University of Glasgow
Department of Computing Science

Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science

North Haugh
St Andrews KY16 8SX

Exception Handling in a Persistent Programming Language

Persistent Programming
Research Report 26

Alan Dearle

Exception Handling in a Persistent Programming Language

P.C. Philbrow and M.P. Atkinson

Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ

Persistent Programming Research Report 26
October 1986

Contents

§1 Introduction

Why should an elegant and expressive algorithm so readily become obscure and verbose when transcribed into a programming language? The responsibility frequently lies with the expansion of code required for dealing with *exceptions* to the algorithm. The resulting code is the aftermath of the conflict between the programmer, and control constructs in the language. To write robust code the programmer must consider all eventualities: the program becomes verbose. To write code that is readable and amenable to maintenance, irrelevant verbosity must be minimized: there is the temptation to sacrifice robustness. The role of the language designer is as a facilitator between these apparently opposing interests. The root of the problem is the need for a suitable control construct. The conflict is resolved by filling the need for a language feature to support *exception handling*. This report describes the exception handling mechanism that has recently been integrated with the persistent programming language PS-algol.

We preceeded the design work on a PS-algol exception handling mechanism by reviewing the exception handling facilities of existing languages. The purpose of this exercise was to gain an insight into what facilities have been deemed desirable in the past, to identify common themes, to observe how the exception component related to the whole, to learn from their successes, and to anticipate pitfalls in our own venture. The first part of this paper derives from this survey. The survey is not intended to be exhaustive, rather to be illustrative of the influences on the present design.

The design proceeded through a number of iterations, and an experimental implementation. These have been described elsewhere. The initial designs were strongly influenced by the ideas of procedural abstraction, characterized by CLU, and by the simplicity of the Poly mechanism. These influences remain in varying degrees. As the iterations converged it became clear that we were seeking to introduce not one but two new ideas to PS-algol. These eventually materialized in the notions of *system events* and of *programmer exceptions*. Events and exceptions are not only orthogonal to the language as a whole, but also to one another. The requirements placed on an exception mechanism for PS-algol, the considerations that led to the present design, and a description of the model itself form the third part of this document.

In §4 we describe our current implementation of exception and event handling in PS-algol. Conclusions appear as §5.

The two appendices form supplements to the *PS-algol Reference Manual*. Together they form a programmer's guide to the new exception handling features of PS-algol.

§2 Survey

Much of the seminal work on exception handling issues in programming languages dates from the mid-seventies with the work of Goodenough [10] and Levin [12]. Several languages had already made attempts to come to terms with forced deviations from algorithms, notably perhaps PL/1 with its ON condition, and Algol 68 to a certain extent with its *event routines*, invoked on the occurrence of unusual transput events [22]. Goodenough drew together and rationalized the prevailing ideas, and proposed a set of powerful language features. Subsequent designs frequently refer back to aspects of Goodenough's work but in general they do not consider the full range of his proposals to be necessary.

In this survey we restrict ourselves to describing exception handling as it has manifested itself in eight successful incarnations: CLU, Poly, Standard ML and Pose 3 ML, Mesa, Ada, Modula-2+, and Yemini and Berry's replacement model. Be aware that phrases, such as "raising an exception" frequently have a slightly different colour to them depending on the language being discussed. We try to point out these differences where they occur.

The survey revealed a number of terms that are prevalent in discussions on exception handling. We will set these down before we set out.

A *single-level* model requires that exceptions raised by an invocation are handled by the invoker. This contrasts with *multi-level* models, in which an exception is free to propagate indefinitely back up the call chain. There are grey areas between these two groups that are usually due to an interaction with the scope rules of the language.

In *resumption* models the *signaller* (to use CLU terminology) of the exception continues to exist while the handler is initiated, and can be resumed where it left off when the handler completes. The idea here is that the handler can solve the problem that the signaller encountered, but which it wasn't in a position to fix for itself (perhaps because it required knowledge of some global data structure). It is easiest to think of resumption (especially in the context of single-level models) in terms of the handler being an implicit parameter of the signaller. It is clear that a language with first-class procedures such as PS-algol already has resumption for free. For PS-algol the issue is whether additional syntactic support should be provided.

Resumption models are usually contrasted with *termination* models, although models described as being resumptive typically support termination as well (but not *vice versa*). Termination models disallow the possibility of resuming from the point at which the exception was signalled.

A number of recurring issues were brought out by the survey process:

- Is additional syntactic support necessary to build exceptions into the procedural abstractions of the language?
- Can exceptions be parameterized?
- Is an exception an object in the language?

The question of flow of control is paramount:

- How much expressive power should be entrusted to the programmer (where does control pass when an exception has been handled)?
- How does exception handling relate to existing control constructs (where can exceptions be raised, where can handlers be placed)?

We tackle some of these issues in the context of PS-algol in the subsequent section.

§2.1 CLU

CLU [13, 14] incorporates exceptions into the specification of its procedural abstractions. There is the notion of *exception names*, and exceptions can be parameterized. The exceptions that can be raised by a procedure invocation are declared as part of the procedure heading:

```
fetch= proc( n: int )
        returns( string )
        signals( fuss( string ) )
```

In effect CLU is able to specify a number of return paths. The normal route is anonymous, whereas exceptional routes are named. To reflect this a CLU procedure may terminate by a **return** statement or by a **signal** statement, for example the following segment of code may occur in the body of the procedure specified above:

```
...
if ok() then return( "normal termination" )
else signal fuss( "exceptional termination" )
...
```

CLU distinguishes between *signalling* exceptions and *raising* an exception. An exception can be signalled anywhere in a program. The result of signalling an exception is to terminate the current procedure activation and force an exceptional return. The exception is then raised at the point of invocation, and can now be picked up by a handler in the caller. Exceptions can only arise from invocations. All operators in CLU are invocations.

CLU handlers are statically associated with invocations (and therefore with particular exceptions). This gives the compiler the opportunity to generate more efficient code. In addition, it is asserted, programs are more readable. It also has the potential to eliminate the class of errors caused by the programmer forgetting about the possibility of a particular exception, although CLU doesn't exploit this.

CLU restricts the association of handlers to statements only, they cannot be associated with expressions. They assert that there is insufficient demand for handlers on expressions to warrant the extra work required of the language.

Handlers are specified thus:

statement except handler_list end

where handler_list handles a subset of exceptions raised by invocations textually contained in statement. Unhandled exceptions from statement or from handler_list, are propagated beyond the **except** statement.

Each handler in the handler_list is introduced by the system word **when** or **others**. A list of exception names can be supplied for each handler so long as their arguments are the same type (or are to be ignored). The **others** handler can choose to receive the exception name as a string. An example:

```
begin
...
S1 except when fuss: S2 end
...
end
        except when fuss: S3
                others: S4
        end
```

Here *fuss* raised during the course of the statement S1 is handled by S2. *Fuss* raised in S2 is handled by S3, any other exception raised in S1 or S2 is handled by S4.

The **signal** statement terminates a procedure activation and the exception is raised at the point of invocation. CLU provides a cousin to **signal**, the **exit** statement, that can be used to raise an exception immediately, within the same activation. **Exits** are only a local transfer of control. CLU requires that **exits** are handled locally by the procedure.

CLU requires that exceptions raised by an invocation are handled by the invoker. This is termed a *single-level* mechanism. CLU has the syntactic apparatus for ensuring statically that a procedure handles all exceptions that could be raised by it, yet it provides a hole through the single-level in the form of the language-defined exception named *failure*. *Failure* is implicit in every procedure heading. It can be

raised explicitly but is more usually raised automatically should an exception not be caught during an activation, in which case the name of the offending exception name is converted to a **string** and becomes the parameter to *failure*. It is interesting to note that CLU deemed it necessary to allow exception names (albeit as **strings**) to propagate freely up the call chain, despite the violation of procedural abstraction that this might entail. *Failure* can propagate back freely and so is *multi-level* in nature.

CLU fits the termination model of exception handling.

§2.2 Poly

Poly [16] avoids introducing exception names and sidesteps the single/multi-level issue. An exception in Poly is an exception condition together with an identifying string. Exceptions arise implicitly, generated by the system, or explicitly using the **raise** statement:

raise *string_expression*

Unlike the **signal** statement of CLU, there is no forced termination of the procedure activation. The exception is raised immediately. It differs from CLU's **exit** in that, if no local handler is found, the exception may propagate back up the call chain.

A handler can be associated with any block. If an exception is raised during the normal execution of the block, the handler is entered, and its result is substituted for that of the block. The handler is a **proc** taking a **string** and yielding a value whose type matches that of the block to which it is attached. As there is only one (notional) exception, there can be only one handler; there is no special support for discriminating on the basis of the excepted value. The syntax is therefore simple and unencumbered. The handler appears at the tail end of the block, introduced by the system word **catch**.

As an example, we could write a function that increments its integer argument, up to some limit. If the limit is reached it instead raises an exception. The example shows a handler attached to two invocations of this function:

```
let inc= proc( i: integer )integer
begin if i = limit then raise "limit" ; i + 1 end
...
let res= begin
                let tmp= inc( 1 )
                inc( 2 )
        catch   proc( name: string )
                begin
                        print( "Caught a " )
                        print( name )
                        ()
                end
        end
```

It is not clear what the type of the **raise** statement is. For example the equivalent in ML of

```
if test() then 3 else raise "fuss"
```

is permitted because the **raise** statement is of arbitrary type.

Identifiers declared within the block to which the handler is attached (*tmp* in the above example), aren't available to the handler as they are liable to be uninitialized. Exceptions raised by the handler propagate to the enclosing block. Exceptions raised but not handled within a procedure terminate the activation and appear at the point of invocation.

Poly is essentially multi-level in character, with a limited form of parameter passing. It is another example of the termination model. The simplicity of the Poly exception model has much to recommend it.

§2.3 Standard ML

In standard ML [17] exception conditions are bound statically to particular *exception identifiers*. An *exception* is an object from which its associated exception identifier can be extracted. An exception, together with its value (its *excepted value*) is called an *exception packet*. Neither exceptions nor packets are themselves values.

ML supplies a number of predeclared exceptions. New exceptions can be declared by the programmer, using a syntax similar to that of other declarations in the language, for example:

```
exception bucket: string ;
exception foot: int ;
```

Exceptions may be raised by the system, or expicitly:

```
raise bucket "water" ; ...
raise foot 9 ;
```

These may be caught by specifying a handler. This is a double filter, discriminating first on the exception identifier and secondly on the excepted value:

```
handle bucket ( "water".doThis() | "mud".doThis() ) ;
```

Or generally, **handle exid match**.

