: BETA Terminology

Table of Contents

Copyright Notice

Mjølner Informatics Report
August 1999

BETA Terminology

The following is a short description of important concepts used in the BETA language. Please note, that these descriptions are deliberately informal. The precise meanings of these terms must be found in [MMN 93].

Contents

Modelling

Object-oriented programming

A *program execution* is viewed as a *physical model* or *representation* of part of the world. *Objects* on the computer model phenomena in the world; *attributes* of objects model properties of phenomena.

Computer
Real world
Object
Phenomenon
Attribute
Property
Pattern
Concept

BETA program execution

A collection of *objects*. Some represent phenomena while others are simply part of the implementation.

Object

Computer representation of a real world phenomenon. Its structure consists of attributes and actions.

Pattern

Computer representation of a real world concept. Objects defined according to the pattern are called *instances* or *pattern defined objects*: A pattern is to its instances as a concept is to its phenomena.

Singular object

An object representing a singular "one-of-a-kind" phenomenon - the object is not defined as an instance of some pattern.

State of an object

The combined values of its *measurable properties* at some point in time.

Measurable property

A property which has a measurable value. The value may vary over time.

Part object

An object which is part of another object. Part objects are used to model part or aggregation hierarchies.

Separate object

An autonomous self-contained object which is not a part object.

Reference to separate object

An attribute which is a reference to a separate object.

Kinds of actions

The actions of a phenomenon in a real world system often take place *concurrently* (i.e. in parallel) with those of other phenomena in the system. A single phenomenon normally *alternates* among its own actions.

Declarations and Object Descriptors

Declaration or attribute declaration

An association or *binding* of a name to some entity. The syntactic construct used is

the colon ":" as in, <name>: <entity**>**. For attributes of an object descriptor, these are sometimes referred to as the *attribute name* and *attribute description*, respectively.

Pattern declaration

A declaration binding a *pattern name* to an object descriptor, describing the structure of *instances of the pattern*. Pattern declarations serve as templates for generating objects having a given structure.
Syntax is:
```
<name>: <object-descriptor>
```

Singular object declaration

Declaration of a singular object binding the object name to the singular object description (an object descriptor).
Syntax is:
```
<name>: @<object-descriptor>
```

Attribute reference

An occurrence of an attribute's name in an object descriptor.

Local attribute reference of a pattern

A reference from within a pattern's object descriptor to an attribute declared inside the same object descriptor.

Global attribute reference

Any attribute reference which is not local.

Object-descriptor

Used to describe the structure of objects and consists of a *prefix part* and a *main part*.

Prefix part

Part of object descriptor used to specify the superpattern of the descriptor. The prefix part is specified by a pattern name (or is empty).

Main part

Used to describe the additional structure of objects. Has the syntactic form(# E #) and consists of an *attribute part and an action part.*

Attribute part

Part of object descriptor used to describe the object's attributes. Consists of a list of *attribute declarations*.

Action part

Part of object descriptor used to describe the actions to be performed when the object is executed. Consists of three parts: *enter-part*, *do-part*, *exit-part*.

Enter part

Part of *action part* describing the *enter parameters*.

Do part

Part of *action part* consisting of a list of *imperatives*.

Exit part

Part of *action part* describing the exit*parameters*.

Program

An object descriptor that can be compiled and executed.

Reference Attributes

Reference attribute

An attribute that denotes an object. Reference attributes can be either *static references* or *dynamic references*.

Static reference

A reference attribute that constantly denotes the same object. Such objects are often referred to as *static objects*. In cases where these objects are used to model part (or aggregation) hierarchies, they are referred to as *part objects*, that is, they are part of an *enclosing object*.

Static reference declaration

Used to define static reference attributes.
Syntax is:

```
<name>: @<ptn.name or obj.descriptor>
```

Dynamic reference

A reference attribute that denotes a object. The reference is variable in that it may denote different objects over time. Initially it denotes *NONE* which represents "no object."

Dynamic reference declaration

Used to define dynamic reference attributes.
Syntax is:

```
<name>: ^<pattern name>
```

Indexed collection of static / dynamic references

A repetition (or *array*) of object references referred to by a single name plus an index. The size of a repetition A is denoted by A.range. A[1] refers to the first element in the repetition, A[A.range] to the last.
Syntax is:

```
Name: [eval] @<ptn.name or obj.descriptor>  Name: [eval] ^<ptn.name>
```
The size of the repetition can be dynamically extended by:
```
<number> -> A.extend
```

Qualification or qualifying pattern

The pattern name appearing in a reference attribute declaration. It restricts the set of objects that can be denoted by the reference.

