DELTA PROJECT
REPORT NO. 13

ISSN 0105-8517

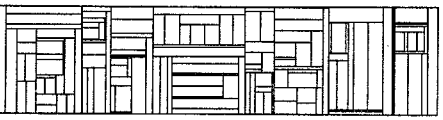
A PETRI NET DEFINITION OF A SYSTEM DESCRIPTION LANGUAGE

by
Kurt Jensen
Morten Kyng
Ole Lehrmann Madsen

This paper has also been published in: Semantics of Concurrent Computation, G. Kahn (ed.), Lecture Notes in Computer Science vol. 70, Springer Verlag 1979, 348-368.

DAIMI PB-96
November 1981 (second edition)

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



A PETRI NET DEFINITION OF A SYSTEM DESCRIPTION LANGUAGE

Abstract

This paper introduces a language, Epsilon, for the description of systems with concurrency, and presents a formal definition of Epsilon's semantics. The language is based on Delta, the first major attempt to create a language solely aimed at system description without the restrictions placed on languages executable on digital computers. The design of Delta was itself heavily influenced by the experience from the development and use of Simula.

It is not obvious what kind of semantics a system description language should have. The situation is more complex than with normal algorithmic languages and none of the existing semantic approaches appear to be satisfactory.

To clarify the situation, we first describe the language Epsilon, which contains only a few basic primitives. Then we define the semantics of Epsilon by means of a formal model based on Petri nets. The model called "Concurrent systems" is an extension of Petri nets with a data part and with expressions attached to transitions and to places. The model is a further development of formalisms proposed by R.M. Keller and A. Mazurkiewicz. The expressions attached to places is a novel feature and is used to define continuous transformations on the data part. The semantics of a given system description is defined in terms of firing sequences of the corresponding concurrent system.

1. INTRODUCTION

The purpose of this paper is to introduce a system description language, Epsilon, based on the Delta language [Delta 75], and to present a formal definition of its semantics by means of a model based on Petri nets ([Petri 73, 75, 76], [Peterson 77]).

During the last decade there has been an increasing need to understand, control and design large, complex systems of men and machines, and thus a demand for concepts and formal languages to conceive and describe systems with concurrency. Mathematics was for a long time the most important tool for analysing such systems. With the advent of the electronic computer, simulation programming languages have become a popular tool for the description and analysis of complex systems, untractable by mathematics. It turned out that the process of carefully creating a system description (i.e. a program) was often at least as beneficial as actually generating the described model (i.e. a program execution in the computer store) and gathering statistics about it. This experience gave birth to the idea of creating a language specially designed for the description of systems, a language without the restrictions necessarily placed on any language executable on a digital computer. The first major attempt in this direction is the Delta language which was created at the Norwegian Computing Center and which draws heavily on the experience from the development and use of Simula ([Simula 70], [Nygaard 70]). Delta is a tool for system specialists and for other groups working with and influenced by data processing systems. It is also a tool for research workers inside other fields such as biology, medicine, and physics.

The conceptual framework behind Delta introduces a large number of new and interesting ideas. The actual language however is not quite as carefully worked out. It is large and complicated, and the relation between some of the language elements is diffuse. The semantics is only informally described although the outline of an abstract machine for "execution" of Delta descriptions has been defined by programming it in the language itself.

It is not clear what kind of information one wants to extract from system des-

criptions, i.e. what kind of semantics system description languages should have. The situation is more complex than what is found in normal algorithmic languages and none of the existing semantic approaches appear to be satisfactory. As an attempt to clarify the situation, we describe a system description language called Epsilon containing only a few basic primitives and we define its semantics by means of a formal model based on Petri nets. Petri nets are chosen because they contain a number of concepts which are closely analogous to Delta. Moreover, Petri nets are themselves useful tools for the description of systems with concurrency. The semantic model is a further development of formalisms defined in [Keller 76] and [Mazurkiewicz 77].

Since Epsilon is intended for describing systems of interacting objects, such as men and machines, an Epsilon system consists of a nested structure of objects. An object is characterised by the actions it executes and a selected set of attributes, which may be variables, procedures and objects.

State transformations can be described by means of algorithms or by means of equations. An algorithm is used in cases where it is adequate to describe the way in which a given state transformation is carried out. Equations are used, when it is adequate to describe state transformations implicitly by means of the properties which a given set of variables should fulfil, and when it is less relevant how the properties can be achieved.

The variables of an object may be observed by other objects and will then always have well defined values, i.e. intermediate values appearing during an algorithmic state transformation will not be observable from other objects. Interaction between objects may also take place by means of interrupts, that is an object may require another object to execute some specific actions.

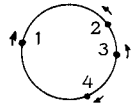
Epsilon also contains a concept of time which can be used to describe continuous state transformations.

The use of equations, observation of non-local variables and the time concept are the main features that distinguish Epsilon from normal programming languages. Epsilon is not a complete language, but it does contain the new and essential ideas from Delta.

In section 2 Epsilon is informally introduced by means of an example. Sections 3, 4, and 5 define the semantic model, the abstract syntax, and the formal semantics of Epsilon respectively. Finally section 6 evaluates the approach and discusses future work.

2. INFORMAL DESCRIPTION OF EPSILON

The main features of Epsilon will be introduced by means of an example describing four balls following a circular orbit. The balls may move in both directions or stand still. Elastic collisions may appear between the balls. Two balls which collide will exchange their velocity (speed and direction). An observer may place a "wall" in front of a ball and in this way negate its velocity. It is assumed that no other forces influence the system, i.e. no friction, no gravitation and no loss of energy. All the balls have the same mass and size.



The following semiformal description of the system, Fig. [2.1], introduces the objects and the kind of actions they perform. Formal language elements are written in capital letters with keywords underlined and informal language elements (i.e. not specified in detail) are written in small letters.