Both handler and raise phrases must be in the same scope as the exception

declaration. Failing this, the universal handler "?" can be used, but as the type of the parameters aren't determinable statically they are no longer available:

```
stg() ? 0 ; {Where stg() normally returns an int}
```

An exception that is not caught by a handler propagates dynamically back towards the top level; ML exceptions are multi-level. Handlers can be placed on any expression.

An example:

```
exception bucket: string ;
val f x=
        let     exception bucket string
        in      raise bucket "water"
        end ;

(f 0 handle bucket ( "water"."wet" ) ) ? "dry" ;
{Prints 'dry' rather than 'wet' because the exception in the packet is bound
to the 'bucket' declaration local to f}
```

Should no discrimination on the excepted value be necessary, there is a derived form:

```
expression1 trap excptnidntfr expression2
```

A derived form of the **raise** expression can be used for exceptions of type **unit**:

```
exception rhubarb: unit ;
escape rhubarb ;
```

This is again a termination model of exceptions. It is multi-level, and incorporates full parameter passing.

§2.4 ML Pose 3

In the Pose 3 UNIX implementation of ML by Cardelli [7], exception packets are in fact string values, called *exception strings*. Exceptions are raised thus:

```
escape "fuss" ; ...
escape stringexpression ;
```

There are three forms of handler (*trap* constructs):

```
expression1 ? expression2 ;
{Like the universal handler in the standard}
```

```
expression1 ?? expression2 expression3 ;
{expression2 yields a list of strings. expression3 is evaluated
if the exception string is found in the list}

expression1 ? s expression2 ;
{The exception string is available to expression2 as the string "s"}
```

The functionality of Pose 3 exceptions is close to that of Poly. There is again effectively one anonymous exception, and this takes a string parameter.

§2.5 Mesa

Mesa [18] goes beyond the termination model and attempts to incorporate more of the flexibility requirements identified by authors such as Goodenough. In particular, Mesa includes resumption and retry as part of its exception model.

Mesa has a type constructor, resembling that provided for procedures, which is used for the declaration of exceptions. Like procedures, exceptions can be parameterized and can return values:

```
bucket: SIGNAL[ s: STRING ] RETURNS[ i: INTEGER ] ;
-- Declares a new exception variable called "bucket", that
-- takes a string and returns an integer
```

Raising an exception can resemble a procedure call but is prefixed with the system word **SIGNAL** or **ERROR**. The call can be used as an expression or as a statement. **ERROR** can also be used by itself to raise a system-defined exception. The exception is raised immediately and not simply signalled (in the CLU sense). Raising the exception *bucket*:

```
puddle← SIGNAL bucket[ "water" ] ;
```

An exception variable must be initialized with a unique value, generated by **CODE**:

```
bucket← CODE ;
```

Exceptions are caught and handled using constructs called *catch phrases*. Catch phrases can be placed on procedure calls:

```
puddle← rain[ 3 ! bucket ⇒
                    BEGIN
                    WriteLine[ s ] ;
                    RETRY
                    END ] ;
--Which means, should the exception "bucket" be raised by the call of "rain",
--catch it, print the string "s" (the parameter associated with "bucket" in the
--declaration of "bucket"), then execute the entire statement again
```

Catch phrases can also be attached to compound statements:

```
rhubarb: SIGNAL= CODE ;
BEGIN ENABLE rhubarb ⇒CONTINUE ;
--If "rhubarb" is raised inside this block, jump to the next statement following the block
END ;
```

A number of different exceptions and handlers can be specified in a catch phrase, by separating them with semicolons (the lot must be bracketed by **BEGIN** ... **END** if the **ENABLE** form is used). Exceptions can share a handler by writing a comma-separated list.

Exceptions are caught by comparing their signal code (generated by **CODE**) with that of each signal value in the catch phrase. Uncaught exceptions propagate up the call chain (multi-level). All exceptions are caught by the name **ANY** although any parameters will not then be available.

After the handler has done its business, it needs to specify where execution is to proceed. There are three options in Mesa: **CONTINUE**, **RETRY** and **RESUME**. **CONTINUE** corresponds to the flow of control found in termination models, that is it means "go to the *statement* following the one to which this catch phrase belongs". **RETRY** means "go back to the beginning of the *statement* to which this catch phrase belongs". **RESUME** is analogous to **RETURN** in Mesa procedures, and similarly can be used to return values. Its effect is to make the point at which the exception was raised look like a procedure call.

Mesa incorporates a multi-level resumption model of exceptions together with full parameter-passing. Mesa has exception variables. It appears to be a very powerful model, but the semantics can be difficult to grasp.

§2.6 Ada

Ada [1] is another language content with the termination model. Its exceptions are named and new ones can be declared by the programmer. Ada exceptions cannot be parameterized.

An example segment of code is given below. Exceptions may be raised explicitly with the **raise** statement. Handlers are associated with a block, rather as in Poly. The system word **exception** is placed at the tail end of a block and is followed by a number of handlers. If an exception is raised between the **begin** and **exception**, a handler is sought between **exception** and **end**. Unhandled exceptions, or exceptions raised by a handler, appear in the enclosing block:

```
declare CUSTARD exception ; --Looks rather like any other declaration
begin
        declare RHUBARB: exception ;
        begin
                raise RHUBARB ;
        exception
                when RHUBARB ⇒
                        PUT( "Rhubarb" ) ;
                        raise ; --Propagate "RHUBARB" to the enclosing block
                when CUSTARD ⇒
                        PUT( "Will not happen" ) ;
        end ;
exception
        when others ⇒PUT( "Prunes" ) ; --"others" catches any exception
end ; --This prints: RhubarbPrunes
```

Exceptions may propagate out of scope, in which case only **others** will catch it, as above. They can however find their way back into scope again.

Ada exceptions are static creatures; consider this recursive procedure:

```
procedure PUDDING( I: INTEGER ) is
        CUSTARD: exception ;
begin
        if I = 0 then raise CUSTARD ;
        else PUDDING( I - 1 )
        end if ;
exception
        when CUSTARD ⇒
                PUT ( "Prunes" ) ;
                raise ;
end PUDDING ;
```

Unlike local variables, a new instance of *CUSTARD* isn't created for each activation, there is only one *CUSTARD*. Thus a call of *PUDDING*(4) gives us five Prunes. Compare this with the equivalent in ML:

```
val rec pudding i=
        let     exception custard
        in      ( if i = 0 then escape custard
                else pudding( i - 1 ) )
                        trap
                        (
                                output( terminal,"Prune" ) ;
                                escape custard
                        )
        end ;

pudding( 4 ) ;
```

This will give us just one Prune and *custard* (deriving from the innermost invocation) exploding at the top level.

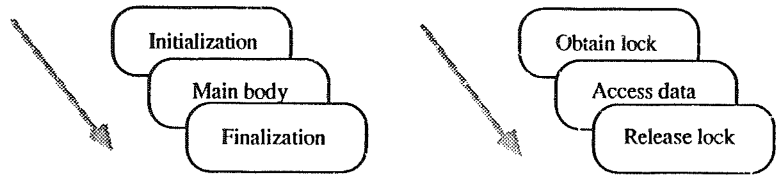
Ada's exceptions can propagate back up several levels of procedure calls before they need to be handled; it is multi-level.

§2.7 Modula-2+

The Modula-2+ system [21] incorporates a multi-level termination model of exceptions. *Exception names* and associated parameter type are declared. At most one parameter is allowed and this must be of the same type as could be returned by a function. Exception names obey the normal scope rules but, as in Ada, they are treated as static constants. The **TRY**statement is used to attach handlers to statement sequences. A **RAISE** statement takes an exception name, and, if appropriate, the single argument.

Optional **PASSING** and **RAISES** clauses are available to assist in the documentation of code.

An interesting feature of Modula-2+ is the introduction of the idea of *finalization*. Programming style frequently follows the pattern:



If the second phase terminates prematurely, it is usual that the finalization phase will still need to be done. There are already a number of implicit finalization actions commonly performed, such as the restoration of context on return from a procedure. Modula-2+ goes further by introducing a **FINALLY** construct that allows the programmer to specify finalization actions. The requirement for language support in this area is obviously related to exception handling issues, and this is the context in which Modula-2+ introduces finalization. It is less clear why **FINALLY** actions should only be entered following an exception.

§2.8 Yemini and Berry's replacement model

Yemini and Berry [23] describe their *replacement* model of exception handling, using Algol 68 as a vehicle. They continue the emphasis on a modular approach to exception handling as initiated by CLU. The replacement model purports to support a large range of handler responses within a simple framework, and is expression-oriented. An axiomatic semantic definition of the model has been constructed, thus making it amenable to formal proof techniques. (In this context, Luckham and Polak [13], have been able to provide an axiomatic semantics for Ada exceptions by extending Ada specifications with exception propagation declarations.)

Yemini and Berry identify the five handler responses that appear commonly in the literature on exception handling. These are: resumption, termination, retry, propagation and transfer of control.

The language construct that characterizes the replacement model is the **replace** completer. This enriches the range of possible handler responses to encompass (with a little help from the host language) all the responses mentioned above. A handler can either replace the signalling expression, or it can replace the **proc** invocation (the signaller) in which the signal occurred. Therefore the signaller can be resumed or terminated respectively. The **replace** completer is used in the termination case.

Yemini and Berry demonstrate the handler responses the replacement model can achieve by using an example procedure called *convert* (reproduced in Figure 2.1). *Convert* takes a vector of integers that are assumed to be character codes, and returns a string formed from them.

```
proc convert= (ref[]int code)string
        signals(exc(int)(char,string) badcode):
begin
        string s:= "";
        for i from lwb code to upb code do
        int code i= code[i];
        s:= s+   if code i≤char hi and code i≥char lo then
                          repr code i
                 else
                          badcode( code i )
                 fi
        od;
        s
end
```

Figure 2.1

They provide an example to illustrate each of the five responses. The examples for propagation and transfer of control are dependent on features of the host language - Algol 68's **skip** and **goto**. The retry example supports just one retry before giving up. The examples for resumption and termination show the characteristic feature of the model from which the other responses are partly derived. Consider:

```
do       ...
         print( convert( nums ) );
         ...
od
on badcode=(int i )(char,string):
         "?"
no
```

The syntax shows that the exception is parameterized by an integer. The *badcode* handler can either replace the signalling expression with a **char**, or the signaller itself with a **string**. (The third possibility is that a transfer of control occurs before the handler is completed, for example another exception, or a **goto**.) By omitting the **replace** completer we are choosing to replace the signalling expression with the character "?".

