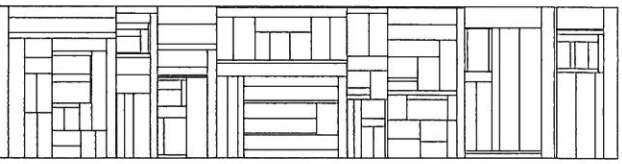
ISSN 0105-8517

The BETA Programming Language

Bent Bruun Kristensen
Ole Lehrmann Madsen
Birger Møller-Pedersen
Kristen Nygaard

DAIMI PB – 229
November 1987

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88 *Telex: 64767 aausci dk*



THE BETA PROGRAMMING LANGUAGE[1]

Part 1: Abstraction Mechanisms
Part 2: Multi-Sequential Execution

Bent Bruun Kristensen, Aalborg University Centre
Ole Lehrmann Madsen, Aarhus University
Birger Møller-Pedersen, Norwegian Computing Center
Kristen Nygaard, University of Oslo

Abstract. The BETA programming language is a modern language in the SIMULA 67 tradition. It supports the object-oriented perspective on programming and contains comprehensive facilities for procedural and functional programming. BETA replaces classes, procedures, functions and types by a single abstraction mechanism called the *pattern*. Patterns may be organized in a classification hierarchy by means of *sub-patterns*. The notion of virtual procedure is generalized to virtual pattern. Virtual patterns combined with sub-patterns make it possible to delay the specification of an attribute in a pattern. Attributes may then have different bindings in different sub-patterns. BETA also provides a unified framework for sequential, coroutine and concurrent execution. This paper is a tutorial introduction to BETA.

[1]To appear in: *Research Directions in Object Oriented Programming.* Edited by B. D. Shriver and P. Wegner, MIT Press, 1987.

Part 1: ABSTRACTION MECHANISMS

1 Introduction

BETA is a modern language in the SIMULA 67 ([SIMULA]) tradition. It supports the object oriented perspective on programming and contains comprehensive facilities for procedural and functional programming. Research is going on with the aim of including constraint oriented constructs. BETA replaces classes, procedures, functions and types by a single abstraction mechanism called the *pattern*. It generalizes virtual procedures to virtual patterns, streamlines linguistic notions such as nesting and block structure, and provides a unified framework for sequential, coroutine, and concurrent execution. The resulting language is smaller than SIMULA in spite of being considerably more expressive.

Instances of patterns, called *objects*, may be used as variables, data structures, procedure/function activations, coroutines and concurrent systems. Patterns may be organized in a classification hierarchy by means of subpatterns. Virtual patterns combined with sub-patterns make it possible to delay the specification of an attribute in a pattern. Attributes may then have different bindings in different sub-patterns. This corresponds to dynamic binding of methods in Smalltalk ([SMALLTALK]). In contrast to Smalltalk, the use of virtual patterns may be checked at compile time, even though the binding is done at run-time.

The first part of this paper describes the abstraction mechanisms and the mechanisms for sequential execution. The second part describes the mechanisms for supporting multiple action sequences, including alternation (generalized coroutine sequencing) and concurrency. Part one is a revised version of [BETA 83b]. Part two is a revised version of [BETA 85a]. The appendix contains a summary of the BETA syntax.

2 Basic Constructs

2.1 Objects and Patterns

A BETA program execution is a collection of *objects*. In a BETA program objects are described by *object-descriptors* having the following form:

```
(#                  {begin object description}
  D1; D2; ...Dn     {list of declarations}
enter In            {input parameters}
do Imp              {imperatives to be executed}
exit Out            {output parameters}
#)                  {end object description}
```

D_1, D_2, ..., D_n are declarations of attributes. *In* is a list of input parameters. *Imp* is an imperative that describes the actions to be performed when the object is executed as a procedure, coroutine or process. *Out* is a list of output parameters produced as a result of object execution. The enter-part corresponds to value-parameters and the exit-part to result-parameters.

An object descriptor may be used to describe a *pattern* of objects. A pattern consists of a named object descriptor:

```
C: (# D1; D2; ...Dn
   enter In
   do Imp
   exit Out
   #)
```

Patterns serve as templates for generating objects (instances). The objects generated as instances of C will all have the same *structure*. That is, the same set of declarations, the same enter-part, the same do-part and the same exit-part.

Declarations serve to describe attributes which may be objects and/or patterns. For example, objects that represent points (e.g. on a screen) have two *Integer* attributes.

Point: (# *x*, *y*: @*Integer*#)

The above pattern for *Point* has empty enter-, do-, and exit-parts and a single declaration that uses the predefined pattern *Integer*. Given this pattern, *Point*-objects may be declared by

*P*1, *P*2: @*Point*

If *Point* objects are to be moved around, this may be described by adding the pattern attribute *Move* within the pattern *Point*:

```
Point:
(# x, y: @Integer;  {two object attributes}

   Move:  {a pattern attribute}
   (# dx, dy: @Integer
   enter (dx, dy)
   do x + dx → x;
      y + dy → y
   #)
#)
```

A *Point* object *P*1 may then be moved by executing an instance of *P*1's *Move*- attribute:

(11,22) → *P*1.*Move* {Execute *P1.Move* with parameters (11,22)}

The pattern *Point* has two object attributes and a pattern attribute, but no enter-, do-, or exit-parts. Pattern attributes of a pattern correspond to interface operations in Smalltalk classes. Object attributes correspond to instance variables. The do-part of an object has no counterpart in Smalltalk and is used when the object is executed as a procedure, coroutine or process. The do-part is invoked by invoking a pattern name as in *P1.Move* which invokes the do-part of the *Move*-pattern in *P1* as a procedure. *P1.Move* describes that a temporary instance of the *Move*-pattern will be created and causes the do-part of this instance to be executed.

The do-part of an object represents a very important extension of the notion of objects that allows patterns to be executed as procedures and

to be used in modelling ongoing processes and in system simulation.

2.2 Evaluation

The basic mechanism for specifying sequences of object execution steps is called an *evaluation.* An evaluation is an imperative that may cause changes in state and produce a value when its executed. The notion of an evaluation provides a unified approach to assignment, function-call and procedure-call.

Examples of evaluations are

$$(11,22) \rightarrow P1.Move$$

$$x + dx \rightarrow x$$

The evaluation

$$x + dx \rightarrow x$$

specifies an ordinary assignment (the assignment of $x + dx$ to x). An evaluation may specify *multiple assignment* as in:

$$3 \rightarrow I \rightarrow J$$

where 3 is assigned to I and the value of I is assigned to J.

The evaluation

$$(11,22) \rightarrow P1.Move$$

specifies a procedure-call. The value $(11, 22)$ is assigned to the enter-part of $P1.Move$ and $P1.Move$ is then executed with these enter parameters. Note that the pattern $P1.Move$ is invoked as an instance.

The general form of an evaluation is:

$$E_1 \rightarrow E_2 \rightarrow ... \rightarrow E_n$$

where $n > 0$. Each E_i is either an object denotation (object or pattern)

or an evaluation list $(F_1, F_2, ..., F_m)$ where each F_j is an evaluation. The execution of an evaluation takes place as follows:

- E_1 is executed,
- a *value transfer* from E_1 to E_2 is carried out,
- E_2 is executed,
- ...
- E_{n-1} is executed
- a *value transfer* from E_{n-1} to E_n is carried out,
- E_n is executed.

Execution of E_i means executing the do-part of E_i, if E_i is an object-denotation. If E_i is an evaluation list then it is the empty action. A value-transfer from E_i to E_{i+1} means assignment of the elements of the exit-list of E_i to corresponding elements of the enter-list of E_{i+1}. If one or both of E_i, E_{i+1} are evaluation-lists, then these lists take the place of enter/-exit-lists. A value-transfer from E_i to E_{i+1} is legal if the corresponding elements of their enter/exit-list are assignable. For the predefined patterns, we have that Integer-objects are assignable to Integer-objects etc. Note that the recursive definition of assignment means that the do-part of objects being assigned during a value-transfer are also executed.

A BETA pattern is among other things a unification of procedures and functions. In figure 1 an example of a BETA program that demonstrates the use of patterns in a procedure/function like manner is given:

The program contains the declaration of two patterns: *Power* and *Reciprok*, and two objects of the pattern *Real*: A and B. The do-part of the program consists of the *evaluation-imperative*

$$(3.14, 2) \rightarrow Power \rightarrow Reciprok \rightarrow (A, B)$$

The execution of this evaluation-imperative takes place as follows: The values 3.14 and 2 are assigned to the input parameters X, n of *Power* (described by **enter** (X, n)), the do-part of *Power* is executed, the output

```
(#
   Power:  {Compute X^n where n > 0}
   (# X, Y: @Real; n: @Integer;
   enter (X, n)
   do 1 → Y;
      (for inx: n repeat Y * X → Y for)
   exit Y
   #);

   Reciproc:  {Compute (Q, 1/Q)}
   (# Q, R: @Real;
   enter Q
   do (if (Q = 0)
       // True then 0 → R
       // False then 1/ Q → R
       if)
   exit (Q, R)
   #);

   A, B: @Real;
do (3.14, 2) → Power → Reciproc → (A, B)
    { A = 3.14^2, B = 3.14^-2}
#)
```

Figure 1: Example of using patterns as procedures/functions

parameter Y of *Power* (described by **exit** Y) is assigned to the input parameter Q of *Reciproc*, the do-part of *Reciproc* is executed and finally the output-parameters Q, R of *Reciproc* are assigned to A,B. $\{\ldots\}$ is a comment.

