PB - 214

J.L. Knudsen: Static Exception Handling in Beta

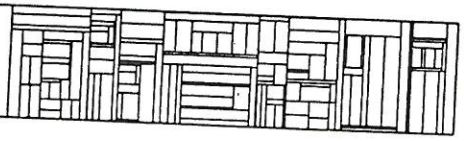
TRYK: RECAU (06) 12 83 55

ISSN 0105-8517

Static Exception Handling in Beta

Jørgen Lindskov Knudsen

DAIMI PB - 214
June 1986



Static Exception Handling in Beta

Jørgen Lindskov Knudsen
Computer Science Department
Aarhus University, Denmark

Abstract

The concept of static exception handling have previously been investigated in a general setting. In this paper, we will investigate the introduction of static exception handling in the programming language Beta. The aim is to show that static exception handling is a valuable contribution to object-oriented programming. Furthermore, the paper illustrates the evolution of a programming language feature from the general design phase to a concrete proposal for inclusion into a full-fledged programming language (in this case Beta).

Introduction

This paper will discuss static exception handling in the programming language Beta. It is assumed that the reader is familiar with the Beta language as described in references /1/ and /2/. Furthermore, it is assumed that the reader is familiar with static exception handling as described in references /3/ and /4/.

The discussion will be divided into four parts. First of all, some problems in Beta are discussed. These problems are related to the coexistence of blockstructure, prefixing and virtual binding. These problems need to be resolved in order to introduce hierarchical, co-operative exception handling as discussed in reference /4/. Secondly, sequels in uni-sequential Beta are discussed. Here it is illustrated that sequels can be included in the Beta language merely by defining a sequel pattern. Thirdly, sequels in multi-sequential Beta are discussed. Exception handling in multi-sequential Beta is particular interesting as it gives rise to a discussion of issues of inter-process exception handling which have not been dealt with in the two previous articles (references /3/ and /4/). Finally, implementation of sequels in basic Beta is discussed.

1 Discussion of Prefixing, Virtual Binding, and Blockstructuring

The programming language Beta is a blockstructured language with extensive support of object-oriented programming. This support is primarily through the concepts prefixing and virtual binding. In this section, we will discuss the relationship between blockstructure, prefixing and virtual binding, and thereby reveal some irregularities of the Beta language, and finally propose some slight changes in order to remove these irregularities such that the three mechanisms become more orthogonal.

One of the mechanisms for specifying the blockstructure of a Beta program is nested singular instances, specified in the action-part. In order to ease the following discussion, we will use the term "block" or "inner block" as a synonym to "singular instance, specified in the action-part".

Let us start by considering prefixing and blockstructure. Let us consider example 1. The example shows two pattern declarations, **A** and **A'**.

```
A : Pattern (# Z : Pattern(# ... #);
               P : Pattern Z(# ... #);
            Do
              :
              B : (# R : Pattern P(# ... #);
                   Do
                     :
                   #);
              :
            #)
A': Pattern A(# Z' : Pattern Z(# ... #); ... Do ... #);
```

Example 1:

In the attribute part of **A**, the patterns **Z** and **P** are declared (**P** prefixed with **Z**). The action part of **A** contains an inner block **B**. In this block, a pattern attribute **R** (prefixed with **P**) is declared.

The pattern **A'** is prefixed with **A** and contains a pattern declaration of **Z'** in its attribute part (**Z'** prefixed with **Z**).

All of these declarations (of **A**, **Z**, **P**, **R**, **A'** and **Z'**) are legal and can be given a clean semantics (given in ref./1/). Let us make one slight change, as can be seen in example 2.

```
A : Pattern (# Z : Pattern(# ... #);
              P :< Z;
            Do
              :
              B : (# R : Pattern P(# ... #);
                   Do
                     :
                   #);
              :
            #);
A': Pattern A(# Z' : Pattern Z(# ... #);
                P ::< Z';
              Do
                :
              #);
```

Example 2:

The only difference between example 1 and 2 is that we have made the binding of **P** into a virtual binding. In **A**, an initial virtual binding of **P** is specified (**P :< Z**)[1], and a further binding of **P** (**P ::< Z'**) is specified in the subpattern **A'**. Again, all of these declarations are legal, and can be given a clean semantics (given in ref./1/). However, making one additional change (as in example 3) creates problems. (Beta has resolved this problem, see later.)

The only difference between example 2 and 3 is that we in block **B** has further bound **P** to **R**. This further binding creates ambiguities in the program. The problem arises when we consider an instance **a** of **A'**. During execution of the action part of **a**, we might reach the invocation of **@P** in block **B**. At that point **P** is further bound twice (once in **B** and once in **A'**), and there is no guarantee that either of them is a subpattern of the other. That is, the binding of **P** is ambiguous.

This ambiguity can be resolved by means of some ad-hoc rules but it can easy be seen that such rules become highly complex and not very intuitive. It should also be noted that this usage of virtual binding leads to bindings of a very dynamic nature (the bindings will depend of the actual control flow). This is in contrast with virtual binding in Beta (as described in reference /1/), where the virtual binding of an attribute is constant with respect to one particular instance. It should be noted that the above problem is avoided in Beta by only allowing further bindings to appear in subpatterns of the pattern containing the initial virtual binding.