An Epsilon system consists of a nested structure of objects. The present system consists of six such objects: four ball objects (lines 2–10), the observer object (11–18) and the system object (1–19) containing the other objects. An object is characterised by a set of attributes (objects, procedures and variables) and an action part.

The declaration (11–18) introduces one object with the name OBSERVER. It has no attributes and its action part (13–17) is a repeated execution of a LET-imperative (14–16). The declaration (2–10) introduces four objects with the names BALL(1), ...,(BALL(4). Each BALL only has one attribute: its position on the orbit (3). Its action part (4–9) is a repeated execution of a LET-imperative (5–8).

The state of an object is a pair consisting of the values of its variables and its stage of execution. The state of an Epsilon system is the set of states for all objects. All objects execute their actions in concurrency.

An object may both observe and change the values of its own variables, but it may only observe variables in other objects.

Objects can interact by means of interrupts; that is, an object, A, may require an object, B, to execute a specific procedure, P, which must be an attribute of B. Each object controls when it wants to accept possible interrupting procedures for execution. Thus, objects may synchronise their actions in two ways: by observing variables and by sending and accepting interrupts. An asynchronous set of objects may be described by restricting all their synchronisation to take place by interrupts. A synchronous set of objects may be described by using observation of variables to synchronise actions.

By means of nesting of objects it is possible to restrict the observability of object-attributes. These and other scope rules will not be defined in this paper.

```
 1 SYSTEM BEGIN
 2 OBJECT BALL(1..4):
 3   BEGIN VAR POSITION : REAL;
 4     REPEAT
 5       LET POSITION = new-position DEFINE POSITION
 6         WHEN crash DO exchange-velocity (crashing-neighbour)
 7         ACCEPT OBSERVER : change-direction
 8       ENDLET
 9     ENDREPEAT
10   END BALL;
11 OBJECT OBSERVER:
12   BEGIN
13     REPEAT
14       LET observe
15         WHEN ready DO INTERRUPT selected-ball BY change-direction
16       ENDLET
17     ENDREPEAT
18   END OBSERVER;
19 END SYSTEM
```

Figure [2.1]

An object alternates between executing two kinds of actions: event-actions and continuous-actions. Event-actions are instantaneous, indivisible and executed by one object, but possibly in concurrency with actions executed by other objects. Continuous-actions are time consuming, interruptable, and executed in cooperation with continuous-actions executed by other objects. An imperative containing a continuous-action specifies an equation to be fulfilled by the system state but gives no details about how this can be achieved.

Actions are described by imperatives.

The execution of a LET-imperative starts with the execution of a continuous-action. For the LET-imperative (5-8) of a BALL this means that the equation "POSITION = new-position" is constantly satisfied while this continuous-action is executed. The variables following DEFINE (here POSITION) may be changed in order to satisfy the equation. The WHEN-clause (6) describes that the execution of the continuous-action will be stopped when the boolean-expression "crash" becomes true. If this happens the imperative following DO (6) is executed. Similarly the ACCEPT-clause (7) describes that the continuous-action may be interrupted if the OBSERVER sends the interrupt "change-direction". If this happens the BALL will execute the procedure "change-direction". In both cases the execution of the LET-imperative is stopped.

When the OBSERVER executes its LET-imperative (14-16), the equation

"observe" will be satisfied. At certain moments of time, the boolean-expression "ready" will be satisfied and the OBSERVER will interrupt one of the BALLs (15).

The system-object executes no actions.

A typical situation in the system will be that all four BALLs are moving around the circular orbit with the OBSERVER merely "observing" them. The BALLs will execute the continuous-action described by the LET-imperative (5-8) and the OBSERVER the one described by the LET-imperative (14-16). The system state will satisfy the equations "POSITION = new position" and "observe".

If two BALLs collide the following event-actions will take place: The boolean-expressions "crash" will be true in the two colliding BALLs. Both of the BALLs will execute the actions described by "exchange-velocity (crashing-neighbour)". Similarly event-actions will be executed if the OBSERVER decides to change the direction of one of the BALLs. The "selected-ball" will then be interrupted and it will execute the procedure "change-velocity".

In Figure [2.2] a more detailed description of the system is given. The description of the BALLs has been extended in order to specify how velocity and position vary. More attributes have been added to each BALL (3-12). Note, however, that some language elements are still informal.

The actions "exchange-velocity" and "change-direction" have been specified as procedures. A sequence of micro-imperatives enclosed within ≪, ≫ describes one event-action (5, 12, 13) and is thus an instantaneous and indivisible action. Micro- imperatives are normal algorithmic imperatives such as assignment, selection and repetition. The equation "POSITION = new-position" has been specified in detail (15).

TIME is an implicitly defined variable contained in the system object and representing time in the system modelled. TIME is continuously increased. (See 5.4 for a precise description of TIME.)

The boolean-expression "crash" has been separated into two parts (16-17), one for a crash with its left neighbour, and one for a crash with its right neighbour. For the sake of brevity we have not described "left/right-crash" in detail.

EXECUTE is a procedure call (16, 17). PUT is a call-by-value parameter transfer (8, 16, 17); the local variable I in EXCHANGE-VELOCITY is assigned the number of the left/right neighbour. Each BALL has an integer attribute with the name B (2) which for BALL(i) (i = 1, 2, 3, 4) has the value i. P_0, T_0 are used to hold the values of POSITION and TIME at the last collision.