The do-part of *Power* consists of two imperatives: an evaluation-imperative assigning Y the value 1 and a for-imperative. The index-variable, *inx*, steps through the values 1,2,...,n. The do-part of pattern *Reciproc* consists of an if-imperative.

BETA will contain a number of predefined patterns for commonly used data types such as *Integer, Boolean, Char* and *Real* and their operations. The patterns +, −, ∗, etc. will denote the usual operations on integers. Similarly for the other predefined patterns. For example the following declaration declare 3 *Integer*-objects

I, J, K: @*Integer*

The standard infix notation for *Integer* expressions can be used:

$1 + I \rightarrow I;\ (I * J) + 12 \rightarrow K$

It corresponds to the following evaluation using function calls:

$(1, I) \rightarrow + \rightarrow I;\ ((I, J) \rightarrow *, 12) \rightarrow + \rightarrow K$

2.3 Control Structures

The iteration control structure of BETA is called a *for-imperative* and has the following form:

(**for** *Index*: *Range* **repeat** *Imperative-list* **for**)

where *Index* is the name of an *Integer*-object and *Range* is an *Integer*-evaluation. *Range* is evaluated prior to the execution of the for-imperative and determines the number of times that *Imperative-list* is executed. *Index* will step through the values 1, 2, ..., *Range*. The scope of *Index* is the *Imperative-list*. *Index* cannot be assigned to.

The selection control structure is called an *if-imperative* and has the following form:

(if E_0
// E_1 **then** I_1
// E_2 **then** I_2
...
// E_n **then** I_n
if)

where E_0, E_1, E_2, ..., E_n are evaluations. E_0 is first evaluated, and that value is tested for equality with E_1, E_2, ..., E_n in an arbitrary order. If $E_0 = E_j$ then I_j may be selected for execution. If one or more alternatives are selectable, then one of these is chosen randomly. If no alternative may be selected, the execution continues after the if-imperative.

The *equality test* between two evaluations E' and $E"$ is carried out as follows:

- Let E' and $E"$ be "primitive" objects, that is instances of *Integer, Boolean, Char* and *Real.* Here the usual test for equality of values is used.
- Let E' and $E"$ be two "compound" objects. The do-parts of E' and $E"$ are executed. Then the exit-parts of E' and $E"$ are tested for equality. This implies that user defined equality is supported.
- Let E' and $E"$ be two evaluation lists. Corresponding elements of the lists are tested for equality.
- Let E' have the form $A_1 \rightarrow ... \rightarrow A_n$ and let $E"$ have the form $B_1 \rightarrow ... \rightarrow B_m$. E' and $E"$ are executed as described in the previous section. In addition A_n and B_m are tested for equality. The possible do-parts of A_n and B_m are only executed one time.

An imperative in the do-part of an objetct descriptor may be *labelled*

L: Imperative

where L is a name. The scope of the label is the *Imperative* to which the label is attached. That is, L may only be referred to within the *Imperative.*

The execution of a labelled imperative may be terminated by executing a *leave-* or *restart-imperative* within it. If **leave** L is executed, the execution continues after the imperative labelled by L. If **restart** L is executed, the execution continues at the imperative labelled by L, i.e. the execution of the labelled imperative is repeated.

Consider the following example:

```
L: (if ...
   // ... then
      M: {2}
         (for ... repeat
              (if ...
               // ... then leave L
               // ... then restart M
          if)for)
   // ... then ...
   if) {1}
```

An execution of **leave** L implies that execution continues at {1}. An execution of **restart** M implies that execution continues at {2}.

Figure 2 describes the pattern *Register*, which is similar to Hoares *Small-IntSet*. The pattern *Register* describes a category of objects. Each *Register*-object consists of the attributes *Table, Top, Has, Insert* and *Remove*. *Table* is an consisting of 100 *Integer*-objects. The action part of a *Register*-object is the single evaluation $0 \rightarrow Top$. *Has*-objects consist of the attributes *Key* and *Result*.

The do-part of the program consists of an execution of the evaluation *R*, which has the effect of initializing the *Register*-object *R*. Then the element 5 is inserted into *R*. Attributes in objects are denoted by a dot: *object.attribute*. Thus *R.Insert* denotes the *Insert* attribute of *R*. Finally it is tested if 5 is a member of *R*.

2.4 Object Kinds and Construction Modes

BETA has three *kinds* of objects: *system*, *component* and *item*. The kind of an object specifies how the objet can be used.

```
(# Register:
   (# Table: [100]@Integer; Top: @Integer;

      Has:  {Test if Key in Table[1 : Top]}
      (# Key: @Integer; Result: @Boolean;
      enter (Key)
      do False → Result;
         Search:
         (for inx: Top repeat
               (if ((Table[inx] = Key) → Result)
                // True then leave Search
          if)for)
      exit (Result)
      #);

      Insert:  {Insert New in Table}
      (# New: @Integer;
      enter (New)
      do (if ((New) → Has)  {Check if New is in Table}
          // False then   {New is not in Table}
             Top + 1 → Top;
             (if (Top ≤ Table.Range)  {Table.Range = 100}
              // True then New → Table[Top]
              // False then   {Overflow}
       if)if) #);

      Remove:  {Remove Key from Table}
      (# Key: @Integer;
      enter (Key)
      do Search:
         (for inx: Top repeat
               (if Table[inx] // Key then
                  (for i: Top − inx repeat
                        Table[inx + i] → Table[inx + i − 1];
                   for);
                  Top − 1 → Top;
                  leave Search
          if)for) #);
   do 0 → Top;
   #);  {end Register}

   R: @Register;  { declaration of a Register-object}
do R;  {initialize R}
   (5) → R.Insert;  {insert 5 into R}
   (if ((5) → R.Has) // True then ...  {5 is in R} if)
#)
```


Figure 2: Hoares SmallIntSet

- A system object may be executed concurrently with other system objects.
- A component object (coroutine) may alternate execution with other components.
- An item object is a dependent action sequence contained in a system, component or item.

In this part of the paper objects of kind item have been described. Objects of kind component or system will be described in part 2.

Item objects may be created statically by declaration, inserted inline in the action-part, or created dynamically by a "new"-imperative. These three different ways of creating objects are called *construction modes.* The construction mode gives rise to three sorts of items respectively called static item, inserted item and dynamic item. The notions of kind and construction mode are orthogonal since objects of kind component or system may also be created either statically, inserted or dynamically.

2.4.1 Static Items

A pattern may be used to declare an item in the following way:

E: @P

E is *static reference* denoting an object (called a *static item*) generated according to the pattern P. Such a static item is an integral part of a surrounding object of which it is an attribute. The declaration

$P1, P2$: @$Point$

specifies two static references *P1, P2.*

A declaration may specify a sequence (array) of static items, called an *item-repetition*:

X: [100] @P

X consists of the P-items $X[1]$, $X[2]$, ..., $X[100]$.

2.4.2 Inserted Items

Consider the pattern C in the following evaluation:

$$E \rightarrow C \rightarrow A$$

Its execution assigns the output of E to the enter-part of an instance of C and causes the instance to be executed. Finally the exit-part of the C-instance is assigned to A.

The C-object (called an *inserted item*) will be an integral part of the surrounding object. This inserted item will the be executed when control reaches the evaluation imperative. The state of this C-item will be undefined prior to each execution of it. The notion of inserted item is similar to an in-line procedure call.

The evaluations

$$(11,22) \rightarrow P1.Move$$
$$(3.14,2) \rightarrow Power \rightarrow Reciproc \rightarrow (A,B)$$

specifies an inserted item for each of the patterns *P1.Move, Reciproc* and *Power.*

With this semantics of inserted items, it may be seen that inserted items cannot be used to describe recursive procedures since this will lead to an infinite recursion. This is analogous to static items which cannot be used for describing recursive data structures. Below the notion of dynamic items will be introduced. Dynamic items may be used for describing recursive data structures and recursive procedures.

2.4.3 Dynamic Items

The static and inserted items described above are static in the sense that they are generated at the same time as and as a permanent, inseparable part of the object in which they are declared. Storage requirements for static and inserted items may be computed at compile time. It is possible to generate items dynamically during a program execution. The follow-

ing declaration specify attributes, which are *dynamic references* to such items:

X: ↑ P

P is the qualification of the references. It specifies that X may denote (refer to) a P-item, a sub-item of P (see section 3) or NONE, which means no item. A dynamic reference is similar to a reference in SIMULA.

A dynamic P-item may be generated by execution of a "new"-imperative $\&P$. An object generated in this way is called a *dynamic item.*

A dynamic reference may be given a value in the following way:

$\&P\square \rightarrow X\square$; {a P-item is generated and its reference is assigned to X}

This corresponds to $X \leftarrow P$ *New* in Smalltalk.

$\&P\square$ specifies that a reference to the newly created item is returned. In contrast, an evaluation with no box, like $\&P$, describes that the item is executed.

An evaluation of the form $\&P$ (i.e. without a box) is similar to a procedure call in ALGOL 60 in the following sense: Each execution of $\&P$ implies creation of an instance of P and a subsequent execution of this instance.