We can compare this with the termination example in which *convert* is terminated and its invocation replaced with an empty string:

```
do       ...
         print( convert( nums ) );
         ...
od
on badcode=(int i )(char,string):
         "" replace
no
```

Every closed construct from which an exception may arise - a procedure or a block - has a **signals** clause attached (Figure 2.1 again).

We could have rewritten these two examples without any exception handling apparatus: the resumption by passing a handling procedure explicitly to *convert* and the termination example by using existing conditional expressions in conjunction with termination codes. The issue is whether the extra syntactic support is warranted. It is not proven that resumption warrants it. It is likely that termination has benefited. There may be a conceptual advantage in relating resumption and termination by this provision of similar syntax.

§2.9 In summary

We conclude this section with a table to summarize the characteristics of the exception models discussed above (Figure 2.2).

	Handler responses supported					
	Termination	Resumption	Retry	Expression-oriented	Multilevel	Parameters
CLU	✓					✓
Poly	✓			✓	✓	
Standard ML	✓			✓	✓	✓
Pose 3 ML	✓			✓	✓	
Mesa	✓	✓	✓	✓	✓	✓
Ada	✓				✓	
Yemini & Berry	✓	✓		✓		✓
Modula-2+	✓				✓	✓

Figure 2.2 Summary of exception models as they appear in a number of languages. Note that the purpose of this table is to suggest the flavour of each model. In a number of cases there is room for debate. For example CLU is essentially single-level yet it allows the system exception "failure" to be multi-level. Again, the "retry" response can be coded in a number of these languages although no special language support is provided

§3 Exception handling in PS-algol

In this section we present our model of exception handling in PS-algol. We lead into this by discussing the pressures that helped to shape our design. The PS-algol model is presented, together with some simple examples that illustrate some of its features. (A programmer's guide to PS-algol's exceptions and events appears as Appendices 1 and 2.) This is followed by a consideration of the relationship between events and terminating exceptions. The section is concluded with a discussion of exception handling issues that arise from persistence.

§3.1 Influences on the design

§3.1.1 Requirements

The following requirements were placed on the prospective exception handling mechanism for PS-algol:

- Termination should be the primary handler response.
- Some programmers identified a need for resumption.
- Handlers should be expression-oriented.
- Exceptions must be able to propagate.
- Exceptions should be parameterizable to provide context.
- The design should show an economy of concepts.
- Exception handling must be orthogonal to the rest of the language.
- Enhanced readability and maintainability.
- Provide encouragement to the construction of robust programs.

Other issues became apparent as the design proceeded: should exceptions be objects in the language? Should handlers be objects in the language? What consequences are there for persistence?

§3.1.2 Support for resumption

To what extent could the language already meet the exception handling requirements placed upon it? We have already seen that resumption can most easily be understood when it is thought of as implicitly passing a handler to an abstraction. If the handler is a procedure, and procedures have full civil rights, as they do in PS-algol, then we are already in a position to model resumption in our language. A procedure may be passed to an abstraction as a parameter, or by being placed in a shared data structure. The issue here for PS-algol is whether additional syntactic support is desirable. Yemini and Berry opted for the syntactic sugar, yet to understand the behaviour of their resumption mechanism we find ourselves translating it into terms of the passing of procedural parameters. Syntax should be driven by observed needs. Current patterns of use in PS-algol have not demonstrated that a need exists. We allow the principle of economy to reject the possibility of additional syntax.

§3.1.3 Matching exceptions with handlers

Exceptions must be matched with their handlers. In Mesa this is done by declaring special exception objects and assigning a value to them. The match is achieved by comparing the value of the exception raised with the values of the exceptions specified in the catch phrase. Ada and ML also declare exceptions for the same end, although in their case they are static constants. If we were to introduce exceptions as a new type, we would open up an entire can of worms, bearing the label "data type completeness". Poly succeeds in avoiding the need to name exceptions by having a single exception condition that is always associated with some string value. The string is made available to the one possible handler. It is left to the programmer to distinguish the nature of the exception using the normal string comparison operations available in the language. This mechanism is simple and appealing, but does not support parameterized exceptions - the programmer must resort to a shared data structure. This obviates readability and leads to high inter-module coupling.

In PS-algol there are the complementary notions of structures and the data type **pntr** that can rove freely over the universe of possible structure instances. Only when a structure field is dereferenced do we need to know the actual type of the object we have in hand, to this end every **pntr** object carries with it its structure class identifier. It is by this mechanism that PS-algol achieves dynamic binding. The fields of a structure can be any combination of valid PS-algol types. By replacing the excepted string value of Poly with a **pntr** value, we can effectively achieve parameterization.

To discriminate between exceptions we can use the existing **pntr** comparison operators of PS-algol. For the programmer to be able to do this the excepted value needs to be made available to the handler. It is unsatisfactory that the programmer should have all the responsibility for discrimination. It is easy to imagine extensive case or **if** clauses littering the code. There is a clear need here for some syntactic support for the discrimination process.

In Poly the handler is drawn from a subset of the possible procedure types expressible in the language. Its return type must match that of the type of the expression on which it is placed, and it must take a single string parameter; it is the latter that makes the excepted value available to handler. A direct analogue in PS-algol would require a single **pntr** parameter. The idea of handlers-as-procedures has a *prima facie* appeal, the big advantage is that we are able to wrap handlers up concisely, pass them around and store them, without having to introduce a new data type. It is not clear whether we would need this

degree of functionality. The alternative is to allow the handler body to be an appropriately typed clause and to provide a means of formally declaring the variable to hold the excepted value. There are circumstances in which the excepted value is ignored, and so there is no wish to declare it formally: if this were to be supported by the existing procedure heading syntax we would have to specify that only two groups of procedure types - those taking just one **pntr** parameter and those with no parameters - are valid in the handler context. There is benefit in having syntax that constrains the parameter type of the procedure. The programmer isn't misled into writing programs that won't compile.

§3.2 The PS-algol model

Consideration of the requirements and points covered in the previous section, led us to the syntax that is set out in Appendix 1.2.

There is no additional syntactic support for resumption, but we provide a new statement to support termination, and a new handler construct, used to determine the flow of control following termination.

To terminate a closed construct we have a terminating **raise** statement. An exception always has an associated excepted value of type **pntr**. Thus:

```
raise foot( "big" )        !In the scope of the declaration: structure foot( cstring size )
                           !The excepted value is an instance of foot
```

It is a statement rather than an expression because to make it of arbitrary type (determined by context) would be alien to PS-algol.

Handlers are placed dynamically as the program proceeds. The runtime system maintains a virtual stack of handlers.* The **when** statement pushes a handler onto this stack. It is a property of the handler where control passes on its completion. Upon exiting from a closed construct, all the handlers loaded from within that construct are popped from the stack. An exception is fielded by popping handlers from the handler stack until a match is found. Thus at any one time an operation is surrounded by an onion skin of protective layers. So we might write:

```
write "This block completed ",
begin
        when any do "with a terminating exception"
        raise nil
        "normally"        !Never reached
end,"'n"
```

This illustrates a number of features. The handler construct is

*The stack model of exception handling is characterized more precisely by a stack of handler stacks. A handler stack being associated with every closed construct.

expression-oriented, with a void statement being a special case. The handler replaces the unexecuted remainder of the block. The keyword **any** is provided as a wildcard to catch any exception (in fact this would be the only way of catching this exception because the structure class of nil is not expressible in PS-algol). To field a specific exception, or group of exceptions, a list of structure names would replace the **any**. For example:

```
structure foot( cstring size )
when foot do ...
...
when foot, error.record do ...
```

If we wish to field the exception and to identify it ourselves, or if we wish to otherwise interrogate the excepted value, we can declare a local constant to hold it:

```
structure foot( cstring size )
when foot, error.record as e do if e is foot then ...
...
when foot as e do write "That was a ",e( size )," foot'n"
```

Each time we use the **when** statement we wrap a tighter protective layer around the code that is subsequently to be executed:

```
let p = proc()    !raises foot(cstring) or error.record(cstring,cstring,cstring)
begin
      ...
end

when any do ...                                               !A
when foot, error.record as e do ...                           !B
when foot as e do
        raise if e( size )="" then error.record( ... ) else e  !C
p()
```

If the call of *p* causes a *foot* to be raised, the handler at C is entered. If C propagates *foot* (notice how we can explicitly propagate an exception) then the handler at B is entered. If B propagates *foot* further (or B or C or *p* raises some other exception, perhaps as a result of a system event), the catch-all handler at A will pick it up. If *p* had raised *error.record*, the handler at C would have been popped from the handler stack, but the handler at B is able to field it. Of course if *p* completed without raising an exception, all three handlers would have been popped at the end of the block.

Resumption from exceptional conditions detected by user programs is achieved through the use of first-class procedures. To be able to resume from exceptional conditions detected by the PS-algol machine itself (it is convenient to refer to them as *system events* to distinguish them from exceptions that invariably cause a termination) we make a data structure visible in the standard environment. We are then able to use first-class procedures ourselves to model resumption from system events. The PS-algol standard constant *events* is a pointer to a structure

containing a collection of *event procedures*. Event procedures "handle" events. When the PS-algol machine detects an event (some hardware condition, or perhaps a runtime error such as a failed dereference) a corresponding procedure from the *events* structure is called. Letting the procedure return normally allows us to resume from where we left off when the event was detected. By having event procedures that return values we are able to provide replacement values for failed operations. It should be noted that we consider the details of the event mechanism to be experimental, and that we anticipate refining them when we have user experience.

This model for resumption from system events is a generalization of an idea present in Algol 68. Algol 68 had event routines defined in the standard environment that responded to particular transput conditions. The programmer could replace these event routines with procedures of their own.

Figure 3.1 is a simple example illustrating how an event procedure might be used. The example is of a screen-saver that is toggled when a keyboard interrupt is received. The first interrupt clears the screen and causes the time to be repeatedly printed at random positions. A second interrupt restores the screen and returns us to the main program.

```
events( Interrupt.event ):=
begin
        let hiding := false
        let save = image X.dim( screen ) by Y.dim( screen ) of off

        proc()                                                    !My interrupt routine
        if hiding then hiding := false else
        begin
                let the.time= time()
                copy screen onto save
                xor screen onto screen

                hiding := true                                    !← A
                while hiding do
                begin
                        let x = random() rem X.dim( screen )
                        let y = random() rem Y.dim( screen )
                        print the.time at x,y                !Display
                        for i = 1 to 100 do { }              !Delay
                        print the.time at x,y using xor      !Erase
                end
                copy save onto screen
        end
end

...     !Main program
```

Figure 3.1 A screen-saver toggled by keyboard interrupts

Figure 3.1 also serves to illustrate a problem associated with the use of asynchronous events. Consider the effect of receiving the second interrupt immediately before *hiding* has been set true at A. The second activation of the interrupt routine will erroneously conclude that it is the first. Havoc ensues. What we have done is to introduce a form of concurrency. If we are to safely share data structures in a concurrent system, we will need support from the language. Krablin [11] has confronted concurrency issues in persistent programming languages, and has devised the necessary constructs in a concurrent version of PS-algol. We can safely handle asynchronous events by using the orthogonal **atomic** statement or its equivalent.