Each BALL starts its actions by an initialisation of its local variables (13). The initial positions, p(B) (13) for the BALLs are assumed to be modulo the length of the orbit. The BALLs cannot pass each other hence it is sufficient to compare their positions directly (without modulo).

```
 1 SYSTEM BEGIN
 2 OBJECT BALL(B:1..4):
 3   BEGIN VAR POSITION, VELOCITY, P0, T0: REAL;
 4     PROCEDURE CHANGE-DIRECTION:
 5       BEGIN ≪VELOCITY:=-VELOCITY; P0:=POSITION; T0:=TIME≫END;
 6     PROCEDURE EXCHANGE-VELOCITY:
 7       BEGIN VAR I: 1..4;
 8         INTERRUPT BALL(I) BY NEW-VELOCITY PUT (V:=VELOCITY);
 9         LET TRUE ACCEPT BALL(I): NEW-VELOCITY ENDLET
10       END;
11     PROCEDURE NEW-VELOCITY:
12       BEGIN VAR V: REAL; ≪VELOCITY := V; P0:=POSITION; T0:=TIME≫END;
13     ≪POSITION:= p(B);  VELOCITY:=v(B); P0:=POSITION; T0:=TIME≫;
14     REPEAT
15       LET POSITION=P0 + VELOCITY * (TIME-T0) DEFINE POSITION
16         WHEN leftcrash DO EXECUTE EXCHANGE-VELOCITY PUT (I:=B ⊖ 1)
17         WHEN rightcrash DO EXECUTE EXCHANGE-VELOCITY PUT (I:=B ⊕ 1)
18         ACCEPT OBSERVER : CHANGE-DIRECTION
19       ENDLET
20     ENDREPEAT
21   END BALL;
22 OBJECT OBSERVER: ....
23   END OBSERVER;
24 END SYSTEM
```

Figure [2.2]

3. CONCURRENT SYSTEMS

Concurrent systems is a semantic model based upon Petri nets.

3.1 Petri nets

A Petri net PN = (P, T, PRE, POST) is a 4-tuple, where P is a set of places, T is a set of transitions, PRE and POST are functions from T into subsets of P. Moreover

1) $P \cup T \neq \emptyset$

2) $P \cap T = \emptyset$

3) $\forall\ t \in T\ [PRE(t) \cap POST(t) = \emptyset]$

A marking is a function $m: P \to \{0, 1\}$. A place p is marked if $m(p) = 1$. If $m(p) = 0$, p is unmarked. For each $t \in T$ the set $COND(t) = PRE(t) \cup POST(t)$ are conditions for t. Two transitions t_1 and t_2 are independent iff $COND(t_1) \cap COND(t_2) = \emptyset$.

A nonempty subset of mutually independent transitions, $X \subseteq T$, has concession in a marking m iff

$$\forall p \in P \quad \begin{bmatrix} p \in PRE(X) & \Rightarrow m(p) = 1 \wedge \\ p \in POST(X) & \Rightarrow m(p) = 0 \end{bmatrix}$$

where $PRE(X) = \cup \{PRE(t) \mid t \in X\}$ and $POST(X) = \cup \{POST(t) \mid t \in X\}$.

When X has concession in m it may fire. If it fires a new marking m' is reached, such that

$$\forall p \in P \quad \begin{bmatrix} p \in PRE(X) & \Rightarrow m'(p) = 0 \wedge \\ p \in POST(X) & \Rightarrow m'(p) = 1 \wedge \\ p \notin COND(X) & \Rightarrow m'(p) = m(p) \end{bmatrix}$$

where $COND(X) = \cup \{COND(t) \mid t \in X\}$. m' is said to be directly reachable from m, which we write as $m \longrightarrow m'$ or $m \xrightarrow{X} m'$.

3.2 Predicates and relations

Let V be a set of variables, which each may take values in a domain F. Let $A \subseteq V$ be given, and let $[A \rightarrow F]$ denote all total functions from A to F. The set of predicates over A is defined as $[[A \rightarrow F] \rightarrow \{true, false\}]$ and is denoted by $PRED_A$. The set of binary relations over A is defined as all subsets of $[A \rightarrow F] \times [A \rightarrow F]$ and it is denoted by REL_A.

3.3 Expressions attached to transitions

While Petri nets are excellent models for the control flow in a language, they are less suited as models for state transformations in the data part. To remedy this situation we augment Petri nets with a data part containing a set of variables and we attach to each transition t an expression of the form: WHEN GUARD(t) DO REL(t) where GUARD(t) is a predicate over a subset of variables and REL(t) is a binary relation defining a set of possible transformations on the same subset of variables. This set of variables is called the scope of t and is denoted by SC(t).

A transition t has concession only when GUARD(t) is satisfied by the current values of the variables in SC(t). If t fires, one of the possible transformations contained in REL(t) is performed upon the variables in SC(t). Transitions can fire concurrently only if they have disjoint scopes. Thus the firing rule for a subset of transitions is modified by an added requirement on concession for each transition and an added requirement on independence for each pair of transitions.

3.4 Expressions attached to places

As a second extension we attach to each place p an expression of the form: LET EQ(p) DEFINE VAR(p) where EQ(p) is an equation over a subset of variables. This set of variables is called the scope of p and is denoted by SC(p). VAR(p) is a subset of SC(p).

We define a control state to be a marking, while a data state is a set of values for the variables. A system state is a pair consisting of a control state and a data

state, such that the data state satisfies all equations attached to places marked in the control state.

An equation is satisfied in a data state iff the predicate constructed from it in the usual way evaluates to TRUE.

The firing of a set of transitions can now be described as the following two steps:

A) The control state is changed according to the modified firing rule, while the data state is changed according to the relations attached to the firing transitions.

B) The equations attached to places marked in the new control state are established. This is done by changing the values of some of the variables in $\cup \{VAR(p) \mid m(p) = 1\}$.

The "intermediate state" between A and B is not considered a system state since there may be equations, which are unsatisfied although they are attached to marked places. Formally we will define the firing of a set of transitions as an instantaneous and indivisible action leading directly from one system state to the next without any intermediate state.

