: An Overview of BETA

Table of Contents

Copyright Notice 1

1 Introduction 3
Powerful abstraction mechanisms 3

2 Patterns and Objects 6

3 Singular objects 9

4 Subprocedure 10

5 Control patterns 12

6 Nested Patterns 13

7 Virtual Pattern 15

7.1 Virtual procedure pattern 17

7.2 Virtual class pattern 18

8 Multiple Threads 20

8.1 Coroutines 22

8.2 Concurrency 23

8.2.1 Monitor example 24

8.2.2 Rendezvous example 25

9 Inheritance 27

10 Other Issues 30

11 References 31

Copyright Notice

Mjølner Informatics Report
August 1999

1 Introduction

BETA is a modern object-oriented language from the Scandinavian school of object-orientation where the first object-oriented language Simula [DMN70] 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 [Mey88] and C++ [Str86], with most type checking being carried out at compile-time. It is well known that it is not possible to obtain all type checking a compile time without sacrificing the expressiveness of the language. BETA has an optimum balance between compile-time type checking and run-time type checking.

The purpose of this paper is to present an overview of the BETA language and give examples on the use of most constructs. The reader is assumed to be familiar with one or more object-oriented languages like Simula, Eiffel, C++, or Smalltalk [GR83]. For more details about BETA see [KMMN83,KKMN87,MMN92].

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 enhances reusability of designs and implementations.

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 applies 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.

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.

The notion of virtual pattern covers virtual procedures like in Simula, Eiffel and 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++.

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 my be assigned different procedures). Since patterns may be used as classes, it is also possible to have variables that can be assigned classes, etc.

BETA does not only allow for passive objects as in Smalltalk, 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 *Ada-like rendezvous*. The user may easily define new concurrency abstractions including schedulers for processes.

BETA supports the three main subfunctions of abstraction:

Identification of 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.

Classification.

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

Composition (aggregation).

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

- *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 composition*: An attribute may be a reference to another object. Reference composition forms the basis for modeling arbitrary relations between objects.
- *Localization (block-structure)*: An attribute of an object may be a (nested) pattern. This is known from Algol 60 as *block-structure*. The block-structure makes it easy to create arbitrary nested pattern. This makes it possible for objects to have local patterns used as classes, procedures, etc. Local patterns greatly enhances the modelling capabilities of an object-oriented language.

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.

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 have been just as important as the language itself. For a description of the conceptual framework, see [KM93]

2 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. Subclass is called deferred class in Eiffel and derived class in C++. Virtual procedure is called deferred routine in Eiffel, and virtual function in C++. Qualified reference is called reference field in Eiffel, and reference in C++. The mentioned constructs are also the basic elements of Smalltalk. Virtual procedure correspond to method, and qualified reference to instance variable. For the latter a major difference is that instance variables have no qualification (type). In addition to virtual procedures, Simula, C++, and BETA have non-virtual procedures .

In this section, the BETA version of the above constructs will be described and compared with other languages using the Simula terminology. 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.

```
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 has the following meaning:

- 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 objects. A part-object is a fixed part of its enclosed object and is generated together with the enclosing object. Simula and Smalltalk does not have part objects, whereas Eiffel and C++ have some support.
- `Dept` is a dynamic reference that either has the value `NONE` or refers to a separate instance of the pattern `Department`. A dynamic reference is similar to a qualified reference in Simula.
- 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 procedure attributes in Simula.
- `ComputeSalary` is a virtual pattern attribute. 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 procedure in Simula.
- `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.

```
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 symbols `::<` describes that the definition of `ComputeSalary` from the superpattern `Employee` is extended. The extension of a virtual pattern is in this example similar to redefining a virtual procedure in Simula. Note, however, that the virtual concept of BETA differs in an important way from that of most object-oriented languages. Further details are given below.

The above examples have shown examples of instantiating 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
```

The above examples have also shown an example of a dynamic reference (like `dept: ^Department`). Such a reference is initially `NONE`, i.e. it refers to no object. A dynamic reference to instances of `Worker` may be declared as followed:

```
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 constant 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 `theForeman`. This form of assignment is called value assignment. In this paper it will only be used for instances of simple patterns like `Integer`. 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 OO constructs may be expressed in BETA. In the following sections, examples of the more unique constructs will be given.

---

3 Singular objects

Often there is only one object of a given type. In most languages it is still 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. Singular action objects are similar to blocks in Algol 60 and Simula. Examples of singular action objects are given below in section 5.

4 Subprocedure

The previous section has showed 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 `shared` should then be performed as follows:

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

This pattern of actions must be used whenever `shared` 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. Consider the following pattern `entry`:

```
 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. Consider the following subpattern of `entry`:

```
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 as follows:

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

Semaphore is 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 8 below, further details about concurrency in BETA will be given.

