ISSN 0105-8517

A Hierarchical, Co-operative Exception Handling Mechanism

Jørgen Lindskov Knudsen

DAIMI PB - 204
January 1986

PB - 204 J.L.Knudsen: Exception Handling Mechanism

A Hierarchical, Co-operative Exception Handling Mechanism

Jørgen Lindskov Knudsen
Computer Science Department
Aarhus University, Denmark

Abstract

One of the most dominant philosophies within programming disciplines is the philosophy of layered systems. In a layered system (or hierarchical system) the layers are thought of as each implementing an abstract machine on top of the lower layers. Such an abstract machine in turn implements utilities (e.g. data-structures and operations) to be used at higher layers.

This paper will focus on exception handling in block-structured systems (as a special case of layered systems). It will be argued that none of the existing programming language proposals for exception handling support secure and well-behaved termination of activities in a block-structured system. Moreover, it is argued that certain termination strategies within block-structured systems cannot be implemented using the existing proposals. As a result of this discussion and as a solution to the problems, a hierarchical, co-operative exception handling mechanism is proposed.

Introduction

The area of exception handling has been the topic of many papers during the last ten years. The foundation of the work has been the pioneering work by J.B. Goodenough[9] and most proposals for exception handling within programming languages owe a lot to this work. As a consequence hereof the proposals share at least one characteristic, namely that exceptions and their handlers are defined separately and handlers are associated with exceptions either by binding imperatives (e.g. bindings that are executed at run-time), or by dynamic bindings (e.g. exception handlers are found in the dynamic context of the raising statement).

In this paper we will propose an exception handling mechanism that does not share the above characteristic. We will examine the possibility of static binding of handlers to exceptions. To be more precise, we examine the possibility of declaring the exception and its handler together. Moreover, we examine static definition of the termination level. That is, static determination of which parts of the system should be destroyed as a consequence of raising a specific exception.

The origin of the static approach to exception handling is the **sequel** concept, first introduced by R.D. Tennent[24,25]. The origin of the sequel concept is the program-point concept, introduced by Landin[13,14]. The static approach to exception handling is introduced by this author in a previous paper[10] that was published independently of a paper by R.D. Tennent, who examined the introduction of sequels into Pascal[26]. (Note, that R.D. Tennent uses the term **exit** instead of **sequel** in reference 25 and 26.)

The present paper extends the proposal to take into account several aspects of exception handling within block-structured systems. When an exception occurs in a block-structured system, an exception handling wave will go through a specific part of the system. This exception handling wave consists of exception handling within specific blocks (those affected by the exception handling wave). In order for this exception handling within a specific block to be secure and well-behaved when it is part of an exception handling wave, one must assume that outer blocks are in a consistent state that makes exception handling in this particular block possible. If consistency of outer levels is not ensured, handling of exceptions is difficult merely by the fact that the particular block cannot assume anything about the state of outer blocks. This is because the exception can occur at any of a number of different places (i.e. the state of outer blocks may be any of a number of different states). Furthermore, the exception handling in this particular block must ensure that the block is in a consistent state before inner blocks are allowed to perform any exception handling actions. (Note, that these consistent states are specific for exception handling and may not be consistent states if no exceptions had occurred.) Finally, the exception handling in this particular block must perform some specific actions, related to the handling of the exception in that block.

In order to cope with exception handling as described above, the static approach to exception handling is extended to include so-called prefixed and virtual sequels. It will be shown that the structure obtained by using prefixed and virtual sequels is well-suited for convoluting exception handling in inner blocks with exception handling in outer blocks. Traditional approaches to exception handling (as the Goodenough proposal) are only well-suited for sequential exception handling in block-structured systems. That is, first perform exception handling in the innermost block involved, then in the next to innermost block, and so on until the outermost block involved is reached.

Exception handling has been dealt with from the viewpoint of verification, too. One notable approach in this direction is the work by F. Cristian[4,5]. The exception handling mechanisms discussed in reference 4 are similar to the Clu mechanisms and those discussed in reference 5 are similar to those of Ada. The aspect of verification of exception handling is important but outside the scope of this paper.

Organization of the paper

The static approach to exception handling is introduced in section 1. Section 2 contains a discussion of the problems of termination in block-structured systems and concludes that new language constructs are needed. Section 3 is the core of the paper and presents the proposal for a hierarchical, co-operative exception handling mechanism that is able to cope with the problems discussed in section 2. Moreover, a discussion of other proposals for hierarchical exception handling is given.

Appendices A,B and C are only introductory material and can be skipped by the knowledged reader. Appendix A is an introduction to prefixing and is intended to be read in connection with section 3.1. Appendix B is an introduction to virtual binding and is intended to be read in connection with section 3.2. Finally, appendix C is a taxonomy for exception handling and gives an introduction to the area of exception handling and clarifies several of the terms from the area. Appendix C is intended to be read either before, or in parallel with, reading the rest of the paper.

1 An Introduction to Static Exception Handling

The static approach to exception handling[10] is an attempt to contribute to a better understanding of writing structured programs with exception handling. The kernel of the static approach is the sequel concept.

1.1 The Sequel Concept

A sequel is an abstraction of the goto-statement and is based on the procedure concept.

A sequel is declared in the following way:

sequel S(...) **begin** ... **end**

A **sequel definition** defines three aspects of exception handling: Firstly, it defines the **name** of an exception. Secondly, it defines the **handler** associated with the exception. Finally, it defines the **termination level** of the exception

Semantically, the sequel concept is similar to the procedure concept except for the transfer of control after the sequel body has been executed. When a sequel is invoked and the sequel body has been executed, control is transferred to the termination point of the block in which the sequel is declared.

A **sequel invocation** will initiate the execution of the body of the sequel. **If** the body terminates successfully, the encloser* of the sequel will be terminated immediately

A sequel invocation will not terminate successfully if, during execution of the body of a sequel, another sequel is invoked. That is, this second sequel will take over and the termination level of the entire exception handling will be the termination level of this second sequel invocation. This applies of course recursively if this second sequel invocation is interrupted by a third sequel invocation.*

The consequences of this semantics are twofold: Firstly, during handling of an exception occurrence it might be discovered that the exception cannot be handled locally — that is, invocation of a sequel declared in an outer block. Secondly, it might be possible to discover that the exception can be handled more locally — that is, invocation of a sequel declared in an inner block.

The example in figure 1 illustrates the use of sequels for exception handling. In the example, three sequels are declared: *TableError*, *SearchError* and *ItemFound*. They are declared in three different blocks and function *TableSearchAndCount* has a formal sequel parameter *ItemNotFound*. The sequel represents three aspects of exception handling: definition of the exception, association of the exception statically with a handler (the sequel body), and definition of the termination level. The semantics of the sequel concept are such that if *TableError*, *SearchError* or *ItemFound* is invoked and terminates successfully then it will result in termination of block B_1, block B_2, or the current invocation of function *TableSearchAndCount*, respectively. The semantics of sequels as parameters are such that if *ItemNotFound* is invoked within *TableSearchAndCount(A',X',SearchError)* and terminates successfully, then it will result in termination of block B_2 whereas within *TableSearchAndCount(A',X',TableError)* it will result in termination of level B_1. The difference is due to the termination level of *SearchError* and *TableError*, respectively.

In the previous paper on the static approach to exception handling, numerous examples of the use of the sequel concept are given and the semantics of the sequel concept are discussed in detail.

Note that we use the terms **exception** and **exception occurrence** when we discuss certain events during program execution, whereas we use the terms **sequel** and **sequel invocation** when we discuss the specific language construct, designed to be used for static exception handling.

1.2 Static vs. Dynamic Exception Handling

In this section we will give a brief comparison of the static approach to exception handling with the dynamic approach to exception handling taken by Clu[16]. We have chosen Clu because it is a very strict proposal. I.e. the dynamic aspects of the proposal are limited compared with other dynamic approaches such as the proposal by Goodenough[9], PL/I[17] and Ada. Moreover, the Clu proposal has been the inspiration of a proposal for an exception handling mechanism in Pascal[3].

