The Mjølner BETA System

BETA Language Introduction

Mjølner Informatics Report

MIA 94-26(1.2)

August 1996

Introduction

This report is a an introduction to the BETA language. The BETA language is presented to someone who is familiar with one or more object-oriented language such as C++ or Eiffel.

The overall aspects of the BETA language is presented. The presentation focuses on the concepts and ideas behind the design of BETA, and includes examples on the use of most constructs. The tutorial contains sections on basic constructs, patterns and objects, singular objects, subprocedure, control patterns, nested patterns, virtual patterns, coroutines, concurrency, and inheritance.

For more details about the BETA language than presented in this tutorial please see [Madsen 93]. For a tutorial on the Mjølner BETA System, please see [MIA 94-24]

Acknowledgment

This introduction is based on a chapter in the book [Knudsen 94] written by Ole Lehrmann Madsen.

Contents

1 Language Concepts

BETA is a modern object-oriented language from the Scandinavian school of object-orientation where the first object-oriented language Simula was developed. BETA supports the object-oriented perspective on programming and contains comprehensive facilities for procedural and functional programming. BETA has powerful abstraction mechanisms for supporting identification of objects, classification and composition. BETA is a *strongly typed* language like Simula, Eiffel and C++ with most type checking being carried out at compile-time. It is well known that it is not possible to obtain all type checking at compile time without sacrificing the expressiveness of the language. BETA has an optimum balance between compile-time type checking and run-time type checking.

1.1 Powerful Abstraction Mechanisms

BETA has powerful abstraction mechanisms that provide excellent support for *design* and *implementation*, including data definition for persistent data. The powerful abstraction mechanisms greatly enhance reusability of designs and implementations.

The pattern

The abstraction mechanisms include *class*, *procedure*, *function*, *coroutine*, *process*, *exception* and many more, all unified into the ultimate abstraction mechanism: the *pattern*. In addition to the pattern, BETA has *subpattern*, *virtual pattern* and *pattern variable*. This unification gives a uniform treatment of abstraction mechanisms and a number of new ones. Most object-oriented languages have classes, subclasses and virtual procedures, and some have procedure variables. Since a pattern is a generalization of abstraction mechanisms like class, procedure, function, etc., the notions of subpattern, virtual pattern and pattern variable also apply to these abstraction mechanisms. In addition to the above mentioned abstraction mechanisms, the pattern subsumes notions such as generic package and task type as known from Ada.

Subpattern

The subpattern covers subclasses as in most other object-oriented languages. In addition, procedures may be organized in a subprocedure hierarchy in the same way as classes may be organized in a subclass hierarchy. Since patterns may also be used to describe functions, coroutines, concurrent processes, and exceptions, these may also be organized in a pattern hierarchy.

Virtual pattern

The notion of virtual pattern covers virtual procedures as in C++. In addition, virtual patterns cover virtual classes, virtual coroutines, virtual concurrent processes, and virtual exceptions. Virtual classes provide a more general alternative to generic classes as in Eiffel or templates as in C++.

Pattern variable

BETA includes the notion of pattern variable. This implies that patterns are first class values, that may be passed around as parameters to other patterns. By using pattern variables instead of virtual patterns, it is possible dynamically to change the behavior of an object after its generation. Pattern variables cover procedure variables (i.e. a variable that may be assigned different procedures). Since patterns may be used as classes, it is also possible to have variables that can be assigned classes, etc.

Coroutines and concurrency

BETA does not only allow for passive objects as in C++ and Eiffel. BETA objects may also act as coroutines, making it possible to model alternating sequential processes and quasi-parallel processes. BETA coroutines may be executed concurrent (non pre-emptive scheduling in current implementation). The basic mechanism for synchronization is semaphores, but high-level abstractions for synchronization and communication, hiding all details about semaphores, are easy to implement, and the standard library includes *monitors*, and *rendezvous*. The user may easily define new concurrency abstractions including schedulers for processes.

BETA supports the three main subfunctions of abstraction: identification, classification, and composition as described in the following.

1.2 Identification of Objects

Class-less objects

It is possible to describe objects that are not generated as instances of a class pattern, so-called “class-less objects”. This is in many cases useful when there is only one object of a kind. In most object-oriented languages, it is necessary to define superfluous classes for such objects. In analysis and design, it is absolutely necessary to be able to describe singular objects without having them as instances of classes.

1.3 Classification

Classification is supported by patterns, subpatterns, and virtual patterns that make it possible to describe classification hierarchies of objects and patterns (objects, classes, procedures, functions, coroutines, processes, exceptions, etc.).

1.4 Composition (Aggregation)

Objects and patterns may be defined as a composition of other objects and patterns. The support for composition includes:

Whole-part

- *Whole-part composition*: an attribute of an object may be a part-object. This makes it possible to describe objects in terms of their physical parts.

Reference

- *Reference composition*: an attribute may be a reference to an object. Reference composition is the basis for modeling arbitrary relations between objects.

Localization

- *Localization*: an attribute of an object may be a (nested) pattern—also known as block-structure. The block-structure makes it easy to create arbitrary nested patterns. This makes it possible for objects to have local patterns used as classes, procedures, etc. Local patterns greatly enhance the modeling capabilities of an object-oriented language.

1.5 Inheritance

In BETA, inheritance is not only restricted to inheritance from superpatterns. It is also possible to inherit from a part-object. Virtual patterns in the part-object may be redefined to influence the enclosing object. Multiple inheritance is supported through inheritance from multiple part-objects. This gives a much cleaner structure than inheritance from multiple superpatterns.

1.6 Conceptual Framework

BETA is intended for modeling and design as well as implementation. During the design of BETA the development of the underlying conceptual framework has been just as important as the language itself.

Modeling

BETA is a language for representing/modeling concepts and phenomena from the application domain and for implementing such concepts and phenomena on a computer system. Part of a BETA program describes objects and patterns that represent phenomena and concepts from the application model. This part is said to be representative since BETA elements at this level are meaningful with respect to the application domain. Other parts of a BETA program are non-representative, since they do not correspond to elements of the application domain, but are intended for realizing the model as a computer system.