In figure 3 an example of using dynamic items is given. One pattern describes a recursive function for computing factorial. Another pattern describes a linked list of *Integers.*

References may be compared in an if-imperative of the following form:

```
(if X□
// Y□ then  {X and Y refer to the same item}....
// NONE then  {X refers to no item}...
if)
```

2.4.4 Summary of Construction Modes

To sum up, objects of kind item may be generated in three different ways:

```
(#
   Factorial:
   (# N, fac: @Integer
   enter N
   do (if (N ≤ 1)
      // True then 1 → fac
      // False then ((N − 1) → &Factorial) * N → fac
      if)
   exit fac
   #)

   Link: {Link describes a linked list}
   (# succ: ↑ Link; {tail of this Link}
      elm: @Integer {head element of this Link}

      Insert: {Insert an element before this Link}
      (# E: @Integer; R: ↑ Link
      enter E
      do &Link□ → R□; {R denotes a new instance of Link}
         E → R.elm; {E = R.elm}
         succ□ → R.succ□; {tail of this Link = tail of R}
         R□ → succ□ {R = tail of this Link}
      #)
   #)

   head: @Link; {A static Link item}
do 0 → head.elm;
   (for inx: 4 repeat
        inx → &Factorial → head.Insert
    for);
    {head = (0 24 6 2 1)}
#)
```

Figure 3: Example of a recursive procedure and a recursive data structure

```
(#
   P:
   (# I, J: @Integer
   enter (I, J)
   do I + J → I
   exit (I, J)
   #);

   E: @P;  {declaration of a static item}
   X: ↑ P;  {declaration of reference to a dynamic item}
   N, M: @Integer;
do  {generation of a dynamic P-item denoted by X}
   &P□ → X□;

    {an evaluation involving static, inserted and dynamic items}
   (3, 4) → E → P → E → &P → X → P → (N, M);
#)
```

Figure 4: Example of dynamic items

- As a statically allocated attribute, called a static item.
- As a statically allocated action, called an inserted item.
- As a dynamically allocated attribute or action, called a dynamic item.

We have mentioned the similarities between inserted items and inline procedure calls and between dynamic items and ALGOL-like procedure activation records. We also note that a static item may be used as a static subroutine.

In figure 4 examples are given of the three different construction modes. The last evaluation involves two executions of the static item E, execution of two different inserted items (specified by the two Ps), and execution of two different dynamic items (the one denoted by X and an anonymous one generated during the evaluation by $\&P$).

```
(# aPoint: @(# X,Y: @Real#);  {a singular static item}
do 5.0 → aPoint.X; 3.0 → aPoint.Y;

    {a singular inserted item modelling an ALGOL-like inner block}
   (# Z: @Real
   do  {switch the coordinates}
      aPoint.X → Z; aPoint.Y → aPoint.X;
      Z → aPoint.Y
   #);

#)
```

Figure 5: Example of singular items

2.5 Singular Objects

When describing an object it is possible to describe its properties directly without referring to a pattern. This corresponds to specifying the type together with a variable in PASCAL ([PASCAL]) and to inner blocks in ALGOL 60 and SIMULA and the "prefixed blocks" of SIMULA. An object being described directly is called a *singular object.* An object described as an instance of a pattern is called a *pattern defined object.* Figure 5 is an example of a BETA program with singular objects.

2.6 Block Structure and Scope Rules

BETA is a language belonging to the ALGOL family with respect to block structure, scope rules and type checking. In ALGOL and SIMULA a procedure or block may have local procedures and/or blocks. In BETA object descriptiors may be textually nested. Block structure is an important mechanism for structuring individual components of a large program. Block structure and sub-patterns are complementary mechanisms for structuring objects and patterns. For more examples of using block structure in BETA see [Madsen 86].

For large programs it is necessary to have a mechanism for separating the program into (separately translatable) modules. It is well know that such

a mechanism for "programming in the large" ([DeRemer and Krohn 76]) cannot be handled with block structure, which is primarily useful for "programming in the small". BETA has no such modularization mechanism. Mechanisms for organization of large programs are considered to be orthogonal to the BETA language. The reason is that how a program is combined from various modules from libraries, other programmers etc. has nothing to do with the *intention* of the program. The mechanism for program modularization is described in [BETA 83a]. The idea is that any sentential form (string of terminal and nonterminal symbols) generated from a nonterminal of the BETA grammar may be a module. This can be applied to any language where the syntax can be described by a context free grammar. The modularization mechanism also supports the separation of a BETA program into interface modules and implementation modules.

With respect to scope rules, BETA also follows the ALGOL tradition, since all names in textually enclosing object descriptors are visible. In addition to the ALGOL scope rules most languages supporting classes have a rule that protects certain attributes of an object from being accessed remotely. In SIMULA this is handled by the hidden/protected mechanism. In Smalltalk instance variables cannot be accessed from outside the object. BETA contains no such protection mechanism. This is handled by the modularization mechanism mentioned above.

3 Classification Hierarchies

Patterns may be organized in a classification hierarchy. An object description may include a *super-pattern* (often called *prefix-pattern* or simply *prefix*). This specifies that objects generated according to the description have all the properties described by the super-pattern. A pattern having a super-pattern is described in the following way:

$C1$: C
($\# D'_1; D'_2; ... D'_m$
enter In'
do Imp'
exit Out'
$\#$);

where C is the super-pattern of $C1$. $C1$ is also said to be a *sub-pattern* of C. Any $C1$-object will then have the same properties as C-objects in addition to those specified between ($\#$ ... $\#$), called the *main-part* of $C1$.

A $C1$-object will have attributes corresponding to the declarations D_1, ..., D_n and D'_1, ..., D'_m. The enter-part of a $C1$-object is a concatenation of In and In'. The exit-part of a $C1$-object is a concatenation of Out and Out'.

The action part of a $C1$-object is combination of Imp and Imp'. This combination is described by means of the **inner**-imperative: The execution of a $C1$-object starts by execution of the imperative Imp in the C. Each execution of an **inner**-imperative during the execution of Imp implies an execution of Imp'.

Consider the example in figure 6. Instances of $C1$ have the attributes a, b, c. Execution of a $C1$-instance implies execution of $11 \rightarrow a$, $22 \rightarrow c$ and $33 \rightarrow b$.

Execution of **inner** in the main-part of an object is the empty action. For example executing an instance of the C-pattern in figure 6 will result in execution of $11 \rightarrow a$, and $33 \rightarrow b$. That is, execution of **inner** is the empty action.

```
C:
(# a, b: @Integer
do 11 → a;
   inner
   33 → b
#);

C1: C(# c: @Integer do 22 → c#)
```

Figure 6: Example of a pattern and a sub-pattern

Record: (# *Key*: @*Integer*#);

Person: *Record*(# *Name*: @*String*; *Sex*: @*SexType*#);

Employee: *Person*(# *Salary*: @*Integer*; *Position*: @*PositionType*#);

Student: *Person*(# *Status*: @*StatusType*#);

Book: *Record*(# *Author*: @*Person*; *Title*: @*TitleType*#);

Figure 7: Example of sub-patterns

In figure 7 is given some pattern declarations illustrating sub-patterns. All *Record-* , *Person-*, *Employee-*, *Student-*, and *Book*-objects may be viewed as *Record*-objects and they all posses the attribute *Key*. Similarly *Person-*, *Employee-*, *and Student*-objects may be viewed as *Person*-objects and they all posses the attributes *Key*, *Name* and *Sex*.

The example in figure 8 illustrates the combination of action-parts. The example contains two patterns and two evaluations. The pattern *Cycle*[2] repeatedly executes an inner-imperative and may be used as a cycle control structure. The pattern *CountCycle* is prefixed by *Cycle*. This has the effect that when executing an instance of *CountCycle*, its do-part will also be repeatedly executed. The first evaluation illustrates the use of *CountCycle* as a control structure. The effect of this is shown in the second evaluation.

[2]A construct of the form (L: $Imp_1; Imp_2; \ldots; Imp_n$: L) is a labelled compound imperative

```
Cycle: (# do (Loop:  inner; restart Loop : Loop)#);


CountCycle: Cycle
(# inx: @Integer
enter (inx)
do inner;
   inx + 1 → inx;
#)


L: (1) →
      CountCycle
      (# F: @Integer
      do (if inx // 10 then leave L if);
         inx → &Factorial → F;
          {Factorial is computed for inx in [1, 9]}
      #)


L: (1) →
         (# inx: @Integer;
            F: @Integer
         enter (inx)
         do
            (Loop:
                  (if inx // 10 then leave L if);
                  inx → &Factorial → F;
                  inx + 1 → inx;
                  restart Loop
             : Loop)
         #)#)
```

Figure 8: Example of combination of action-parts

```
ForAll:
(# Current: @Integer;
do (for inx: Top repeat
        Table[inx] → Current;
        inner
 for) #)
```

Figure 9: Iterator on the *Register*-pattern

Figure 9 describes a pattern *ForAll* which may be added as an attribute to the *Register*-pattern. It defines a control abstraction that makes it possible to step through all the elements of a *Register* and perform an operation upon each element.

This operation may be used in the following way:

$$R.ForAll(\#\ \textbf{do}\ Current \rightarrow DoSomething\#)$$

Current will then step through the values of R and each value will one by one be assigned to *DoSomething*. It may be inconvenient that the name of the index variable always has to be *Current*. This may changed by adding the following attribute to *ForAll*:

$$Index : (\#\ \textbf{exit}\ Current\#)$$

ForAll may now be used as follows:

$$R.ForAll(\#\ I:\ @Index\ \ \textbf{do}\ I \rightarrow DoSomething\#)$$

The idea of combining action-parts by means of **inner** originates from SIMULA where it is used for prefixing of classes. In [Vaucher 75] it is proposed to extend this idea for prefixing of procedures.