The Clu proposal consists of three parts. Firstly, each procedure or iterator (hereafter called routines) declares which exceptions it might raise during execution of the body of the routine. All exceptions raised within a routine are propagated to the immediate caller which *must* handle the exception. This rule is not followed strictly: Any exceptions not handled by the caller are converted into the universal exception *Failure*. This exception

* By encloser is meant the program unit (e.g. block, procedure invocation) in which the sequel is declared

* Please note, that this semantics of sequel invocation is not concistent with the semantics described in section 2.9 of the previous paper[10]. In fact, the semantics described there does not solve the problems discussed and the semantics described here is the only realistic alternative. The semantics of section 2.9 in the previous paper should therefore be abandoned

```
declare (* block $B_1$ *)
  ...
  function TableSearchAndCount(A:Table; X:Item;
                              sequel ItemNotFound(Item)):TableIndex;
  declare
    var I : TableIndex;
    sequel ItemFound(J:TableItem);
    begin
      A[J].Occurrences := A[J].Occurrences + 1;
      TableSearchAndCount := J;
    end;
  begin
    I := Hash(X);
    loop
      if     A[I].Item = X        then ItemFound(I)
      elseif A[I].Item = NullItem then ItemNotFound(X)
      end;
      I := (I mod TableMax) + 1;
    end;
    TableSearchAndCount := I;
  end;
  ...
  sequel TableError(X:Item);
  begin ... end;
  ...
begin
  ...
  declare (* block $B_2$ *)
    ...
    sequel SearchError(X:Item);
    begin ... end;
    ...
  begin
    ...
    ItemPos := TableSearchAndCount(A', X', SearchError);
    ...
  end;
  ...
  ItemPos := TableSearchAndCount(A', X', TableError);
  ...
end
```

Figure 1: *An example of sequels used for static exception handling*

can be propagated through any routine-call without being handled. Secondly, exceptions are handled by the so-called *catch-phrases* attached to routine-calls. Thirdly, in order to be able to do local exception handling (i.e. exception handling internally in a routine) it is possible to raise exceptions that *must* be handled within the routine itself and cannot be propagated. If the exception is not handled locally, it is converted into the previously mentioned *Failure* exception. (Local exceptions are raised by the **exit**-statement whereas non-local exceptions are raised by the **signal**-statement.)

To illustrate the differences between the dynamic approaches and the static approach to exception handling, two examples are given (figure 2 and figure 3). The examples are the same (useless) program, formulated in Clu and using the static approach, respectively. The example program is taken from reference 15.

```
sign = proc(x:int)returns(int)signals(zero,neg(int))
          if     x<0  then signal neg(x)
          elseif x=0  then signal zero
          else             return(x)
       end sign
       .
        .
         .
         begin
           a := sign(x) except when neg(i:int)
                                   S1;
                                   exit done
                        end
           b := sign(y) except when neg(i:int)
                                   S2;
                                   exit done
                        end
         end except when done
                         S3
             end
```

Figure 2: *A Clu example with exception handling*

These examples illustrate several differences. First of all, the static approach gives a more clear separation of the specification of the standard execution and the exceptional execution (textually separated). Secondly, specifically with respect to Clu, the examples show that within the static approach there is no need for two different ways of raising exceptions, as in Clu (**signal** (non-local exceptions) and **exit** (local exceptions)). This is because the static definition of the termination level makes it possible for the programmer to explicitly state the difference between local and non-local exception at the time of definition of the exception. (I.e. in the static approach, the termination level is a property of the exception declaration and not the raising statement.) Thirdly, except for the sequel concept no additional concepts need to be introduced. Finally, although not explicit in the examples above, it is impossible, using the static approach, to raise an exception without it being handled. This avoids the need for such mechanisms as *Failure* exceptions with special propagation rules as in Clu.

```
function sign(x:integer; sequel zero; sequel neg(integer)):integer;
begin
  if     x<0  then neg(x)
  elseif x = 0 then zero
  else              sign := x
  end
end
.
 .
  .
  declare
    sequel done; begin S3 end;
    sequel neg1(i:integer); begin S1; done end;
    sequel neg2(i:integer); begin S2; done end;
    sequel zero; begin ... end;
  begin
    a := sign(x,zero,neg1);
    b := sign(y,zero,neg2)
  end
```

Figure 3: *The Clu example implemented using sequels*

1.3 Discussion of Derived Definition

In addition to the sequel concept, derived definition is presented in the previous paper. Derived definition is not directly tied to exception handling, but is a very useful generalization of Ada's derived type and generic definition[18]. However, the application of derived definition to sequels has shown some advantages that justify a discussion of derived definition in connection with exception handling.

Derived definition can be illustrated by modifying the example in figure 3. Assume that we within the **declare**-block several times want to handle the exceptions *neg* and *zero* as in the assignments to *a* and *b*. This fact can be expressed explicitly by derived definition as illustrated in figure 4.

That is, derived definition is used to specialize a parameterized program unit (such as a function, procedure, sequel, ...) by providing (some of) the parameters such that these parameters are constantly bound within the scope of the derived definition. The concept of "curried" functions (see for instance reference 23) is an example of this specialization technique used for functions.

Unfortunately, the previous paper on the static approach did not discuss the semantics of derived definition applied to sequels. The problem is the definition of the termination level. Fortunately, the nature of the sequel concept leaves only two possibilities. Let us discuss the example of figure 5.

The termination level is either block B_1 (i.e. the encloser of *S*), or block B_2 (i.e. the encloser of the derived definition of *S'*). In the latter case, the relationship between *S* and *S'* is only a matter of "lending" code. We find that *S'* is a specialization of *S* and as such, *S'* must inherit *all* of the properties of *S*, including the termination level. That is, the termination level of *S'* must be the termination level of *S* (i.e. block B_1).

All in all, *S'* is a shorthand for *S* in that some of the parameters of *S* may be bound in *S'* and furthermore the remaining parameters of *S* may be specialized in *S'*.

In a recent report on a conceptual framework for programming languages[11], the derived

```
function sign(...):...;
begin
  (* as in example 3 *)
end;
.
 .
  .
  declare
    (* declarations of sign, neg1, neg2 and zero as in example 3 *)
    function a-sign(x:integer) is new sign(x,zero,neg1);
    function b-sign(x:integer) is new sign(x,zero,neg2);
  begin
    .
     .
     a := a-sign(x); ... x := b-sign(z);
     .
     .
     b := b-sign(y); ... y := a-sign(r);
    .
    .
  end
```

Figure 4: *An example of the use of derived definition on sequels*

```
declare (* block B_1 *)
  ...
  sequel S(...); begin ... end;
  ...
begin
  ...
  declare (* block B_2 *)
    ...
    sequel S'(...) is new S(...);
    ...
  begin
    ...
    .. S'(..);
    ...
  end;
  ...
end
```

Figure 5: *What is the termination level of S'?*

definition concept is discussed in further detail as a general language mechanism supporting specialization.

As it will be shown in the following, the hierarchical, co-operative exception handling mechanism contains derived definition as a special case and we therefore do not discuss derived definition of sequels any further.

2 Discussion of Termination in Block-Structured Systems

Many of the proposals for exception handling mechanisms are based on a termination model. That is, as a consequence of the occurrence of an exception some part of the system is terminated. In this section we will discuss the problems of termination in block-structured systems as a motivation for a proposal for a hierarchical, co-operative exception handling mechanism.