The BETA language as presented in this introduction is for describing objects and patterns. The objects and patterns constitute the logical structure of a program execution. The physical structure of a program execution is handled by other components of the Mjølner BETA System. A tutorial on using the this system is given in [MIA 94-24].

2 Basic Constructs

The most fundamental elements of BETA are objects and patterns. This section describes the basic patterns and values, simple assignments, control structures, variable declarations, repetitions and patterns used as composite types.

2.1 Simple Types and Values

The simple types (or also called basic patterns) are `integer`, `boolean`, `char`, and, `real`. The following table shows the simple types with examples of values, including `text constant`. Notice, that `text` is not a simple type in BETA, but a pattern defined in the basic BETA environment called `betaenv`.

Type	Value
`integer`	`7, -4, 0x4FFC, 2x101101`
`boolean`	`true, false`
`char`	`'c'`
`real`	`3.141, -1.234E3`
`text constant`	`'abc'`

2.2 Simple Static Variables

In BETA, a static variable (also called a static reference) is declared like:

```
i: @integer;
r: @real;
```

Variables of the simple types can only be declared static, see below for dynamic references.

2.3 Simple Assignments

Simple value assignments in BETA goes left to right:

Value assignmen

```
2 -> i          (* assign the value 2 to i *)
i -> j          (* assign the value of i to j *)
i*j -> k        (* assign the value of i*j to k *)
(i,j) -> (x,y)  (* assign the value of i to x and
                 *        the value of j to y
                 *)
```

2.4 Control Structures

BETA has two build-in control structures: `if` and `for`, both having two forms. The simple `if` imperative with one boolean expression:

if

```
(if <expression> then
    <imperatives>
 else
    <imperatives>
if)
```

and the `if` with several alternatives:

```
(if <expression>
 // <expression> then <imperatives>
 // <expression> then <imperatives>
 …
 else
   <imperatives>
if)
```

where `//` means equals.

The simple `for` imperative just iterates a given number of times:

for

```
(for <expression> repeat <imperatives> for)
```

but the `for` imperative may implicitly declare an iteration variable, only available inside the `for` loop, by:

```
(for <variable>: <expression> repeat <imperatives> for)
```

The `for` loop always starts in 1 and stops at `<expression>`. The loop can be terminated or restarted using labels, see below.

The following BETA code is a general object-descriptor (or descriptor for short):

descriptor

```
(# <declarations>
enter <enter-list>
do <imperatives>
exit <exit-list>
#)
```

A descriptor consists of type and variable declarations, an enter part for parameters (`enter <enter-list>`), a do-part for the action (`do <imperatives>`), and finally an exit part for the results (`exit <exit-list>`). All elements are optional.

A descriptor can be labeled, and the descriptor can be restarted and/or left using the label:

labeled descriptor

```
L: (# … leave L … restart L #)
```

In general any imperative can have a label:

labeled imperative

```
L: <imperative>
L: (if  … leave L …  if)
L: (for … leave L … for)
```

`leave L` implies that control is transferred to immediately *after* the labeled imperative/descriptor. `restart L` implies that control is transferred to immediately *before* the labeled imperative/descriptor.

2.5 Static and Dynamic Variables

Reference attributes

In BETA variables are two examples of *reference attributes*—static references that constantly denote the same object, and dynamic references that may denote different objects.

Static Reference

Examples of static reference variables are:

```
i: @integer (* i refers to a simple type: integer *)

p: @A       (* an instance of A is automatically generated and
             * p always refers to this object *)

s: @(# … #) (* an instance of (# … #) is automatically generated
             * and s always refers to this singular object *)
```

Dynamic reference

Examples of dynamic reference variables are:

```
i: ^integerObject

p: ^A
```

Assignments between dynamic references can be done using the reference operator ‘`[]`’ (read *box*):

Reference assignment

```
p1[] -> p2[]  (* reference assignment *)
```

Dynamic reference variables are initially `NONE` i.e. refers to nothing. Objects can be created using the new operator ‘`&`’:

```
&A[] -> p[]  (* create an instance of A and assign the reference
              * to p *)
```

It is illegal to declare dynamic references to simple types:

```
i: ^integer (* ILLEGAL *)

r: ^real    (* ILLEGAL *)
```

Instead use `integerObject`, `charObject`, `booleanObject`, or `realObject` defined in the Mjølner BETA System basic `betaenv` environment.

2.6 Repetitions

In BETA it is possible to declare a repetition of static (simple types) or dynamic references. A repetition is declared like:

```
R: [10] @integer  (* repetition of 10 static references *)

P: [10] ^A        (* repetition of 10 dynamic references *)

R[1] -> i         (* value assignment *)

P[1][] -> x[]     (* reference assignment *)

RR: [1] @integer  (* repetition of 1 static reference *)

R -> RR           (* repetition assignment:
                   * all values from R is copied into RR
                   * RR is automatically extended if needed
                  *)

R.range           (* the size of the repetition *)

n -> R.extend     (* extends the repetition with n elements *)

n -> R.new        (* allocates a new repetition with n elements *)
```

The range of a repetition is `1` to `R.range`, thus repetitions always start with 1.

2.7 Composite Types (Records)

Using the object-descriptor it is possible to declare composite types:

```
point:  (# x,y: @integer #) (* point is a composite type
                             * consisting of two integers *)
p: @point    (* static declaration of a point *)
p.x          (* remote access to x *)

circle:      (* composite type using simple and composite types *)
  (# center: @point;
     radius: @integer;
  #)
```

The declaration of `point` and `circle` above is in general called a *pattern* declaration. The pattern will be described in details in the following sections.

Pattern

3 Patterns and Objects

Most object-oriented languages supporting the object-oriented perspective have constructs such as class, subclass, virtual procedure, and qualified reference variable. These constructs all originated with Simula. Eiffel and C++ include these constructs although a different terminology is used. In addition to virtual procedures BETA also has non-virtual procedures.

In this introduction, the BETA version of the above constructs will be described and compared to other object-oriented languages. The example used in the following is a company with different kinds of employees, including salesmen and workers. `employee` is an abstract superpattern describing the common properties of all employees.