Figure 3.2 shows how we might make use of a system exception raised by one of the system event procedures. The system exception *StructureAccess* is raised by the event procedure of similar name. The event is caused by dereferencing an object of the wrong class. (See also Appendix 2.2.) We show how a routine might be written that extracts a library procedure from a database. The routine assumes that most people will have used the same convention in storing their library routines, that is they wrap it in an instance of a particular structure class. It is nevertheless capable of coping with libraries that don't adhere to the convention.

```
let recover = proc( cpntr p ->proc() )
begin
        !Generate a procedure to do the job
        !Call the compiler
        !Do it
end

let lib = proc( cstring key,tab ->proc() )
begin   when StructureAccess as e do recover( e )

        structure proc.container( proc() the.proc )
        s.lookup( key,tab )( the.proc )
end
```

Figure 3.2 A routine to extract procedures from libraries

This technique may be of particular value in a program development environment. The developer may build an impenetrable fire-wall to protect themself, by for example, replacing the default *events(Uncaught.event)* (see Appendix 2.2) with a procedure that calls a browser, or they may have an arrangement as in Figure 3.3.

```
let test.program = proc(); ...

let finished := false
repeat
begin   when any as e do ...      !Lookup you favourite browser
                                  !and browse the excepted value

        test.program()
        finished := true
end                               !Then try again ...
while ~finished
```

Figure 3.3 Protection from insecure code

§3.3 The relationship between events and exceptions

In the PS-algol model we've been describing, an event causes an immediate procedure call. An exception causes a termination and unwinding followed by what can be thought of as a procedure call: a transfer of control followed by a procedure call (or *vice versa*). In PS-algol the type of the exception handling procedure is constrained by the syntax to fall within a subset of the normally expressible types. A terminating raise is a selective procedure call returning to an enclosing context. The call of the event procedure is a special case of the terminating raise (no net transfer of control).

If this difference between events and exceptions were to be abstracted away, there may be interesting implications for what a procedure call is. Does a generalized procedure have greater flexibility in where it returns to? The terminating raise syntax provides a means by which a procedure call can include a recommendation of where the procedure is to return to.

§3.4 Issues arising from persistence

An exception in PS-algol is a transient condition. Excepted values may be any **pntr**, and this is exploited in handler selection: handlers discriminate dynamically between exceptions on the basis of excepted value. Inasmuch as exceptions are transient conditions, and excepted values are normal PS-algol objects, there are no new implications for persistence. Environments will be retained, or recovered, as necessary. The POMS Persistent Object Management System [6] ensures that any problems come out in the wash.

Can we abstract over the idea of exceptions? We have exceptions that occur dynamically (related to the state of active data), could exceptions occur statically (because of a value in a static data structure that is in persistent store) [4, 5]? Dæmons inhabiting persistent store might detect exceptional conditions. We've been placing handlers along the route taken by the program as its fate unfolds; what would placing handlers on static data look like?

We have been thinking of exceptions as being deviations from an algorithm. Could we expand the idea to include deviations from a value, or range, or state?

Our approach in the present model has been to lay down finer and finer, increasingly specific, handlers as we proceed dynamically. The handlers persist so long as the dynamic history persists. This is necessarily transient. What if we were to lay down layers of handlers around some static state - shells or skins of protective handlers? Exceptional conditions might be detectable by some dæmon. The "test and raise" would map onto one of these dæmons. The nature of the test (the constraints on some stored object) might be a property of a particular object, rather than a general directive to a dæmon. The nature of the exception might also be left as a property of the object. There are two extremes: we could have a multitude of dæmons, one per object; or more sensibly, one kind of dæmon with a constraint/exception pair associated with each object.

But stored data is by nature static (or is it, in a world full of dæmons that are forever rearranging and improving things?). We have been implying that stored data can exist that doesn't conform to its constraint criteria. These ideas are more useful if it is possible to change the constraints on objects themselves. This is meaningful if the constraint is placed on a class of objects. So we may store a description of a class of objects, with this description including a specification of constraints placed on instances of such objects, alongside real instances of that class. Now the constraint is changed. The stored instances become invalid. Next time the instance is accessed, or perhaps a constraint-checking dæmon finds it, an exception is raised. The exception would have been specified as part of the class description. There are two possibilities for placing the handler. In a system in which it is dæmons that detect the constraint violation, the handler could either be specified as part of the constraint, or as part of the definition of the dæmon. In a system in which the violation is detected by a user program, the handler could again be associated with the constraint, or as part of the program. There is no real difference here between dæmons and user programs. The dæmon could just be an ordinary program that browses over the whole store (or at least over some well defined region of the store - the store may be infinitely large, and there are rules of access to be considered). A default handler could be provided for the case when no appropriate handler has been supplied. This would catch any constraint violation. Similarly, a constraint violation with no specifically provided exception could raise the general exception *ConstraintViolation*.

These considerations of exceptions in an essentially static context naturally extend back into the dynamic context of the present work. Many of these issues have already been explored by Owoso [19].

§4 Implementation

In this section we give an overview of the implementation of programmer exceptions and of system events. For background material and for an explanation of terminology, we refer the reader to the *PS-algol Abstract Machine Manual* [3]. We include also some thoughts on efficiency.

§4.1 The exception mechanism

Our implementation of the exception handling mechanism involved modifications to the compiler and to the runtime system. The only change to the PS-algol abstract machine is the introduction of two new exception processing opcodes: RAISE and LOAD_HANDLER. There are no new types of heap object, and no modifications to the structure of the existing object types; in particular, frames remain unaltered.

§4.1.1 Compiler

We assume here some familiarity with the structure of the present PS-algol compiler. For an exposition of single-pass recursive descent compiling, on which it is based, we refer the reader to Davie & Morrison [9].

The table of reserved words in the lexical analyzer has been expanded. Two new recognizer procedures (*when.decl* and *raise.clause*) go into the syntax analyzer, and two new opcode-generating procedures into the code generator (*raise.op* and *load.hndlr.op*).

There are minor additions to *proc.decl* and to the other block-recognizing procedures, in support of handler type-checking, and to manage the transfer of control following the completion of a handler.

The compiler always arranges for space on top of the pointer stack to accommodate the excepted value. Handlers themselves are compiled as procedures that take a single **pntr** parameter. This will be the excepted value. When an exception is raised at runtime, handlers are popped from a virtual handler stack that reflects the dynamic history of the program. Each popped handler is given the opportunity to field the exception. The list of structure names given in the handler is expanded into a series of tests. These tests form a *guard*. Each test is on the class identifier of the excepted value: is it the same as the class identifier from which the structure name was taken? The compiler does this by planting code at the start of the handler's code vector. This code is composed of jump instructions and the structure-comparing IS opcodes already present in the abstract machine. A RAISE is generated for the case when a match fails. If the guard is passed, then the match has succeeded, and the handler-proper is entered. If the match fails, encountering the RAISE opcode propagates the exception back to the next handler.

Each **when** statement results in the generation of a LOAD_HANDLER opcode. This new opcode expects two values in the byte stream. The first is a procedure number, identifying the whereabouts of the handler closure in the vector of closures associated with the current code vector. This procedure number is immediately available to the compiler as it has been maintaining a count.

The second value is a little trickier to obtain. It is needed by the interpreter to calculate the return address from the handler. This must correspond to the block exit of the enclosing block. Recursive descent compiling frequently requires code to be generated before all the facts have been gathered. There is the apparent paradox that we need to have completed compilation of the block, and so have determined where it ends, before we can compile this component of the block. The compiler gets round this kind of problem by buffering the code vectors. Holes can be left in the byte stream that can be filled in as knowledge is acquired. This is typically required for forward jumps. We make use of existing tools in the compiler to back-fill the return address offsets: A hole is left in the code vector following each LOAD_HANDLER. In any given block there may be a number of LOAD_HANDLER opcodes (each corresponding to a **when** statement). The return address holes are chained together. On reaching the end of the block we can follow the chain back to the first hole, filling in the return address offsets as we go. See Figure 4.1.

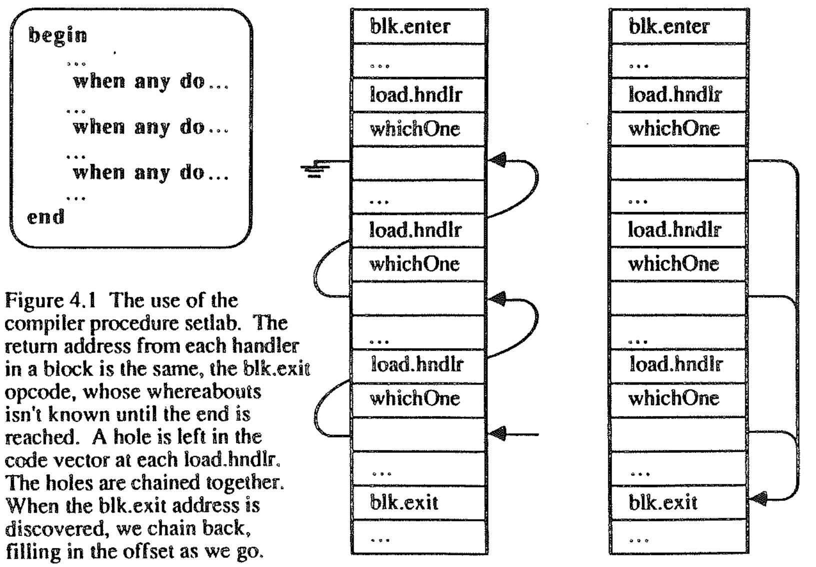


Figure 4.1 The use of the compiler procedure setlab. The return address from each handler in a block is the same, the blk.exit opcode, whose whereabouts isn't known until the end is reached. A hole is left in the code vector at each load.hndlr. The holes are chained together. When the blk.exit address is discovered, we chain back, filling in the offset as we go.