Let us assume that we have a block-structured system as outlined in figure 6 and let us

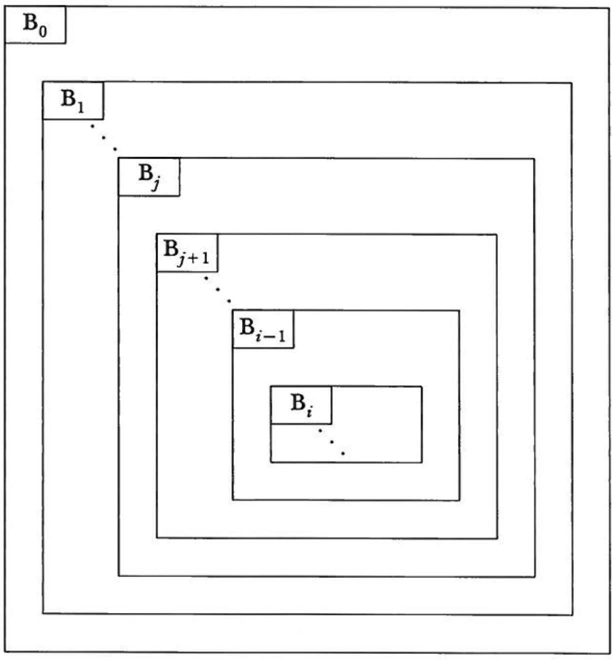


Figure 6: *A block-structured system*

further assume that an exception occurrence has been identified in block B_i and therefore an exception (named E) has been raised in block B_i.

Assuming that the exception handling mechanism being used follows a termination model, two problems arise: How do we determine the termination level? (i.e. the outermost block that must be terminated as a consequence of the exception occurrence), and how do we terminate each intermediate block?

The problem of how to determine the termination level has been discussed in length in the previous paper[10] and will furthermore be summarized in section C.2.3. We will therefore only discuss the termination process here.

Let us assume that the termination level is found to be block B_j ($j < i - 1$; i.e. there is at least one intermediate block between the raising block B_i and B_j). That is, blocks

$B_i, B_{i-1}, \ldots, B_{j+1}, B_j$ have to be terminated. The most common termination process is the following: Terminate blocks $B_i, B_{i-1}, \ldots, B_{j+1}$ abruptly (i.e. no local clean-up actions possible) after which block B_j is allowed to do some local clean-up actions (specified in the exception handler for the exception) before it terminates.

In many cases this is too abrupt since the intermediate blocks $B_i, B_{i-1}, \ldots, B_{j+1}$ have no opportunity to do local clean-up actions. Several of the proposals do therefore contain mechanisms that is oriented towards this situation (see section 3.5).

Unfortunately, there exist a range of exception handling strategies that cannot be implemented using any of these mechanisms. Let us again consider the system outlined in figure 6 in which exception E is raised in block B_i and where the termination level is block B_j.

Now let us further assume that exception E represents an erroneous computational state of the program and that we want some debugging information to be written on a specific file X in the case of E being raised. Let us further assume that clean-up handlers $H_{j+1}, \ldots, H_{i-1}, H_i$ (as discussed above) are specified in blocks $B_{j+1}, \ldots, B_{i-1}, B_i$.

The problem is: Where do we specify the opening of the file X. One solution (which we for obvious reasons will abandon) is to open the file at initialization of block B_j regardless of whether E will ever happen. If we specify the opening in handler H_j in block B_j, the handlers $H_{j+1}, \ldots, H_{i-1}, H_i$ are unable to write debugging information on X since it is not open when these handlers are activated (since the handlers are activated in sequence: $H_i, H_{i-1}, \ldots, H_j$). Then, the only possible solution is to specify the opening in handler H_i. If, on the other hand, it could happen that E were raised in block B_{i-1}, handler H_{i-1} has to be able to open X. This implies that all handlers might specify opening of X. This implies further that a very careful programming style is needed in order to ensure that the file is opened when needed and that only one handler actually opens the file.

In general, the problem is that in some cases, a block must be in a consistent state before any termination process of the inner blocks can be initiated in order to ensure a consistent termination of these inner blocks. That is, the termination of a block is a three-phased process: Firstly, the state of the block must be made consistent. Secondly, the inner blocks are allowed to terminate consistently and finally, the block itself may perform some local clean-up actions, ensuring further consistency.

In the example above, the consistent state with respect to exception handling of E in block B_l, is that file X is open and that the blocks B_j, B_{j+1}, $\ldots$, B_{l-1} have all done their preample debugging.

One final requirement is that the clean-up actions of each intermediate block can be made dependent on the identity of the exception that initiated the termination process.

—oOo—

We find that the two termination processes outlined above are very different and will therefore use two different terms to identify them: **Abrupt termination** and **smooth termination.**

By **abrupt termination** is meant that blocks B_i, B_{i-1},-$\ldots, B_{j+1}$ are all terminated immediately, after which the handler in block B_j is executed, and then block B_j is terminated

By **smooth termination** is meant that the blocks B_i,-$B_{i-1}, \ldots, B_j$ are each given the opportunity to do some local clean-up actions before they are terminated

We have not seen smooth termination discussed elsewhere although its relevance is demonstrated by the above discussion. In section 3.5 other proposals for hierarchical exception handling will be discussed (AML/X[20], Mesa[19], Multics PL/I[22], Clu[16], Ada[18], ANSI/IEEE Pascal proposal[27] and Taxis[21]).

In the next section we will introduce an extension to the static approach to exception handling. An extension that allows smooth termination without sacrificing the static behavior of the approach.

The prime source of inspiration to the proposal can be found in an article on prefixed procedures[28]. Prefixing used as a general program structuring mechanism is a very important aspect of the programming language Beta[12], and is discussed in depth in reference 11.

3 Hierarchical, Co-operative Exception Handling

Inspired by the above discussion we want to define an exception handling mechanism that allows hierarchical exception handling. Furthermore, the multi-level termination process should involve co-operation of the involved levels during the termination process (i.e. smooth termination).

Such an exception handling mechanism can be obtained by replacing the derived definition concept by the far more powerful prefixing concept. For an introduction to prefixing, see appendix A.

3.1 Prefixed Sequels

A prefixed sequel is similar to a derived sequel — the main difference is that the body-part of the sequel can be expanded by the prefixed sequel. Assume that sequel *S* is defined in an outer block, then sequel *S'* can be defined as a prefixed sequel by means of the following declaration in an inner block:

sequel S'(...) **prefix** S(...) **begin** ... **end**;

We call *S'* the prefixed sequel and *S* the prefix of *S'*.

A **prefixed sequel** is a specialization of another sequel. The termination level of a prefixed sequel is the termination level of the prefix. The body of a prefixed sequel is a specific combination of the body of the prefix and the body of the prefixed sequel itself

Figure 7 illustrates the prefixed sequel concept by repeating figure 5 now formulated using prefixed sequels.

The differences between the two examples are the INNER statement in *S* and the prefix- and body-part of *S'*. The semantics of *S* are the same here as in figure 5 — in the case of *S* being invoked explicitly, the INNER statement is equivalent to the *Skip* statement. The semantics of *S'*, on the other hand, are as follows: The termination level of *S'* is the termination level of the prefix (i.e. block B_1). When *S'* is invoked, control is immediately passed to the prefix *S* which then executes *code1*. In this case, execution of the INNER statement means execution of the body of the prefixed sequel *S'*. Execution of *S'* means execution of *code3* after which control is returned to *S* which in turn executes *code2*. If *S*

```
declare (* block B1 *)
  ...
  sequel S(...) begin code1; INNER; code2; end;
  ...
begin
  ...
  declare (* block B2 *)
    ...
    sequel S'(...) prefix S(...); begin code3 end;
    ...
  begin
    ...
    .. S'(..);
    ...
  end;
  ...
end
```

Figure 7: *The example in figure 5 implemented using prefixed sequels*

terminates successfully, then blocks B_1 and B_2 will be terminated immediately. Note, that the body of *S'* will not be executed if either no INNER-statement is present in the body of *S*, or if the INNER-statement in the body of *S* is not executed.