Pattern Employee

```
employee:
  (# name: @text;
     birthday: @date;
     dept: ^Department;
     totalHours: @integer;
     registerWork:
       (# noOfHours: @integer
       enter noOfHours
       do noOfHours + totalhours -> totalHours
       #);
     computeSalary:<
       (# salary: @integer
       do inner
       exit salary
       #);
  #);
```

The elements of the `employee` pattern have the following meaning:

Elements of Employee

- The attributes `name`, `birthday`, `dept` and `totalHours` are reference attributes denoting instances of the patterns `text`, `date`, `department` and `integer` respectively.
- `Name`, `birthday`, and `totalHours` refer to *part-object*s. A part-object is a fixed part of its enclosed object and is generated together with the enclosing object. Part-objects are also found in Eiffel and C++.
- `Dept` is a *dynamic reference* that either has the value `NONE` or refers to a separate instance of the pattern `department`.
- The attributes `registerWork`, and `computeSalary` are pattern attributes describing actions to be executed. They correspond to procedures in most other languages. The enter-part describes the input parameters of a pattern and the exit-part describes its output parameters. `registerWork` has one input parameter `noOfHours` and `computeSalary` has one output parameter, `salary`.
- `registerWork` is a non-virtual pattern attribute. This means that its complete description is given as part of the description of `employee`. It is similar to non-virtual functions in C++.
- `computeSalary` is a virtual pattern attribute (specified by using the ‘`:<`’ symbol). Only part of its description is given since the computation of the salary is different for salesmen and workers. The description of a virtual pattern may be extended in subpatterns of `employee`. A virtual pattern attribute is similar to a virtual function in C++.

Class and procedure patterns

- `employee`, `registerWork` and `computeSalary` are all examples of patterns. `employee` is an example of a pattern used as a class and is therefore called a class pattern. `registerWork` and `computeSalary` are examples of patterns used as procedures and are therefore called procedure patterns. Technically there is no difference between class patterns and procedure patterns.

The following patterns are subpatterns of `employee` corresponding to salesmen and workers.

Subpatterns of Employee

```
worker: employee
  (# seniority: @integer;
     computeSalary::<
       (# do noOfHours*80+seniority*4->salary; 0->totalHours #)
  #);
salesman: employee
  (# noOfSoldUnits: @integer;
     computeSalary::<
       (# do noOfHours*80+noOfSoldUnits*6->salary;
             0->noOfSoldUnits->totalHours
       #)
  #)
```

- The class pattern `worker` adds the attribute `seniority` and extends the definition of `computeSalary`. The salary for a worker is a function of the `noOfHours` being worked and the `seniority` of the worker.
- The class pattern `salesman` adds the attribute `noOfSoldUnits` and describes another extension of `computeSalary`. The salary for a salesman is a function of the `noOfHours` being worked and the `noOfSoldUnits`.
- The symbol ‘`::<`’ describe the fact that the definition of `computeSalary` from the superpattern `employee` is extended.

Part object

The above examples have shown instantiation of objects from patterns in the form of part-object attributes (like `birthday: @date`). An instance of, say `worker`, may in a similar way be generated by a declaration of the form:

```
mary: @worker
```

Dynamic reference

The above examples have also shown a dynamic reference (like `dept: ^department`). Such a reference is initially `NONE`. A dynamic reference to instances of `worker` may be declared as follows:

```
theForeman: ^worker
```

`theForeman` may be assigned a reference to the object referred by `mary` by execution of the following imperative:

```
mary[] -> theForeman[]
```

Note that the opposite assignment (`theForeman[]->mary[]`) is not legal since `mary` is a static reference. An instance of `worker` may be generated and its reference assigned to `theForeman` by executing the following imperative:

```
&worker[] -> theForeman[]
```

A few additional comments about constructs used so far:

- The symbol `&` means new.
- The symbol `->` is used for assignment of state.
- An expression `R[]` denotes the reference to the object referred by `R`, whereas an expression `R` denotes the object itself. The above assignment thus means that the qualified reference `theForeman` is assigned a reference to the generated instance of `worker`.
- An assignment of the form `mary->theForeman` means that the state of the object referred by `mary` is enforced upon the state of the object referred by `theFore-`

man. This form of assignment is called *value assignment*. If X and Y are integer objects then X -> Y means that the value of X is assigned to the object Y.

In this section, it was shown how the most common object-oriented constructs may be expressed in BETA. In the following sections, examples of the more unique constructs will be given.

4 Singular Objects

Often there is only one object of a given type. In most languages it is necessary to make a class and generate a single instance. In BETA it is possible to describe a singular object directly. There is only one president of our company and he may be described as the following singular object:

```
   president: @employee(# computeSalary::< (# do BIG ->salary #) #)
```

The declaration `president` is similar to the declaration of `mary`. The difference is that in the declaration of `mary`, a pattern name (`worker`) describes the objects whereas a complete object description is used to describe the president.

The `president` object is an example of a singular data object corresponding to an instance of a class pattern. It is also possible to describe singular action objects corresponding to an instance of a procedure pattern. Examples of singular action objects are given below in section 6.