Remote access

Used to denote attributes within an enclosing object.
Syntax is:
```
reference.attribute
```

Computed Remote access

Used to denote attributes within objects that are returned as the result of evaluations.
Syntax is:
```
(evaluation).attribute
```

Pattern Attributes

Pattern reference

A reference attribute that denotes a pattern. The structure of the pattern is represented locally using a *structure object*. Such objects include a reference back to the object of which the pattern is an attribute. This reference is called the *origin* of the pattern.

Pattern reference declaration

Used to define a pattern.
Syntax is:
```
<name>: <object descriptor>
```

Pattern variable declaration

Used to define pattern variable attributes. A pattern variable may denote different patterns during the execution. The qualification restricts the set of patterns which may be denoted by the pattern variable.
Syntax is:
```
<name>: ##<pattern name>
```

Class pattern

Generally, a pattern used to model physical objects.

Procedure pattern

Generally, a pattern used to model action sequences.

Function pattern

A procedure pattern which computes and returns a value. Such patterns always have an exit part.

Basic pattern

A pattern that is predefined within the BETA language. Examples are integer, real, boolean, and char. Relevant operations include: +, -, *, div, mod, and, or, not, true, false, =, <, >, <>, <=, >=.

Imperatives

Imperative

Describes an action; *executing* the imperative causes the action. Imperatives appear in the do-part of an object. Kinds of imperatives include *evaluations*, *reference assignments*, *dynamic object creation*, and *control structures*.

Evaluation imperative

An imperative that can cause state changes and may produce a value when executed.

Value assignment

An evaluation imperative that sets (changes) the value of an attribute.
Syntax is:
```
3 -> I
```

Reference assignment

An *imperative* used to change the value of a dynamic reference.
Syntax is:
```
objRef[] -> dynObjRef[]
```
objRef may be any object reference but dynObjRef *must be a dynamic object reference.*

Pattern assignment

An *imperative* used to change the pattern denoted by a pattern variable.
Syntax is:
```
ref## -> dynPatRef##
```
Ref may be the name of a pattern variable, the name of an object, or the name of a pattern but dynPatRef *must be a dynamic pattern reference.*

Multiple assignment

An evaluation imperative that causes several assignments.
Syntax is:
```
3 -> I -> J
```

Dynamic object creation / generation

Imperatives used to create new *dynamic objects*.
Syntax is:
```
&Pat   or   &Pat[]
```

Value equality

True when two references denote objects that have the same state.
Syntax is:
```
A = B
```

Reference equality

True when two references denote the same object.
Syntax is:
```
A[] = B[]
```

Pattern equality

True when two pattern references denote the same pattern.
Syntax is:
```
A## = B##
```
Note that < and <= are also defined for pattern comparisons based on the inheritance hierarchy.

Procedure call

An evaluation imperative that causes invocation of a procedure pattern.
Syntax is:
```
&ProcPat
```
or
```
(arg1,arg2) -> &ProcPat
```

Function call

An evaluation imperative that causes invocation of a function pattern.
Syntax is:
```
(arg1,arg2) -> &FuncPat -> result
```

Control structure

An imperative that controls the flow of executions.

For imperative

A control structure used to support *iteration*. A list of imperatives are executed repeatedly while an index steps from 1 up to the number of iterations.
Syntax is:
```
(for Index: Range repeat       Imperative-list
for)
```

General-if imperative

A control structure used to support *selection*. Based on evaluating a condition evaluation and comparing it to the values of a number of selection evaluations, one of a set of imperative-lists is executed.
Syntax is:
```
(if E0   // E1 then I1   // E2 then I2   E   // En then In   else I   if)
```

Simple-if imperative

A control structure used to support boolean *selection*. Based on evaluating a condition evaluation and testing if it is true or false, one of two imperative-lists is executed.
Syntax is:
```
(if E then      I1   else I2  if)
```

Labelled imperative

A means of naming an imperative. References to the label (via *jump imperatives*) can be made from within the imperative.
Syntax is:
```
L: Imperative
```
or
```
L: (# ... do ... #)
```

Jump imperative

Causes flow of control to "jump" to another location. A jump imperative is one of a *Leave imperative* or a *Restart imperative*.

Leave imperative

Causes termination of the execution of a labelled imperative; execution resumes after the labelled imperative. This imperative can only appear within the labelled imperative.
Syntax is:
```
leave L
```

Restart imperative

Causes restarting of the execution of a labelled imperative, that is, jump is to the start of the imperative. Can only appear within the labelled imperative.
Syntax is:
```
restart L
```

Block Structure and Scoping

Block structure

The nesting of one structure in another in the text of a program. In BETA, object descriptors and imperatives can be nested inside of other object descriptors and imperatives. It is the job of the programmer to use *indentation* to make such nesting visible to readers. In the following example, Deposit's object descriptor is nested inside of Account's.
```
Account:    (# Deposit:         (# E         do E         #);    #);
```

Declaration of a name

An association of a name with some defining expression.
Syntax is:
```
<name>: E
```
Recall that colon ":" always signals a declaration of some kind.