In general, assume that we have sequels

$$S_0, S_1, \ldots, S_i, \ldots, S_n$$

where S_i is prefixed with S_{i-1} (as illustrated in figure 8). Then the termination level of S_i is the same as the termination level of S_0. When S_i is invoked, control is passed to S_0 which executes pre_0 and then passes control to S_1 (as a result of the INNER statement). S_1 then executes pre_1, etc. until control reaches S_i which executes pre_i then INNER as a *Skip* statement (since S_i is the invoked sequel), then executes $post_i$, and then passing control to S_{i-1} which executes $post_{i-1}$, and so on until control reaches S_0 which executes $post_0$. If this computation terminates successfully, then the blocks $B_0, B_1, \ldots, B_i$ are terminated immediately. That is, the execution history of invoking S_i is:

```
exe(S_i) => exe(pre_0)
               exe(pre_1)
                 .
                  .
                   .
                    exe(pre_i)
                    exe(Skip)
                    exe(post_i)
                   .
                  .
                 .
               exe(post_1)
            exe(post_0)
            terminate(block B_i)
            terminate(block B_i-1)
            .
            .
            terminate(block B_0)
```

```
sequel $S_0$ declare
          $decl_0$
        begin
          $pre_0$;
            INNER;
          $post_0$
        end;
.
.
 sequel $S_1(\ldots)$ prefix $S_0(\ldots)$ declare
                              $decl_1$
                            begin
                              $pre_1$;
                                INNER;
                              $post_1$
                            end;
.
.
.
  sequel $S_i(\ldots)$ prefix $S_{i-1}(\ldots)$ declare
                                 $decl_i$
                               begin
                                 $pre_i$;
                                   INNER;
                                 $post_i$
                               end;
.
.
   sequel $S_n(\ldots)$ prefix $S_{n-1}(\ldots)$ declare
                                  $decl_n$
                                begin
                                  $pre_n$;
                                    INNER;
                                  $post_n$
                                end;
```

Figure 8: *Multi-layered prefixing of sequels*

Additionally, within a prefixed sequel, the identifiers, etc. that are declared local to the prefix (and local to its prefix, etc.) are accessible. That is, S_i may use global identifiers, and all identifiers declared locally in $S_0, S_1, \ldots, S_{i-1}, S_i$ (i.e. declared in decl_0, decl_1, ..., decl_i).

Similarly to derived definition, it is possible to let a prefixed sequel bind some (or all) of the parameters defined in the prefix.

The control flow of prefixed sequels is complex, but this complexity is justified by four factors. First of all, all prefixed sequels follow the same pattern of control flow; I.e. it is therefore a matter of comprehending this uniform pattern of control flow initially, and then utilize this pattern when approaching any actual prefixed sequel. Secondly, the problem of hierarchical exception handling as discussed above contains inherited complexity that must be reflected in *any* solution, whether it is explicitly programmed using one of the traditional methods, or by means of a specific language construct as prefixed sequels. Thirdly, as pointed out by Dijkstra[7] understanding of programs is eased if the static structure of the program reflects directly the structure of the corresponding computation. It is our claim that the static properties of prefixed sequels directly reflect the structure of the corresponding computation, and that the benefits of the proposal outweight the initial difficulty in learning the concept. Moreover, implementation of prefixed sequels does not cause severe problems. Finally, the complexity of prefixed sequels (and virtual sequels, see later) is found to be comparable to the complexity of the exception handling mechanisms of languages like Ada.

Unfortunately, prefixed sequels are only a partial solution to the problem discussed in section 2. Indeed, prefixed sequels represent a hierarchical, co-operative exception handling mechanism that solves the problem of where to specify the opening of file X (namely in pre_0 — and X can then in turn be closed in $post_0$). Moreover, as the scope of the identifiers declared in the prefix includes the prefixed sequel, declaring X local to S_0 would prevent unintended access to X (i.e. accesses other than for debugging). The deficiency of prefixed sequels is that invoking S_i in block B_i will cause smooth termination of blocks $B_i, B_{i-1}, \ldots, B_1, B_0$, but invoking S_j ($j < i$) in block B_i will cause abrupt termination of blocks $B_i, B_{i-1}, \ldots, B_{j+1}$. Initially, this seems to be a problem of only minor concern, but if S_j is invoked within a routine declared in block B_j but invoked from block B_i, the problem becomes apparent. Of course, returning to a dynamic approach would give an easy solution, but would abandon the merits of the static approach. We therefore extend the static approach (slightly) to cope with the problem — the extension is termed virtual sequels. For an introduction to virtual binding, see appendix B.

3.2 Virtual Sequels

A virtual sequel is very similar to an ordinary sequel or a prefixed sequel, and can be declared in one of the following ways:

```
sequel S(...) virtual                  begin ... INNER ... end;
sequel S(...) virtual prefix S'(...)   begin ... INNER ... end;
```

The first declaration declares a virtual sequel by giving its default body (handler). (We refer to this first declaration as the initial virtual binding.) The second declaration combines the prefixing and virtual concepts — that is, declares a virtual sequel by giving its default body (handler) as a prefixed sequel.

A **virtual sequel** allows inner blocks to augment its body (handler) in such a way that **all** invocations of the sequel will invoke the **augmented** handler as long as the inner block is active. The termination level of a virtual sequel is the termination level of the initial virtual binding

The inner blocks may issue a further binding of the virtual sequel by means of the following declaration:

sequel S(...) **virtual bind begin** ... INNER ... **end**

The example in figure 9 illustrates the virtual sequel concept. In block B_1, S is declared

```
declare (* block B1 *)
  ...
  sequel S(...) virtual begin ... INNER ... end
  ...
begin
  ...
  declare (* block B2 *)
    ...
    sequel S(...) virtual bind begin ... INNER ... end
    ...
  begin
    ...
    .. S(..)
    ...
  end;
  ...
end
```

Figure 9: *Virtual binding of sequels*

virtual and in block B_2, a further binding of S is done.

A virtual sequel is a sequel that allows inner blocks to augment its handler. In many respects, the semantics of sequel S in block B_2 is the same as sequel S' in figure 7 — that is, sequel S in block B_2 can be considered as prefixed with sequel S in block B_1. The difference between the semantics of S in block B_2 in figure 9 and S' in figure 7 is that in figure 9, invoking S in block B_2 (either directly as in figure 9 or indirectly through a routine declared in block B_1) will always lead to smooth termination of block B_2.

In general, if sequel S is declared virtual in block B_0 then any inner block is allowed to augment the sequel by a further binding (as S in block B_2 in figure 9). The semantics of a virtual sequel define the termination level to be the termination level of the prefix, and moreover a partial handler that is concerned with the clean-up actions in that block. Furthermore, inner blocks are allowed to augment the handler to ensure local clean-up actions in those blocks. In some respects, virtual sequels resemble dynamic exception handling but it is very important to note that when using virtual sequels it is impossible to nullify previous handlers — it is only possible to augment the previous handlers to take care of local clean-up. Even the termination level is static. We find that virtual sequels are a far more structured tool for co-operative exception handling than the dynamic approaches to exception handling as discussed in section 3.5. The advantages of virtual sequels are that the static structure of the program reflects the desired computation directly, and moreover that the co-operation among the handlers during the smooth termination process is far more structured than can be obtained using any of the dynamic approaches.

3.3 Default Exception Handling

Default exception handling covers two distinct notions: *Default Exception Handler* and *Default Smooth Termination* (see appendix C). In several proposals these notions are dealt with by means of specialized constructs such as default handlers in Taxis, and *Others*-clauses in Clu and Ada.

In the static approach it is always the case that an exception has a handler (namely the body of the sequel) and a notion of default handlers is therefore superfluous. The notion of virtual sequels can though be thought of as default binding of a handler in case no further bindings are defined.