One of the differences between SIMULA/BETA and Smalltalk is the question of "typing" of variables. In Smalltalk variables have no type and may refer to any object. In SIMULA/BETA references are *qualified* by means of a pattern name. The qualification specifies that the reference may refer only to objects that have been specifed by means of that pattern. The

notion of qualification is defined as follows:

A pattern $C1$ is said to be *qualified* by a pattern A if

- A is $C1$ or
- the super-pattern C of $C1$ is qualified by A.

Similarly an object is qualified by a pattern A if it is an instance of A or if a possible super-pattern used in the description of the object is qualified by A.

Consider

C: (# ...#); R: ↑ C;
$C1$: C(# ...#); $R1$: ↑ $C1$;
$C2$: $C1$(# ...#); $R2$: ↑ $C2$;

Here $C2$ is qualified by $C2$, $C1$ and C. Furthermore $C1$ is qualified by $C1$ and C. Finally C is qualified by C. The reference R may refer instances of C, $C1$ and $C2$. The reference $R1$ may refer instances of $C1$ and $C2$. Finally $R2$ may refer instances of $C2$ only.

```
KeyMax:
(# Rec :< Record;
   M: @Rec;
enter (M)
do
   (if (M.Key > MaxKey) // True then
      M.Key → MaxKey;
      inner
if) #)
```

Figure 10: Example of a virtual pattern

4 Virtual patterns

The virtual concept was introduced in SIMULA, as was the subclass mechanism. In BETA the virtual concept is generalized and is through the pattern concept available for all kinds of objects, not only procedure activation records.

A pattern attribute may be declared as virtual. This implies that only a part of the properties of the virtual pattern is known and that the full specification of it may be deferred. The known properties will correspond to a prefix of the virtual pattern. This prefix is specified in the declaration of the virtual pattern. A virtual pattern may thus be viewed as a "pattern parameter" of the surrounding pattern. A virtual pattern declaration has the following form:

$$V :< A$$

V is declared as a virtual pattern with *qualifying pattern* A. The pattern V may be *bound* to any sub-pattern of A. In case that the virtual pattern is not bound then it has a *default-binding*, which is its qualifying pattern.

The *KeyMax* pattern in figure 10 has a virtual pattern attribute. The pattern *Rec* of *KeyMax*-objects is virtual and known to be prefixed by *Record* (defined in section 3). Any *Rec*-object in *KeyMax* is therefore known to have all the properties of *Record*-objects. E.g. *M.Key* is legal since any *Record*-object has a *Key* attribute. The effect is that *Rec* may be regarded and used as as "pattern-parameter" of *KeyMax*. *Rec* may in

```
PersonCount: KeyMax
(# Rec :: Person;
do
   (if M.Sex
    // Male then MaleCount + 1 → MaleCount
    // Female then FemaleCount + 1 → FemaleCount
 if) #);

P: @PersonCount
```

Figure 11: Example with a virtual binding

sub-patterns of *KeyMax* be bound to sub-patterns of *Record.*

The virtual pattern attributes of a pattern P may be bound in sub-patterns of P. A binding of a virtual pattern may have the form of a *final binding*

```
either
    V :: A1
or
    V :: A1(# ... #)
```

Assume that this declaration appears in a descriptor that is prefixed by pattern P. $V1$ must then be a virtual pattern attribute of P. $A1$ must be a pattern that is prefixed by the qualifying title (A) of $V1$ (since V is declared by $V :< A$). The binding has the effect that the pattern $V1$ is identical to $A1$ (or $A1$(# ... #)). Objects generated according to $V1$ will be $A1$-objects (or $A1$(# ... #)-objects), *even if the generation is specified in the prefix.*

Consider figure 11 which is a continuation of the previous example. The bind-construct in *PersonCount* refers to the virtual pattern *Rec* in the super-pattern *KeyMax.* The qualifying pattern of *Rec* is *Record* and *Rec* is bound to the pattern *Person.* This means that *Rec* is an ordinary pattern attribute in all *PersonCount*-objects, i.e. any *Rec*-object in a *PersonCount*-object is known to be *Person*-object. In the specification of the super-pattern *KeyMax*, however, only properties of *Rec* specified in its *Record*-prefix may be assumed. The instance P of *PersonCount* has an object attribute M which is an instance of *Person* since *Rec* is bound

```
Find:
(# Key,index: @Integer; NotFound :< Object;
enter (Key)
do 1 → index;
   Search:
   (if (index ≤ Top)
   // True then
      (if Table[index] // Key then
          inner;
          leave Search
       if);
      index + 1 → index;
      restart Search
   // False then NotFound;
 if) #);
```

Figure 12: Example of general control abstraction

```
Has: Find
(# Result: @Boolean;
   NotFound :: (# do False → Result#);
do True → Result
exit (Result)
#);
```

Figure 13: Example of a specialization of *Find*

to *Person*.

In figure 12 is given an example of a control abstraction using a virtual pattern. This *Find* pattern may be added as an attribute to pattern *Register*. Find will search for an element identical to *Key*. If such an element is found an inner will be executed. Otherwise the item specified according to the virtual pattern *NotFound* will be executed[3].

Find may be used to implement pattern *Has* as shown in figure 13.

In figure 14 *Register* has been revised by adding two virtual pattern at-

[3]The predefined pattern *Object* used in *"NotFound :< Object" is assumed to prefix any pattern*, thus *NotFound* may be bound to any pattern. (On the other hand *"Object"* has no attributes and no actions, so no information is given about a virtual qualified by *"Object"*.)

tributes: *Content* and *Overflow*. Pattern *Register* may then be used to define specialized *Registers* consisting of elements prefixed by *Record*.

Note that pattern *Has* uses the specification that *Content*-objects have a *Key* attribute. A *Student* register may be declared as in figure 15.

Suppose that the pattern *Record* in addition to *Key* has a pattern attribute *Display*, to display the value of *Key* in some form. Within *Register* it would then be possible to have the register displayed by

(**for** *inx*: *Top* **repeat** *Table*[*inx*].*Display* **for**)

This will, however, only display the *Key*, even if the *Register* is a *Person* or *Book* register. Instead the *Display* attribute may be declared as a virtual pattern as in figure 16. Then the for-imperative above will have the effect that the actual binding of *Display* is executed. A binding of *Display* in *Person* is shown in figure 17.

This binding of *Display* is *final* in the sense that *Display* is no longer virtual. It would be desirable to be able to extend the specification of *Display* in *Student* and *Employee* to get such objects displayed too. This is possible by using the *further binding* construct

$V ::< A1$

The rules for using a further binding are the same as for using a final binding. In a further binding the specification of the virtual pattern V is extended to be a virtual pattern qualified by $A1$. V is thus still a virtual pattern and may be further specified in sub-patterns of the surrounding pattern.

Display may then be specified as shown in figure 18.

Now *Display* is bound to *PersDisp*, but it is still a virtual pattern. A new further binding may then be added in *Student* and *Employee*.

The use of patterns and sub-patterns makes it possible to construct a broad variety of abstractions. There are however certain limitations and disadvantages of this approach: There is an assymmetry between the use of inner and virtual patterns to construct control abstractions. Note

```
Register:
(# Content :< Record; Overflow :< Exception;
   Table: [100]@Content; Top: @Integer;

   Has:
   (# Subject: @Content; Result: @Boolean;
   enter (Subject)
   do False → Result;
      Search:
      (for inx: Top repeat
            (if ((Table[inx].Key = Subject.Key) → Result)
             // True then leave Search
       if)for)
   exit (Result)
   #);

   Insert:
   (# New: @Content;
   enter (New)
   do (if ((New) → Has) // False then
         Top + 1 → Top;
         (if (Top ≤ Table.range)
          // True then New → Table[Top]
          // False then Overflow
    if)if) #);

   Remove: ...;
   ForAll: ...;
   Find: ...;
do 0 → Top;
#);
```

Figure 14: The *Register* pattern parameterized with virtual patterns

```
StudentReg: Register
(# Content :: Student;
   Overflow :: Exception(# ...#);

   UpdateStatus: Find
                 (# Status: @StatusType;
                    NotFound :: (# ...#)
                 enter (Status)
                 do Status → Table[index].Status
                 #)
#)
```

Figure 15: Example of a specialization of the *Register* pattern

```
Record:
(# Key: @Integer;
   RecDisp: (# do {display the Key value} inner #);
   Display :< RecDisp
#)
```

Figure 16: Virtual *Display* attribute

```
Person: Record
(# Name: @String; Sex: @SexType;
   Display :: RecDisp(# do {display Name and Sex} #)
#)
```

Figure 17: Binding of the virtual *Display* attribute

```
Person: Record
(# Name: @String; Sex: @SexType;
   PersDisp: RecDisp(# ...#);
   Display ::< PersDisp;
#)
```

Figure 18: Further binding of the virtual *display* attribute

the pattern *Find* where if the desired element is found, inner is executed, otherwise the virtual pattern *NotFound* is executed. In section 5 of [BETA 83b] a generalization of virtual patterns is proposed in order to deal with these problems. In [BETA 87] the notion of virtual patterns is further exploited.

Part 2: MULTI-SEQUENTIAL EXECUTION

The BETA programming language supports a number of different organizations of action sequences. These are: — sequential action-sequences, where procedures are executed sequentially — *alternation* which is a generalization of coroutine sequencing as found in SIMULA 67 — *concurrency* which is found in languages like CONCURRENT PASCAL, CSP and Ada. In this part of the paper the constructs for multi-sequential execution, that is alternation and concurrency will be described. Communication between concurrent components takes place by synchronized execution of items. This is similar to a rendezvous in Ada. BETA has no constructs for guarded input/output as do CSP and Ada. Instead a combination of alternation and block structure is used.