Application of a name

Any occurrence of a name in a program which is not a declaration. Note that this does not include keywords of the BETA syntax (e.g. if, for, repeat, do), but does include predefined pattern and attribute names (e.g. char, putInt, stream).

Scope of a declaration

The part of the program text "covered" by the declaration, that is, where applications of the declared name refer to the given declaration. In BETA, the scope of a declaration is the object descriptor it appears in. The exception to this is that the declaration may be "hidden" by declarations of the same name in nested object descriptors or labelled imperatives. Note that the declared name can also be applied outside its object descriptor using remote access. We say that a name is *local* to the object descriptor in which it is declared and *global* to any nested object descriptors (for which it is not hidden).

Inserted Objects

Inserted item

A means of generating (and executing) a procedure object allocated as part of the enclosing object.
Syntax is:

```
A -> P -> B
```

or

```
A -> P(# E #) -> B
```

This differs from dynamic generation, &P, in that the instance of P is generated only once rather than each time the imperative is executed. Note that inserted items should not be used to define recursive procedures. That is, an inserted instance of P may be specified in the action part of P.

Inheritance

Direct subpattern

A pattern P is a *direct subpattern* of Q if P extends (specialises) the definition of Q. Q is called the *directsuperpattern* of P and instances of P are also instances of Q.
Syntax is:

```
P: Q(# E #)
```

Q is called the *prefix pattern* (or simply *prefix*), while the contents of (# E #) is called the *main-part* of P. The prefix Q means that P's object descriptor *inherits* all of Q's declarations in addition to any new ones defined in P's main-part.

Subpattern

A pattern P is a *subpattern* of Q if it is either a direct subpattern of Q or a subpattern of a direct subpattern of Q. Likewise, Q is a *superpattern* of P if it is either a direct superpattern of P or a superpattern of the direct superpattern of P. A pattern can have at most one direct superpattern.

Abstract superpattern

A pattern used only as a superpattern for other patterns, that is, it is not intended to be used to generate objects. If P is declared without the use of a superpattern, P: (# E #), then P is assumed to be a subpattern of the most general abstract

superpattern, Object. Note that the basic patterns, Integer, Real, Boolean, Char and Real are not subpatterns of Object.

Superpattern as qualification

If R is a dynamic reference qualified by the pattern Q (i.e. R: ^Q) and Q is a superpattern of P, then instances of both P and Q can be assigned to R. However, only attributes of Q (and of superpatterns of Q) can be accessed using remote access through R. That is, if attribute A is declared in the main part of P, then the remote access R.A is illegal.

Action specialisation

The use of a subpattern to extend the action part of a pattern. Action specialisation can involve any or all of the enter-part, exit-part and do-part. The enter and exit parts of instances of P (again, a subpattern of Q) consist of Q's enter and exit parameters together with those defined by P. Extending the do-part of Q requires the use of the **inner** imperative in Q's action part. Executing the do-part of an instance of P proceeds by executing Q's do-part and executing P's do-part each time inner is encountered.
Syntax is:

```
Q: (# E do E inner E #);  P: Q(# E do E #);
```

Virtual Patterns

Virtual pattern

A pattern attribute V of a pattern Q is *virtual* if it is only partially defined in Q. That is, the definition of V can be extended in subpatterns of Q.
Syntax is:

```
Q: (# V:< S #)  Q: (# V:< S0(# E #) #)  Q: (# V:< (# E #) #)
```

In the first of the three forms, we say that the virtual V is *qualified* by the pattern S, in the second and third forms, we say that V is *directly qualified*.

Further binding of a virtual pattern

The means by which a virtual attribute V of a pattern Q is extended in a subpattern P of Q.
Syntax is:

```
P: Q(# V::< S1 #)  P: Q(# V::< S1(# E #) #)  P: Q(# V::< (# E #) #)
```

S1, S1(# E #), or (# E #) is called the *extended descriptor* of V. If we're using either the first or second form, and if V is qualified by S in the pattern Q, then S1 must be a subpattern of S. In the case of the third form there are no constraints on Q's declaration of V. If X is an instance of P, then X.V specialises (that is, adds properties to) the definition of V in Q. Note that V is now a virtual pattern in P (as well as Q) and can continue to be further bound in subpatterns of P.

Final binding of a virtual pattern

The means by which a virtual attribute V of a pattern Q is extended in a subpattern P of Q, and at the same time made non-virtual.
Syntax is:

```
R: P(# V:: S2 #)  R: P(# V:: S2(# E #) #)  R: P(# V:: (# E #) #)
```

Final binding is identical to further binding, except that with final binding, V is no longer virtual.

BETA Terminology Mjølner Informatics