dead marking becomes the first marking of the succeeding instantaneous mode.

Following the informal description of continuous and instantaneous mode given above, we now present a more formal definition of dynamic behaviour, for the kind of Equation nets used in our formal semantics (section 6.1 - 6.7).

As usual we use

$$m_1 \xrightarrow{X} m_2$$

to denote that the marking m_1 by a single step of concurrent transitions X is transformed to the marking m_2 , according to the normal firing rules of high-level Petri nets.

We say that a marking m is dead iff it is dead with respect to \rightarrow , i.e. if no transition can fire in m . A firing sequence is a sequence of markings $f = (m_i)_{1 \leq i \leq n}$ where $1 \leq n < \infty$, such that for each $1 \leq i < n$ there exists a step X_i satisfying $m_i \xrightarrow{X_i} m_{i+1}$.

A marking m is said to be consistent iff each place-equation evaluates to true, for all tokens present at that place.

We say that two markings m_1 and m_2 are equivalent $m_1 \equiv m_2$ iff for each place p $m_1(p)$ is identical to $m_2(p)$ except for those components of the token-colour(s), which represent the values of the TIME variable and the changeable variables in the place-equation of p .

Below we define the transformation of markings in such a way, that all TIME variables are synchronously increased, i.e. in any marking they all have equal values (c.f. section 1.3). We use $\text{TIME}(m)$ to denote this common value in marking m .

For two markings m_1 and m_2 with $\text{TIME}(m_1) < \text{TIME}(m_2)$ and a function f mapping the closed interval $I = [\text{TIME}(m_1), \text{TIME}(m_2)] \subset \mathbb{R}$ into markings, we define

$$m_1 \stackrel{f}{\sim} m_2 \iff \begin{cases} f(\text{TIME}(m_i)) = m_i \text{ for } i = 1, 2 \quad \wedge \\ \forall t \in I [\text{TIME}(f(t)) = t \quad \wedge \\ \quad f(t) \text{ is consistent} \quad \wedge \\ \quad f(t) \equiv m_1 \quad \wedge \\ \quad t \neq \text{TIME}(m_2) \Rightarrow f(t) \text{ is dead}] \end{cases}$$

We then use the notation

$$[m_1, m_2]_f = \{f(t) \mid t \in I\}$$

Let an Equation net with initial marking m_0 be given. A dynamic behaviour (starting at $t_0 = \text{TIME}(m_0)$, having events at $E = \{t_i \mid 1 \leq i \leq n\}$ where $1 \leq n < \infty$, and ending at t_{n+1}) is a function b mapping the closed interval $I = [t_0, t_{n+1}] \subset \mathbb{R}$ into firing sequences, satisfying

- a) $t_0 \leq t_1 < t_2 < \dots < t_i < t_{i+1} < \dots < t_n \leq t_{n+1}$
- b) $b(t)$ is a single marking for all $t \in I \setminus E$
- c) $\text{LAST}(b(t_i)) \stackrel{f_i}{\sim} \text{FIRST}(b(t_{i+1}))$
for all $1 \leq i < n$
and for $i = 0$ iff $t_0 \neq t_1$
and for $i = n$ iff $t_n \neq t_{n+1}$

where f_i maps the interval $[t_i, t_{i+1}]$ into markings and is defined by

$$f_i(t) = \begin{cases} \text{LAST}(b(t_i)) & \text{if } t = t_i \\ b(t) & \text{if } t_i < t < t_{i+1} \\ \text{FIRST}(b(t_{i+1})) & \text{if } t = t_{i+1} \end{cases}$$

A marking is reachable iff it is contained in a behaviour.

Descriptions not involving model-time

The complexity of dynamic behaviour is mainly due to the combination of explicit discrete descriptions (event-statements) with implicit descriptions based on the continuous increase of model-time (equational-statements). This combination is appropriate for the description of some systems, but for other systems we can avoid the use of model-time. When this is the case, the changes in continuous mode also take place as a number of discrete steps and the definition of dynamic behaviour can be simplified as shown below.