5 Introduction

In the first part of this paper the abstraction mechanisms of BETA, *pattern, sub-pattern* and *virtual pattern* were presented. In this second part of the paper the treatment of *action sequencing* between objects in a program execution is presented. Action sequencing appears in several ways in programming languages. The simplest mechanism is *sequential execution*, where procedures are executed sequentially and the dynamic structure of active procedure activations is organized as a stack.

For many applications it is more natural to organize a program execution as several sequential processes. This mode of execution is called *multi-sequential execution.* Several language constructs that support *multiple action sequences* have been proposed. In [Conway 63] the notion of

coroutine sequencing is proposed. Coroutine sequencing is different from sequential execution since a coroutine may temporarily suspend execution and later be resumed. This means that coroutines are useful to support program executions with multiple action sequences. The usefulness of coroutines is demonstrated by the languages SIMULA and MODULA 2 [Wirth 82]. The *generator* concept of the language Icon [Griswold et al. 81] also demonstrates the usefulness of the coroutine concept.

The sequencing between coroutines is *deterministic* and *explicit*, since the programmer specifies as part of the coroutine when it shall suspend its actions and which coroutine is to take over. A *semi-coroutine* always returns to the caller. A *symmetric coroutine* may transfer the control to any other coroutine.

In a number of situations a program execution has to deal with multiple action sequences that go on *concurrently*. Coroutines are not suitable to support such concurrent action sequences. In the coroutine situation, each coroutine has exclusive access to common data and there is no need for synchronization. However, to explicitly handle the sequencing between a large number of symmetric coroutines requires a strict discipline of the programmer. In the concurrent situation, it is often necessary to be able to deal with *non-determinism*: For example, the case in a system with multiple processors. The needs for handling concurrent action sequences have resulted in a number of languages that support concurrency: CONCURRENT PASCAL ([Brinch-Hansen 75]), CSP [Hoare 78] and Ada [Ada 80]. In CSP and Ada, non-determinism is represented by means of *guarded input/output commands*. That is, within an object (process, task) describing one action-sequence, it is possible to wait for one or more communications with other objects.

The BETA language supports sequential execution, alternation and concurrency. In addition a general way of specifying *compound objects* has been introduced. Alternation and compound objects are useful structuring mechanisms. Used together they may among others be an alternative to guarded commands. Below we briefly summarize the BETA constructs treated in this part.

As mentioned in part 1, objects may be of three *kinds*: *system, compo-*

nent, or *item*. In part 1, the constructs for generation and execution of items were described.

Objects of kind system support non-deterministic multiple action sequencing in the form of concurrency. Systems may be executed by means of a *concurrent-imperative*. This imperative is similar to Dijkstra's par-begin, parend, but the constituent parts of a concurrent-imperative are systems (like in CSP) and not arbitrary imperatives.

Systems may communicate by means of synchronized execution of items. One system (the *sender*) may request another system (the *receiver*) to execute an item. The receiver must execute an accept-imperative before the communication takes place. The sender must name the receiver. The receiver specifies that the sender must belong to some restricted set of systems. This set may as extremes consist of one specific system or the set of all systems.

Synchronization between systems is similar to the handshake in CSP or rendezvous in Ada. In CSP both the sender part and the receiver part in a communication must be named. In Ada only the sender must name the receiver whereas the receiver accepts all systems. The BETA approach includes these two extremes as special cases. Concurrency and synchronization are described in section 6.

By means of textual nesting (block structure) it is possible to specify *compound systems*. A system may specify concurrent or alternating execution of one or more internal objects. Such a compound system will then have several ongoing action sequences. External systems may communicate directly with the internal systems without synchronizing with the enclosing system. Internal systems may execute items belonging to an enclosing system. Such items are executed one at a time. In this aspect an enclosing system is functioning similar to a monitor in CONCURRENT PASCAL. Few programming languages support compound systems in a general sense. In Ada, for example, it is possible to specify compound systems in the form of nested tasks. However, the communication with internal tasks is limited. It is not possible in Ada to call entry procedures of internal tasks of a task. Compound systems are described in section 7.

A number of activities may be modelled by means of compound systems consisting of several concurrent action sequences. Examples of this are machines consisting of several parts each executing an independent action sequence. In other cases an activity may be characterized by performing several action sequences, but at most one at a time. The activity will then shift between the various action sequences. An example of this is a cook making dishes. This involves several ongoing activities by the cook who constantly shifts between those requiring his attention. An activity like this may be described in BETA by means of objects of kind component and a mode of execution called *alternation.*

Alternating execution of components is specified by means of the *alternation-imperative.* Alternating execution of a list of components means that only one of the components will execute its action-part at a time. The components not executing actions are delayed at well defined points. Interleaving (shift of execution to another component) may only take place when a component is (1) at the beginning of its action-part, (2) attempting to make a communication, or (3) has suspended its execution of actions.

A processor handling several devices may naturally be described by alternation. The processor may be a compound system consisting of a number of alternating components each serving its own device. The use of components and alternation is further described in section 8.

6 Concurrent Execution of Systems

In this section generation of objects of kind *system* and concurrent execution of systems is described. The following example shows a BETA program containing three systems, *Slave*1, *Slave*2, and *Master*. *Slave*1 and *Slave*2 are generated according to the pattern *Slave*. *Slave*1 and *Slave*2 are static systems. This mode of generation is quite analogous to the generation of static items as demonstrated in part 1 of the paper. *Master* is also a a *static system*, but described directly without referring to a pattern.

The action-part of the program consists of *the concurrent-imperative*

(|| *Master* || *Slave*1 || *Slave*2 ||)

A concurrent-imperative specifies concurrent execution of the systems. The concurrent-imperative terminates when all the involved systems have *suspended* execution of their actions. A system is suspended when it either has executed a suspend-imperative or has finished executing its action-part.

```
(#
   Slave: (# ...#);
   Slave1: @ || Slave;
   Slave2: @ || Slave;
   Master: @ || (# ...#);
do
   (|| Master || Slave1 || Slave2 ||)
#)
```

6.1 System Communication

Communication between systems takes place by means of synchronized execution of items. A system S may request a system R to execute a specific item belonging to R. The system R must signal that it is willing to execute that specific item if requested by S. The program fragment in figure 19 illustrates two communicating systems:

```
S: @ ||
   (# ...
   do ...E1 → R>?M → E2...
   #);
R: @ ||
   (# M: @(# ...with Sdo ...#);
      ...
   do ...; <?M; ...
   #);
```

Figure 19: Sketch of two communicating systems

The system S may execute a *request-imperative* of the form:

$$E1 \rightarrow R{>}?M \rightarrow E2$$

which means that the system R is requested to make a synchronized execution of the item M. The parts $E1 \rightarrow$ (called the *predecessor* of M) and $\rightarrow E2$ (called the *successor* of M) are optional. M is a static item attribute of R and may be declared as follows:

$$M\text{: @(\# ...with } S\text{do ...\#)}$$

The with-part of M specifies that M may be executed synchronized with the system S. In order to signal that R is willing to execute M, R may execute an *accept-imperative* of the form:

$$<?M$$

A *communication* is carried out when S is executing $R{>}?M$ and R is executing $<?M$. Both S and R are blocked until the communication takes place. A communication has the effect that S and R together execute the evaluation:

$$E1 \rightarrow M \rightarrow E2$$

This takes place as follows:

1. If M has a predecessor (i.e. $E1 \rightarrow$ is present), then the execution of $E1$ is carried out by S independently of R. During the *value-*

transfer between $E1$ and M, S and R must be synchronized.

2. M is executed by R independently of S.

3. If M has a successor (i.e. $\rightarrow E2$ is present), then S and R must be synchronized during the value-transfer between M and $E2$. $E2$ is then executed by S independently of R.

4. S and R must be synchronized at at least one point in time during the communication.

S is called the *sender* and R the *receiver*.

A communication is thus similar to a rendezvous in Ada. The evaluation of the input values ($E1 \rightarrow$) is carried out by the sender independently of the receiver. During the transfer of the input values, the sender and receiver must be synchronized. Similarly for the output values ($\rightarrow E2$). If no input values (output values) are present, then the sender and receiver need only be synchronized during the transfer of the output values (input values). If neither input values nor output values are present, then the sender and receiver must be synchronized at at least one point in time.

The example in figure 20 shows a BETA program with communicating systems. The *Master*-system transmits a sequence of values to the two *Slave*-systems. Each *Slave*-system computes the sum of the values being received. Each value is received and accumulated by a synchronous execution of *Incr*. The transmission of values is terminated by a zero. Then the *Slave*-systems return the sums to the *Master*-system by a synchronous execution of *Result*. Positive numbers are transmitted to *Slave*1 and negative numbers are transmitted to *Slave*2.