In summary: Handlers are compiled as though they were ordinary procedures with a single pointer parameter. The compiler plants guard code at the beginning of the handler code vector that will result in the excepted value being raised again unless it satisfies the match criteria. The effect of the raise opcode is to scan down the pointer stack of each frame in the dynamic history, searching backwards through time. The scan terminates as soon as a handler is found. It is the responsibility of the compiler to plant discriminatory guard code in each handler.

§4.1.2 Interpreter

The changes in the interpreter to support programmer exceptions are limited to the main fetch loop. There are two new opcodes: RAISE and LOAD_HANDLER. We describe here the actions taken by these when they are encountered at runtime.

LOAD_HANDLER takes two values from the code stream: an identifying procedure number, and an offset to the exit instruction of the enclosing block.

LOAD_HANDLER creates a PS-algol structure class instance - the *handler structure*. The handler structure is given a recognizable class identifier that cannot be confused with any a user program could create: we use the string "H" from the single character table. Using the procedure number, a reference to the handler's code vector is taken from the vector of closures and placed in this structure. Its static link is made to the current frame. The final field of the handler structure is calculated by converting the end-of-block offset to an offset from the beginning of the current code vector - it is then in the same form as the RA field of frames. Finally the handler structure is loaded onto the pointer stack. Figure 4.2 shows a representation of a handler structure.

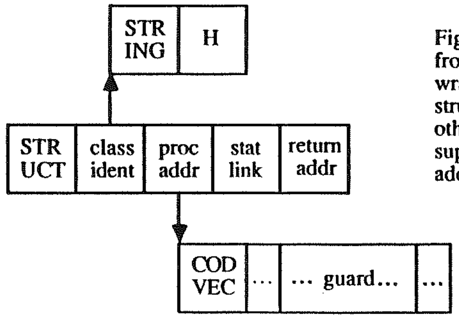


Figure 4.2 Handlers are reached from the pointer stack. Each is wrapped in an easily recognized structure to distinguish them from other objects. This structure also supplies the handler with its return address.

RAISE pops the excepted value from the top of the pointer stack. The dynamic chain of frames is scanned until a handler (any handler) is found. Frames are tidied up and discarded as we go. For each frame, we scan down the pointer stack in search of an object with the class identifier of "H" - that is, we are looking for a handler structure. If we unwind to the global frame (recognized by having a dynamic link of zero) the system event *Uncaught.event* is flagged. If a handler structure is found, we extract the closure and apply it, passing the excepted value as the parameter and supplying the previously calculated (by LOAD_HANDLER) return address as the RA in the new frame. The pointer to the handler structure on the stack is zeroed out, to make sure it is never found again (see later). Be careful that the popped excepted value isn't cleared away by the garbage collector before it finds its way into the new frame.

§4.1.3 The dynamics of raising and handling exceptions

This section draws together the previous two by considering the dynamics of raising an exception.

A pointer expression has been evaluated and the result, the excepted value, is left at the top of the current pointer stack. RAISE pops this value and searches back through the dynamic history for a handler. It begins with the current frame, by scanning down through the pointer stack for a handler structure. If no handler structure is found, the frame is tidied and discarded, the dynamic link is followed, and the pointer stack of this new frame becomes the subject of the scan.

When a handler structure is found, the handler's closure is extracted and is applied. The return address is supplied from the handler structure.

The compiler has planted a guard at the beginning of the handler code vector to test whether the handler is able to field the exception. Failure to pass the guard initiates a RAISE which propagates the excepted value to an older handler. Note that the first frame to be inspected following the failure to pass the guard will be the frame in which the unsuitable handler had itself been found. The pointer stack of this frame was not retracted in case garbage collection strikes. To avoid cyclic attempts at entering the unsuccessful handler, the handler structure pointer on the stack is set to nil immediately before entering a handler. The cycle of "raise » handler entry » test » raise" repeats until the test in the guard code of the handler succeeds, and the code of the handler-proper is executed. The handler terminates normally when a conventional return opcode is performed.

The following sequence of panels illustrate the implementation of exceptions. Consider the segment of PS-algol source code shown in Figure 4.3. The segment includes a block within which two handlers are set up, before encountering a **raise** statement. (*StringTooLong* is an exception raised by the system if an attempt is made to construct a string whose length exceeds the limit imposed by the implementation.) In the example, the exception will not be fielded by the second handler (the one placed to catch *StringTooLong*), but will be caught by the first (**any** matches all possible exceptions). The code of the first handler will

be substituted for the code that would have completed the block.

```
...
begin
    ...
    when any do ...
    ...
    when StringTooLong    do ...
    ...
    raise nil
    ...
end
...
```

Figure 4.3

Figure 4.4 shows the three code vectors that the compiler generates for this segment. The longest corresponds to the main segment. The two shorter ones are the handlers. Opcodes are indicated with characters in the outlined font style. The outlined ellipses indicate opcodes that don't directly concern us here. Code vectors (CODVEC), frames (FRA) and structures (STRUCT) are all self-describing objects found in a PS-algol heap. The procedure numbers in the first vector identify the two handlers. The return offsets give the distance in bytes to the block exit instruction (this offset is shown with bracketing horizontal lines).

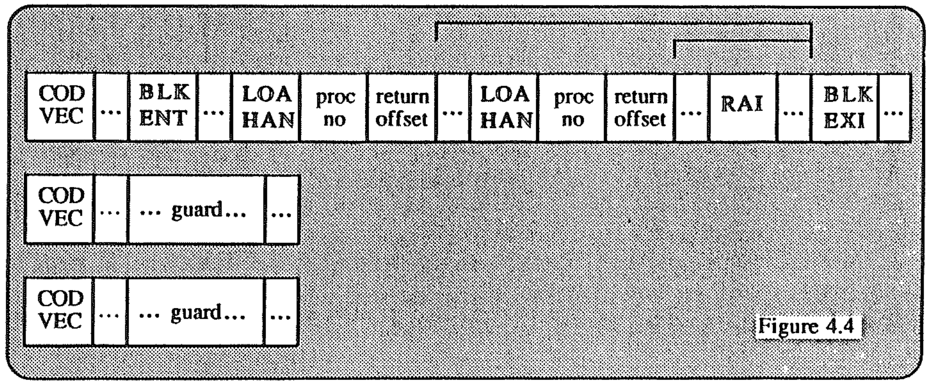


Figure 4.4

In Figure 4.5 we see the principal code vector in the process of being executed. Associated with the activation of this code is a frame object. Every block or procedure call has a frame. Frames hold the two local stacks associated with each block - a pointer stack and a main (non-pointer) stack - where local variables are kept. Frames are linked together in a chain that forms the dynamic history of the program. The PS-machine has a number of registers that contain active pointers. In the diagrams these are shown as triangles. The code pointer in Figure 4.5 is shown just beyond the two in-line values that have been supplied to the LOAD_HANDLER of the first handler. Other registers mark the bases and the tops of the current stacks (msb, psb, msp and psp). A frame that has been put to rest for a while has the height of its stacks encoded within it in order to free the fast registers. The diagrams show the stack top pointers of quiescent frames as empty triangles. In the figure, block entry has already been performed and a new frame created. We can see that it is the currently active frame because the pointer registers are referring to it. We have joined the diagram following the loading of the first handler. A pointer at the top of the active pointer stack is pointing at a handler structure. In turn, the handler structure leads to the code vector of the handler.

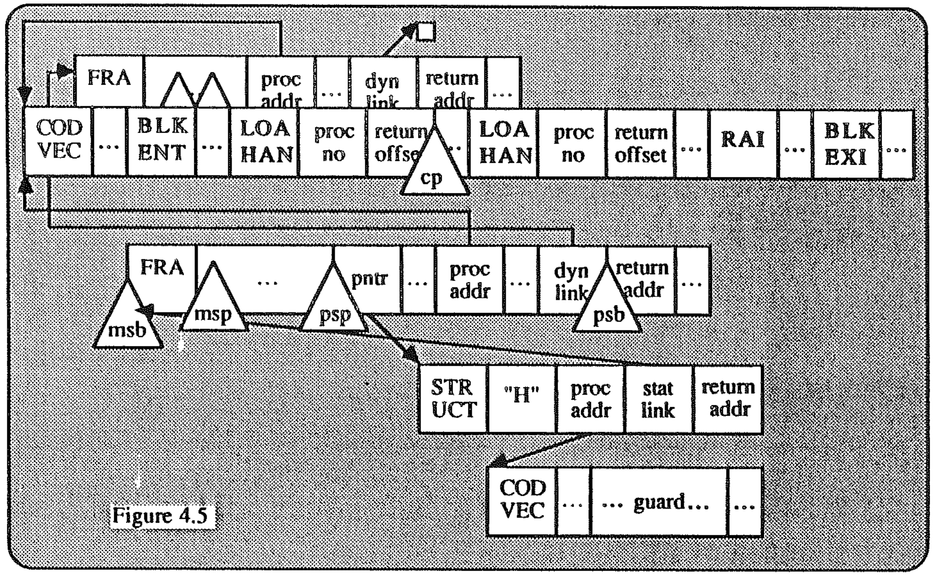


Figure 4.5

By the time we reach Figure 4.6 a handler structure for the *StringTooLong* handler has been loaded onto the stack. In fact we have proceeded to the stage at which an excepted value has been left on the top of the stack, and the RAISE instruction is about to take place.

In Figure 4.7 the RAISE has been performed. The second handler, the last handler to be pushed onto the virtual handler stack, has been located from the frame that had been active. The position of its top-of-stack marker shows the excepted value to have been popped from the pointer stack. The pointer to the handler has been removed (set to nil) from the stack. Control has passed to the code vector of the handler, as shown by the position of the code pointer, cp. We are in fact in the process of executing the guard. The current frame is now that of the handler, and it is to here that the excepted value has been copied. Note how the frames are chained together by their dynamic link.