Unfortunately, the above Beta restriction on the usage of virtual binding makes virtual binding along the blockhierarchy of a program nearly impossible. This can be illustrated by example 4.

The example illustrates that we can create prefix hierarchies that reflect the blockhierarchy of the program, but we cannot let the virtual bindings of a name follow the same

[1]The Beta terminology is slightly different than the one used here. In Beta, **P :< Z** is called a *virtual pattern declaration* but in order to conform with reference /4/ we retain the terminology used there.

```
A : Pattern (# Z : Pattern(# ... #);
                P :< Z;
             Do
                :
                B : (# R : Pattern Z(# ... #);
                       P ::< R;
                    Do
                       @P;
                    #);
                :
             #);
A': Pattern A(# Z' : Pattern Z(# ... #);
                P ::< Z';
              Do
                 :
              #);
a : @A';
```

Example 3:

```
(# Z : Pattern (# ... #);
   P :< Z;
Do
   :
   (# Z' : Pattern Z(# ... #);
      P ::< Z'; (* illegal !!! *)
   Do
      :
   #);
   :
#);
```

Example 4:

blockhierarchy. Prefixed inner blocks make one-level further bindings possible but does not give the full generality. This is illustrated in example 5. Note, the further binding **P ::< Z''**

```
(# A : Pattern (# Z : Pattern (# ... #);
                   P :< Z;
                Do
                   :
                #);
Do
   :
   A(# Z' : Pattern Z(# ... #);
        P ::< Z';
     Do
        :
        (# Z'' : Pattern Z'(# ... #);
           P ::< Z''; (* still illegal !!! *)
        Do
           :
        #);
     :
     #);
   :
#);
```

Example 5:

is still illegal!

As it can be seen from the above discussion, we need to relax the Beta restriction on the usage of virtual bindings.
We will propose the following restriction:

> If an initial virtual binding is present in the attribute part of a pattern **A**, then further bindings can only be specified in subpatterns of **A**.
>
> However, if the initial virtual binding is present in the attribute part of the entity-descriptor associated with a singular entity, then further bindings can be specified in inner blocks (i.e. in the attribute part of singular entities, inserted in the action part of the entity itself).

This restriction would allow for the further binding **P ::< Z'** in example 4, whereas the further binding **P ::< Z''** in example 5 would still be illegal. In fact, the above restriction makes virtual bindings along both the blockhierarchy and the prefixhierarchy possible, but disallows any connections between the two virtual binding mechanisms. That is, one might even use two different syntactic notions, depending on whether it is virtual binding along the prefixhierarchy, or virtual binding along the blockhierarchy. However, we will in the following use the same notion for both kinds of binding since no confusion is possible.

The above restriction does not apply to prefixing, and in that respect prefixing is still more generally usable than virtual binding. However, the above restriction on possible further bindings of virtual bindings has broadened the usability of virtual binding compared with current Beta.

This kind of virtual binding resembles dynamic binding. The reason why virtual binding along the blockhierarchy can be allowed in an otherwise static language is that the kind of

dynamic binding that can be specified by means of virtual binding is very restricted. Let us consider example 6.

```
(# A : Pattern (# P : Pattern (# ... #);
                  VP :< P;
                  R : Pattern (# ... Do ... VP ... #);
               Do
               #);
   a : @A;
Do
   :
   (# Q : Pattern (# ... #);
      VP :< Q;
      S : Pattern (# ... Do ... VP ... #);
   Do
      :
      (# Q' : Pattern Q(# ... #);
         VP ::< Q';
      Do
         :
         S; (*1*)
         :
         a.R; (*2*)
         :
      #);
      :
      S; (*3*)
      :
   #);
   :
#);
```

Example 6:

Note the two **VP**'s. One virtually bound in **A**, and one virtually bound (**VP :< Q**) in the inner block. The action phase executed as a result of the invocation of **S** contains an invocation of **VP**. If **S** is invoked at (*1*) this will result in execution of the action phase of **Q'**, whereas it at (*3*) will result in execution of the action phase of **Q**. However, even though **VP** is virtual in **A**, invoking **VP** through the **a.R** at (*2*) will lead to the execution of the action phase of **P**, not **Q'**. This is in contrast to dynamic binding where it would have been the action phase of **Q'**.

The virtual binding mechanism is restrained in one further respect compared with dynamic binding. The further binding of a virtual pattern must follow a prefixhierarchy (that is, the pattern specified in the further binding must be prefixed with the pattern specified in any previous further/initial binding), whereas using dynamic binding there is no way within the language to ensure that any relation holds between the various bindings of a particular name.

Furthermore, having the possibility of virtual binding along the blockhierarchy of a program gives further opportunities for top-down development for programs. Furthermore, this additional dynamics gives only rise to a slightly more complex implementation since the implementation technique to be used for this kind of virtual binding is almost identical to the one to be used for virtual prefixes in current Beta. The extra complexity is at

block-entry/exit. In a block containing a further binding of a virtual pattern, the virtual pattern association must be relocated to the pattern specified in the further binding. This relocation must be done at block-entry, and the old association must be restored at block exit.