3.5 Concurrent systems

A concurrent system is a triple CS = (CON, INT, INIT) where

- CON, the control part, is a Petri net (P, T, PRE, POST)
- INT, the interpretation, is a pair (DATA, EXP), where
  1) DATA, the data part, is a pair (V,F), where V is a set of variables, which each may take values in domain F.
  2) EXP, the expression part, is a 5-tuple (EQ, VAR, GUARD, REL, SC) consisting of five functions:

$$EQ : P \to \cup \{PRED_A \mid A \subseteq V\}$$
$$VAR : P \to \mathcal{P}(V)$$
$$GUARD : T \to \cup \{PRED_A \mid A \subseteq V\}$$
$$REL : T \to \cup \{REL_A \mid A \subseteq V\}$$
$$SC : P \cup T \to \mathcal{P}(V)$$

such that

$$\forall p \in P\ [EQ(p) \in PRED_{SC(p)} \wedge VAR(p) \subseteq SC(p)]$$
$$\forall t \in T\ [GUARD(t) \in PRED_{SC(t)} \wedge REL(t) \in REL_{SC(t)}]$$

- INIT, the initial system state, is a system state, (m_1, s_1). See below.

The set of all markings, $M = [P \to \{0, 1\}]$, are called control states. The set of all data values, $S = [V \to F]$, are called data states. A pair $(m,s) \in M \times S$ is a system state iff $\forall p \in P\ [m(p) = 1 \Rightarrow EQ(p)(s_{SC(p)}]$, where $s_{SC(p)}$ is the restriction of s to SC(p).

For Petri nets we have defined a set of concepts, such as independence, concession and direct reachability. We will now define similar concepts for concurrent systems. Since the latter definitions are generalisations of the former ones, we will use the same names. To avoid ambiguity, we will prefix the new concepts with "cs-".

Two transitions t_1 and t_2 are cs-independent iff t_1 and t_2 are independent and $SC(t_1) \cap SC(t_2) = \emptyset$.

A nonempty set of mutually cs-independent transitions, $X \subseteq T$, has cs-concession in a system state (m,s) iff

1) X has concession in m

2) $\forall\, t \in X\ [GUARD(t)\ (s_{SC(t)})]$

When X has cs-concession in (m,s), it may fire. If it fires there are two different possibilities:

A) If there exists a system state (m',s') such that

1) $m \xrightarrow{X} m'$

2) $\exists\, s'' \in S\ [\forall t \in X\ [(s_{SC(t)}, s''_{SC(t)}) \in REL(t)] \wedge s_A = s''_A \wedge s''_B = s'_B]$

where $A = V - \cup \{SC(t) \mid t \in X\}$ and $B = V - \cup \{VAR(p) \mid m'(p) = 1\}$

(m',s') is said to be cs-directly reachable from (m,s), which we write as $(m,s) \longrightarrow (m',s')$ or $(m,s) \xrightarrow{X} (m',s')$.

B) If such a system state does not exist, firing of X in (m,s) is a violation.

From now on we will omit the prefix "cs-" and always refer to the definitions of concurrent systems and not to those of Petri nets.

A sequence of system states, $fs = \{(m_i, s_i)\}_{1 \le i \le n}$ where $1 \le n \le \infty$, is a firing sequence iff $(m_i, s_i) \longrightarrow (m_{i+1}, s_{i+1})$ for all i, where $1 \le i < n$. fs is finite iff $n < \infty$. A finite firing sequence is maximal iff no transition has concession in (m_n, s_n) and violating iff a set of transitions may fire as a violation in (m_n, s_n). Moreover by definition $fs.FIRST = (m_1, s_1)$, $fs.LAST = (m_n, s_n)$ if $n < \infty$ else undefined, $|fs| = n$, and $(m,s) \in fs$ iff $(m,s) = (m_i, s_i)$ for some i, where $1 \le i \le n$.

Having now shown that concurrent systems have a rigorous mathematical definition, we will in the rest of this paper use a more informal notation consisting of the normal graphical notation for Petri nets, augmented with expressions as indicated in subsections 3.3 and 3.4. Scopes will always be implicitly defined by the involved subexpressions for EQ(p), GUARD(t), and REL(t). If EQ(p) or GUARD(t) is omitted this is equivalent to the always satisfied equation, which is denoted by TRUE. If REL(t) is omitted this is equivalent to the identity relation. If VAR(p) is omitted this is equivalent to the empty subset of variables.

3.6 Comparison with similar formalisms

Adding a data part to a Petri net and attaching expressions to transitions have also been proposed in [Keller 76] and [Mazurkiewicz 77]. The attachment of expressions to places is primarily inspired by the LET-imperative in Delta, which allows the values of variables to be defined implicitly by means of equations at the

expense of algorithmic transformations. A similar idea is present in assignment systems, [Thiagarajan & Genrich 76].

In the formalism of Mazurkiewicz one is only allowed to attach expressions to transitions in such a way that (Petri net) concurrent transitions get disjoint scopes. Furthermore Mazurkiewicz formalism consists of a scheme and an interpretation. The schemes adhere to the Petri net firing rules. An effect similar to the use of GUARDS are introduced via the interpretation.

In Keller's formalism there is no concurrency between transitions: "if there is any possibility of simultaneous events occurring, such an occurrence can be represented as a sequence of occurrences of events in some arbitrary order." Keller only allows functions on variables, not relations in general. In Keller's terminology this means that our model is "nondeterministic" while his model is "deterministic". Both models are "polygenic".

4. SYNTAX OF EPSILON

In this section the (abstract) syntax of Epsilon is defined using an extended BNF. The use of {A} list{B} means one or more instances of A separated by B, i.e. A, ABA, ABABA, etc.; $list_0$ indicates that the list may be empty. An optional clause is indicated by opt{...}.