Figure 4.8 tells us that the first handler to be tried was unsatisfactory - we

failed to pass the guard and the exception was raised again. No handler had been

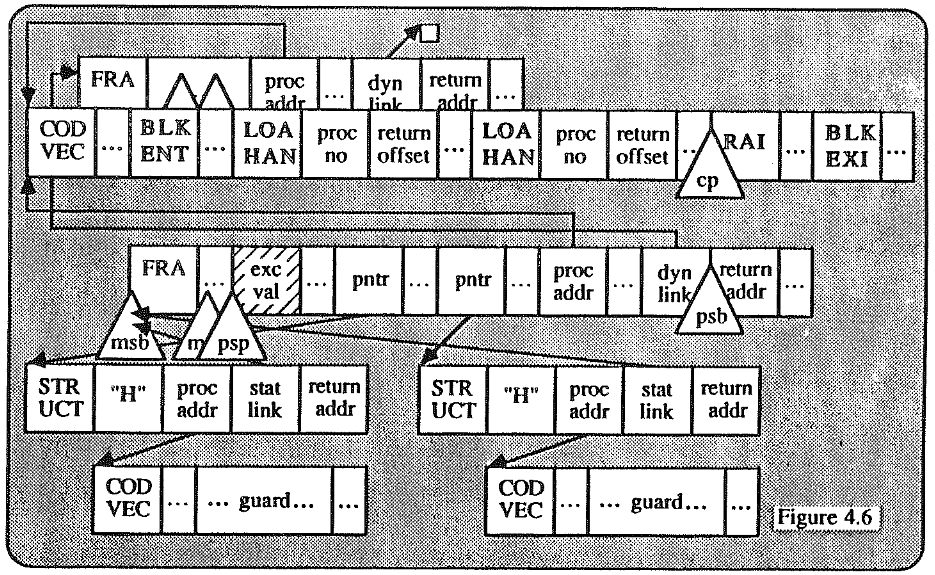


Figure 4.6

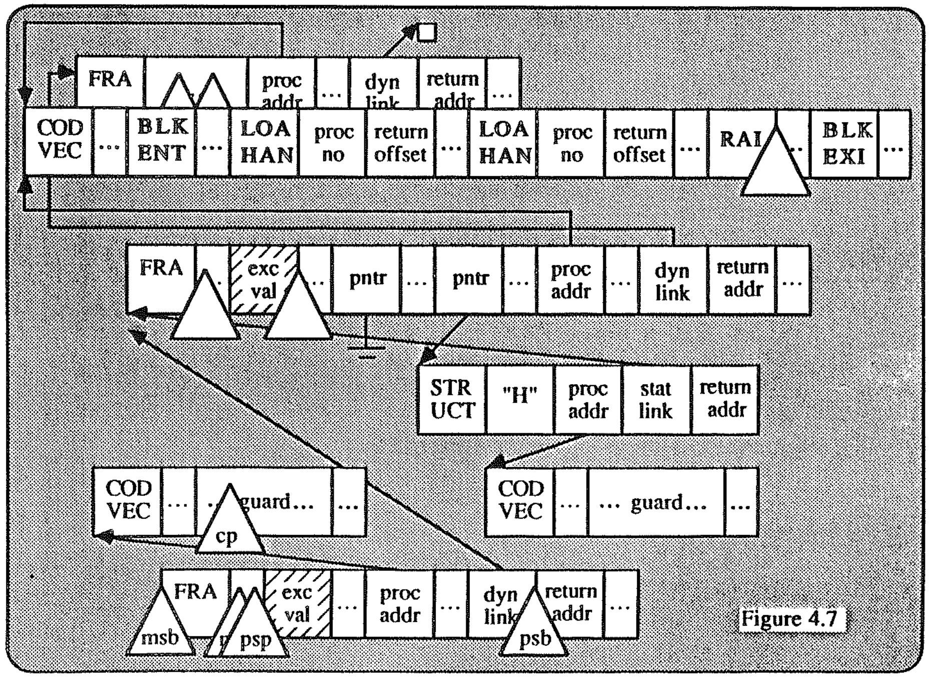


Figure 4.7

found on the frame of the handler and so the previous frame had been tried again. The first handler to be loaded was found successfully. Both handlers have now been removed from the virtual handler stack. The current frame is that of the new handler. The diagram shows that the guard has been passed and the handler-proper is being executed. When the handler completes, its return address will set the cp to the block exit of the principal code vector.

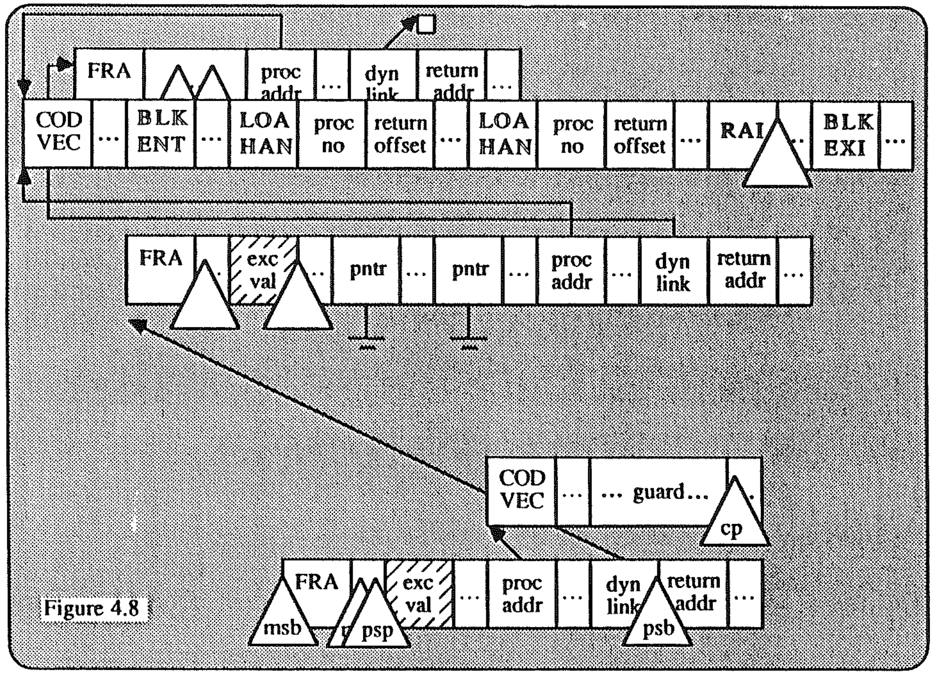


Figure 4.8

Finally, Figure 4.9 shows that block exit successfully completed. The active frame has been reset to that of the main segment and we are executing the final ellipsis from the panel of Figure 4.3.

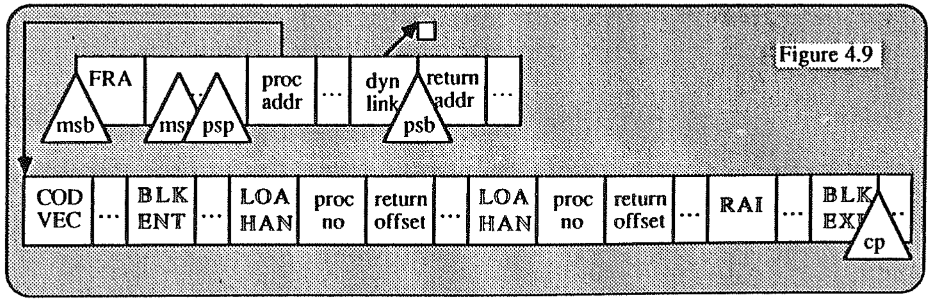


Figure 4.9

§4.2 System events

This section divides roughly into two: compiler-related aspects of the implementation of events, and the runtime mechanism for dealing with events.

§4.2.1 Compiler-related aspects

To support event handling we have made visible a global data structure accessible through the standard **pntr** constant *events*. *Events* has been placed in one of the free slots on the standard frame - it is declared in the standard declarations file read by the compiler. *Events* is a pointer to an instance of the class identifier whose name is *events.strc* (see Appendix 2). *Events.strc* has been put into the PS-algol prelude. It is a structure whose fields are procedures. There is an event procedure corresponding to each possible system event. On detection of an event, its event procedure is called. A structure is the most straightforward way in PS-algol of grouping together procedures of mixed type.

All but two of the event procedures are written in PS-algol. The runtime error interface to the user has hitherto been through the interpreter functions *error()* and *ferror()*. This interface has been retained for some particularly obnoxious system events which pre-empt the application of a PS-algol procedure - for example heap overflow - and for the multitude of "impossible" errors that might be caused through a bug in the implementation. So there are two fields of the system events structure that correspond to these functions. They are there to provide a consistent interface.

A significant proportion of the PS-algol runtime system is itself written in PS-algol, for example the real i/o, the outline line-drawing package and some of the image functions. These detect error conditions and report them. We have modified the real i/o and raster graphics packages (in *tables.S* and *raster.S* respectively) to invoke an appropriate event procedure instead. The default actions mimic the original.

Of those procedures written in PS-algol itself, most make use of the new exception mechanism, and simply raise an appropriate exception. The class identifiers used by these exceptions are also declared in the prelude (see Appendix 2). The program *events.S* wraps all these event procedures up in an instance of *events.strc*, and commits this to the system database.

At runtime the POMS code picks these procedures up and initializes *events*, with the procedures placed in a new instance of *events.strc*. This has the advantage that every program could rely on picking up the same event-handling objects.

§4.2.2 Duplicating the *events.strc*, a standard frame problem

It would have been good to have taken the same structure instance from persistent store as well, so that programs would always have identical components to their standard frames. The problem was in the requirement to make some of the fields writeable. It would not be a good idea to allow programs to make changes to the system that would show up in other people's programs. To achieve identity of *events* the program finalization routines would need to discard changes made to the standard environment. This might be reasonable, it already happens with *screen* and the printing field width variables. Conceptually though it would be better to move *screen*, *events* and any other object that might be modified by user programs into a frame that is local to the user's program (like the current global frame), and which it is understood is initialized afresh by the system for each interpreter incarnation. The present arrangement is rather *ad hoc*. Although in name we have a standard frame, it is only safe if the programmer assumes that their standard environment may have changed since the last time a program was run. We could otherwise write programs that showed up a conceptual inconsistency. Two different "standard" objects can be taken by a user program, packaged in a structure, and stored in a database. A subsequent program can extract these objects and compare them for equality with their analogues in todays standard frame. If we are comparing standard procedures that are built into the interpreter, because of the way the POMS works they will certainly satisfy the equality test; if they are standard procedures implemented in PS-algol they *might* be the same; but if they are one of the standard images, or in our case, the events structure, they will certainly be different. Yet it is thought of as a "standard" environment. It would be useful to have these issues clarified.

§4.2.3 Runtime aspects

There are problems in the provision of diagnostics following an uncaught exception. We are unable to provide a trace back of line numbers, or procedures invoked, because of the tidying up of frames performed during the search for a handler - the dynamic history is necessarily thrown away so that it is not retained unnecessarily over garbage collection, or by the POMS. Our working solution for this was to provide a new private standard procedure, called *post.mortem*, that yields the line number at which the last event (other than *Uncaught.event*) occurred. This is used by the default handler provided for the *Uncaught.event* event.