Informally, the following must be done for each virtual pattern with a further binding in the block:

block entry: Push the current association on the stack
Virtual Pattern Pointer := Further Binding Pattern

block exit: Pop the old association off the stack
Virtual Pattern Pointer := Old Association

Note, that the extra complexity is very minimal (in the order of 2 operations), and the extra cost is only charged on blocks that actually utilizes the virtual binding mechanism.

One problem remains to be discussed: Instances of virtual patterns, and instances of patterns with a virtual prefix (in short called *virtual instances* in the following discussion).

Virtual instances declared in the attribute part of patterns can be dealt with effectively by means of indirect addressing (pointer to the actual entity).

Virtual instances declared in the attribute part of inner blocks introduces additional problems. First of all, Beta has to allocate such instances off-line in order to be able to do compile-time calculation of attribut-offsets of entities (just as above). But one problem remains, namely the actual allocation. Let us illustrate by example 7.

```
(# P : Pattern (# ... #);
   VP :< P;
   v : @VP;
Do
   . (* here we must be able to access the *)
   . (* P specific attributes of v *)
   (# P' : Pattern P(# ... #);
      VP ::< P';
   Do
      . (* here we must be able to access the *)
      . (* P' specific attributes of v *)
   #);
   .
   .
   (# P'' : Pattern P(# ... #);
      VP ::< P'';
   Do
      . (* here we must be able to access the *)
      . (* P'' specific attributes of v *)
   #);
   . (* here we are only able to access the *)
   . (* P specific attributes of v *)
#);
```

Example 7:

The problem is that we must be able to allocate an entity that can be interpreted by any pattern that is bound to **VP** in all inner blocks to the block in which **v** is declared. In many cases, this is possible at compile-time, but in the worst case, the virtual part of **v** has to be dynamically allocated (i.e. off-line).

Summary

Introducing virtual bindings along the blockhierarchy will allow usage of virtual binding more generally than in current Beta. Moreover, the implementation is only slightly more complicated than virtual binding along the prefixhierarchy, and the extra cost is only charged when the mechanism is actually used. However, in order to avoid conflicting further bindings we have to restrict the usage of virtual binding in such a way that virtual bindings along the prefixhierarchies are separated from the virtual bindings along the blockhierarchies. This restriction is a slight relaxation of the restriction imposed on virtual bindings in current Beta.

That is, virtual bindings along the blockhierarchy can be incorporated in the Beta language just by relaxing one language restriction and without making the implementation significantly more complex.

2 Introducing Sequels into Beta

Following the above discussion of prefixing, virtual binding and blockstructure in Beta, we are in position to discuss the introduction of static exception handling in Beta. We will divide the discussion into two steps. Firstly, we discuss introduction of sequels into the uni-sequential part of Beta, and secondly, we discuss the introduction of sequels into the multi-sequential part of Beta.

2.1 Introducing Sequels into Uni-sequential Beta

Let us start by introducing a predefined pattern named **Sequel** that implement the desired behaviour of sequels. In a later section, we will comment on how to implement this pattern in basic Beta. Sequels as described in references /3/ and /4/ will now be specified in Beta as patterns with **Sequel** in their prefix-chain.[2] The semantics of sequels in Beta can now be formulated as follows:

Let **s** be a sequel instance with descriptor **D** such that

- $\mathbf{D} = \mathbf{S}_n(\#\ \ldots\ \#)$ or $\mathbf{D} = \mathbf{S}_n$

  and $\mathbf{A}_0$: **Pattern** (# ... #)
  $\vdots$
  $\mathbf{A}_k$: **Pattern** $\mathbf{A}_{k-1}$(# ... $\mathbf{S}_1$: **Pattern Sequel**(# ... #) ... #)
  $\vdots$
  $\mathbf{A}_m$: **Pattern** $\mathbf{A}_{m-1}$(# ... $\mathbf{S}_n$: **Pattern** $\mathbf{S}_{n-1}$(# ... #) ... #)
  $\vdots$
  $\mathbf{A}_l$: **Pattern** $\mathbf{A}_{l-1}$(# ... #)

  and **a** is an instance with descriptor
  $\mathbf{A}_l$(# ... **s** = **D** ... #) .

Then the termination level of **s** is $\mathbf{A}_k$. That is, execution of **s** in **a** will result in termination of all **a** activities originating from the action part specified in $\mathbf{A}_k$. And execution will be resumed after the *inner*-imperative in $\mathbf{A}_{k-1}$. (The *inner*-imperative that initiated the action part specified in $\mathbf{A}_k$.)

[2]Note that we in the following will use the term sequel in two different ways. When we say "the pattern **Sequel** (capital S)", we refer to the predefined pattern, implementing the sequel behaviour. When we say "a sequel", we refer to any entity with the predefined pattern **Sequel** in its prefix-chain. No confusion should be possible.

- **D = Sequel**

  and **a** in an instance with descriptor
  P(# ... **s** = **D** ... #) .

  Then the termination level of **s** is the encloser of **Sequel**. (We will comment on this case later.)