Default smooth termination cannot immediately be dealt with within the static approach. However, virtual binding can be used to simulate a similar behaviour as illustrated by the example in figure 10.

```
declare (* block B1 *)
  sequel Default virtual begin code1; INNER; code2 end
begin
  .
   .
   declare (* block B2 *)
     sequel S prefix Default begin code3; INNER; code4 end
   begin
     .
      .
      declare (* block B3 *)
        sequel Default virtual bind
        begin code5; INNER; code6 end
        ...
        sequel S' prefix S
        begin code7; INNER; code8 end
      begin
        .
        declare (* block B4 *)
          sequel Default virtual bind
          begin code9; INNER; code10 end
        begin
          ... S' ...
        end
        .
      end
      .
     .
   end
  .
   .
end
```

Figure 10: *Simulation of Default Smooth Termination*

Let us examine what happens when *S'* is invoked in block B_4. Since *S'* is prefixed with *S* which in turn is prefixed with *Default* and *Default* is virtual with further bindings in block B_3 and B_4, the following execution path will be followed:

```
code1; code5; code9; code3; code7; Skip;
code8; code4; code10; code6; code2; terminate(B1)
```

thereby terminating blocks B_2, B_3 and B_4 as well. This usage of virtual prefixing results in a two-phased exception handling technique. The first phase is the default exception handling of all the blocks in which the virtual prefix is further bound (e.g. code1;code5;code9 and code10;code6;code2 in the above execution path). The second phase is exception handling of the particular exception occurrence (e.g. code3;code7 and code8;code4 in the above execution path).

However, the above usage of virtual prefixing is not the ultimate solution to default smooth termination. First of all, if *S* had not been prefixed with *Default*, B_4 would have been terminated abruptly. That is, it is not possible to ensure that default smooth termination always is applied to a specific block without examinating all sequels that might be invoked in inner blocks. That is, default smooth termination will not be a property of the block itself, but a property of some of the sequels that might be invoced in the block, or in inner blocks. Secondly, as a consequence of this simulation, the termination level of *S'* will be the same as the termination level of *Default*. That is, simulating default smooth termination as above will interact with the termination level of *S'*. However, if the initial virtual binding of *Default* were at the same level as *S*, there would not be any problems with this interaction. This, however, leads to a programming style in which any declaration of a sequel is accompanied with an initial virtual binding of a default sequel to be used as a virtual prefix of the original sequel. That is, all sequels will be declared as virtually prefixed.

This programming style is for most practical usages comparable to declaring the original sequel by means of an initial virtual binding. In fact, using virtual sequels is a far more simple mechanism for default smooth termination that the above mentioned method and should be preferred. There is, however, one drawback with using virtual sequels as the basis for default smooth termination: Default smooth termination is still a property of the individual sequels and not a property of the individual blocks. In fact, virtual sequels do not support default smooth termination as defined in appendix C but on the other hand, virtual sequels alliviate the need for direct support of default smooth termination.

In order to support default smooth termination within the static approach, we have to introduce the concept of default sequels. A default sequel is declared in the following way:

sequel S(...) **default begin** ... INNER ... **end**

If a default sequel in invoked explicitly, its semantics are the same as if **default** had not been specified (that is, exactly as the equivalent simple sequel).

However, a default sequel can also be invoked implicitly as part of an exception handling wave. Implicit invocation will take place if the block in which the default sequel is declared, is terminated abruptly. When a default sequel implicitly becomes part of an exception handling wave, it is inserted in the prefix chain of the explicitly invoked sequel. The position of the default sequel in the prefix-chain is determined in the following way:

> Let the prefix-chain of the explicitly invoked sequel *S* be $E_0, E_1, \ldots, E_n$ (i.e. E_i is prefixed with E_{i-1} and $E_n = S$). Let $E_0, E_1, \ldots, E_n$ be declared in blocks

B_0, B_1, ..., B_n. Let the default sequel D be declared in block B. Then $B_j < B < B_{j+1}$ for some j ($0 \leq j \leq n$), and therefore D is implicitly inserted in the prefix-chain: $E_0, \ldots, E_j$, D, $E_{j+1}, \ldots, E_n$. If S is non-virtual, this is read: E_0, D.

Of course, all default sequels that are involved in the exception handling wave is implicitly inserted as above, and their relative positions in the chain are defined by the nesting of their defining blocks. That is, the resulting prefix-chain before the exception handling is initiated will be:

$$E_0, \ldots, E_i,\ D_1,\ E_{i+1}, \ldots, E_j,\ D_2,\ D_3, \ldots, D_k, E_{j+1}, \ldots, E_n,\ D_l, \ldots, D_m$$

That is, all default sequels corresponding to blocks that are abruptly terminated, are inserted in the prefix-chain according to the nesting of their defining blocks.

As it can be seen from above, the static approach needs specialized constructs for default smooth termination — just as Clu, Ada and others need *Others*-clauses to deal with it. This seems to be an inherited property of default smooth termination and we should therefore not be surprised that specialized constructs appear in the static approach too.

3.4 Hierarchical, Co-operative Exception Handling — An Example

In this section we will present yet another example of the usage of the proposed language mechanisms for hierarchical, co-operative exception handling. Unfortunately, such examples are extensive by the nature of the domain, but it is our hope, that the following example will prove valuable.

Let us consider the following scenario. In some part of a program a resource is used. The resource is controlled by the well-known *request-release* scheme. We further assume that the resource is utilized through some utility operations (not important here), and in addition the resource has an *Undo* operation — undo as much as possible of the work done since last *request*.

During the utilization of the resource, the program engages in a specific communication pattern with some third party. We assume that this communication pattern is described in a manner similar to scripts[8]. We assume that the script is controlled by the following operations: *Engage* is used by an object to indicate that it is willing to engage in the script; *Disengage* is used by an object to indicate that it considers its obligations in the script as fulfilled; *ScriptError* is used by an object to indicate that it is not able to fulfill the obligations of the script (e.g. due to some exception occurrence internally in the object); And moreover, there may be additional script control operations (not important here). We assume finally that if the script cannot be fulfilled, then the resource must undo upto last *request*. (This example might seen imaginary but think of the resource as a database system and the script as describing a particular database transaction.)

The problem can now be formulated as follows: How do we handle an exception occurrence during engagement in the script in such a way that, first of all, the script is informed properly and secondly, the resource is informed to undo and then released.

Using the proposed language mechanisms in this paper, the problem can be formulated as outlined in figure 11.

Note first of all the clean separation of exception handling with respect to the resource and the script, respectively. It should also be noted that, although textually separated, the exception handling of the script is convoluted in the exception handling of the resource.

Although this example is programmed using prefixing, it could have been programmed using virtual binding (even with additional benefit). If we had declared *ResourceError* as

```
declare (* block $B_1$ *)
  sequel ResourceError
  begin
    Resource!Undo;
    INNER;
    Resource!Release
  end
begin
  Resource!Request;
  .
   .
    .
    declare (* block $B_2$ *)
      sequel LocalError prefix ResourceError
      begin
        Script!ScriptError;
        INNER;
        Script!DisEngage
      end
    begin
      Script!Engage;
      .
       .
        .
        LocalError; (* Exception occurrence *)
        .
       .
      .
      Script!DisEngage
    end
    .
   .
  .
  Resource!Release
end
```

Figure 11: *Hierarchical Exception Handling — An Example*

virtual in block B_1 and further bound it in block B_2 instead of declaring *LocalError* in block B_2, then we would have had the additional benefit that even if *ResourceError* were invoked indirectly by an operation declared in B_1, exception handling of the script would be performed properly.

3.5 Mechanisms towards Hierarchical Exception Handling in Other Proposals

In this section we will give a brief overview of some of the other proposals for exception handling mechanisms and relate them to our proposal for a hierarchical, co-operative exception handling mechanism.

AML/X

The hierarchical exception handler binding mechanism of AML/X[20] can be used to specify a hierarchical structure of exceptions very similar to the one that can be specified using derived definition of sequels. The main differences between the two proposals are that the AML/X proposal is highly dynamic (e.g. allows computable association of handlers), and furthermore substitution of parameters down the hierarchy is not possible in AML/X. Furthermore, the AML/X proposal is a very divergent proposal that is intended to be useful for many very different ways of handling exceptions. The hierarchical part of the AML/X proposal is essentially an exception renaming mechanism similar to the possibility of renaming exceptions in Ada[18]. That is, the hierarchical structure is not suitable for the specification of smooth termination as discussed above.