5 Subprocedure

The previous sections has shown examples of patterns used as classes and procedures. For class patterns, examples of subpatterns have been given. Subpatterns may also be used for procedure patterns. For attributes, subpatterns may add new attributes and extend definitions of virtual patterns in the superpattern. In addition, a subpattern may specify further imperatives which have to be combined with the imperatives of the superpattern. The combination of the imperatives is handled by the `inner` construct. Consider the following objects:

```
mutex: @semaphore; sharedVar: @integer
```

The variable `sharedVar` is shared by a number of concurrent processes. Mutual access to the variable is handled by the semaphore `mutex`. Update of `sharedVar` should then be performed as follows:

```
mutex.P; m+sharedVar -> sharedVar; mutex.V
```

Abstract procedure pattern

This pattern of actions must be used whenever `sharedVar` and other shared objects have to be accessed. Instead of manipulating the semaphore directly it is possible to encapsulate these operations in an abstract procedure pattern. The pattern `entry` can describe this encapsulation:

```
entry: (# do mutex.P; inner; mutex.V #)
```

Execution of `entry` locks `mutex` before the `inner` and releases it afterwards. `inner` may then in subpatterns of `entry` be replaced by arbitrary imperatives. The subpattern `updateShared` of `entry` updates `sharedVar`:

```
updateShared: entry
  (# m: @integer
  enter m
  do sharedVar+m-> sharedVar
  #)
```

Execution of an imperative

```
123 -> updateShared
```

will then result in execution of the actions

```
mutex.P; sharedVar+123->sharedVar; mutex.V
```

We may now define an abstract superpattern corresponding to a monitor:

```
monitor:
  (# mutex: @semaphore;
     entry: (# do mutex.P; inner; mutex.V #);
     init:< (# do mutex.V(*initially open*); inner #)
  #);
```

A (singular) `monitor` object may now be declared like `shared` below:

```
shared: @monitor
  (# var: @integer;
     update: entry(# m: @integer enter m do var+m->var #);
     get: entry(# v: @integer do var->v exit v #)
  #)
```

Semaphores are the basic mechanism in BETA for synchronization. They can express most synchronization problems, but may be complicated to use. It is therefore mandatory that high level abstraction mechanisms like monitor can be defined. In section 9 below, further details about concurrency in BETA will be given.

6 Control Patterns

Sub (procedure) patterns are used intensively in BETA for defining control patterns (control structures). This includes simple control patterns like `cycle`, `forTo`, etc. It also includes so-called iterators on data objects like `list`, `set` and `register`. A pattern describing a register of objects may have the following interface:

scan is a control pattern

```
register:
  (# has: (# E: ^type; B: @boolean enter E[] do … exit B #);
     insert: (# E: ^type enter E[] do … #);
     delete: (# E: ^type enter E[] do … #);
     scan: (# current: ^type do … inner … #);
     …
  #)
```

A number of details have been left out from the example. These include the representation and implementation of the `register`. A `register` may include instances of the pattern `type`, which has not been specified. `Type` is an example of a virtual class pattern which will be introduced later. For the moment `type` is assumed to stand for the pattern `object` which is a superclass of all patterns, i.e. a `register` may include instances of all patterns. An instance of `register` may be declared and used as follows:

```
employees: @register;
…
mary[]->employees.insert;
(if boss[]->employees.has then … if)
```

The control pattern `scan` may be used as follows:

Using a control pattern

```
0->totalSalary;
employees.scan
  (# do current.computeSalary+totalSalary->totalSalary #);
totalSalary->screen.putint
```

This works as follows:

- The imperative `employees.scan(# … #)` is an example of a singular action object as mentioned in section 4.
- The do-part of `scan` has an `inner` imperative which is executed for each element in the register. The details of this are not shown, but it may be implemented as a loop that steps through the elements of the register and executes `inner` for each element.
- The attribute `current` of `scan` is used as an index variable that for each iteration refers to the current element of the register. This may be implemented by assigning the reference of the current element to `current` before `inner` is executed.
- The effect of executing the above singular action object is that `current.computeSalary+totalSalary->totalSalary` is executed for each element in the register.

7 Nested Patterns

One of the characteristics of Algol-like languages is block-structure, which allows for arbitrary nesting of procedures. The possibility of nesting has been carried over to BETA where patterns can be arbitrarily nested. Block-structure is a powerful mechanism that extends the modeling capabilities of languages. However, besides Simula and BETA, none of the mainstream object-oriented languages supports block-structure. In most object-oriented languages, an object may be characterized by data attributes (instance variables) and procedure attributes. In BETA, an object may in addition be characterized by class pattern attributes.

BETA supports general block-structure

In the examples presented so far, there have been two levels of nesting. The outer level corresponds to class patterns, like `employee`, and the inner level corresponds to procedure patterns, like `computeSalary`. In procedural languages like Algol and Pascal it is common practice to define procedures with local procedures. This is also possible in BETA.

Nested Class Patterns

The possibility of nesting classes is a powerful feature which is not possible in languages like C++ and Eiffel. The following example shows a class pattern that describes a product of our company:

```
productDescription:
  (# name: @text;
     price: @integer;
     noOfSoldUnits: @integer;
     order:
       (# orderDate: @date;
          c: ^customer;
          print:<
            (#
            do name[] -> puttext;
               'Price: '->puttext; price -> putint; ' '->put;
               ' No of units sold: '->puttext;
               noOfSoldUnits->putint; ' '->put;
               orderDate.print;
               C.print;
               inner
            #)
       #)
  #);
```

One of the attributes of a `productDescription` object is the class pattern `order`. An instance of `order` describes an order made on this product by some customer. The attributes of an `order` object include the date of the order, the number of units ordered, the customer ordering the product, and a `print` operation. Consider the objects:

```
P1,P2: @product; o1,o2: @P1.order; o3,o4: @P2.order
```

The objects `o1` and `o2` are instances of `P1.order` whereas `o3` and `o4` are instances of `P2.order`. The block-structure makes it possible to refer to global names in enclosing objects. In the above example, the `print` operation refers to names in the enclosing `order` object. This resembles most object-oriented languages where operations inside a procedure refer to names in the enclosing object. The `print` operation, however, also refers to names in the surrounding `productDescription` object. Execution of say `o1.print` will thus print the values of `P1.name`, `P1.price`, `P1.noOfSoldUnits`, `o1.orderDate`, and `o1.c`.

8 Virtual Pattern

Structural equivalence is used in BETA

In the example in section 3 it was mentioned that a redefinition of a virtual procedure pattern is not a redefinition (overriding) as in C++. In fact a virtual pattern in BETA can only be extended and cannot be completely redefined. The rationale behind this is that a subpattern should have the same properties as its superpattern including which imperatives are executed. Ideally a subpattern should be *behaviorally equivalent* to its superpattern. This will, however, require a correctness proof. The subpattern mechanism of BETA supports a form of *structural equivalence* between a subpattern and its superpattern.