The rationale of the above rule is the following: Introducing a descriptor with **Sequel** as the immediate prefix is in the Beta programming language synonymous to defining a sequel by means of a specialized language construct (such as the one used in references /3/ and /4/).

Let us illustrate this semantics of sequels by means of three examples from references /3/ and /4/, now formulated in Beta with the predefined pattern **Sequel**. The termination level of instances of the sequels *Present* and *Absent* in example 8 is *TableSearchAndCount*

```
TableSearchAndCount : Pattern
                      (# A : @Table;
                         X : @Item;
                         I : @TableIndex;
                         Result : @TableIndex;
                         Present : Pattern Sequel
                                   (# J : @TableIndex;
                                   Enter <J>
                                   Do A.Occurrences[J] + 1 => A.Occurrences[J];
                                      J => Result;
                                   #);
                         Absent : Pattern Sequel
                                   (# J : @TableIndex;
                                   Enter <J>
                                   Do 1 => A.Occurrences[J];
                                      X => A.Item[J];
                                      J => Result;
                                   #);
                      Enter <A,X>
                      Do X => Hash => I;
                         Cycle(# Do (if A.Item[I] =
                                   // X then I => Present;
                                   // nullItem then I => Absent;
                                   if);
                                   I Mod TableMax => I; I + 1 => I;
                         #);
                      Exit <Result>
                      #);
```

Example 8: example 3 from reference /3/

since S_1 in that case is *Present* and *Absent*, respectively. In example 9, the termination level of instances of the sequel *LocalError* is **B1** since S_1 in that case is *ResourceError*. Example 10 is a more realistic example of usage of static exception handling in Beta and we will therefore comment more detailed on that example.

In block **B**, a *Stack* pattern is declared with two virtual sequel patterns *Overflow* and *Underflow*. These virtual sequels are invoked in *Push* and *Pop*, respectively. Furthermore,

```
B1 : (# ResourceError : Pattern Sequel
                    (# Do Resource!Undo; INNER; Resource!Release #);
   Do
     Resource!Request;
     :
     :
     B2 : (# LocalError : Pattern ResourceError
                        (# Do Script!ScriptError; INNER; Script!DisEngage #);
          Do
            Script!Engage;
            :
            :
            LocalError;
            :
            :
            Script!DisEngage;
          #);
     :
     :
     Resource!Release;
   #);
```

Example 9: example 11 from reference /4/

in **B**, a sequel *StackError* is declared and a specialization of *Stack* named *GlobalStack*, in which *Overflow* and *Underflow* are final bound to *StackError*. Finally, a *P* pattern in specified in **B**. In *P*, two sequels *IllegalPush* and *IllegalPop* are declared, and a specialization of *Stack* named *LocalStack*, in which *Overflow* is final bound to *IllegalPush*, and *Underflow* is final bound to *IllegalPop*. Finally, *Stack1* is declared as an instance of *LocalStack*, and *Stack2* is declared as an instance of *GlobalStack*. In the action-part of *P*, two imperatives are shown: *I* => *Stack1.Push* and *J* => *Stack2.Push*. If *Overflow* is invoked during *I* => *Stack1.Push*, then *Overflow* is bound to *IllegalPush*, in which case $\mathbf{S}_1$ is *IllegalPush* and therefore the termination level of *Overflow* is *P*. If, on the other hand, *Overflow* were invoked during *J* => *Stack2.Push*, then *Overflow* is bound to *StackError*, in which case $\mathbf{S}_1$ is *StackError*, and therefore the termination level of *Overflow* is **B**

These three examples show that with the predefined pattern **Sequel** it is possible to specify static exception handling in Beta along the lines of references /3/ and /4/.

In the rest of this section, we will discuss issues of static exception handling particular related to the structure of the Beta programming language. The generality of the Beta language gives rise to some additional considerations.

The first consideration is immediate instances of **Sequel** (case **D** = **Sequel** on page 7) as illustrated in example 11. We will here comment on two different interpretations of **D** = **Sequel**. There are two possibilities: Firstly, the termination level might be **B** (i.e. the termination level of an immediate instance of **Sequel** is the encloser of the *instance*). This would have as consequence that the termination level of **Sequel** would be dynamic (i.e. two instances may have different termination level) whereas the termination level of any subpatterns of **Sequel** would be static. This is in contrast with the static approach to exception handling and we must therefore consider the other alternative. The second alternative is that the termination level is the encloser of the **Sequel** *pattern*. This implies that the termination level of **Sequel** is static and thus consistent with the static approach to exception handling. Furthermore, the behaviour of the first alternative can easily be simulated as illustrated in example 12. This second interpretation is the one used in the semantics on page 7.

```
B : (# Stack : Pattern
            (# Max :< Integer;
               Overflow, Underflow :< Sequel;
               Store : @[Max]Item;
               Top : @Integer;
               Push : Pattern (# I : @Item;
                                 Enter <I>
                                 Do (if Top =// Max then Overflow if);
                                    Top + 1 => Top;
                                    I => Store[Top];
                                 #);
               Pop : Pattern  (# Result : @Item;
                                 Do (if Top =// 0 then Underflow if);
                                    Store[Top] => Result;
                                    Top - 1 => Top;
                                 Exit <Result>
                                 #);
               Empty : Pattern (# Do Exit <Top = 0> #);
            Do (* Stack initialization *) 0 => Top #);
   StackError : Pattern Sequel (# ... #);
   GlobalStack : Pattern Stack(# Overflow :: StackError; Underflow :: StackError #);
   P : Pattern
      (# IllegalPush : Pattern Sequel(# ... #);
         IllegalPop : Pattern Sequel(# ... #);
         LocalStack : Pattern Stack(# Overflow :: IllegalPush; Underflow :: IllegalPop #);
         Stack1 : @LocalStack(# Max :: 100 #);
         Stack2 : @GlobalStack(# Max :: 50 #);
      Do (* P *)
         .
         .
         I => Stack1.Push;
         .
         .
         J => Stack2.Push;
         .
         .
      #);
Do (* B *)
   P;
#);
```