6.2 The Some-Construct

As described above, the with-part of an object specifies that only one specific system may be the sender-part of a communication involving the object. It is often desirable to specify that the sender-part of a communication may be selected from a restricted set of systems.

```
(#
   Slave:
   (# Sum: @Integer;
      Incr: @(# I: @Integer
                enter (I)
                with Master
                do Sum + I → Sum
                #);
      Result: @(# with Master exit (Sum)#)
   do 0 → Sum;
      Loop: Cycle
            (# do <?Incr;
               (if Incr.I // 0 then leave Loop if);
            #);
      <?Result;
   #);

   Slave1: @ || Slave;
   Slave2: @ || Slave;

   Master: @ ||
   (# Pos, Neg: @Integer; V: [100] @Integer;
   do ReadIn;  { read values to V }
      (for inx: 100 repeat
           (if True
           // V[inx]  > 0 then (V[inx]) → Slave1>?Incr
           // V[inx]  < 0 then (V[inx]) → Slave2>?Incr
       if)for);
      (0) → Slave1>?Incr; Slave1>?Result → Pos;
      (0) → Slave2>?Incr; Slave2>?Result → Neg;
   #)

do (|| Master || Slave1 || Slave2 ||)
#)
```

Figure 20: Example of concurrent systems

The construct **some** P may be used to specify that the sender-part may be any system which is qualified by P. (See section 3.)

This may be specified as follows:

M: @(# ... **with some** P ... #)

If $<?M$ is executed, then the sender-part of the communication may be any P-system that executes $R>?M$.

The example in figure 21 illustrates the use of **some**. The systems using *SingleBuf* must be qualified by *Producer* or *Consumer*. Only *Producer*-systems may use *Put* and only *Consumer*-systems may use *Get*. The description of the system *Prod* has the form *Producer*(# ... #), which implies that *Prod* is qualified by *Producer*.

The above approach includes the possibilities of CSP and Ada to specify the sender-part of a communication. In CSP one is forced to name one specific system. In Ada all systems may appear as the sender-part of a communication. In BETA it is in addition possible to restrict the possible sender-systems to a set of systems belonging to a pattern.

```
(#
   Producer: (# ...#);
   Consumer: (# ...#);

   SingleBuf: @ ||
   (# BufCh: @Char;
      Put: @(# enter (BufCh) with some Producer#);
      Get: @(# with some Consumer exit (BufCh)#);
   do Cycle(# do <?Put; <?Get#);
   #);

   Prod: @ || Producer
   (# Ch: @Char;
   do ... {produce a character} Ch → SingleBuf>?Put...
   #);

   Cons: @ || Consumer
   (# Ch: @Char;
   do ...SingleBuf>?Get → Ch {consume a character} ...
   #)

do
   (|| Prod || SingleBuf || Cons ||)
#)
```

Figure 21: Example of partial restrictions on cummunication partners

7 Compound Objects

Just as it is useful to be able to construct compound items it is useful to be able to construct *compound systems.* It should be possible to refine a system into more systems and in this way construct a compound system that consists of multiple action sequences. In this section it is shown that block structure is useful for constructing compound systems.

In BETA the actions to be performed by a system may be distributed among several internal systems. The internal systems may be more or less independent. They may access common data (items in an enclosing system), communicate with each other, communicate with external systems or control communications between external systems and the enclosing system. Below the behavior of compound systems is described.

7.1 Execution of Global Items

Execution of an item belonging to an enclosing system may be requested from an internal system. The actual execution of the item will be performed by the system to which the item belongs. Consequently only one global item of an enclosing system may be executed at a time. If two or more internal systems request execution of a global item, they will be served one at a time. The enclosing system must be executing either a concurrent-imperative or an alternation-imperative (see section 9) in order to execute a request from an internal system.

The example in figure 22 describes a picture, represented by the *Picture*-system. The actual picture is represented by the internal static item *pictureData.* The operations upon *pictureData* are the items *Update* and *Get.* The picture is constantly being displayed on a screen. This is performed by the internal system *Display.* The picture may be changed by means of external requests. The internal system *Control* handles possible external requests and updates the picture. The systems *Display* and *Control* need not synchronize as the *Display*-system always displays the latest version of the picture. The operations upon *pictureData* must be indivisible. The *Display*-system reads the picture by requesting execution of the global item *Get.* Similarly the *Control*-system updates the picture

```
(#
   Picture: @ ||
   (# pictureData: @...;  {representation of the picture}
      Update: @...;  {operation to update the picture}
      Get: @...;  {read the picture}

      Display: @ || (# ...do Cycle(# do Get → Screen>?show#)#);

      Control: @ ||
               (# request: @(# do ...;Update;...#);
               do Cycle(# do <?request#)
               #);
   do (|| Display || Control ||)
   #)
do (|| Picture ||)
#);
```

Figure 22: Example of systems exectuing global items

by requesting execution of the global item *Update*. Since *Get* and *Update* are executed by the *Picture*-system, only one at a time will be executed, that is, execution of each item has exclusive access to the representation of the picture.

7.2 Communication With Internal Systems

From outside a system it is possible to communicate directly with its internal systems without synchronizing with the enclosing system. The example in figure 23, the *Pipe*-system consists of three internal systems, *Fsys, Gsys, Hsys*, each computing a function. The *In*-system delivers values to the *Pipe.Fsys*-system and the *Out*-system receives values from the *Pipe.Hsys*-system. Patterns of the form *TPort* (e.g., *realPort*) specify *T*-items that may be executed synchronized with any system.

```
(#
   In: @ || (# A: @Real; ...do ...(A) → Pipe.Fsys>?X1; ...#);

   Pipe: @ ||
   (#
      Fsys: @ ||
            (# X1,Y1: @realPort; F: (# ...#)
            do Cycle
               (# do <?X1;  {await input value}
                     X1 → F → Y1;  {compute F}
                     Y1 → Gsys>?X2 {await output to Gsys}
               #)#);

      Gsys: @ ||
            (# X2,Y2: @realPort; G: (# ...#)
            do Cycle
               (# do <?X2;  {await input value}
                     X2 → G → Y2;  {compute G}
                     Y2 → Hsys>?X3;  {await output to Hsys}
               #)#);

      Hsys: @ ||
            (# X3,Y3: @realPort; H: (# ...#)
            do Cycle
               (# do <?X3;  {await input value}
                     X3 → H → Y3;  {compute H}
                     <?Y3 {await output of Y3}
               #)#);
   do (|| Fsys || Gsys || Hsys ||)
   #);

   Out: @ || (# B: @Real; ...do ...Pipe.Hsys>?Y3 → B; ...#)
do (|| In || Pipe || Out ||)
#)
```

Figure 23: Example of communication with internal systems

```
(#
   R: @ ||
   (# M: @T1Port;
      N: @T2Port;
      ControlM: @ || (# ...do ...<?M...#);
      ControlN: @ || (# ...do ...<?N...#);
   do (|| ControlM || ControlN ||)
   #);

   S: @ || (# ...do ...R>?M...R>?N...#)
do (|| R || S ||)
#);
```

Figure 24: Example of communication using global items

7.3 Communication Using Global Items

A global item may be accessed in an accept-imperative. This means that an internal system may control the possible communications of an enclosing system. As with execution of a global item, it is the system to which the item belongs that executes the item. This again implies that the system must be executing a concurrent- or alternation-imperative in order that the request can be accepted. In figure 24 possible accepts of the requests $R>?M$ and $R>?N$ from the S-system are controlled by the internal systems *ControlM* and *ControlN* of the R-system.

8 Alternating Execution of Components

In the previous sections concurrent execution of systems has been described. BETA also contains a generalization of the SIMULA coroutine construct. Consider the following example:

```
(#C1 : @ | (# ...suspend ...#);
   C2 : @ | (# ...suspend ...#);
   C2 : @ | (# ...suspend ...#)
do (| C1 | C2 | C3 |)
#)
```

C1, C2, and C3 are objects of kind *component*. Components may be executed alternating as specified by the imperative:

```
(| C1 | C2 | C3 |)
```

Alternating execution means that at most one of the components is executing its action part at a time. The components not executing actions are delayed at well defined points in their action sequence. These points are the same at which interleaving (i.e. shift of execution to another component) may take place. Interleaving may take place (1) at the beginning of the action-part of the component, (2) when the component is attempting to make a communication, and (3) when the component has suspended its action-sequence.

The example in figure 25 describes a bounded buffer implemented using alternation. The internal components *Put* and *Get* control the communication with the *Buffer*-system. The imperative *Pause* specifies that this is a point where interleaving may take place. *Pause* is not specified here, but may be implemented as a communication with some system in the environment, such as a timer-system.

Since the execution of the *Put* and *Get* components is alternating, each component has exclusive access to the buffer representation. Interleaving may only take place at *<?InCh*, *<?OutCh* and *Pause*.

```
(#
   Buffer:
   (# S: [100] @Char; In, Out : @Integer;
      InCh, OutCh: @CharPort;

      Put: @ |
      (# do
         Cycle(# do
         (if (In = Out)
         // False then  { buffer not full}
            <?InCh;  {Await input of char to InCh}
            InCh → S[In]; (In mod 100) + 1 → In
         // True then Pause { buffer is full }
       if) #)#);

      Get: @ |
      (# do
         Cycle(# do
         (if (In = (Out mod 100 + 1))
         // False then  { buffer is not empty}
            S[(Out mod 100) + 1 → Out] → OutCh;
            <?OutCh;  {Await output of char from OutCh}
         // True then Pause { buffer is empty }
       if) #)#);

   do 1 → In; 100 → Out;
      (| Put | Get |)
   #);

   Prod: @ || (# ... do ... Ch → Buf>?InCh ... #);
   Buf: @ || Buffer;
   Cons: @ || (# ... do ... Buf>?OutCh → Ch ... #);
do (|| Prod || Buf || Cons ||)
#)
```

Figure 25: Bounded buffer with alternating components

8.1 Coroutine Sequencing

In SIMULA, an object may also be used in a procedure-like manner by means of the call/detach-imperatives. This use of objects is called *semi-coroutines* in SIMULA. SIMULA also supports full *symmetric coroutines* by means of the resume-imperative. One disadvantage of using SIMULA coroutines (objects) as procedures is that it is not possible to transfer parameters to and from the object at the point of a call, detach or resume.