5 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:

```
Register:
  (# has: (# E: ^ type; B: @ boolean enter E[] do ... exit B #);
     insert: (# E: ^type enter E[] do ... #);
     delete: (# E: ^type enter E[] do ... #);
     scan: (# thisElm: ^ type do ... inner ... #);
     ...
  #)
```

A number of details, has been left out from the example. This includes 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 below. 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
   ...
do mary[]->employees.insert;
   (if boss[]->employees.has//true then ... if)
```

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

```
0->totalSalary;
employees.scan(#do thisElm.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 3.
- The do-part of scan has an *inner* imperative which is executed for each element in the register. The details of this is 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 `thisElm` 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 `thisElm` before *inner* is executed.
- The effect of executing the above singular action object is that `thisElm.computeSalary->S` is executed for each element in the register.

6 Nested Patterns

One of the characteristics of Algol-like languages is block-structure, which allows for arbitrary nesting of procedures. In Simula it is in addition possible to nest classes although there are some restrictions when using nesting. 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 main stream 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 Simula and BETA, an object may in addition be characterized by class pattern attributes.

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 of course also possible in BETA.

Nested class patterns

The possibility of nesting classes is a powerful feature which is not possible in languages like Smalltalk, 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 {print name, price, noOfSoldUnits, orderDate, C}
             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 no. of units ordered, the customer ordering the product, and a `Print` operation. Consider the objects:

```
P1: @ Product;
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 is like in most object oriented languages where one inside a procedure refers 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`.

---

7 Virtual Pattern

In the example in section 2, it was mentioned that a redefinition of a virtual procedure pattern is not a redefinition (overriding) as in Simula. 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 that are executed. Ideally a subpattern should be behavioral 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:

```
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):*

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

As it 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:

```
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:

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

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 Smalltalk the programmer must explicitly invoke the actions from the superpattern by means of `super`. 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. In Simula there may only one *inner* in a class and it must appear at the outermost level of imperatives.

---

7 Virtual Pattern

7.1 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 includes the code `noOfHours*80` and`0->`*`totalHours`*. This code may instead be defined in the definition of `ComputeSalary` in `Employee` as shown below:

```
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 `Salesmen` have an *inner* to make it possible to make further extensions of `ComputeSalary` in subpatterns of `Worker` and `Salesman`.

7.2 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:

```
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 know 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 thisElm.seniority > old
           // true then thisElm.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 `thisElm` of the `scan` operation is used in this way by the operation `findOldestSeniority` which computes the oldest seniority of the register. The expression `thisElm.seniority` is legal since `thisElm` 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`.

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 provides an alternative to generic classes as found in Eiffel. A further discussion of virtual class patterns may be found in [MM 89].

8 Multiple Threads

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:

```
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          suspend;GetOrder;
        ProcessOrder; suspend;
        DeliverOrder; suspend
  #)#)
```

The component object may be invoked by an imperative

```
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 8.1 below another example of using coroutines will be given.

It is also possible to execute component objects concurrently. By executing
`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 8.2.

8 Multiple Threads

8.1 Coroutines

Deterministic coroutines have demonstrated their usefulness through many years of usage in e.g. Simula. In [DH72] many examples are given. Further examples of coroutines in BETA may be found in [KKMN88] and [MMN92]. 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 its is possible to sort these registers according to a given criteria like the total hours worked by the employee. Suppose that we want to produce a list of names of all employees sorted according 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 thisElm[]->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// truethen
     e1.print; pReg.getNext->e1[]
     elsee2.print; hReg.getNext->e2[]
     if)
  #)
```

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.

8.2 Concurrency

As previously mentioned, it is possible to perform concurrent execution of components by means of the fork operations 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 4 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 4 above. The library also includes patterns defining synchronization in the form of rendezvous in CSP [Hoa74] and Ada [ADA82].

8.2.1 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; ... #)
#)
```

The procedure pattern `conc` is another example of a high-level concurrency pattern from the Mjølner BETA library. It corresponds to the parbegin/parend imperative of Dijkstra [Dij68] in the sense that it does not terminate execution until execution of components being started (by `S1.start`, etc.) have terminated their execution.