Consider the following patterns:

Patterns A and AA

```
A: (# V:< (# x: … do I1; inner; I2 #) #);
AA: A(# V::< (# y: … do I3; inner; I4#) #)
```

The pattern `A` has a virtual procedure attribute `V`. `V` has an attribute `x` and its do-part contains the execution of `I1; inner; I2`. The subpattern `AA` of `A` extends the definition of `V`. The extended definition of `V` in `AA` corresponds to the following object-descriptor (except for scope rules):

Combined descriptor

```
(# x: …; y: … do I1; I3; inner; I4; I2 #)
```

As may be seen the `V` attribute of `AA` has the attributes `x` and `y` and the do-part consists of `I1; I3; inner; I4; I2`. The definition of `V` is an extension of the one from `A` and *not* a replacement.

The subpattern `AB` of `A` describes another extension of `V`:

Pattern AB

```
AB: A(# V::< (# z: … do I5; inner; I6 #) #)
```

Here `V` corresponds to the following object descriptor:

```
V: (# x: …; z: … do I1; I5; inner; I6; I2 #)
```

The definition of `V` may be further extended in subpatterns of `AA` also as shown in the definition `AAA`:

Pattern AAA

```
AAA: AA(# V::< (# q: … do I7; inner; I8 #) #)
```

The definition of `V` corresponds to the following object-descriptor:

```
V: (# x: …; y: …; q: … do I1; I3; I7; inner; I8; I4; I2 #)
```

As may be seen, the pattern `V` is a combination of the definitions of `V` from `A`, `AA` and `AAA`.

Final binding

The definition of `V` may be extended using a final binding (::) in subpatterns of `A` as shown in the definition `AC`:

```
AC: A(# V::(# q: … do I2; inner; I4 #) #)
```

The final binding of `V` means that `V` cannot be extended in subpatterns of `AC`. The extended definition of `V` in `AC` corresponds to the following object-descriptor (except for scope rules):

```
(# x: …; y: … do I1; I3; inner; I4; I2 #)
```

The virtual mechanism in BETA guarantees that behavior defined in a superpattern cannot be replaced in a subpattern. This form of structural equivalence is useful when defining libraries of patterns that are supposed to execute a certain sequence of actions. In C++, the programmer must explicitly invoke the actions from the superclass by means of superclass::functionname. This is illustrated by the example in the next section.

The inner construct is more general than shown above, since a pattern may have more than one inner and inner may appear inside control structures and nested singular object descriptors.

8.1 Virtual Procedure Pattern

The attribute computeSalary of pattern employee is an example of a virtual procedure pattern. In this example the do-part of the virtual definition in employee is very simple, only consisting of an inner-imperative. The extended definitions of computeSalary in worker and salesman both include the code noOfHours*80 and 0->totalHours. This code may instead be defined in the definition of computeSalary in employee as shown below:

ComputeSalary is a virtual procedure pattern

```
employee:
  (# …
     computeSalary:<
       (# salary: @integer
       do noOfHours*80->salary; inner; 0->totalHours
       exit salary
       #)
  #);
  worker: employee
    (# …
       computeSalary::< (# do seniority*4+salary->salary; inner #)
    #);
  salesman: employee
    (# …
       computeSalary::<
         (#
         do noOfSoldUnits*6+salary ->salary;
            0 ->noOfSoldUnits;
            inner
         #)
    #)
```

The extended definitions of computeSalary in worker and salesman have an inner to enable further extensions of computeSalary in subpatterns of worker and salesman.

8.2 Virtual Class Pattern

Virtual patterns may also be used to parameterize general container patterns such as the register pattern described above. For the register pattern we assumed the existence of a type pattern defining the elements of the register, i.e. elements of a register must be instances of the pattern type. The pattern type may be declared as a virtual pattern attribute of register as shown below:

type is a virtual class pattern

```
register:
  (# type:< object;
     insert:< (# e: ^type enter e[] do …#)
     …
  #)
```

The declaration type:< object specifies that type is either the pattern object or some subpattern of object. In the definition of register, type may be used as an alias for object, e.g. references qualified by type are known to be at least Objects. Since object is the most general superpattern, type may potentially be any other pattern. The virtual attribute type may be bound to a subpattern of object in subpatterns of register. The following declaration shows a pattern workerRegister which is a register where the type attribute has been bound to worker:

```
workerRegister: register
  (# type::< worker;
     findOldestSeniority:
       (# old: @integer
       do scan
           (# do (if current.seniority > old then
                         current.seniority->old
           if)#)
       exit old
       #)
  #);
```

In the definition of `workerRegister`, the virtual pattern `type` may be used as a synonym for the pattern `worker`. This means that all references qualified by `type` may be used as if they were qualified by `worker`. The reference `current` of the `scan` operation is used in this way by the operation `findOldestSeniority` which computes the oldest seniority of the register. The expression `current.seniority` is legal since `current` is qualified by type which in `workerRegister` is at least a `worker`.

In subpatterns of `workerRegister` it is possible to make further bindings of `type` thereby restricting the possible members of the register. Suppose that `manager` is a subpattern of `worker`. A `manager` register may then be defined as a subpattern of `workerRegister`:

```
managerRegister: workerRegister(# type::< manager #)
```

In the definition of `managerRegister`, `type` may be used as a synonym for `manager`, i.e. all references qualified by `type` are also qualified by `manager`.

General parameterized patterns

Virtual patterns make it possible to define general parameterized patterns like `register` and to restrict the member type of the elements. In this way virtual class patterns provide an alternative to templates as found in C++.