There are a number of better and more comprehensive solutions available, although they tend to be expensive. A record of the dynamic history could be frozen at the time of the event: expensive of processor and requiring a quantity of store to be allocated that could not be determined statically. We could delay the loss of dynamic history until after the exception has been fielded - but this would require two traversals of the path. Less ambitiously, we could clear the history but retain for a while a pointer to the signallers frame. In the same vein, we could have a special pointer in each frame that is filled in with a link to the child frame as the

exception unwinds, and that might be ignored by the POMS. All of the last three solutions could provide the wherewithall to support interactive post mortem diagnostic aids (a browser), either for PS-algol programmers, or for PS-algol implementers. A doubly-linked structure for browsing could be built by retaining the dynamic growth, along with the unwinding path. The *post.mortem* PS-algol procedure could be enhanced by causing it to initiate such a debugger.

Events that are detected by those parts of the PS-algol system that are written in PS-algol, can call the appropriate event procedure direct; no further support is needed. Unfortunately there are two other classes of events: those detected by the PS-algol machine itself (that is, within an opcode), and those that are reported by the machine that is host to the PS-machine. The latter are essentially asynchrorous and are the least tractable.

The problem with these two classes is that of making sure the PS-machine is in a consistent state before invoking the event procedure - that the machine's registers are correctly set (stacks the right size, the code pointer pointing at a real opcode). Because of the structure of the interpreter - the fetch loop is one large switch statement with most of the opcodes written as in-line code - it would, in a large number of cases, be possible to tidy up from the current opcode, set up the frame for the event procedure by performing an APPLY operation, and then to jump to the top of the fetch loop. But as the interpreter has grown it has become more structured, and parts of opcodes now involve function calls to quite sophisticated routines: this means that a simple jump to the top of the fetch loop is no longer feasible. This problem has been solved during the implementation of the *edgeViolation* mechanism in the **print** statement, by making a nested call of the *execute_code()* function whose body is the fetch loop [2]. However, we are obliged to provide a means by which the asynchronous host system events can be supported. Having provided that support there is nothing to be gained from using a separate mechanism for the events detected by the PS-machine.

Both these two classes of events initiate an event handler through the same basic mechanism. We introduce two interpreter functions: *sys_error()* and *sys_event()*. *Sys_error()* is called, when an event is detected, to set the flag *sepend* to a non-zero value, that identifies the event. *Sys_error()* is called on receipt of a signal from the underlying host machine, or directly by the PS-machine.* At the same time, any parameters that are to be passed to the event procedure are placed in a dummy PS-algol frame, accessed through the PS-machine pointer *separams*. The frame is a proper PS-algol object, made accessible to the garbage collector. It is created, with a fixed size, at interpreter initialization, by the PS-machine function

*The version 4.0 implementation of the PS-algol interpreter doesn't call *sys_error()* when the PS-machine detects an event, in the way described here. Rather, there is an immediate call to *sys_event()*, thus setting up the new frame and code pointer value before completion of the offending opcode. This requires great circumspection to be sure that the completing opcode doesn't damage the registers.

init_events(). It is made accessible to the garbage collector by placing it in the dummy PS-algol vector of pointers, called *save_vec* in the PS-machine, which the garbage collector always checks.

At the earliest opportunity following detection, we return to the top of the fetch loop. At the top of the fetch loop we always need to check the state of the events-pending flag. There is no safe way of coding round this problem. if we are to support asynchronous hardware events. A performance degradation of up to 5% has been measured, because of the frequency of the test, but this is tolerable. Having been caught by the flag, the flag is cleared, and the interpreter function *sys_event()* is called.

The event's identification is passed to *sys_event()*. This identification is used by *sys_event()* to index into a static table called *se_map*. Each entry in *se_map* describes an event procedure. Two fields of the entry describe how to restore the local stacks to a consistent state. A third supplies an offset in the PS-algol *events* structure, to the event procedure needed. Note that there isn't a one-to-one correspondence between events and event procedures - several events map onto the same procedure. The final two fields describe the number of main and pointer stack parameters that are to be copied from *separams* to the frame being created for the event procedure.

Sys_event() is like a modified APPLY opcode. It differs in that the closure is extracted from the PS-algol *events* structure, and that the parameters are taken from *separams* rather than from the local stacks. In addition, if the event is identified as being due to an uncaught exception, the return address from the event handler is set to be the FINISH opcode of the main program. Where it is feasible, the mechanism described allows control to resume, when the event procedure returns, where it left off - at the opcode following the event. In those circumstances where this is not possible, we enforce termination by orthogonal use of the new exception raising mechanism in the default procedure, and by making the event procedure field a constant.

In summary: from the implementation point of view there are three classes of events:

- those detected by the parts of the PS-algol runtime system that are implemented in PS-algol itself,
- those detected directly by the PS-machine, and
- asynchronous events detected by hardware.

In the first class, the appropriate event procedure is called directly, having first been found through the *events* structure, now located in the standard environment. The remaining two share the same mechanism: a flag is set with some identifying value, and the opcode continues to completion or, at least, resets the PS-machine to a consistent state. The next fetch cycle detects that an event has been flagged, and an application of the appropriate event procedure is initiated.

§4.3 Future optimizations

To a certain extent we sacrificed efficiency of operation to achieve compatibility with the existing system. For example a previous incarnation managed the handler stack in a new area of the frame; the present incarnation has a virtual handler stack that is interweaved with the normal pointer stack, this has increased the cost of searching for handlers but has kept the machine simple. The current system always involves a dynamic search for a handler. We anticipate that there will be frequent opportunity for an optimizing compiler to associate a potentially raised exception statically with its handler, and to generate appropriate in-line code. For example:

```
structure abandon( cstring problem )
when abandon as the do write the( problem )
...
if ... do raise abandon( "sinking ship" )
```

So long as there is no path through the code that could set up a relevant intervening handler, the **raise** statement can be replaced by a direct call to the handler, immediately followed by a jump instruction. In this case, the guard placed in the handler would be redundant, and the handler call could be flattened out. If this in line replacement is to be done safely in the context of asynchronous system events, we need to be sure that no event procedure can raise *abandon* (in this example). It is as though there is always the potential for a procedure call between specifying a handler, and the end of the closed construct. In the context of first class procedures this problem is non-trivial.

There are two observations that can be made relating to optimizers that flatten blocks. The first is to note the possibility of statically associating exceptions and handlers, as touched on above. The second is to draw attention to the question of how to achieve the appropriate dynamics: Consider removing the block entry and exit from a block containing the handler A. The return address from A will need to be set to a point in the middle of the resulting sequence, rather than to an exit opcode. Recall that one of the effects of an exit opcode is to pop the handlers introduced in that block, from the notional handler stack. This will need to be mimicked in our new sequence, perhaps by setting the handler entry in the stack to nil.

The use of formalizing techniques such as those used by Clenaghan [8] to describe continuation semantics, may provide further clues to ways of approaching optimization.

§5 Conclusions

We have described an exception handling mechanism for the persistent programming language PS-algol. By making the mechanism orthogonal to the rest of the language we were able to integrate exceptions into the existing system without disrupting the user community. This is especially significant in that the PS-algol environment is heavily dependent on persistent objects, both data and procedures, that have been accumulated across several versions of the language. PS-algol has been able to evolve in a persistent environment.

We have introduced two new statements to support a multi-level, expression-oriented, termination model of exceptions, based on the idea of handler stacks. These are the **raise** and the **when** statements. The **when** statement is desirable syntactic support to maintain local stacks of exception handlers. The **raise** statement is necessary to escape from block constructs. By allowing the exception condition to be associated with a **pntr** value, we achieve the semantics of parameterized exceptions. The model supports the handler responses of termination, resumption, retry and propagation.

We maintain that first-class procedures are sufficient to provide resumption semantics, and that no further syntactic support is desirable, in PS-algol at least.

We have made visible a global PS-algol data structure containing procedures that are invoked on the occurrence of specific system events. By combining existing features of PS-algol with the new exception mechanism we are able to provide familiar responses to system errors, yet allow the development of more sophisticated responses. Where appropriate, resumption has become one such possible response.

First-class procedures are sufficient to provide much of the behaviour required of an exception mechanism. Provision of a terminating raise statement has been the only significant development required of the language.

Acknowledgements

This work was funded in part by an ICL URC grant, and in part by SERC grant GR/D29895 and Alvey/SERC grant GR/D43259.

Many of our colleagues in Glasgow and St Andrews have provided useful input, in the form of comments, suggestions and observations. In particular Fred Brown has made significant contributions to the task of implementation.

References

1. *Ada Reference Manual.* Springer-Verlag, July 1980
2. ARMOUR, I. PS-algol print command: implementation. Working paper, Department of Computing Science, University of Glasgow, September 1986
3. ATKINSON, M.P. *et al. PS-algol Reference Manual - Second Edition, Persistent Programming Report 12,* Department of Computing Science, University of Glasgow and Department of Computational Science, University of St Andrews, 1985
4. BORGIDA, A. Flexible data exceptions. In *VLDB11*, ed. Bubenko. August 1985
5. BORGIDA, A. Accomodating exceptions to (type) contraints in information systems. In *Proceedings of the Appin workshop* Persistence and Data Types, *Persistent Programming Research Report 16*, ed. Atkinson, M.P., Buneman, O.P. and Morrison, R. Department of Computing Science, University of Glasgow, December 1985
6. BROWN, A.L. AND COCKSHOTT, W.P. *The CPOMS Persistent Object Management System, Persistent Programming Research Report 13.* Universities of Glasgow and St Andrews, 1985
7. CARDELLI, L. ML under Unix. *Polymorphism 1*, 3, December 1983
8. CLENAGHAN, K. Readable continuation semantics in an executable first-order equational language. Department of Computing Science, University of Glasgow, October 1986
9. DAVIE, A.J.T. AND MORRISON, R. *Recursive Descent Compiling.* Ellis Horwood, 1981
10. GOODENOUGH, J.B. Exception handling: issues and a proposed notation. *Commun. ACM 21*, 12 (December 1975), 683-696
11. KRABLIN, L. Building flexible multilevel transactions in a distributed persistent environment. *Proceedings of the Appin workshop* Persistence and Data Types, *Persistent Programming Research Report 16*, ed. Atkinson, M.P., Buneman, O.P. and Morrison, R. Department of Computing Science, University of Glasgow, December 1985
12. LEVIN, R. Program structures for exceptional condition handling. Carnegie-Mellon University, June 1977
13. LISKOV, B AND SNYDER, A. Exception handling in CLU. *IEEE Trans Software Engineering*, November 1979
14. LISKOV, B. *et al. CLU Reference Manual.* CSG Memo 161, Draft, MIT, July 1978
15. LUCKHAM, D.C. and POLAK W. Ada exception handling: an axiomatic approach. *ACM Trans Programming Languages and Systems 2*, 2 (April 1980), 225-233
16. MATTHEWS, D.C.J. *Poly Manual.* Technical Report 63, University of Cambridge, February 1985
17. MILNER R. A Proposal for Standard ML. *Polymorphism 1*, 3 (December 1983)
18. MITCHELL, J.G., MAYBURY, W. and SWEET, R. *Mesa Language Manual.* CSL-78-1, Xerox PARC, February 1978
19. OWOSO, G.O. *Data Description and Manipulation in Persistent Programming Languages.* University of Edinburgh, 1984
20. PERSISTENT PROGRAMMING GROUP. *PS-algol Abstract Machine Manual, Persistent Programming Research Report 11.* Universities of Glasgow and St Andrews, 1985
21. ROVNER, P., LEVIN, R. and WICK, J. On extending Modula-2 for building large, integrated systems. Digital Systems Research Center, Palo Alto, 11th January 1985
22. VAN WIJNGAARDEN *et al.* Report on the Algorithmic Language Algol 68. *Numerische Mathematik 14* (1969), 79-218
23. YEMINI, S. and BERRY, D.M. A Modular Verifiable Exception Handling Mechanism. *ACM TOPLAS 7*, 2 (April 1985), 214-243

Appendix 1
Addendum to the PS-algol Reference Manual

A1.1 Introduction

An exception in PS-algol is an exception condition together with an associated excepted value. The excepted value is always of type **pntr**.