Mesa

The exception handling mechanism in Mesa[19] contains the so-called *Unwind*-mechanism that to some degree alleviates the problem of hierarchical exception handling as discussed in section 4. Intuitively, the *Unwind*-mechanism behaves as if the exception *Unwind* is raised within each intermediate block just before it terminates. The termination level of *Unwind* is always the same as the block itself. This means that if each intermediate block specifies a handler for exception *Unwind*, local clean-up actions can be specified in that handler. The remaining problem is that the clean-up actions in this way are always the same irrespective of which exception originally caused the termination process to be initiated. In this respect the *Unwind*-mechanism is similar to default sequels.

The Multics System[22] has extended the PL/I exception handling mechanism to include an *Unwind*-mechanism similar to that of Mesa.

Clu, Ada, a.o.

Most other proposals alleviate the problem of hierarchical exception handling by allowing handlers to reraise exceptions and using a particular programming style. The method is as follows: Each intermediate block defines its own handler for the exception if local clean-up is needed. This handler contains the local clean-up actions and immediately before the terminating end of the handler, the propagated exception is reraised (e.g. Clu and Ada). This method is possible because of the dynamic or computable association of handlers.

Another way to alleviate the problem of hierarchical exception handling is by means of the so-called *Others*-clause (in languages like Ada and Clu). An *Others*-clause in an exception handler specifies that the handler is ready to handle any exception that is propagated to it. The body of the *Others*-clause will be executed if an exception is propagated to the handler and not otherwise handled by that handler. That is, the *Others*-clause specifies default exception handling. The effect of *Unwind* can now be implemented by reraising the propagated exception at the end of the body that is attached to the *Others*-clause.

The ANSI/IEEE proposal for extending Pascal

The joint ANSI/IEEE proposal for introduction of exception handling mechanisms into Pascal (by the joint ANSI X3J9 and IEEE P770 standards committee)[27] contains a hierarchical structuring mechanism for exceptions that is useful both for default exception handling and for renaming of exceptions as the AML/X proposal. It is important to note that the hierarchical structure is a structure of the names of exceptions and not a hierarchical structure of the exception handlers (as the proposals in this paper). Therefore, although the ANSI/IEEE proposal is an interresting proposal, it does not support hierarchical exception handling as discussed in section 3.

Taxis

The exception handling mechanism of Taxis[21] contains similar ideas to those presented in this paper. But there are some basic differences. The exception handling model is single-level termination with dynamic association of handlers. This implies that smooth termination as discussed above does not apply to the Taxis proposal. Moreover, all exceptions are raised implicitly when either pre- or post-conditions of transactions are violated, or the return-value of the transaction does not conform with the specification. The user can specify which exception should be raised when a specific pre- or post-condition is violated, whereas a pre-defined exception is raised when the return-value is nonconformable.

Default exception handling is done by associating a default handler with the exception at the time of definition of the exception. (That is, *static* association of the default handler.)

The hierarchical part of the proposal is divided into two parts. Both exceptions and handlers are organized in a specialization hierarchy but the two hierarchies are independent in the sense that any handler may be associated with any exception (of course, the parameters must match).

The hierarchical structure of the exceptions is used in the same way as the hierarchical structure of exception names in the ANSI/IEEE proposal. That is, a handler associated with a specific exception is capable of handling occurrences of that exception and *any specializations hereof*. The hierarchical structure of the handlers is used to structure the handlers in the same way as transactions (in fact, handlers are transactions).

3.6 Final Remarks

We have presented a hierarchical, co-operative exception handling mechanism as an extension to the static approach to exception handling. The proposal is divided into three parts: Prefixed sequels that make it possible to specify smooth termination; Virtual sequels that make it possible to augment a handler in inner blocks; And finally default sequels that make default exception handling possible within the static approach and thereby extend the possibilities of smooth termination.

Implementation of the static approach has not been tried yet. But since the underlying principles of prefixing and virtual binding are well-known and implemented elsewhere (e.g. Simula67 and Beta), we do not find that implementation of the proposal will give rise to significant difficulties. Implementing sequels is similar to implementing procedures with a non-local goto (to the end of the encloser) immediately before the terminating end.

This proposal is only a proposal for a new language construct, not an entire language proposal. We find that the proposed language constructs can be incorporated into any language in the Algol family (e.g. Algol60, Pascal, Ada). That is, in any statically scoped language. The extra complexity introduced hereby is felt to be in the same line as the extra complexity that is introduced by the exception handling mechanisms of, say, Ada.

With that in mind, we find that prefixing and virtual binding is a generally useful structuring mechanism (as discussed in reference 11 and 2). We would therefore prefer that this proposal was incorporated in a language in which prefixing and virtual binding were general structuring mechanisms. The proposal would in that way be part of a more homogenous language design. As examples of languages of this sort are Galileo[1], Taxis[21], Simula67[6] and Beta[12]. (Note that Galileo and Taxis do not support virtual binding.)

Acknowledgements

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

References

1. A. Albano, L. Cardelli, R. Orsini, "Galileo: A Strongly-Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, **10** (2), (June 1985), 230-260.

2. A. Borgida, "Features of Languages for the Development of Information Systems at the Conceptual Level", *IEEE Software*, **2** (1), (Jan. 1985), 63-72.

3. I.D. Cottam, "Extending Pascal with One-entry/Multi-exit Procedures", *SIGPLAN Notices*, **20** (2), (Feb. 1985), 21-29.

4. F. Cristian, "Exception Handling and Software Fault Tolerance", *IEEE Transactions on Computers*, **31** (6), (June 1982), 531-540.

5. F. Cristian, "Correct and Robust Programs", *IEEE Transactions on Software Engineering*, **20** (2), (Mar. 1984), 163-174.

6. O.-J. Dahl, B. Myhrhaug, K. Nygaard, "Simula 67, Common Base Language", *Publication No. S-22*, Norwegian Computing Center, 1970.

7. E.W. Dijkstra, "Notes on Structured Programming", in O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, "*Structured Programming*", Academic Press, London 1972.

8. N. Francez, B. Hailpern, "Script: A Communication Abstraction Mechanism", *RC 9995 (44379)*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

9. J.B. Goodenough, "Exception Handling: Issues and a Proposed Notion", *Communications of the ACM*, **18** (12), (Dec. 1975), 683-696.

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

11. J.L. Knudsen, K.S. Thomsen, "A Conceptual Framework for Programming Languages", *DAIMI PB-192,* Computer Science Department, Aarhus University, April 1985.

12. 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.

13. P.J. Landin, "A Correspondance between ALGOL 60 and Church's lambda-notation: Part I & II", *Communications of the ACM,* **8** (2&3), (Feb. & Mar. 1965), 89-100 & 158-165.

14. P.J. Landin, "The Next 700 Programming Languages", *Communications of the ACM,* **9** (3), (Mar. 1966), 157-166.

15. P.A. Lee, "Exception Handling in C Programs", *Software — Practice & Experience,* **13** (5), (May 1983), 389-405.

16. B.H. Liskov, A. Snyder, "Exception Handling in Clu", *IEEE Transactions on Software Engineering,* **5** (6), (Nov. 1979), 516-558.

17. M.D. MacLaren, "Exception Handling in PL/I", *SIGPLAN Notices,* **12** (3), (Mar. 1977).

18. Military Standard, "Ada Programming Language", American National Standards Institute, Inc., *ANSI/MIL-STD-1815A-1983,* 22. January 1983.

19. J.G. Mitchell, W. Maybury, R. Sweet, "Mesa Language Manual", *CSL-79-3, Xerox PARC,* April 1979.

20. L.R. Nackman, R.H. Taylor, "A Hierarchical Exception Handler Binding Mechanism", *Software — Practice & Experience,* **14** (10), (Oct. 1984), 999-1007.

21. B.A. Nixon, "A Taxis Compiler", *CSRG Technical Report #33,* Department of Computer Science, University of Toronto, April 1983.

22. E.I. Organic, "*The Multics System*", The MIT Press, 1972.

23. J.E. Stoy, "*Denotational Semantics: The Scott—Strachey Approach to Programming Language Theory*", The MIT Press, 1977.