For two markings m_1 and m_2 in an Equation net without model-time, we define

$$m_1 \rightsquigarrow m_2 \iff \begin{cases} m_1 \text{ and } m_2 \text{ are consistent} \\ m_2 \equiv m_1 \\ m_1 \text{ is dead} \end{cases}$$

Let an Equation net with initial marking m_0 be given. When the net do not involve model-time a dynamic behaviour is a sequence of markings $b = (m_i)_{0 \leq i \leq n}$ where $0 \leq n < \infty$, such that for each $0 \leq i < n$ there exists a step $m_i \rightarrow m_{i+1}$ or a step $m_i \rightsquigarrow m_{i+1}$. A marking is reachable iff it is contained in a behaviour.

If all reachable markings are non-dead, \rightsquigarrow cannot be applied. Then each behaviour consists of a single instantaneous mode, and it is simply a firing sequence, defined by the normal firing rules of high-level Petri nets.

An Equation net (with or without model-time) is consistent iff each reachable marking is consistent. We shall demand all Epsilon descriptions to yield consistent Equation nets.

Explanation and motivation

The formal definitions of behaviour and consistency, given above, are based on a number of choices, and we now finish this section

by explaining and motivating the most important of them.

The semantics presented in this paper redefines the sequential event concept in Delta [Holbæk-Hanssen, Håndlykken & Nygaard, 75] by means of the concurrency concept in Petri nets. Delta allows a nearly unlimited use of external variables in event-actions, and when we first introduced concurrent event-actions in Epsilon, we preserved this nearly unlimited use of external variables [Jensen, Kyng & Madsen, 79b]. As one of the penalties of this choice, the concurrency relation could not be determined from the net structure alone.

In our present opinion this was not just a technical "net-problem", but an indication of irregular language concepts, allowing too complicated relations between event-actions. The notion... of external event-actions, described in this paper, allows us to retain the sufficient expressional power, and at the same time drastically simplifies the ties between different event-actions. Now, execution of event-actions are concurrent iff the corresponding transitions are concurrent in the net structure, i.e. according to the normal firing rules of high-level Petri nets.

We have also redefined the relation between event-actions and the immediately succeeding equational-actions. Again the starting point was a technical problem related to the net-model: In Delta and the early version of Epsilon instantaneous mode achieved consistent markings by a two-step transformation. First the old marking was modified according to the transition firing (event-actions) and then the obtained "pre-marking" was turned into a consistent marking, by imposing the place-equations of marked places (equational-actions).

When concurrency of event-actions was reduced to normal Petri net concurrency of the corresponding transitions, this was a strong argument in favour of reducing also the transformations in instantaneous mode to be single steps, following the normal firing rules of high-level Petri nets. If this is done, consistency of markings must be achieved, by demanding each transition to

be defined in such a way, that it establishes the place-equations of its output places, and does not destroy the place-equations of places being concurrently marked. Equation nets with this property are said to be smooth.

According to the old semantics, with two-step transformations in instantaneous mode and non-smooth Epsilon-descriptions, some of the changes due to the first step (event-actions) might be "cancelled" by the second step (equational-actions). This was an unsatisfactory situation, adding unnecessary complexity to the system description.

Earlier we considered Epsilon descriptions yielding smooth Equation nets to be an important subclass of all Epsilon descriptions. However, going through a large number of examples, we found that nearly all of them were smooth, and those which were not could easily be rewritten to be so, usually gaining clarity. For this reason we now demand all Epsilon descriptions to be smooth, c.f. the definition of consistency, and we reduce the transformations in instantaneous mode to be defined only by the normal firing rules of high-level Petri nets.

6.9 EVALUATION OF OUR SEMANTIC APPROACH

In this chapter we have defined a formal semantics covering most aspects of Epsilon. We have used a new semantic approach built on denotational semantics, high-level Petri nets and equation-systems. The formal semantics has been a strong help during the language design. It has improved our understanding and treatment of non-determinism, concurrency, scope-rules, and the difference between continuous and instantaneous mode. It has revealed a number of inconsistencies and helped us to simplify some of the language constructs. Two examples of this are discussed in the preceding section. Numerous other examples can be found in [Jensen, Kyng & Madsen, 79a].