In BETA parameter transfer to objects used as procedures and objects used as semi-coroutines is identical. In figure 26 is shown an example of using a component as a semi-coroutine.

The pattern *nextChar* describes a coroutine that reads an ASCII-file from a disc. The file is organized as a sequence of blocks of fixed length. The last block of the file will only be partially filled with characters. The rest of the block is filled with ASCII *null* characters and will contain at least one *null*. The file contains at least one block. As the last character of the file, *nextChar* will return the ASCII character fileSeparator (*fs*). *NextChar* will in addition ensure that the last line before *fs* is terminated by a lineFeed(*lf*) by always returning a *lf* before the final *fs*.

The declaration *nextCh*: @ | *nextChar* specifies the generation of a *static component*. The component is an instance of the pattern *nextChar* and has the name *nextCh*. *NextCh* may be executed in an alternation-imperative. Execution of the imperative **suspend** within *nextCh* implies that the execution of the do-part of *nextCh* is temporarily suspended. Control returns to the caller of *nextCh* and the exit-list of *nextCh* is evaluated. Successive calls on *nextCh* will then resume the execution of *nextCh* at the point following **suspend**.

8.2 Motivation for Alternation

As mentioned in the introduction it is often the case that an activity may be characterized by performing several activities, but at most one at a time. One example is a cook making dishes; another example is a processor serving a number of devices.

```
(#
   nextChar:
   (# F : @file; B : [blockSize] @Char;
      chr : @Char; inx : @Integer;
   do F.readBlock → B;  {a file contains at least one block}
      nextBlock:
      (if F.endFile // False then
         (for i : blockSize repeat B[i]  → chr; suspend for);
         F.readBlock → B;
         restart nextBlock
       if);
      1 → inx;
      lastBlock:
      (if (B[inx] ≠ null) // True then
         B[inx] → chr; suspend;
         inx + 1 → inx; restart lastBlock
       if);
      lf → chr; suspend;  {ensure linefeed before endfile}
      fs → chr
   exit (chr)#);

   nextCh : @ | nextChar;  {declaration of a component}
   ch : @Char;
do ...
   loop :Cycle
         (# do (| nextCh → ch |);
               (if ch // fs then leave loop
if) #)#)
```

Figure 26: Example of a component modelling a semi-coroutine

Alternation makes it possible to structure a program using components without explicitly having to synchronize access to common data. Most programs using alternation could be modelled by programs using concurrency. However concurrency implies that each system executes actions with a positive speed. On a single processor this implies time sharing using a clock. Some implementations of concurrency avoid this by shifting to another process only at the point of a communication. If this is the case the program actually consists of alternating components and not of concurrent systems. In a concurrent program no looping system system can monopolize the processor whereas this is the case with an alternating program.

As mentioned in the introduction guarded input/output commands may be used to handle the non-determinism of communication in a system of concurrent components. A component involved in communications with more than one component may then wait for one or more communications at the same time.

In many situations the different communications are more or less independent. Assume that the system A communicates with the systems B, C, D and E. Then it may be the case that the communications with B and C are performed in sequence and the communications with D and E are performed in another sequence, but there is no sequencing between the two groups of communications. In such a situation it may be more natural to describe the communication sequences by means of alternating components. In the above example the system A can be described as a compound system consisting of two alternating components, one for communication with B and C and another for communication with D and E. In the buffer example, *Put* takes care of a sequence of $InCh$-communications and *Get* takes care of a sequence of $OutCh$-communications.

By using alternation and compound objects instead of guarded commands one often avoids mixing logically independent action sequences into one action sequence. This may make the structure of the resulting program more clear. In addition it simplifies implementation. A CSP-process and an Ada-task may have several open communications waiting at a time. When a communication takes place possibly other open communications of the involved objects must be closed. In BETA each object may wait

for at most one communication. No open communications need to be closed when a communication takes place. Finally in CSP and Ada the use of input and output as guards is not symmetric: It is only possible to have input-commands (accept-statements) in a guarded command. The possibility of allowing output-commands as guards in CSP is mentioned in [Hoare 78]. However, symmetric use of input-/output-guards greatly complicates the implementation.

Alternation is not an alternative to concurrency, but a supplement. A number of activities are by nature alternating and non-deterministic and such activities should not be modelled by concurrent systems, coroutines or guarded commands. However experience with BETA is necessary to find out if alternation is a general useful sequencing mechanism.

9 Conclusion

The BETA programming language has been developed as part of the BETA project. The purpose of this project is to develop concepts, constructs and tools in the field of programming and programming languages. BETA has been developed from 1976 on and the various stages of the language are documented in [BETA 76-85].

The application area of BETA is programming of large software systems including embedded as well as distributed computing systems. For this reason a major goal has been to develop constructs that may be efficiently implemented. Furthermore it has been attempted to develop a language with a few basic but general constructs.

There are many reasons for proposing a new programming language. BETA is intended to be a modern language in the SIMULA tradition. With the present state of art it is possible to design a more simple, flexible and powerful language based upon SIMULA. The SIMULA class/subclass constructs have only been successfully carried over to languages intended for exploratory programming, such as Smalltalk and its successors FLAVORS ([Cannon 82]) and LOOPS ([Bobrow and Stefik 83]). These languages are "typeless" since variables are not qualified by a class like in SIMULA and BETA. The advantages of "types" should be well known: the compiler can catch errors that otherwise would have been discovered one at a time on run-time, the program is easier to understand, and more efficient code can be generated. No language has succesfully combined the class/subclass constructs with constructs for handling concurrency and alternation. CONCURRENT PASCAL was an attempt, but is lacking subclasses and virtuals.

SIMULA is a system description and a programming language. The DELTA language ([DELTA]) is a system description language only, allowing description of full concurrency, continuous change and component interaction, developed from a SIMULA conceptual platform. BETA started from the system concepts of DELTA, but is a programming language, drawing upon a large number of contributions to programming research in the 1970s.

BETA supports the object oriented perspective on programming. A major part of the BETA project has been to develop the basic concepts underlying object oriented programming. In this paper, however, only the the BETA programming language has been described. The basic concepts will be the subject of another paper.

The pattern construct is a very general abstraction mechanism. In practice most patterns are either used as classes, procedures, functions or types. Few patterns are useful for more than one of these purposes. For abstract super-patterns[4] it often appears that no decision is made about the use of its sub-patterns. The main advantage of having only one abstraction mechanism is a uniform treatment of abstraction mechanisms and their instances. A consequence of having only one abstraction mechanism is that hierarchical classification is also available for patterns used as procedures, functions or types. Another consequence is that the virtual concept is not only available for procedures. Virtual patterns may model SIMULA-like virtual procedures and Smalltalk methods. In addition they may model virtual classes which are not available in SIMULA and Smalltalk. Virtual patterns may also be used for modelling formal procedures and formal types.

BETA has evolved for many years. Experience from use will probably influence the language further. Certain parts of the language will definitely be further developed.

- The value concept of BETA needs to be further developed. Some ideas are described in [BETA 83b]. The notion of enumeration types from Pascal should be supported. Consider

  $$color = (red, green, white)$$

  In BETA *color, red, green*, and *white* will be modelled as patterns. *Red, green*, and *white* will be sub-patterns of *color*. It should also be possible to specialize a pattern like *red* into sub-patterns *pink* and *ruby*. This will make it possible to describe a hierarchy of types and values.

- BETA does not support multiple inheritance. Currently no pro-

[4]In Smalltalk terminology, an abstract super-pattern (super-class) is a pattern (class) that is only used as a super-pattern. That is, no instances are created.

posal for supporting multiple inheritance has been found acceptable. The main problem is to replace the inner mechanism for combination of action parts. The solution proposed in [Thomsen 86] is a promising approach.

- In addition to object oriented programming, BETA supports procedural programming and to a limited extent also functional programming. Work is going on to improve the support for functional programming. In addition support for logic programming will be included. All these programming styles will not be supported equally well. The overall perspective will be object oriented. The functional and logical programming will primarily be aimed at describing measurable properties of objects and transitions between meaningful states.

- In section 4 on virtual patterns, it was mentioned that there is an asymmetry between the use of inner and virtual patterns. In a future version of BETA, the generalization of virtual patterns proposed in [BETA 83b] will be incorporated.

- BETA has no constructs for modularization and protection of data representation (see section 2.6). Instead a general language independent mechanism has been developed ([BETA 83a]) based on the context-free grammar of a language. The idea is that a program may be split into arbitrary fragments (modules). A fragment may be a string of terminal and nonterminal symbols generated from any nonterminal of the grammar. By defining a partial ordering between the fragments, it is possible to perform separate context-sensitive analysis (static semantics). The method may be used to split a module into an interface-part and an implementation-part thereby providing protection of data representation. Finally the mechanism may be used as a basis for constructing a system for handling different versions/variants of a program.

- For a large class of systems, e.g. operating systems, database systems, the possibility of exchanging components during the life span (execution) of the system is a natural and inherent property of the system. In order to do this it is necessary to have a model of a machine that includes both the program execution and the processor performing the program execution. Work is going on to

develop such a model together with language constructs for expressing this in BETA. A machine is viewed as an ensemble making a performance (program execution) according to a play (program) described in some language (programming language).