24. R.D. Tennent, "Language Design Methods besed on Semantic Principles", *Acta Informatica,* **8** (2),(1977), 97-112.

25. R.D. Tennent, "*Principles of Programming Languages*", Prentice-Hall, Inc., 1981.

26. R.D. Tennent, "Some Proposals for Improving Pascal", *Computer Languages,* **8** (3/4), (1983), 125-137.

27. T.N. Turba, "The Pascal Exception Handling Proposal", *SIGPLAN Notices,* **20** (8), (Aug. 1985), 93-98.

28. J.G. Vaucher, "Prefixed Procedures: A Structuring Concept for Operations", *INFOR*, **13** (3), (Oct. 1975).

A Introduction to Prefixing

The concept of prefixing originates from the programming language Simula67[6]. It was further developed by J.G. Vaucher into a structuring mechanism for operations[28] and in the programming language Beta it is used as a general program structuring mechanism[12].

We would like here to give a short general introduction to the concept of prefixing. The foundation of prefixing is the concept of some sort of descriptor* that may consist of a formal parameter part, a declarative part, and an action part. Examples are classes, procedures, functions and sequels.

Let us consider the example in figure 12. In order to make prefixing possible, we

```
P = descriptor($FP_1$,$FP_2$,...,$FP_n$)        - - formal parameter part
      $DP_1$;
      $DP_2$;
      ...                                       - - declarative part
      $DP_m$;
    begin
      $S_1$;
      ...
      $S_{i-1}$;
        INNER;                                  - - action part
      $S_{i+1}$;
      ...
      $S_l$
    end
```

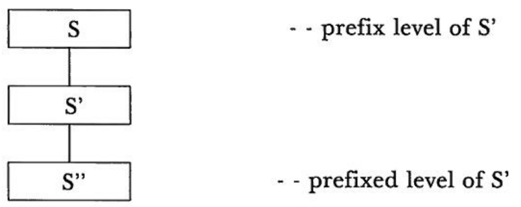
Figure 1: *A descriptor*

have introduced a special kind of action (called INNER). The semantics of INNER will be explained soon.

Figure 13 shows how a new descriptor *P'* can be prefixed with *P*. The overall principle of prefixing is that the formal parameters of *P'* are the union of the formal parameters of *P* and those specified in *P'*. The identifiers declared in *P'* are the union of those of *P* and those specified in the declarative part of *P'*. Finally, the actions of *P'* are the actions of *P* with the action part of *P'* merged into the action part of *P* at the point of the INNER statement.

Figure 14 illustrates the semantics of prefixing. However it should be noted that *P"* is not literally the semantics of *P'* since the bindings in *P* are done in the environment of *P* only whereas the bindings in *P'* are done in an environment similar to that of *P"*. That is, *P* cannot use the bindings in *P'* but *P'* can use the bindings in *P*.

The control pattern of prefixed procedures can be illustrated by figure 15. As an abstraction mechanism, the prefixed procedures *S*, *S'* and *S"* expresses a three layered system:



*For a further discussion of descriptors in programming languages, see reference 11.

```
P' = prefix P                                  - - prefix part
     descriptor(FP'_1,...,FP'_r)               - - formal parameter part
        DP'_1;
        DP'_2;
        ...                                    - - declarative part
        DP'_s;
     begin
        S'_1;
        ...
        S'_{j-1};
          INNER;                               - - action part
        S'_{j+1};
        ...
        S'_t
     end
```

Figure 2: *A prefixed descriptor*

```
P" = descriptor(FP_1,..,FP_n,FP'_1,..,FP'_r)
        D_1;                                   - - concatenation of
        ...                                    - - formal param. parts
        D_m;
        D'_1;                                  - - concatenation of
        ...                                    - - declarative parts
        D'_s;
     begin
        S_1;
        ...
        S_{i-1};
          S'_1;
          ...                                  - - merge action parts
          S'_{j-1};                            - - (i.e. substitute
            INNER;                             - - INNER in P with the
          S'_{j+1};                            - - action part of P')
          ...
          S'_t;
        S_{i+1};
        ...
        S_l;
     end
```

Figure 3: *P" illustrates the semantics of P' in figure 13 — except for the binding aspect of prefixing*

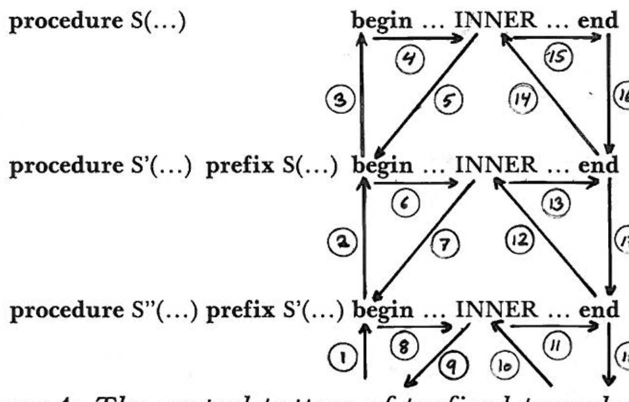

procedure S(...) begin ... INNER ... end
procedure S'(...) prefix S(...) begin ... INNER ... end
procedure S"(...) prefix S'(...) begin ... INNER ... end
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Figure 4: *The control pattern of prefixed procedures*

in which *S'* is a specialization of *S* and *S"* a further specialization of *S'*. For a discussion of prefixing as a general abstraction mechanism, see reference 2 and 11.

The execution of each prefixed procedure can be seen as a five phased execution. When the control enters the prefixed procedure, it is immediately passed to the prefix levels to let those execute their initial actions. When control is passed back from the prefix levels, the prefixed procedure itself executes some initial actions (e.g. initialize local data-structures and bringing them into a consistent state). Then, when control reaches an INNER statement, it will be passed to the prefixed level (if any). When the prefixed levels have terminated their actions, control is passed back to the prefixed procedure that then executes some finalization actions (e.g. local clean-up, collecting results from the prefixed levels, etc.). Finally, when the prefixed procedure has executed all of its actions, control will be passed to the prefix levels.
This pattern of control can be pictured as in figure 15.

Note, that in the case of the prefixed level being non-existent, the INNER statement acts as a Skip/NoOp statement (see INNER in *S"* of figure 15).

B Introduction to Virtual Binding

The origin of the concept of virtual binding is the programming language Simula67[6] and is further developed as part of the programming language Beta[12]. In both languages a virtual binding can be introduced in a class definition and further binding is only allowed in sub-class definitions of that class.

We extend the concept of virtual binding slightly. First of all, we assume in this paper that virtual bindings can be introduced in ordinary blocks as well. Furthermore, we allow further bindings of virtual bindings in outer blocks to appear in inner blocks. That is, virtual bindings introduced in ordinary blocks are only allowed to be further bound in inner blocks. (In fact this is not an extension of the original virtual binding concept but rather a consequence of considering blocks as anonymous program units and inner blocks as anonymously prefixed with the outer block.)

Let us discuss the block-structured system outlined in figure 16 and let us further assume that identifier *E* is bound to some descriptor *D* within block *S*. If *E* is bound non-virtually within *S*, then all invocations of *E* within *S*, *S'*, *S"* and *S'"* will result in execution of the action part of *D*. If, however, *E* is bound to *D'* within *S'*, invoking *E* in *S* will result in execution of the action part of *D* whereas invocation of *E* in *S'*, *S"* or *S'"* will result in

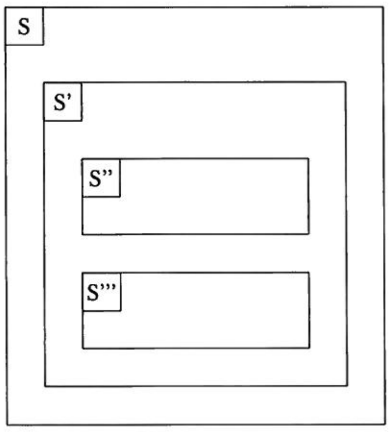


Figure 5: *A block-structured system*

execution of the action part of *D'*.[*] If *E* is bound to *D'* in *S'* and *D'* is prefixed with *D* then invocation of *E* in *S* will result in execution of the action part of *D* only, whereas invocation of *E* in *S'*, *S"* or *S'''* will result in execution of the action part of both *D* and *D'* (as explained in the previous section on prefixing).