The formal semantics allows us to present and discuss the different language constructs in a more precise way, and thus it enhances language dissemination. It allows us to perform rigorous analysis of small system descriptions as shown in [Kyng, 82].

Acknowledgements

The work reported in this paper is partly funded by the Danish Technology Council, Grant No. 1981-133/440-81.177. Comments from a number of students improved the readability of this paper.

REFERENCES

[Brauer, 80]

Brauer, W. (ed.): Net theory and applications. Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Hamburg 79. Lecture Notes in Computer Science vol. 84, Springer Verlag 1980.

[Bødker & Hammerskov, 82]

Bødker, S. and Hammerskov, J.: Graphical System Description. Thesis. DAIMI IR-33, 34, 35, Computer Science Department, Aarhus University, February 1982. (In Danish).

[Dahl, Dijkstra & Hoare, 72]

Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R.: Structured Programming. Academic Press, London 1972.

[Dahl, Myhrhaug & Nygaard, 70]

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

[Gordon, 79]

Gordon, M.J.C.: The denotational description of programming languages. Springer Verlag 1979.

[Genrich & Lautenbach, 81]

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

[Hansen, 78]

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

[Hansen & Madsen, 82]

Hansen, N.D. and Madsen, K.H.: Formal semantics by a combination of denotational semantics and high-level Petri nets. To appear in the proceedings of the Third European Workshop on Applications and theory of Petri nets, Informatik-Fachberichte, Springer Verlag.

[Hoare, 78]

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

[Holbæk-Hanssen, Håndlykken & Nygaard, 75]

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

[Håndlykken & Nygaard, 80]

Håndlykken, P. and Nygaard, K.: The DELTA System Description Language - Motivation, Main Concepts and Experiences from Use. Paper presented at the GMD Symposium on Software Engineering Environments, Lahnstein, Germany, June 1980.

[Ichbiah et al, 80]

Ichbiah, J.D. et al: Reference manual for the ADA programming Language. Proposed standard document. United States Department of Defense, July 1980.

[Jensen, 78]

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

[Jensen, 81a]

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

[Jensen, 81b]

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

[Jensen, 82]

Jensen, K.: High-level Petri nets. To appear in the proceedings of the third European Workshop on Applications and theory of Petri nets, Informatik-Fachberichte, Springer Verlag.

[Jensen & Kyng, 80]

Jensen, K. and Kyng, M.: Petri nets and semantics of system descriptions. DAIMI PB-116, Computer Science Department, Aarhus University, April 1980.

[Jensen, Kyng & Madsen, 79a]

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

[Jensen, Kyng & Madsen, 79b]

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

[Keller, 76]

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

[Kristensen, Madsen, Møller-Pedersen & Nygaard, 81]

Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B. and Nygaard, K.: A survey of the BETA programming language. Norwegian Computing Center, Oslo, report no. 698, November 1981.

[Kyng, 82]

Kyng, M.: Specification and verification of networks in a Petri net based language. DAIMI PB - 153, Computer Science Department, Aarhus University, September 1982. (To appear in the proceedings of the third European Workshop on Applications and theory of Petri nets, Informatik-Fachberichte, Springer Verlag).

[Lautenbach, 75]

Lautenbach, K.: Liveness in Petri nets. Interner Bericht ISF-75-02.1, GMD Bonn, July 1975.

[Mazurkiewicz, 77]

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

[Petri, 75]

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

[Petri, 76]

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

[Stoy, 77]

Stoy, J. E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press 1977.

[Tennent, 76]

Tennent, R.D.: Language Design Methods Based on Semantic Principles. Acta Informatica 8 (1977).

[Wang, 75]

Wang, A.: Generalized types in high-level programming languages. Research Reports in Informatics, No. 1, University of Oslo, 1975.