- A set of attributes (patterns and references) realizing concepts and phenomena from a given application area may be viewed as a language for describing systems belonging to that application area. Often it would be desirable to describe such systems using another syntax than that of BETA. For this purpose a BETA programming system should contain a tool that given a set of attributes and a description of the syntax will produce a new front end of the BETA compiler.

Despite the above list we think that BETA has reached a state where it may be usable for programming computer systems.

An experimental version of BETA has been implemented on various Motorola 68000 based machines including Macintosh, SUN and Sysware SUS. Work is going on to design and implement a programming environment for BETA. This takes place in a number of projects in Scandinavia ([Mjølner],[Scala]).

Acknowledgement. The BETA project was initiated in 1976 as part of what was then called *The Joint Language Project* (JLP). People from The Regional Computing Center, Aarhus University, Computer Science Department, Aarhus University, Institute for Electronic Systems, Aalborg University Centre and The Norwegian Computing Center, Oslo participated in the JLP-project. The initiative for the JLP was taken in the autumn of 1975 by Bjarner Svejgaard, Director of The Regional Computing Center.

In addition to the JLP team members, a large number of people especially from Oslo and Aarhus have participated in our discussions. Bruce Moon, University of Canterbury, and Dag Belsnes, The Norwegian Computing Center have been most influential. Also the team members of Scala and Mjølner have contributed.

The work reported here has been supported by *The Royal Norwegian Council for Scientific and Industrial Research*, grant no. ED 0223.16641

(the Scala project), and by *The Danish Natural Science Research Council* grants no. 11-3106 and FTU 5.17.5.1.25.

Finally we want to thank the referees and the editors of this book for their assistance in improving the readability of this paper. In particular we want to acknowledge the contributions of *Peter Wegner*.

10 References

1. [Ada] *Ada Reference Manual.* Proposed Standard Document, United States Department of Defense, July 1980.

2. [ALGOL] P. Naur (ed.): Revised Report on The Algoritmic Language ALGOL 60. *Regnecentralen. Copenhagen, 1962.*

3. [BETA 76] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: BETA Project Working Notes 1-8. *Norwegian Computing Center, Oslo and Computer Science Department, Aarhus University, Aarhus, 1976-1982.*

4. [BETA 83a] B.B. Kristensen, O.L Madsen, B. Møller-Pedersen, K. Nygaard: Syntax Directed Program Modularization. In: *Interactive Computing Systems* (ed. P. Degano, E. Sandewall), North-Holland, 1983.

5. [BETA 83b] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Abstraction Mechanisms in the BETA Programming Language. *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, January 24-26 1983, Austin, Texas.*

6. [BETA 83c] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: From SIMULA 67 to BETA. *Proceedings of the Eleventh SIMULA 67 User's Conference, September 1983, Paris. Norwegian Computing Center, 1983.*

7. [BETA 85] B.B. Kristensen, O.L. Madsen, B. Møller Pedersen, K. Nygaard: Multi-sequential Execution in the BETA Programming Language. *Sigplan Notices, Vol. 20, No. 4, April 1985.*

8. [BETA 87] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Classification of Actions or Inheritance also for Methods. *Proceedings of the Second European Conference on Object Oriented Programming, Paris, June 1987.*

9. [Bobrow and Stefik 83] D.G. Bobrow, M. Stefik: The LOOPS Manual. *Xerox Corporation 1983.*

10. [Brinch-Hansen 75] P. Brinch-Hansen: The Programming Language Concurrent PASCAL. *IEEE Transactions on Software Engineering* **SE-1**, *2 (June 1975), 149-207.*

11. [Cannon 83] H. Cannon: Flavors, A Non-Hierarchical Approach to Object-Oriented Programming. *Draft 1982.*

12. [CLU] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert: Abstraction Mechanisms in CLU. *Comm. ACM 20, 8 (1977), 564-576.*

13. [Conway 63] M.E. Conway: Design of a Separable Transition – Diagram Compiler. *Comm. ACM* **6**, *7 (1963), 396-408.*

14. [DELTA 75] E. Holbæk Hansen, P. Haandlykken, K. Nygaard: System Description and the DELTA Language. *Norwegian Computing Center, Oslo, 1975.*

15. [DELTA 81] P. Haandlykken, K. Nygaard: The DELTA System Description Language: Motivation, Main Concepts and Experience from use. *In: Software Engineering Environments (ed. H. Hunke), GMD, North-Holland, 1981.*

16. [DeRemer and Krohn] F.L. DeRemer, H. Krohn: Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering, SE-3, 1 (Jan 1977), 69-84.*

17. [Dijkstra 75] E. W. Dijkstra: Guarded Commands,Nondeterminacy, and Formal Derivation of Programs. *Comm. ACM 18, 8 (1975), 1-82.*

18. [Griswold et al. 81] R.E. Griswold, D.R. Hanson, J.T. Korb: Generators in ICON. *ACM Transactions on Programming Languages and Systems* **3**, *2 (April 81), 144-161.*

19. [Hoare 72] C. A. R. Hoare: Proof of Correctness of Data Representation. *Acta Informatica 4 (1972), 271-281.*

20. [Hoare 78] C.A.R Hoare: Communicating Sequential Processes. *Comm. ACM* **21**, *8 (1978), 666-677.*

21. [Madsen 86] O.L. Madsen: Block Structure and Object Oriented Languages. *Sigplan Notices, October 1986. (Also in: B.D. Shriver, P. Wegner (ed.):* Research Directions in Object Oriented Programming, *MIT Press, 1987).*

22. [Mjølner] H.P. Dahle, M. Løfgren, O.L. Madsen, B. Magnusson: The Mjølner Project — A Highly Efficient Programming Environment for Industrial Use. *Mjølner report no. 1, Oslo, Malmø, Aarhus, Lund 1986.*

    The Mjølner project is a joint venture between Elektrisk Bureau a.s., Oslo, Telelogic AB, Malmø and Sysware aps, Aarhus; in cooperation with The Norwegian Computing Center, Oslo, University of Lund, Lund, Aalborg University Centre, Aalborg, and Aarhus University, Aarhus. It has been initiated by Nordforsk (the Nordic cooperative organization fo applied research), Copenhagen, which also coordinates the project. The project is partly funded by a grant from The Nordic Fund for Technology and Industrial Development. The purpose of the project is to develop a programming environment supporting object-oriented design and programming.

23. [PASCAL] N. Wirth: The Programming Language PASCAL. *Acta Informatica 1 (1971), 35-63.*

24. [Scala] The Scala project is a joint venture between University of Oslo, Norwegian Computing Center, Elektrisk Bureau a/s and Norsk Data a/s, funded by NTNF. The purpose of the project is further development of the BETA language, implementation of a BETA environment (in cooperation with the Mjølner project), and implementation of BETA on ND machines.

25. [SIMULA] O.J. Dahl, B. Myrhaug, K. Nygaard: SIMULA 67 Common Base Language. *Norwegian Computing Center, Oslo, 1968, 1970, 1972, 1984.*

26. [Smalltalk] A. Goldberg, D. Robson: *Smalltalk 80: The Language and its Implementation.* Addison Wesley 1983.

27. [Thomsen 1986] K.S. Thomsen: *Multiple Inheritance, A Structuring Mechanism for Data, Processes and Procedures. Computer Science Department, Aarhus University, DAIMI PB-209, 1986.*

28. [Wirth 82] N. Wirth: *Programming in MODULA 2.* Springer Verlag, Berlin, New York, 1982.

29. [Vaucher 75] J. Vaucher: Prefixed Procedures: A Structuring Concept for Operations. *Infor, vol. 13, no. 3, October 1975.*

Appendix

In this appendix a summary of the BETA constructs will be given. The following abbreviations are used: P, $P0$: pattern title (name), D_i: declaration, In, Out, E_i, $range$: evaluation, OS: pattern title or object descriptor, Imp, I_k: imperative, X, X_i: object, T, T_i: item, C_i: component, S_i: system. Constructs marked by a * are not used in this paper.

Object descriptor	$P0$ $(\# D_1; D_2; \ldots D_n$ **enter** In **do** Imp **exit** out $\#)$

Declarations	
pattern	P: $P0(\# \ldots \#)$
virtual pattern	$V :< P$
further binding	$V ::< P$
final binding	$V :: P$
static item reference	$T1$: $@OS$
dynamic item reference	$T2$: $\uparrow P$
static component reference	$C1$: $@ \mid OS$
dynamic component reference*	$C2$: $\uparrow \mid P$
static system reference	$S1$: $@ \parallel OS$
dynamic system reference*	$S2$: $\uparrow \parallel P$
object repetition	R: $[range] \ldots$

Evaluations	
inserted item	OS
dynamic item	$\&OS$
inserted component*	$\mid OS$
dynamic component*	$\& \mid OS$
inserted system*	$\parallel OS$
dynamic system*	$\& \parallel OS$
object execution	X
object reference	$X\square$
assignment evaluation	$E1 \rightarrow E2 \rightarrow ... \rightarrow En$
evaluation list	$(E_1, E_2, ..., E_n)$

Imperatives	
repetition	(**for** *inx*: *range* **repeat** *Imp* **for**)
selection	(**if** $E0$ // $E1$ **then** $I1$ // $E2$ **then** $I2$ ... // En **then** In **if**)
label	L: Imp (L: $I1; I2; ...In : L$)
jumps	**restart** L **leave** L
alternation	$(\mid C1 \mid C2 \mid \ldots \mid Cn \mid)$
concurrency	$(\parallel S1 \parallel S2 \parallel \ldots \parallel Sn \parallel)$

Communication	
communication request	$R>?T$
communication accept	$<?T$