Example 10: example 10 from reference /3/

```
B : (# S : @Sequel;
   Do
      .
      .
      S; (* what is the termination level of S? *)
      .
      .
   #);
```

Example 11:

```
B : (# S : @Sequel(# ... #);
   Do
      :
      :
      S; (* with termination level B *)
      :
      :
   #);
```

Example 12:

The second consideration is sequels as attributes to static instances. Let us consider example 13. Assume that we in an entity **R** executes *Stack.Push*, and assume further that

```
Stack : (#
      :
      :
      Overflow : Pattern Sequel(# ... #);
      :
      :
      Push : Pattern (# I : @Item;
                        Enter <I>
                        Do (if Top =// Max then Overflow if);
                           Top + 1 => Top;
                           I => Store[Top];
                        #);
      :
      :
   Do (* Stack *)
      0 => Top;
   #);
```

Example 13:

Stack.Top = *Stack.Max*. Then *Stack.Overflow* will be executed, terminating the action part of *Stack* **but not** terminating *Push*, since *Push* is not an activity originating from the action part of *Stack*.

This underlines one fundamental principle of static exception handling in Beta: Invocation of a sequel in a Beta program will only affect the activities originating from the action part of the termination level — all other activities are unaffected by the exception occurrence. In example 13 this implies that *Push* must be designed otherwise if *Push* should be terminated when *Overflow* is executed, as illustrated in example 14.

```
Push : Pattern (# I : @Item;
                  PushError : Pattern Sequel(# ... Do ... Overflow #);
               Enter <I>
               Do (if Top =// Max then PushError if);
                  Top + 1 => Top;
                  I => Store[Top];
               #);
```

Example 14:

Let us consider example 15 that illustrates the consequences of the co-existence of

prefixed instances and sequel attributes of such prefixed entities. Then, if execution of the

```
(# A : Pattern P(#
                  .
                  .
                  .
                  S : Pattern Sequel (# ... #);
                  .
                  .
                  .
               Do
                  .
                  .
                  .
                  S;
                  .
                  .
                  .
               #);
   a : @A; (* Static instance *)
   a' : #A; (* Dynamic instance *)
Do
   .
   .. => A => ..
   .. => a => ..
   .. => .@A => ..
   .. => .a' => ..
   .
#);
```

Example 15:

action part of any instance of **A** reaches the invocation of **S**, the action part of **S** will be executed, and if it terminates successfully,[3] the execution of all actions originated from the action part of the particular instance of **A** will be terminated (including the action part of **A**). Whether or not the particular instance of **A** will become inaccessible after the sequel invocation, is a property of its mode of generation and not a direct consequence of the sequel invocation.

Note, that it is only the action part of **A** that is terminated by the invocation of **S**. That is, execution will continue after the *inner*-imperative in **P** which caused the execution of the action part of **A**. This semantics of sequel invocations is first of all consistent with the treatment of blocks in reference /4/, and furthermore consistent with the declaration of **S** as an attribute of **A**, and not of **P**. That is, **S** belongs to the abstraction level of **A** and is unknown to the abstraction level of **P**. In other words, **S** must handle the exception occurrence in such a way that control can be passed securely to the **P** level. If not, **S** should invoke another sequel either declared in **P**, in one of **P**'s superpatterns, or declared in an outer scope.

Finally, example 16 outlines the structure of a Beta program with static exception handling, utilizing both prefixing and virtual binding. With the above discussion, the example requires only a few comments:

In the instance **BetaProgram**, patterns **A** and **A'**, and instances **a**, **a'**, **t** and **r** are declared. In the prefix **BetaPredefinedEnvironment**, the **Sequel** pattern is declared and a sequel instance **Halt**. Sequel **Halt** will terminate the entire **BetaProgram** (including the action part specified in **BetaPredefinedEnvironment**). Sequel **t** will terminate the entire **BetaProgram** (as **Halt**), whereas sequel **r** will terminate the action part specified in **BetaProgram** (returning control to **BetaPredefinedEnvironment**). Sequel **S** will terminate