Syntax

```
1  <Epsilon-system> ::= SYSTEM <object descriptor>
2  <object-descriptor> ::= BEGIN opt{<decl>;} <imp> END
3  <decl> ::= <decl>; <decl> | VAR {<id>} list{,} : <type>
4      |OBJECT <id> opt{(<id> : <range>)} : <object-description>
5      |PROCEDURE <id>: BEGIN opt{<decl>;} <imp> END
6  <imp> ::= EMPTY | <imp>; <imp> | EXECUTE <proc-id> <put> <get>
7      | INTERRUPT <object> BY <proc-id> <put>
8      | LET <equation> opt{DEFINE {<var-id>} list{,}}
9          {WHEN <boolean-exp > DO <imp>} list0
10         {ACCEPT  <object> : <proc-id> <get>} list0 ENDLET
11     | REPEAT <imp> ENDREPEAT |«<micro-imp>»
12 <object> ::= <object-id> | <object-id>(<range-exp>)
13 <micro-imp> ::= EMPTY | <micro-imp>; <micro-imp> | <var-id> ::= <exp>
14     | IF <selection> FI | DO <selection> OD
15 <selection> ::= {<boolean-exp> → <micro-imp>} list{□}
16 <put> ::= opt{PUT({<var-id> := <exp>} list{,})}
17 <get> ::= opt{GET({<var-id> := <var-id>} list{,})}
```

The syntax of identifiers, types, ranges, equations, and the various expressions will not be specified. <id> is used when an identifier is declared. <object-id> is an application of an identifier declared as an OBJECT, etc. A precise definition of the context dependent parts of the syntax will not be given in this paper. The scope rules resemble those of Algol 60. Only VAR and PROCEDURE declarations may appear in a procedure. Recursive procedure calls are not allowed, neither directly using EXECUTE nor indirectly by means of INTERRUPT (see also sections 5 and 6).

PUT and GET-clauses are call-by-value and call-by-result parameter transfers. In PUT (GET) the leftside (rightside) of the assignments indicates local variables in the procedure whereas the rightside (leftside) is an expression (variable) evaluated at the place of the procedure activation (EXECUTE or ACCEPT).

5. SEMANTICS OF EPSILON

In this section we use concurrent systems to define the semantics of Epsilon. This is done by defining a syntax-directed translation of Epsilon descriptions into concurrent systems. In this paper the translation is only informally defined. It could be formalized using an attribute grammar with concurrent systems as attribute values. In [Pearl 78] a similar syntax-directed translation is formally defined by means of a van Wijngaarden grammar.

The set of behaviours for a concurrent system, defined in section 5.4 by means of firing sequences, constitutes the semantics of the corresponding Epsilon description.

For each language construct a corresponding concurrent system is given. The concurrent system for objects and procedures are defined in section 5.1. For each variable appearing in the description of an object or a procedure there will be a corresponding variable in the concurrent system.

It is important to distinguish between imperatives defined by <imp> (section 4, lines 6-11) and micro-imperatives defined by <micro-imp> (section 4, lines 13-14).

Imperatives and micro-imperatives belong to different levels in an Epsilon description. The action-part is described by imperatives. Imperatives of the form, ≪MIC-IMP≫, are called general-assignment-imperatives. Although defined by a sequence of micro-imperatives a general-assignment-imperative describes one indivisible and instantaneous event-action.

The actual choice of micro-imperatives in Epsilon is of less importance for this paper. We have chosen Dijkstra's guarded commands because they are useful and to illustrate that nondeterministic control structures are simple to define in terms of concurrent systems.

We shall use two levels of concurrent systems to model imperatives and micro-imperatives respectively. At the imperative level there is a single high level concurrent system containing a concurrent subsystem for each imperative in the system.

At the micro-imperative level there is a separate low level concurrent system for each general-assignment-imperative. Each low level concurrent system contains a concurrent subsystem for each micro-imperative in the corresponding general-assignment-imperative. The concurrent subsystems representing imperatives and micro-imperatives are defined in section 5.2 and 5.3 respectively.

A general-assignment of the form ≪MIC-IMP≫ will in the high level concurrent system be represented by one transition having an attached expression of the form DO REL. REL is defined by means of the firing sequences in the low level concurrent system corresponding to MIC-IMP.

The use of Petri nets often leads to large and unstructured nets. This disadvantage can be diminished by the use of levels of concurrent systems. There is no reason to have only two levels. The concurrent systems corresponding to micro-imperatives may use operations like +, *, etc. These will be part of the Epsilon language, and might also be defined using concurrent systems.

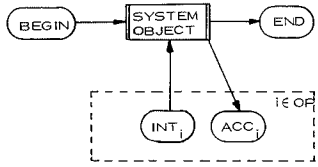
The use of concurrent systems at different levels resembles the use of morphisms in Petri nets [Petri 76] and the "structured nets" of [Kotov 78].

5.1 Objects and procedures

Each object is characterized by a set of attributes (objects, procedures and variables) and by an action part.

The actions in the system object are represented by a concurrent system of the form:

[5.1]



Dashed rectangles are used to indicate a set of identical concurrent subsystems. The text in the upper, righthand corner indicates the name and range of a variable used to give places, transitions, and expressions inside each concurrent subsystem a common subscript. In this simple situation each concurrent subsystem consists of only two places. The range, OP, is the set of all pairs of identifiers, OBJ:PROC, which occurs in one or more ACCEPT-clauses for the system object.

The special rectangle surrounding "SYSTEM OBJECT" indicates a closed concurrent subsystem (i.e. a concurrent subsystem, where no place in the subsystem is a condition for a transition outside the subsystem). The closed concurrent subsystem may have more conditions outside the subsystem than shown (cf. [5.7]).

Each procedure attribute in the system object is represented by a concurrent system of the form

[5.2]

where PROC is the name of the procedure.

Each object attribute in the system is represented by a concurrent subsystem constructed by the same rules as the system object.

Thus in the (high level) concurrent system representing an Epsilon system there is a concurrent subsystem for each object and for each procedure.

The closed concurrent subsystem in [5.1] and [5.2] are constructed from the imperatives contained in the object and the procedure respectively. Initially all BEGIN places for objects are marked, all other places are unmarked, and the values of variables are determined by language defined defaults.