9 Coroutines and Concurrency

A BETA object may be the basis for an execution thread. Such a thread will consist of a stack of objects currently being executed. An object which can be used as the basis for an execution thread has to be declared as an object of kind component as shown in the following declaration:

Components with execution threads

```
A: @| activity
```

The symbol "`|`" describes that the object `A` is a component. A component (thread) may be executed as a coroutine or it may be forked as a concurrent process. Consider the following description of `activity`:

```
activity:
  (#
  do cycle
      (#
      do getOrder; suspend;
         processOrder; suspend;
         deliverOrder; suspend
  #)#)
```

The component object may be invoked by an imperative

Coroutine execution

```
A
```

which implies that the do-part is executed. The execution of `A` is temporarily suspended when `A` executes a suspend-imperative. In the above example this happens after the execution of `getOrder`. A subsequent invocation of `A` will resume execution after the suspend-imperative. In the above example this means that `processOrder` will be executed. If `B` is also an instance of `activity`, then the calling object may alternate between executing `A` and `B`:

```
cycle(# do A; … B; … #)
```

The above example shows how to use components as deterministic coroutines in the sense that the calling object controls the scheduling of the coroutines. In section 9.1 below another example of using coroutines will be given.

It is also possible to execute component objects concurrently. By executing

Concurrent execution

```
A[]->fork; B[]->fork
```

the component objects `A` and `B` will be executed concurrently. As for the deterministic coroutine situation, `A` and `B` will temporarily suspend execution when they execute a suspend-imperative. Further examples of concurrent objects will be given below in section 9.2.

9.1 Coroutines

Deterministic coroutines have demonstrated their usefulness through many years of usage. Below we give a typical example of using coroutines.

Suppose we have a register for the permanent workers and another one for the hourly paid workers. Suppose also that it is possible to sort these registers according to a given criterion like the total hours worked by the employee. Suppose that we want to produce a list of names of all employees sorted according to the total hours worked. This may be done by merging the two registers. A `register` object has a `scan` operation that makes it possible to go through all elements of the register. Instead we define an operation of `register` in the form of a coroutine `getNext`, which delivers the next element of the register when called:

```
register:
  (# …
     getNext: | @
       (# elm: ^employee
       do scan(# do current[]->elm[]; suspend #);
          none->elm[]
       exit elm[]
       #);
  #);
  pReg: @permanentRegister; hReg: @hourlyPaidRegister;
  …
  pReg.getNext->e1[]; hReg.getnext->e2[];
  L: cycle
    (#
    do (if e1[] = none then (*empty hReg*); leave L if);
       (if e2[] = none then (*empty pReg*); leave L if);
       (if e1.totalHours < e2.totalHours then
           e1.print; pReg.getNext->e1[]
        else
           e2.print; hReg.getNext->e2[]
       if)
    #)
```

Suspending and resuming

The attributes `getNext` of the objects `pReg` and `hReg` have their own thread of execution. When called in an imperative like `pReg.getNext->e1[]`, the thread is executed until it either executes a `suspend` or terminates. If it executes a `suspend`, it may be called again in which case it will resume execution at the point of `suspend`. The first time `getNext` is called, it will start executing `scan`. For each element in the register, it will suspend execution and exit the current element via the exit variable `elm[]`. When the register is empty, `NONE` is returned.

9.2 Concurrency

As previously mentioned, it is possible to perform concurrent execution of components by means of the fork operation as sketched in the following example:

```
(# S1: @| (# … do … #);
   S2: @| (# … do … #);
   S3: @| (# … do … #)
  do S1[] -> fork; S2[] -> fork; S3[] -> fork; …
#)
```

The execution of `S1`, `S2` and `S3` will take place concurrently with each other and with the object executing the `fork` operations. Concurrent objects may access the same shared objects without synchronization, but may synchronize access to shared objects by means of semaphores. In section 5 above the pattern `semaphore` has been described. It is well known that a semaphore is a low level synchronization mechanism which may be difficult to use in other than simple situations. For this reason the Mjølner BETA library has a number of patterns defining higher level synchronization mechanisms. This library includes a `monitor` pattern as described in section 5 above. The library also includes patterns defining synchronization in the form of rendezvous as in Ada.

Monitor Example

The following example describes a company with a number of salesmen, workers and carriers. The salesmen obtain orders from customers and store them in an order pool. The workers obtain orders from the order pool, process them and deliver the resulting item in an item pool. The carriers pick up the items from the item pool and bring them to the customer. Salesmen, workers and carriers are described as active objects whereas the order- and item pools are represented as monitor objects.

```
(# salesman: employee
     (# getOrder: (# … exit anOrder[] #)
     do cycle (# do getOrder -> jobPool.put #)
     #);
   S1,S2, …: @|salesman;
   jobPool: @monitor
     (# jobs: @register(# type::< order #);
        put: entry
          (# ord: ^order enter ord[] do ord[] ->jobs.insert #);
        get: entry
          (# ord: ^order do jobs.remove -> ord[] exit ord[] #)
     #);
   worker: employee
     (# processJob: (# … enter anOrder[] do … exit anItem[] #)
     do cycle(# do jobPool.get -> processJob -> itemPool.put #)
     #);
   W1,W2,…: @| worker;
   itemPool: @monitor(# … #);
   carrier: employee
     (# deliverItem: (# enter anItem[] do … #)
     do cycle(# do itemPool.get ->DeliverItem #)
     #);
   C1,C2, …: @| carrier;
do jobPool.init; itemPool.init;
   conc(# do S1[]->start; … W1[]->start; … C1[]->start; … #)
#)
```

Procedure pattern conc

The procedure pattern conc is another example of a high-level concurrency pattern from the Mjølner BETA library. It does not terminate execution until components being started (by S1[]->start, etc.) have terminated their execution.

Rendezvous Example

Next we show an example of using the library patterns for describing synchronized rendezvous. The example shows a drink machine that provides coffee and soup. A customer operates the machine by pushing either makeCoffee or makeSoup. If make-Coffee has been pushed, then the customer may obtain the coffee by means of get-Coffee. Similarly if makeSoup has been pushed then the soup may be obtained by means of getSoup.

The system pattern has a port attribute which may be used to define synchronization ports. The drink machine described below has three such ports, activate, cof-feeReady, and soupReady. A port object has a pattern attribute entry which may be used to define procedure patterns associated with port. For the port activate, two procedure patterns makeCoffee and makeSoup are defined. For coffeeReady and soupReady, the procedure patterns getCoffee and getSoup are defined.