[3]In the following, we will assume that sequels terminate successfully when invoced. This is just to ease the discussion and does not limit the validity of the discussion.

```
BetaPredefinedEnvironment : Pattern (#
                                      :
                                      Sequel : Pattern (# ... #);
                                      Halt : @Sequel;
                                      :
                                    #);
BetaProgram : BetaPredefinedEnvironment
              (# A : Pattern (# S : Pattern Sequel(# ... #);
                                VS :< Sequel;
                                x : @VS;
                             Do
                                :
                             #);
                 A' : Pattern A(#
                                S' : Pattern Sequel(# ... #);
                                S'' : Pattern S(# ... #);
                                VS ::< S';
                             Do
                                :
                             #);
                 a : @A;
                 a' : @A';
                 t : @Sequel;
                 r : @Sequel(# ... #);
              Do
                 :
              #);
```

Example 16:

the action part specified in **A** both when invoked in **a** and **a'**. Sequel **S'** will terminate the action part of **A'**, whereas sequel **S''** will terminate the action part specified in **A**, too.

Instances of **VS** are a little irregular, and indicates the need for constraints on the usage of the **Sequel** pattern. Instances of **VS** in **a** (e.g. **x**) will terminate the entire **BetaProgram** (as **Halt** and **t**) whereas instances of **VS** in **a'** (e.g. **x**) will terminate the action part specified in **A'**. If there were no further binding of **VS** in **A'**, instances of **VS** in **a'** would have terminated the entire **BetaProgram**. Although we cannot enforce the constraint on the usage of **Sequel** in current Beta, we strongly recommend that the **Sequel** pattern is only used for prefix specification in order to avoid the irregularities of instances of, say, **VS**.

This situation is very similar to the concept of abstract classes in Smalltalk-80 (reference /5/). Abstract classes in Smalltalk-80 are classes of which no immediate instances may be created — only instances of subclasses of the abstract class.

Summary

In summary, introducing sequels in uni-sequential Beta is possible without too many compromises. The only problems are attributed to sequels being introduced as a predefined pattern **Sequel** and not as a new concept with its own rules. Fortunately, these irregularities are only minor borderline cases that can be eliminated by one additional rule saying that the predefined pattern **Sequel** can only be used as prefix in pattern declarations. Earlier descriptions of the Beta programming language discussed mechanisms for putting constraints on the possible usages of certain patterns, but such mechanisms are not part of current Beta. If such mechanisms were avaiable, we would only allow **Sequel** to be used as prefix and thereby avoiding the problems of immediate instances of **Sequel**. However, since these irregularities can be avoided by trivial programming tricks as shown above, we will allow their presence and make use of their flexibility when appropriate.

Basically, this is all there is to sequels in uni-sequential Beta and with the above mentioned restriction on the usage of virtual binding, the discussion in reference /4/ is applicable to Beta.

It should be noted, that the concept of default sequels discussed in reference /4/ is not contained in the present proposal for sequels in Beta. The reason is that the concept of default sequels is a very specialized concept, and as such it does not conform well into the Beta framework of very general language constructs.

2.2 Introducing Sequels into Multi-sequential Beta

Objects in Beta are the containers of individual control-threads in a Beta program. Objects may be compound objects. In order to ease the following discussion we define four different states that an object may be in during its entire lifetime.

Active: We say that an object is in *Active* state when it is executing the imperatives in its action part (excluding any *ObjectExecution*-imperatives).

Dormant: We say that an object is in *Dormant* state when it is executing an *ObjectExecution*-imperative (i.e. either (*/.../.../.../*) or (*//.../.../.../.../*)).

Suspended: We say that an object is in *Suspended* state when it either has executed a *suspend*-imperative, has finished executing its action part, or the *ObjectExecution*-imperative it is part of, has become detached.

We say that an *ObjectExecution*-imperative will become detached, if either a *detach*-imperative is executed by one of the objects, denoted in the *ObjectExecution*-imperative, or a sequel declared in the outer scope to the *ObjectExecution*-imperative is invoked during execution of the *ObjectExecution*-imperative.

Terminated: We say that an object is in *Terminated* state when it has finished executing its action part, or a sequel declared local to the object has been invoked and terminated successfully.

In order to analyze exception handling in multi-sequential Beta, we only have to consider objects in states *Active* and *Dormant*. The states *Suspended* and *Terminated* are irrelevant to this discussion since no further actions of the objects will be executed in these states (except for resumption in the *Suspended* state afterwhich the object converts into *Active* state)

An object in *Active* state may in principle engage in three different activities. Firstly, it may execute its own action part and the action part of local instances. Secondly, it may engage in communication with some other object, or finally, it may request the execution of global instances. Note, that communication is synchronized mutual exclusive execution of the action part of an instance local to one of the communication partners.

An object in *Dormant* state may engage in two different activities. Firstly, it may execute the action part of one of its local instances on request from one of the objects specified in the *ObjectExecution*-imperative. Such executions are mutual exclusive. Secondly, it may engage in communication with some other object, either a global object, or one of the other objects of the *ObjectExecution*-imperative.

In the following, we will discuss the different cases in which sequels might be executed in multi-sequential Beta. Furthermore, we will discuss the consequences of invoking the sequels in the different cases. We will assume, that we have an object **A** with a local sequel attribute, named **S** (i.e. the termination level of **S** is **A**. We will divide the discussion into two parts. Firstly, we will assume that **A** is in *Active* state, and secondly, we will assume that **A** is in *Dormant* state.