5.2 Imperatives

Each imperative is represented by a concurrent system of the form:

[5.3]

EMPTY-imperative

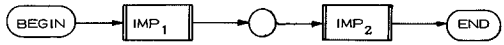
The EMPTY-imperative is represented by

[5.4]

Sequencing

The imperative "IMP_1; IMP_2" is represented by

[5.5]

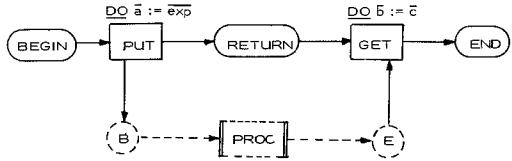


This should be understood as the concurrent system obtained by identifying the END place of the concurrent system representing IMP_1 with the BEGIN place of the concurrent system representing IMP_2. Similar remarks will tacitly be assumed for all following compositions of concurrent systems.

Procedure call

The imperative "EXECUTE PROC PUT($\bar{a}$:= $\overline{exp}$)GET($\bar{b}$:= $\bar{c}$)" is represented by

[5.6]



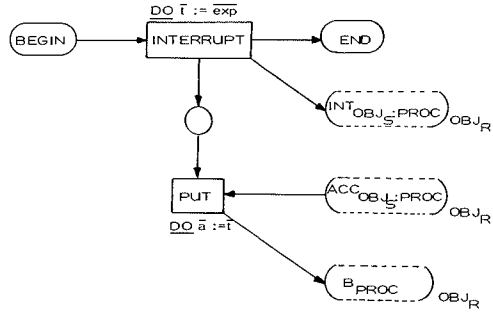
where the subsystem containing B, E, and PROC is the one introduced by the declaration of PROC, see [5.2].

The "dashed" places, transitions, and directed lines are used to indicate that the corresponding concurrent subsystem is not local to the procedure call, i.e. the subsystem is shared by all calls of the procedure (and all interrupts). The PUT-clause specifies a call-by-value parameter transfer, where the variables $\bar{a}$ (local to PROC) are assigned the values of the expressions $\overline{exp}$ (evaluated in the calling environment). Analogously the GET-clause specifies a call-by-result parameter transfer, where the variables $\bar{b}$ (in the calling environment) are assigned the values of the variables $\bar{c}$ (local to PROC). Note that recursive procedure calls are not allowed.

Interruption

Consider the imperative "INTERRUPT OBJ BY PROC PUT($\bar{a}$:= $\overline{exp}$)" executed by an object OBJ_S. When OBJ is a simple object identifier containing no range expression ($OBJ = OBJ_R$), the INTERRUPT-imperative is represented by

[5.7]



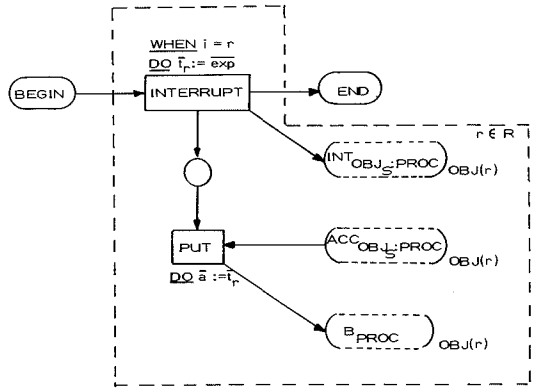
The places $INT_{OBJ_S:PROC}$ and $ACC_{OBJ_S:PROC}$ belong to the concurrent system representing OBJ_R (see [5.1]). B_{PROC} is the BEGIN place for the procedure PROC in object OBJ_R (see [5.2]). When $INT_{OBJ_S:PROC}$ is marked, an interrupt from OBJ_S with procedure PROC is waiting to be executed by OBJ_R. When this interrupt is accepted by OBJ_R, $ACC_{OBJ_S:PROC}$ becomes marked (see [5.9]).

$\bar{t}$ is a set of auxiliary variables used to remember the values of $\overline{exp}$ until OBJ_R is ready to receive them. Each INTERRUPT transition has its own set of auxiliary variables.

Normally OBJ_S is allowed to proceed immediately, without waiting for OBJ_R to accept and execute the interrupting procedure PROC. OBJ_S will be delayed only when $INT_{OBJ_S:PROC}$ is marked already. Note that recursive interrupts are not allowed.

When OBJ contains a range expression (OBJ = OBJ(i)), the INTERRUPT-imperative is represented by

[5.8]



R is the set of values, which the range expression may take.

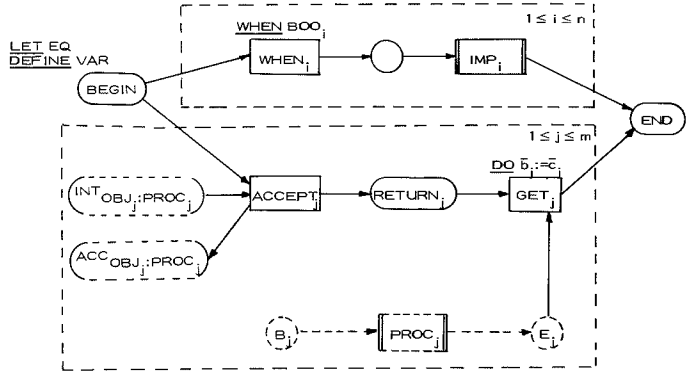
LET-imperative

The imperative

"LET EQ DEFINE VAR
WHEN BOO_1 DO IMP_1
⋮
WHEN BOO_n DO IMP_n ($n \geq 0$)
ACCEPT OBJ_1:$PROC_1$ GET($\bar{b}_1 := \bar{c}_1$)
⋮
ACCEPT OBJ_m:$PROC_m$ GET($\bar{b}_m := \bar{c}_m$) ($m \geq 0$)
ENDLET"

is represented by

[5.9]



where the subsystem containing B_j, E_j and $PROC_j$ is the one introduced by the declaration of $PROC_j$ in the object, OBJ, executing the LET-imperative, see [5.2]. $INT_{OBJ_j:PROC_j}$ and $ACC_{OBJ_j:PROC_j}$ are defined in [5.1].