An execution of a port-entry operation like aDrinkMachine.makeCoffee will only be executed if the drinkMachine has executed a corresponding accept by means of ac-tivate.accept.

- Initially a drinkMachine is ready to accept either makeCoffee or makeSoup.
- If e.g. makeCoffee is executed, then when “the coffee has been made”, the drinkMachine is willing to accept the operation getCoffee. This is signaled by executing an accept on the port coffeeReady. Technically this is implemented by assigning a reference to coffeeReady to the port reference drinkReady. The do-part of drinkMachine then makes an accept on drinkReady.
- When the operation getCoffee, has been executed, the drinkMachine is again ready to accept a new operation associated with the activate port.

```
drinkMachine: system
  (# activate: @port;
     makeCoffee: activate.entry
       (# do … coffeeReady[]->drinkReady[] #);
    makeSoup: activate.entry(# do … soupReady[]->drinkReady[] #);
    coffeeReady, soupReady: @port;
    getCoffee: coffeeReady.entry(# do … exit someCoffee [] #);
    getSoup: soupReady.entry(# do … exit someSoup [] #);
    drinkReady: ^port
  do cycle(# do activate.accept; drinkReady.accept #)
  #)
```

The `drinkMachine` may be used in the following way:

```
aDrinkMachine: @| drinkMachine
…
aDrinkMachine.makeCoffee; … aDrinkMachine.getCoffee;
aDrinkMachine.makeSoup; … aDrinkMachine.getSoup;
```

As may be seen the use of the patterns `system`, `port` and `entry` makes it possible to describe a concurrent program in the style of Ada tasks that synchronize their execution by means of rendezvous. A `port` object defines two semaphores for controlling the execution of the associated entry patterns. The actual details will not be given in this language introduction.

It is possible to specialize the `drinkMachine` into a machine that accepts further operations:

```
extendedDrinkMachine: drinkMachine
  (# makeTea: activate.entry(# do … teaReady[]->drinkReady[] #);
     teaReady: @port;
     getTea: teaReady.entry(# … exit someTea[] #)
  #)
```

The `extendedDrinkMachine` inherits the operations and protocol from `drinkMachine` and adds new operations to the protocol.

The basic mechanisms in BETA for providing concurrency are component-objects (providing threads), the fork-imperative (for initiating concurrent execution) and the semaphore (for providing synchronization). As has been mentioned already, these mechanisms are inadequate for many situations. The abstraction mechanisms of BETA make it possible to define higher-level abstractions for concurrency and synchronization.

More information

Please see the manual [MIA 90-8] for details about the concurrency library.

10 Inheritance

The subpattern mechanism combined with the possibility of redefining/extending virtual procedures is widely recognized as a major benefit of object-oriented languages. This mechanism is often called inheritance since a subpattern is said to inherit properties (code) from its superpattern. Inheritance makes it easy to define new patterns from other patterns. In practice this has implied that subpatterns are often used for sheer inheritance of code without any concern for the relation between a pattern and its subpatterns in terms of generalization/specialization. The use of multiple inheritance is in most cases justified in inheritance of code and may lead to complicated inheritance structures.

Classification anc inheritance

In BETA subpatterns are intended for representing classification and inheritance of code is a (useful) side effect. In BETA it is not possible to define a pattern with multiple superpatterns corresponding to multiple inheritance. There are indeed cases where it is useful to represent classification hierarchies that are not tree structured. However, a technical solution that justifies the extra complexity has not yet been found.

Inheritance from part-objects

BETA does support multiple inheritance, but in the form of inheritance from part-objects. A compound object inherits from its parts as well as its superpattern. The reason that this has not been more widely explored/accepted is that in most languages inheritance from part-objects lacks the possibility of redefining/extending virtual procedures in the same way as for inheritance from superpatterns. Block-structure and singular objects make this possible in BETA.

Assume that we have a set of patterns for handling addresses. An address has properties such as street name, street number, city, etc., and a virtual procedure for printing the address. In addition we have a pattern defining an address register.

```
address:
  (# streetName: @text; streetNo: @integer; city: @text; …
     print:<
       (#
       do inner;
          streetName->puttext;
          streetNo->putint; (*etc.*)
       #);
  #);
addressRegister: register(# element::< address #)
```

We may use the `address` pattern for defining part-objects of `employee`/`company` objects:

```
employee:
  (# name: @text; {the name of the employee*)
     adr: @address(# print:: (# do name->puttext #)#)
  #);
company:
  (# name: @text; (*the name of the company*)
     adr: @address(# print:: (# do name->puttext #) #)
  #);
```

The object `adr` of `employee` is defined as a singular `address` object where the virtual `print` pattern is defined to print the name of the `employee`. As can be seen it is possible to define a part-object and define its virtual procedures to have an effect on the whole object. The `company` pattern is defined in a similar way.

It is possible to handle the address aspect of employees and companies. An example is an address register:

```
AReg: @addressRegister;
…
employee1.adr[]->AReg.insert; employee2.adr[]->AReg.insert;
company1.adr[]->AReg.insert; company2.adr[]->AReg.insert;
AReg.scan(# do current.print #)
```

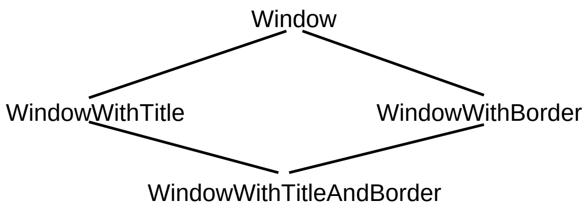
The `AReg` register will contain `address` objects which are part of either `employee` objects or `company` objects. For the purpose of the register this does not matter. When the `print` procedure of one of these `address` objects is invoked it will call the print procedure associated with either `employee` or `company`. The scanning of the `AReg` register is an example of invoking the `print` pattern.

The example shows that in BETA inheritance from part-objects may be used as an alternative to inheritance from superpatterns. The choice in a given situation depends of course on the actual concepts and phenomena to be modeled. In the above example it seems reasonable to model the address as a part instead of defining `employee` and `company` as specializations of `address`.