Virtual bindings behave differently. Let us assume that *E* is virtually bound to *D* in *S*. The virtual binding of *E* means that if *E* is further bound to *D'* in, say, *S'* then first of all, *D'* must be prefixed with *D* and secondly, the further binding will have effect on invocations of *E* in *S*. That is, if from within *S'*, *S"* or *S'''*, *E* is invoked in *S*, the action part of both *D'* and *D* will be executed just as if *E* were invoked directly in *S'*, *S"* or *S'''*. This is in contrast to a non-virtual binding of *E* to *D'* in *S'*, in which case invocations of *E* in *S* from within *S'*, *S"* or *S'''* would result in execution of the action part of *D* only, and invocations of *E* directly in *S'*, *S"* or *S'''* would result in execution of the action part of both *D'* and *D*.

E may be further bound to *D"* in *S"*, and to *D'''* in *S'''*. Since *S"* and *S'''* are parallel blocks it is only necessary to demand that *D"* and *D'''* both are prefixed with *D'*. That is, *D"* need not be prefixed with *D'''*, or vise versa.

Note that virtual binding is not the same as dynamic binding. A very important aspect of virtual binding is that it is impossible to nullify the binding of *E* in *S* and replace it with another totally different binding. (As a remark, in Simula67 it is possible to nullify the initial virtual binding which makes virtual binding in Simula67 resemble dynamic binding.) Virtual bindings allow rebindings of the original binding — rebindings that **augment** the original binding by prefixing. We say that the original virtual binding has been specialized.

As indicated above, this presentation of virtual binding is mostly inspired by the definition of virtual binding used in the Beta programming language. In reference 11, a detailed discussion of virtual binding is given.

C A Terminology for Exception Handling

The purpose of this section is to describe a taxonomy for exception handling in order to establish a unified terminology within the domain of exception handling.

The domain of exception handling is characterized by the definition of what to classify

[*]Note that we assume static scoping.

as exception occurrences.

We define an **exception occurrence** as being *"a computational state that requires an extraordinary computation"*. Exceptions are associated with classes of exception occurrences. We say that an exception is **raised** if the corresponding exception occurrence has been reached. We say that an exception is **handled** when the extraordinary computation is initiated. An **exception handler** is the specification of an extraordinary computation. If an exception is raised but not handled at the same level of the program, we say that the exception is **propagated** to an outer level. If the exception is handled, and that handler raises the same exception, we say that the exception is **reraised**. Finally, we say that a specific exception is **associated** with a specific handler when raising the exception results in invocation of the handler.

The taxonomy for exception handling is divided into six categories to be discussed in the following. The most important category with respect to this paper is the identification of the two different termination processes: Abrupt termination and smooth termination.

C.1 Exception Occurrences

The overall spirit of an exception handling mechanism can be identified by considering the way in which the proposal characterizes exception occurrences. In most cases this classification will have a major impact on the proposal.
In PL/I, exception occurrences are *"computational states where hardware limitations have been exceeded"*[17]. In a proposal for exception handling in C, exception occurrences are *"computational states where something unusual has happened"*[15]. In reference 5, exception occurrences are *"invocation of an operation in a computational state that is outside the standard domain of the operation"*. Finally, in the static approach[10], exception occurrences are *"computational states where an alternative computation must be initiated"*.

C.2 Exception Handling Model

Perhaps the most important aspect of exception handling is the exception handling model. The exception handling model is concerned with the control pattern that will be the result of raising an exception. There are three major models: The resumption model, the signalling model, and the termination model.

C.2.1 The Resumption Model

In the resumption model the control pattern in the case of an exception occurrence is very similar to an interrupt. That is, raising an exception will result in suspension of the present computation, invocation and execution of the handler, and then resumption of the suspended computation.

Examples are Goodenough's *Notify* exceptions[9] and PL/I's *condition*[17].

C.2.2 The Signalling Model

The control pattern of the signalling model is similar to the resumption model. The difference is that the handler might specify that resumption should be abandoned and termination initiated; That is, specify a change to the termination model.

The *Signal* exceptions in Goodenough's proposal are examples of exception handling following the signalling model.

C.2.3 The Termination Model

In the termination model raising an exception will result in invocation of a handler and termination of at least the syntactic entity in which the exception is raised. The control pattern of the termination model can be characterized by the termination level and the termination process:

- Termination **Level**
  The termination level may be either single-level or multi-level:

  1. **Single-level** Termination
     in which only the syntactic entity in which the exception is raised, is terminated (e.g. Clu[16]).
  2. **Multi-level** Termination
     in which termination of several levels of syntactical units is possible (e.g. Ada). In multi-level termination, the termination level must be defined somehow. There are at least two possibilities:
     (a) to the level of the declaration of the exception (e.g. sequels[10,26]).
     (b) to the level of the handler (e.g. Ada[18])

- Termination **Process**
  In terms of figure 6: Let us assume that an exception is raised at level i and the termination level is level j. The termination of levels $i,i\text{-}1,\ldots,j+1,j$ can then follow two main patterns: Abrupt termination or smooth termination.

  1. **Abrupt** Termination
     By abrupt termination is meant that levels $i,i\text{-}1,\ldots,j+1$ are all terminated immediately, after which the handler is executed, and then level j is terminated (e.g. Ada).
  2. **Smooth** Termination
     By smooth termination is meant that the levels $i,i\text{-}1,\ldots,j$ are each given the opportunity to do some local clean-up actions before they are terminated. The role of the handler(s) in this case may differ from proposal to proposal. Prefixed sequels is an example of an exception handling mechanism that supports smooth termination directly. Languages like Clu and Ada contain mechanisms that make it possible to implement smooth termination to some degree.

C.3 Association of Handlers

There are three major approaches to handler association: Computed association, dynamic association and static association.

1. **Computed** association
   Here the association is done by means of binding imperatives (e.g. the ON CONDITION-statement in PL/I).

2. **Dynamic** Association
   Here the handler is associated with the exception using dynamic binding of exception handlers to exception names. That is, the handler is found in the dynamic context of the raising statement (e.g. Ada, Clu and many others).

3. **Static** Association
   Here the handler is associated with the exception using static binding. That is, the handler is found in the static context of the raising statement (e.g. sequels).

C.4 Handler Types

The handlers associated with an exception may be of several different types. Some of the possibilities are:

- routines or routine-like (e.g. sequels, AML/X)
- statements or blocks (e.g. Ada, Clu)
- expressions (e.g. Goodenough's proposal, AML/X)
- labels (e.g. AML/X)
- booleans (e.g. AML/X)

C.5 Parameterization of Exceptions and Handlers

The expressive power of the different proposals depends moreover on whether it is allowed to parameterize the exceptions and their handlers, making it possible to communicate information to the handler when raising an exception.

C.6 Default Exception Handling

Default exception handling can follow two different paths:

1. **Default Exception Handler**
   By a default exception handler is meant that a specific handler is invoked if an exception is raised and not handled by another handler. The default handler might be a pre-defined handler (e.g. the implicit **except** statement attached to any routine body in a Clu program), or it might be user-definable (e.g. the default handler mechanism of Taxis).
2. **Default Smooth Termination**
   By default smooth termination is means that a specific level may specify a handler to be invoked if the level is terminated as a consequence of an exception being propagated through the level and not otherwise handled in that level. E.g. default sequels, or the *Others*-clause in Clu and Ada.

C.7 Equivalence of User- and Language-defined Exceptions

Since we are examining language constructs for user-specified exception handling, it is important to observe whether user-defined exceptions and handlers are allowed, and whether these are treated in the same way as the language-defined exceptions.