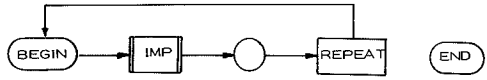
The place $INT_{OBJ_j:PROC_j}$ is marked by OBJ_j when it executes an imperative of the form "INTERRUPT OBJ BY $PROC_j$", see [5.7].

When the ACCEPT-clause contains range expressions, [5.9] is modified analogously to [5.8].

Repetition

The imperative "REPEAT IMP ENDREPEAT" is represented by

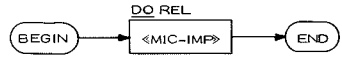
[5.10]



General-assignment-imperative

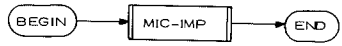
The imperative "≪MIC-IMP≫" is represented by

[5.11]



where the relation REL will be defined below by means of a separate (low level) concurrent system, CS, constructed from the micro-imperative MIC-IMP by the rules defined in section 5.3. The variables of CS and the scope of transition ≪MIC-IMP≫ are the set of variables, A, observed or changed in MIC-IMP. The variables must all be local to the object. CS has the form

[5.12]



Let m_B and m_E be the control states for CS in which only BEGIN and END respectively are marked and let a final state be any state of the form (m_E, u). Let FS(s) be the set of firing sequences, fs, where fs.FIRST = (m_B, s). Recalling that F is the value domain for the variables A we define REL as follows:

$$\forall\, s,r \in [A \to F]\ [(s,r) \in REL \iff$$
$$(\exists\ fs \in FS(s)\ [fs.LAST = (m_E, r)] \wedge$$
$$\forall\ fs \in FS(s)\ [fs \text{ is finite} \wedge (fs \text{ is maximal} \Rightarrow fs.LAST \text{ is a final state})])].$$

Let a data state s be given. If it is possible to go into an infinite loop (an infinite firing sequence exists) or to enter a non-final state where no transition has concession, then the above definition implies that there is no r such that $(s,r) \in REL$. In this case the firing of the transition representing the general-assignment-imperative in the data state s is a violation.

5.3 Micro-imperatives

Like imperatives each micro-imperative is represented by a concurrent system of the form [5.12]. In contrast to imperatives BEGIN and END will be the only places outside the closed concurrent subsystem being conditions for transitions inside.

EMPTY-micro-imperative

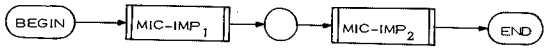
The EMPTY-micro-imperative is represented by

[5.13]

Sequencing

The micro-imperative "MIC-IMP$_1$; MIC-IMP$_2$" is represented by

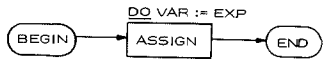
[5.14]



Assignment

The micro-imperative "VAR := EXP" is represented by

[5.15]



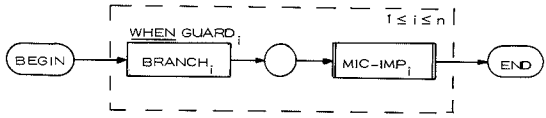
Selection

The micro-imperative (defined in [Dijkstra 75])

"IF GUARD$_1$ → MIC-IMP$_1$
▯ GUARD$_2$ → MIC-IMP$_2$
⋮
▯ GUARD$_n$ → MIC-IMP$_n$ $(n \geq 1)$
FI"

is represented by

[5. 16]

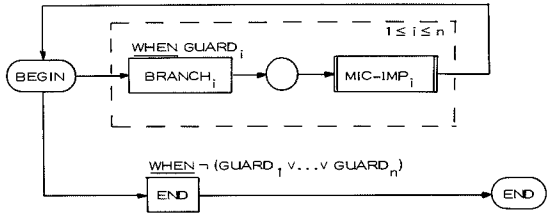


Repetition

The micro-imperative (defined in [Dijkstra 75])

"DO GUARD$_1$ → MIC-IMP$_1$
▯ GUARD$_2$ → MIC-IMP$_2$
⋮
▯ GUARD$_n$ → MIC-IMP$_n$ $(n \geq 1)$
OD"

is represented by



5. 4 Systems

The experience with Delta indicates that a global and continuous time concept is useful and simplifying in the description of a large number of systems, and we have therefore included the variable TIME in Epsilon.

The representation of a continuous time concept by Petri nets has no satisfactory solutions [Petri 76]. For this reason we have chosen not to incorporate incrementation of the variable TIME in the firing rules of concurrent systems, but to treat it separately.

Systems which are most adequately described without a concept of global time, e.g. computer networks and systems containing physical particles with a relativistic behaviour, are described in Epsilon by omitting any use of the global variable TIME.

A behaviour of a concurrent system without TIME is a firing sequence for it starting with the initial system state.

An Epsilon description without global TIME is well-behaved iff all of its behaviours are non-violating.

The behaviours of an Epsilon system with TIME are more complicated, since they are functions from an interval (consisting of the possible values of TIME) into firing sequences. We shall require that in a closed interval of TIME transitions have only concession for a finite number of values of TIME. Furthermore we require that only a finite number of firings take place at any value of TIME. This implies that at most values of TIME no transition has concession and the value of TIME is continuously increased, without changing the marking, until some transition gets concession. Then the value of TIME is kept constant until the concurrent system, by firing of transitions, has reached a system state, where no transition has concession. The value of TIME is again increased and so on. This informal description of behaviours for systems involving global time can be formalised in a way, which contains the corresponding definition for systems without global time as a special instance:

The following definitions are defined relatively to a concurrent system, where FS is the set of firing sequences, (m_1, s_1) the initial state and $t_0 = s_1(\text{TIME})$.