Multiple inheritance using part-objects

In general it is possible to specify multiple inheritance from part-objects since it is possible to have several part-objects like the `address` object above. This form of multiple inheritance provides most of the functionality of multiple inheritance from C++ and Eiffel. It is simpler since the programmer must be explicit about the combination of virtual operations. It does, however, not handle so-called overlapping superclasses. The programmer must also explicitly redefine the attributes of the component classes. This may be tedious if there is a large number of attributes. However, a renaming mechanism for making this easier has been proposed for BETA, but it is not yet implemented in the Mjølner BETA System. Multiple inheritance from part-objects should be used when there is a part-of relationship between the components and the compound. This also covers situations where implementations are inherited. It should not be used as a replacement for multiple specialization hierarchies.

A common example of using multiple inheritance is modeling windows with titles and borders. This may be modeled using block-structure. Since a window may have a title, a border or both, the following class hierarchy using multiple inheritance is often used:



In BETA this can be described using nested patterns:

```
window:
  (# title: (# … #);
     border: (# … #);
   …
  #);
aWindow: @window(# T: @title; B: @border #)
```

The descriptions for `title` and `border` are made using nested patterns. For a given window, like `aWindow`, a `title` object and a `border` object may be instantiated. If e.g. two titles are needed, two instances of `title` are made. This example illustrates another situation where multiple inheritance may be avoided.

Appendix
BETA Quick Reference Card

A summary of all special characters in BETA, and a short list of the syntax of the language is given below along with a short description of their semantics:

Special characters	Semantics
`:`	Pattern Declaration
`: @`	Static object reference declaration
`: ^`	Dynamic object reference declaration
`: ##`	Pattern reference declaration
`: @|`	Static component declaration
`: ^|`	Dynamic component declaration
`: [range]`	Declaration of repetition. `range` must be an integer evaluation
`:<`	Virtual declaration
`::<`	Extended binding of virtual declaration
`::`	Final binding of virtual declaration
`&`	Dynamic creation of item; new
`&|`	Dynamic creation of component
`->`	Assignment
`[]`	Object reference
`##`	Pattern reference
`(#`	Object descriptor begin
`#)`	Object descriptor end
`//`	Selection in if-imperative
`do`	Beginning of action part
`enter exit inner` `leave restart` `(if if) then else` `(for for) repeat` `none this suspend`	Additional keywords (for their usage, see below)

Short syntax	Semantics
`P: (# … do … #)`	Definition of a pattern
`PP: P(# … do … #)`	Definition of a subpattern
`enter …`	Specification of enter-parameters
`exit …`	Specification of exit-parameters
`inner P`	Execute the actions in the subpattern. P is an optional name of an enclosing pattern.
`this(P)`	Denotation of this object
`this(P)[]`	Reference to this object
`E.P`	Remote name
`(E).P`	Computed remote name
`L: Imp`	In action part: labeled imperative
`L: (# … do … #)`	In action part: labeled imperative
`leave L`	Terminate labeled imperative or object instance L
`restart L`	Goto beginning of labeled imperative or object instance L
`suspend`	Component suspension
`E1 -> E2`	Assignment imperative

`(if E then Imp1`
` else Imp 2`
`if)`
	Simple if:
	Evaluation of E, Imp1 is executed if E is true, otherwise Imp2' is executed. ‘else Imp2’ is optional

`(if E`
` // E1 then Imp1`
` // En then Impn`
` else Imp`
`if)`
	General selection imperative:
	Sequential evaluation of E, E1, … En. First Impi is executed where Ei=E If no Ei=E, then Imp is executed 'else Imp' is optional

`(for range repeat`
`     Imp`
`for)`
	Repetition imperative:
	Execute Imp 1..range] times

`(for I: range repeat`
`     Imp`
`for)`
	Repetition imperative:
	I is a locally scoped integer variable within Imp. Execute Imp with I assigned each value in [1..range]

`NONE`	The nil reference value
`R[i:j]`	Repetition slice
`R[i]`	Indexed repetition element
`(e1, e2, …, en)`	Evaluation list

Please note, that the above description is by no means complete, and in some cases ambiguous. The ultimate reference is naturally the BETA grammar as defined in the BETA book [Madsen 93].

References

[Knudsen 94]	J. L. Knudsen, M. Löfgren, O. L. Madsen, B. Magnusson (eds.): *Object-Oriented Environments – The Mjølner Approach*, Prentice Hall, 1994, ISBN 0-13-009291-6.
[Madsen 93]	O. L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993, ISBN 0-201-62430-3
[MIA 90-2]	Mjølner Informatics: *The Mjølner BETA System: BETA Compiler Reference Manual* Mjølner Informatics Report MIA 90-2.
[MIA 90-4]	Mjølner Informatics: *The Mjølner BETA System: Using BETA on UNIX Systems*, Mjølner Informatics Report MIA 90-4.
[MIA 90-6]	Mjølner Informatics: *The Mjølner BETA System: Using BETA on the Macintosh*, Mjølner Informatics Report MIA 90-6.
[MIA 90-8]	Mjølner Informatics: *The Mjølner BETA System: Basic Libraries, Reference Manual*, Mjølner Informatics Report MIA 90-8
[MIA 91-16]	Mjølner Informatics: *The Mjølner BETA System—X Window System Libraries*, MjølnerInformatics Re port MIA 91-16.
[MIA 91-20]	Mjølner Informatics: *The Mjølner BETA System – Persistent Store*, MjølnerInformatics Report MIA 91-20.
[MIA 92-22]	Mjølner Informatics: *The Mjølner BETA System – Container Libraries, Reference Manual*, MjølnerInformatics Report MIA 92-22.
[MIA 94-25]	Mjølner Informatics: *The Mjølner BETA System – Distribution* MjølnerInformatics Report MIA 94-25.
[MIA 94-26]	Mjølner Informatics: *The Mjølner BETA System – BETA Language Introduction* MjølnerInformatics Report MIA 94-26.
[MIA 94-27]	Mjølner Informatics: *The Mjølner BETA System – GUIEnv Libraries* MjølnerInformatics Report MIA 94-27.

Index