Objects in Active state

Let us assume that **A** is in *Active* state. There is now four different cases to consider:

1. **A** may execute the action parts of local instances. During this, a sequel local to **A** might be invoked, resulting in the termination of all activities originated within **A** (as discussed in section 2.1). If the action part of **A** is invoked by an *inner*-imperative, execution of **A** resumes after the *inner*-imperative in the prefix. Otherwise, **A** will go into *Terminated* state. The sequel must have been invoked as a consequence of the execution of the action part of some local instance; That is, the exception occurrence does not involve any other objects.

2. **A** may accept communications with some other object **Q**. The static instance denotation in the *accept*-imperative denotes a local instance **R**. During the syncronized execution of the entity, **A** cannot be terminated by an invocation of **S**, other than invocations originated from **R**, since **A** executes **R** in mutual exclusion. If, through the execution of **R**, **S** is invoked, the communication with **Q** is deadlocked. The only way in which **Q** can be unlocked is if **Q** is in *Dormant* state and one of the objects in the *ObjectExecution*-imperative executes the **A!R** and another accepts invocations of

some **Q**-local sequel (e.g. executes ?**S'**). In that case, **S** has the opportunity of invoking **S'** in **Q**, just prior to its termination of the action part of **A**. The communication need not deadlock if no synchronizations are pending when **R** is terminated by **S**.

Note, that the local instance denoted by the *accept*-imperative might be a static instance of a sequel. In that case, **A** accepts its action part to be terminated by some other object **Q**.

3. **A** may execute an *accept*-imperative where the static instance denotation denotes some global instances. This case will be dealt with later when we discuss **A** in *Dormant* state.

4. **A** may execute a *request*-imperative. If during the synchronized execution of the entity, an exception occurs, we are in the opposite situation to the one discussed in case 2 above, and **A** will therefore be locked in the communication if there are any pending synchronizations in the communication. Note, that this is no worse than the request never being accepted, or some cases of usage of *detach*- or *leave/restart*-imperatives. Note that the acceptor of the communication cannot succeed in requesting **A** to execute some of its own sequels since **A** cannot accept another communication while locked in the communication that caused the initial exception to occur.

   There is one subtle point with respect to synchronized executions of sequels. Synchronized execution of sequels does not give rise to deadlock situations, since they need not synchronize at the delivery of the exit-list as sequels, by their very nature, cannot have exit-lists.

In conclusion, if **A** is in *Active* state, its execution cannot be interrupted by exception occurrences except those that **A** is directly involved in. More importantly, none of the communications that **A** might accept can be asynchronously interrupted by a sequel declared in **A**.

Objects in Dormant state

The situation is more complex when we consider **A** in *Dormant* state. In *Dormant* state, several objects have been initiated by **A** in an *ObjectExecution*-imperative. For the sake of the following discussion, we will call these objects the *running objects*.

Let us consider one of these running objects. Then there are five different cases to consider:

1. One of the running objects may execute the action part of its own local entities. This situation is not different from case 1 of an object in *Active* state.

2. One of the running objects may request the execution of global instances. Since global instances are executed in mutual exclusion, their execution will not be terminated by an invocation of **S** in **A**, other that invocations through the execution of the global instance.

3. One of the running objects may denote a global instance in an *accept*-imperative. This is with respect to sequel invocation not different from the previous case.

4. Two running objects may both engage in the synchronized execution of the same entity. The entity must then be local to one of them. During this synchronized

execution, **S** might be invoced (caused by a third running object). This will cause all running objects to be suspended, including the two synchronized objects, but since they are synchronized with each other, and both are suspended, deadlock is avoided. Note, they might be resumed at some later point, thereby possibly completing the communication.

5. One of the running objects may itself engage in communication with a global object different from **A**. This can be by executing either an *accept*-imperative or a *request*-imperative. In both cases, **S** might be invoked (caused by one of the other running objects) in which case all running objects will be suspended immediately.

   If **S** is invoked while the running object is executing an *accept*-statement (and the synchronized execution of the entity has been initiated), the global object will be locked just as discussed in case 2 of an *Active* object.

   If, on the other hand, **S** is invoked while the running object is executing a *request*-imperative (and the synchronized execution has been initiated), the global object need not be involved; It may just discard any exit-list produced. However, we have to impose one restriction on the sequel **S** in such cases. **S** must not terminate before all pending synchronizations are fulfilled in order to ensure secure synchronization of the system irrespectively of whether **S** is invoced or not. One might say that **S** in such cases must fulfill any synchronizations on the behalf of the running objects.

This concludes the detailed discussion of sequels in multi-sequential Beta. Note, that the above discussion applies just as well if **S** is declared local to an outer block in which the *ObjectExecution*-imperative is present.

3 Implementation of the pattern Sequel in Basic Beta

In the above discussion, we have introduced static exception handling into the programming language Beta by means of a predefined pattern **Sequel**. In this section, we will discuss one approach to implementing this pattern in Basic Beta.