8.2.2 Rendezvous example

The next example shows 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 `makeCoffe` or `makeSoup`. If `makeCoffe` has been pushed, then the customer may obtain the coffee by means of `getCoffee`. 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`, `coffeeReady`, and `soupReady`. A `port` object has a pattern attribute `entry` which may be used to define procedure patterns associated with `port`. For the `portactivate`, 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 `activate.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 signalled 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 `getCoffe` 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
   ...
do aDrinkMachine.makeCoffee; aDrinkMachine.getCoffee;
   aDrinkMachine.makeSoup; aDrinkMachine.getSoup;
```

As it may bee 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 paper.

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 `DrinkMachi-ne` 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 it has been mentioned already, these mechanism are inadequate for many situations. The abstraction mechanisms of BETA makes it possible to define higher-level abstractions for concurrency and synchronization. In this paper some examples have been given. Many researchers have proposed several alternative mechanisms for handling concurrency and synchronization. The motivation for this is due to problems with current proposals.

---

9 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 is 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.

In BETA subpatterns is 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 subpatterns 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.

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 makes this possible in BETA.

Assume that we have a set of patterns for handling addresses. An address has properties such as, street name, street no., 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`- and `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 it 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:

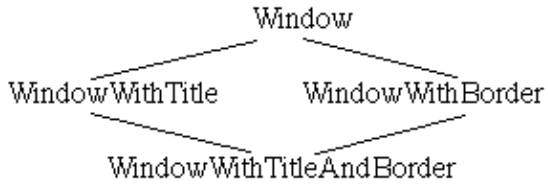
```
   Areg: @ AddressRegister;
   ...
do employee1.adr[]->Areg.insert; employee2.adr[]->Areg.insert;
   company1.adr[]->Areg.insert; company2.adr[]->Areg.insert;
   Areg.scan(#do thisAdress.print #)
```

The `Areg` register will contain `Address` objects which are either part of `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 withe 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 modelled. In the above example it seems reasonable to model the address as a part instead of defining `Employe` and `Company` as specializations of `Address`.

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. 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 more detailed discussion of using part objects for inheritance may be found in [MM 92].

In [Øst90] it is shown how a common example of using multiple inheritance for modeling windows with titles and borders may be handled 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 [Øst90] its is shown how such windows may 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. For details see [Øst90]. This

examples illustrates that another situation where multiple inheritance may be avoided.

---

10 Other Issues

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 paper, is for describing objects and patterns. The objects and patterns constitute the logical structure of a program execution. This physical structure of a program execution is handled by other components of the Mjølner BETA System which is a programming environment supporting BETA. The Mjølner BETA System provides the following:

- Mechanisms for splitting a large BETA program into a set of modules.
- Mechanisms for protecting the attributes of an object like hidden, protected and private in Simula and C++. In addition there are mechanisms for physically separating the interface of a pattern from its implementation
- Mechanisms for defining alternative implementations/variants of a pattern.
- Mechanisms for defining which objects are persistent and which are transient
- Mechanisms for assigning active objects to processors on a computer system. I.e. handling distribution.

For a description of the Mjølner BETA mechanism for modularization, including separation of interface and implementation and alternative implementations, see [Mad92]. For a description of the support for persistent objects, see [AFO93].

Acknowledegement. Jørgen Lindskov Knudsen has given many useful comments on this paper.

11 References

[Ada82] Ada Joint Program Office, United States Department of Defense. Reference Manual for the Ada Programming Language, 1982.

[AFO92] O. Agesen, S. Frølund, M. H. Olsen: Language Support Level for Persistence in BETA. In [LKMM93]

[DH72] O.-J. Dahl, C.A.R. Hoare: Hierarchical Program Structures. In Dahl, Dijkstra, Hoare: Structured Programming, Academic Press, 1972.

[DMN70] O.-J. Dahl, B. Myrhaug, K. Nygaard: Simula 67, Common Base Language. Technical Publication No. S-22, Norwegian Computing Center, 1970.

[Dij68] E. Dijkstra: Co-operating Sequential Processes. In: Genuys (ed.): Programming Languages, Academic Press, 1968

[GR83] A. Goldberg, D. Robson: Smalltalk-80: The Language and its Implementation, Addison Wesley, 1983.

[Hoa74] C.A.R. Hoare: Monitors: An Operating System Structuring Concept. Comm. ACM, Oct. 1974, Vol. 17, No. 10, pp. 549-557

[KM93] J.L. Knudsen, O.L. Madsen: A Conceptual Framework for Object-Oriented Programming. In [LKMN93] and [MMN92].

[KMMN83] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Abstraction Mechanisms in the BETA programming Language. POPL 1983.

[KMMN87]B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language. In Shriver, Wegner (eds.): Research Directions in Object-Oriented Programming. MIT Press, 1987.

[KMMN88] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Couroutine Sequencing in BETA. Hawaii International Conference on System Sciences - 21, Jan. 5-8, 1988.

[LKMM93] M. Løfgren, J.L. Knudsen, O.L. Madsen, B. Magnusson (eds.): Object-Oriented Software Development Environments - The Mjølner Approach, Prentice Hall, 1993, to appear.

[Mad92] O.L. Madsen: The Mjølner BETA Fragment System. In [LKMM93] and [MMK92].

[MM89] O.L. Madsen, B. Møller-Pedersen: Virtual Classes - A powerful mechanism in object-oriented programming. OOPSLA 90.

[MM92] O.L. Madsen, B. Møller-Pedersen: Part Objects and Their Location. TOOLS'7, Dortmund 1992.

[MMN92] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object-Oriented Programming in the BETA programming Language, Computer Science Department, Aarhus University, August 1992, Addison Wesley, 1993, to appear.

[Mey88] B. Meyer: Object-Oriented Software Construction., Prentice Hall 1988.

[Nau60] P. Naur (ed.): Revised Report on Algol 60, A/S Regnecentralen 1960.

[Str86] B. Stroustrup: The C++ Programming Language. Addison Wesley, 1986.

[Øst90] K. Østerby: Parts, Wholes, and Sub-classes. In Smith (ed.): Proceedings of the European Simulation Multiconference, 1990, ISBN 0-911801-1.