A behaviour of a concurrent system is a function $b: J \to FS$, where J is a closed interval of reals and there exists a finite sequence of reals, $t_0 \leq t_1 < t_2 < \ldots < t_{n-1} < t_n \leq t_{n+1}$ with $n \geq 0$, such that

1) $J = [t_0, t_{n+1}] \wedge (m_1, s_1) = b(t_0).\text{FIRST}$

2) $\forall\, t \in J\, [|b(t)| > 1 \Leftrightarrow t \in \{t_1, t_2, \ldots, t_n\}]$

3) $\forall\, i \in \{0, 1, \ldots, n\}\ \forall\, t \in]t_i, t_{i+1}[\ [b(t_i).\text{LAST} \approx b(t) \approx b(t_{i+1}).\text{FIRST}]$

4) $\forall\, t \in J \forall (m, s) \in b(t)\ [s(\text{TIME}) = t]$

5) $\forall\, t \in J - \{t_{n+1}\}\ [b(t) \text{ is maximal}]$

where $(m_1, s_1) \approx (m_2, s_2)$ iff

a) $m_1 = m_2$

b) $\forall\, v \in V\ [s_1(v) \neq s_2(v) \Rightarrow v \in U\ \{\text{VAR}(p) \mid m_1(p) = 1\} \cup \{\text{TIME}\}]$

$B(t)$ is the set of behaviours $b: [t_0, t] \to FS$. An Epsilon description with global TIME is well-behaved iff

$$\forall\, t \geq t_0\ [(B(t) \neq \emptyset) \wedge (\forall\, b \in B(t)\ [b(t) \text{ is finite and non-violating}])].$$

The definition of behaviour given above resembles in some repsects "iterated firing of occurrence" defined in [Moalla, Pulou & Sifakis 78].

6. CONCLUDING REMARKS

In the preceding sections we have presented the kernel of a system description language and a mathematical model used to give a formal definition of the semantics. The presence of the formal definition improves the useability of the language in the description, analysis, and design of systems:

The use as a descriptional tool is enhanced by providing a rigorous basis for an understanding of the language. This basis stresses the use of Epsilon as a tool for hierarchical system description. Furthermore it can be used to isolate the important aspects of given system descriptions and thus be a help in the difficult task of formulating an equivalence-concept for system descriptions.

The formal definition may be used in the analysis of systems to prove global properties of a given system description, e.g. that it is well-behaved. It is an important subject for future work, to formulate suitable global system properties and to develop formal methods for proving them by testing only local properties. This research can draw on a large body of related results in the field of Petri nets. In practical work with large and complicated systems this is most often the only manageable alternative to simulation on a computer system. For a discussion of how to obtain executable programs from system descriptions, see [Kyng 76].

In the design of systems, it will be possible to describe the anticipated design and to analyse it in order to check before implementation, whether it is consistent and has the desired properties. In this way the formulation of the mathematical model and the formal definition improved the design of Epsilon itself.

Related work

Other uses of Petri nets as a semantic model of languages may be found in [Lauer & Campbell 75] and [Pearl 78] where the semantics for path expressions and a process control language respectively are defined. An attempt to define a formal semantics of the control flow in Delta is reported in [Delta 79]. There the semantics of a large part of Delta is defined using a Petri net model, which is described and analysed in [Jensen 78]. Epsilon is designed using the large number of improvements and simplifications of Delta resulting from this work.

Extensions of Epsilon and the model

In order to get a complete system description language Epsilon has to be enlarged in several respects, e.g. structuring facilities, parameter mechanisms, and control structures. Among the obvious candidates for extensions are classes and subclasses with virtuals as known from Simula. The formal definition of these concepts by means of concurrent systems is straightforward. We do, however, also want to include recursive procedures (and interrupts), reference variables, and dynamic generation (and destruction) of objects, and this calls for an enlargement of the semantic model. In [Delta 79] we used infinite nets (but only finite markings).

At present we are considering a solution based on a model containing labelled tokens representing the identity of the different objects and the different procedure activations. This approach is inspired by [Genrich & Lautenbach 79].

Acknowledgements

We want to thank Antoni Mazurkiewicz and Kristen Nygaard for providing the initial inspiration for this work. Moreover we are grateful to Mogens Nielsen and Erik Meineche Schmidt for stimulating discussions and many helpful comments.

References

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

Delta, Jensen, K., Kyng, M. and Madsen, O.L.: Delta Semantics Defined by Petri Nets. DAIMI PB-95, March 1979, (Comp. Sci. Dept., Aarhus University).

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

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

Genrich, H.J. and Lautenbach, K.: The Analysis of Distributed Systems by Means of Predicate/Transition-Nets. Gesellschaft für Mathematik und Datenverarbeitung, Bonn, January 1979 (Draft version).

Jensen, K.: Extended and Hyper Petri Nets. DAIMI TR-5, August 1978.

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

Kotov, V.E.: An Algebra for Parallelism Based on Petri Nets. Mathematical Foundations of Computer Science 1978, J. Winkowski (ed.), Springer Verlag (1978), 39-55.

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

Lauer, P.E. and Campbell, R.H.: Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes. Acta Informatica 5 (1975), 297-332.

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

Moalla, M., Pulou, J. and Sifakis, J.: Synchronized Petri Nets: A Model for the Description of Non-autonomous Systems. Mathematical Foundations of Computer Science 1978, J. Winkowski (ed.),Springer-Verlag (1978), 374-384.

Nygaard, K.: System Description by Simula - An Introduction. Norwegian Computing Center, Oslo, 1970.

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

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

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

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

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

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

Thiagarajan, P.S. and Genrich, H.J.: Assignment Systems - A Model for Asynchronous Computations. Interner Bericht 76-10, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, November 1976.