Basically, we only need to be able to denote the action part corresponding to the termination level. Given this ability, the only thing the pattern **Sequel** needs to do is to issue a *leave*-imperative as its last imperative (with the label denoting the action part corresponding to the termination level).

In order to specify this denotation, we have to introduce some structural attributes. The structural attributes are attributes supplied and maintained by the implementation. The prototype implementation of Beta contains the definition of several structural attributes of which two are of interest here: **SUB**[4] and **ORG**. In order to implement the **Sequel** pattern in Beta, we would like to add one structural attribute **ACTION**, denoting the action part of the entity, and to extend the usability of all the structural attributes. The extension is to allow the structural attributes to be used to specify attribute denotations as composite denotations (i.e. using the usual '.'-notation).[5]

That is, we utilize the following structural attributes:

[4]Actually, **SUB** is called **MAINP** and with a slightly different meaning. The differences are of no concern to the following discussion.

[5]Note, that we in this paper do not propose these structural attributes to be generally usable, but as convenient notations for the implementation. It should be obvious, that the compiler and/or the run-time system easily can deduce these attributes. In order not to burden all entities with these structural attributes, it would be convenient to be able to specify that these attributes are only maintained for instances of particular patterns,

ORG: Denotes the origin of the entity.

SUB: Denotes the entity at the qualification of a possible subpattern of the entity (*none* if no subpattern exists).

ACTION: Denotes the action part of the entity.

As noted above, we are discussing one possible implementation of sequels and not whether the proposed structural attributes should be generally usable programming tools.

Using these structural attributes, we are able to specify the pattern **Sequel** to be:

```
Sequel : Pattern (#
                    Do
                      INNER;
                      (if ( SUB = none ) =
                      // true then leave ORG.ACTION;
                      // false then leave SUB.ORG.ACTION;
                      if);
                    #);
```

It should be noted that this particular usage of the *leave*-imperative is not allowed in current Beta.

In fact, this is all there is to implementing sequels in Beta, since the above discussion of sequels in multi-sequential Beta equally well applies in the case where one of the running objects executes a *leave*-imperative with a label denoting an enclosing block. In other words, the problems discussed must be dealt with in Beta because of the existence of the *leave*-imperative, and not because of the introduction of sequels into the language.

Final Remarks

The above discussion reveals that static exception handling along the lines of references /3/ and /4/ can be introduced in object-oriented programming, examplified by the programming language Beta. The discussion of virtual binding along the block-structure have revealed the usefulness hereof and illustrated that allowing virtual binding along the block-structure does not complicate the implementation significantly.

Introducing the sequel concept in Beta have been shown to be possible. Minor compromisis have been necessary. The generality of the Beta language have been both an advantage and a disadvantage. The prime advantage have been the ability to introduce the sequel concept in Beta merely be defining one pattern. The prime disadvantage have been not to be able to disallow immediate instances of this sequel pattern and some problems concerning virtual instances.

Fortunately, these disadvantages show up in borderline cases, and only realistic usage of the proposal will reveal whether these disadvantages are minor or not.

The discussion of static exception handling in the multi-sequential part of Beta illustrates one important aspect of this proposal — an exception is associated with one and only one control-thread. This implies that exception handling strategies during a rendezvous of control-threads is not an inherent proporty of this proposal — an exception occurrence will only affect one of the involved control-threads. If other exception handling strategies

---

and otherwise not. This would allow the definition, within the language itself, of further structural attributes, without wasting space in those entities that does not utilize them.

are wanted in a particular application, the programmer is free to implement the specific strategy and is not restricted to the language-defined strategy.

The discussion of the implementation of the sequel pattern show that implementation is possible if access is allowed to some of the structural attributes that are normally maintained by the compiler. The implementation have not been tested in the proto-type implementation, yet.

This proposal for introduction of static exception handling in Beta is final in the sence that the next step must be to incorporate the proposal in the proto-type implementation and then resolve the above mentioned problems during the implementation phase and during realistic programming tasks using the proposal.

Acknowledgements

The work reported here have benefited from many discussions with Kristine Stougård Thomsen and Ole Lehrmann Madsen. Thanks are also due to Brian H.Mayoh and Peter Mosses for reading earlier drafts of this paper and giving many valuable comments.

References:

/1/ B.B.Kristensen, O.L.Madsen, B.Møller-Pedersen, K.Nygaard,
Abstraction Mechanisms in the Beta Programming Language,
Proceedings of the 10'th ACM Symposium on Principles of Programming Languages,
Austin, Texas, 1983.

/2/ B.B.Kristensen, O.L.Madsen, B.Møller-Pedersen, K.Nygaard,
Multi-sequential Execution in the Beta Programming Language,
ACM SIGPLAN Notices, **20**(4), 57-70 (April 1985).

/3/ J.L.Knudsen,
Exception Handling — A static Approach,
Software — Practice & Experience, **14**(5), 429-449 (May 1984).

/4/ J.L.Knudsen,
A Hierarchical, Co-operative Exception Handling Mechanism,
DAIMI PB-204, Computer Science Department, Aarhus University, January 1986.

/5/ A.Goldberg, D.Robson,
Smalltalk-80: The Language and its Implementation,
Addison-Wesley Publishing Compagny, 1983.