The PS-algol exception mechanism is orthogonal to other aspects of the language. In particular existing PS-algol sources need not be modified. Syntactic support has been provided to support a termination model of exceptions. Persistent first-class procedures are sufficient to achieve resumption semantics (via the parameter-passing mechanism or by the use of global data structures).

A1.2 Syntax

The PS-algol syntax is extended as follows:

```
<void-sequence> ::= <handler>
<void-clause> ::= <raise.clause>

<handler> ::= when<structure-id.list>[as<identifier>]do<clause>
<structure-id.list> ::=
        any |
        <structure-identifier>[,<structure-identifier>]*

<raise.clause> ::= raise<pntr-exp3>
```

A1.3 Raising exceptions

An exception can only arise by encountering a **raise** statement. The current flow of control is abandoned. A handler is sought by tracing back the dynamic history, this includes block entry and procedure invocation. The value yielded by the handler replaces that of the block enclosing the handler.

Associated with the *exception condition* is an *excepted value* of type **pntr**. The result of the pointer expression in the **raise** statement becomes the excepted value. For example:

```
structure foot( cstring size )
raise foot( "big" )
```

The excepted value here is an instance of the structure class whose name is *foot*. If the structure class of the excepted value has fields, then we have the functionality of parameterized exceptions. This allows us to pass back diagnostics, or partially completed (perhaps expensive) results.

A1.4 Handling exceptions

Exception handlers are placed using the **when** statement. For example:

```
when foot do write "12 inches'n"
```

Here we have provided a handler for an exception whose excepted value is *foot*. *foot* is a structure name that has previously been declared in this scope. The effect of encountering the **when** statement is to push the handler onto a stack. Should an exception subsequently be raised, the handler stack is repeatedly popped until a handler is found that matches the structure class of the excepted value. All handlers pushed onto that stack during a block are popped at the end of the block.

The excepted value can be made available to the handler by using the as<identifier> phrase. Its type is **cpntr**.

A catch-all handler is provided by using the keyword **any** in place of the list of structure names.

We may specify a list of exceptions that can be fielded by the handler, for example:

```
structure arm
when foot,arm do write "limb"
```

The handler <clause> serves as an alternative completer for the enclosing block. The type of the <clause> must match that of the block.

For example:

```
let len=
begin
        structure foot( cint inches )
        when any do 0
        when foot as e do 12+e( inches )
        ...
        raise foot( 3 )
        ...
        12
end
```

Here, the normal-case value of the block is the integer 12. If however the **raise** statement is executed, the result will be 15. If any other exception is raised, the result will be 0.

A1.5 System events and system exceptions

The PS-algol machine may detect a number of system events. These may derive from receipt of hardware signals, for example keyboard interrupt, or from run-time errors such as attempting to apply a nullproc. The action taken, on the occurrence of each possible system event, is determined by a **proc**, called here an *event procedure*. These event procedures have become visible to user programs by wrapping them up in a globally accessible instance of a structure:

```
let events= events.strc( ... )
```

The *events.strc* structure is declared in the standard prelude.

The default action of most event procedures is to raise a corresponding *system exception*. The structure classes used as system exceptions are also declared in the standard prelude. For example, if a program should attempt to read an invalid integer, the system event procedure *events(ReadI.event)* is invoked. The default value for *events(ReadI.event)* is a procedure whose declaration might look like:

```
proc( cstring offender ->int )
begin
        raise ReadI( offender )
        0
end
```

When an event occurs the system invokes the appropriate event procedure. If there is a normal return from the event procedure, the program will resume from the point at which the event occurred. *Events(ReadI.event)* is non-void. By making this a variable field we have a device that allows the programmer to supply an alternative value to the failed read operation. The default event procedure given above will not of course resume, but initiates a search for a handler. It is not appropriate for the user to replace all event procedures; the restriction is enforced by the existing constancy feature. Note that we are using a global data structure here in the same way in which the user may themselves program resumption.

If an exception is raised but is never handled by the user program, the event procedure *events(Uncaught.event)* is invoked. This procedure displays meaningful diagnostics for all system exceptions, and makes an attempt at user-defined exceptions. The full range of system events and system exceptions are described in Appendix 2.

Appendix 2
System events and system exceptions

This appendix describes the event procedures that are provided by the PS-algol system, along with the system exceptions that they may raise. First we set out the structure class declarations that have been added to the standard prelude to support system events. The prelude can be used as a useful quick reference to the range of events and exceptions that may occur and require a response. This is followed by individual descriptions of the default system event procedures.

A2.1 Additions to the standard prelude

```
!The global pntr events refers to an instance of the following structure class:

structure events.strc
(
        cproc( cstring )                              Error.event,
                                                      Ferror.event ;
        proc( cpntr,cint )                            Uncaught.event ;
        proc()                                        TooManyEvents.event,
                                                      Interrupt.event,
                                                      WindowChange.event ;
        cproc()                                       Arithmetic.event ;
        proc( cstring -> int )                        ReadI.event ;
        proc( cstring -> bool )                       ReadB.event ;
        proc( cstring -> string )                     ReadS.event ;
        proc( cstring -> real )                       ReadR.event ;
        proc( cstring,cstring -> pntr )               OpenDatabase.event ;
        cproc()                                       ApplyingNullproc.event ;
        cproc( cpntr,cstring )                        StructureAccess.event ;
        cproc( cint,cint,cint )                       VectorBounds.event ;
        cproc( cint,cint )                            IliffeBounds.event ;
        proc( cpixel,cint -> pixel )                  PixelBounds.event ;
        proc( c#pixel,cint,cint -> #pixel )           SubImage.event,
                                                      Limit1.event ;
        proc( c#pixel,cint,cint,cint,cint -> #pixel ) Limit2.event ;
        proc( cstring,cint,cint -> string )           SubString.event ;
        proc( cstring,cstring -> string )             StringTooLong.event ;
        proc( -> int )                                Decode.event ;
        proc( cpntr -> pntr )                         CursorTip.event ;
        cproc( cstring )                              Menu.event,
                                                      StringToTile.event
)

!The following structure classes are used by event procedures to raise exceptions:

structure TooManyEvents
structure Interrupt
structure WindowChange
structure Arithmetic
structure ReadI( cstring ReadI.char )
```

```
structure ReadB( cstring ReadB.char )
structure ReadS( cstring ReadS.char )
structure ReadR( cstring ReadR.char )
structure OpenDatabase( cstring OpenDatabase.name,OpenDatabase.explain )
structure ApplyingNullproc
structure StructureAccess( cpntr StructureAccess.pntr ; cstring StructureAccess.class )
structure VectorBounds( cint VectorBounds.index,VectorBounds.lwb,VectorBounds.upb )
structure IliffeBounds( cint IliffeBounds.lwb,IliffeBounds.upb )
structure PixelBounds( cpixel PixelBounds.pixel ; cint PixelBounds.index )
structure SubImage( c#pixel SubImage.image ; cint SubImage.start,SubImage.length )
structure Limit1( c#pixel Limit1.image ; cint Limit1.Xoffset,Limit1.Yoffset )
structure Limit2( c#pixel Limit2.image ; cint Limit2.Xoffset,Limit2.Yoffset,Limit2.Xdim,Limit2.Ydim )
structure SubString( cstring SubString.string ; cint SubString.start,SubString.length )
structure StringTooLong( cstring StringTooLong.left,StringTooLong.right )
structure Decode
structure CursorTip( cpntr CursorTip.point )
structure Menu( cstring Menu.reason )
structure StringToTile( cstring StringToTile.reason )
```

A2.2 System event procedures

Many of the default event procedures have been placed in variable fields of the *events* structure. This allows the programmer to replace a default procedure with a custom one of their own. A number of the event procedures return values. This feature allows the programmer to provide an alternative value for a failed operation.

Note that although every program has a constant called *events* globally available to it as standard, it should be thought of as being declared locally to that program and will be initialized with a different instance of the *events.strc* structure. It is guaranteed that the semantics of calling the default procedure in any one event procedure field, will not vary between programs, but identity of closures is not guaranteed.

events(Error.event), events(Ferror.event)
: These two procedures correspond to the old error reporting interface. They print their **string** argument and **abort** the program. They cannot be replaced by the programmer.

events(Uncaught.event)
: If an exception is allowed to propagate all the way back to the top level of the program, without being intercepted by a handler, this event procedure is invoked. It is parameterized by the excepted value and the line number at which the exception was raised. If the exception is recognized as a system exception, the default procedure will print out the name of the exception and the values in the fields of the excepted value; it will explain the significance of these values as it does so. If the exception is not recognized as a system exception, it will print the class identifier of the excepted value instead. The line number at which the exception arose is printed. If the exception had been raised by an event

procedure then the line number at which that event occurred is also given. To return from this procedure following the *Uncaught.event* event, is to complete the program.

events(TooManyEvents.event)
This is initiated should an event overflow occur, that is, a second event occurred before the event procedure for the first had been called successfully. The default action is to raise the system-defined exception *TooManyEvents.*

events(Interrupt.event)
Invoked on receipt of a keyboard interrupt (usually caused by typing Control-C). To be compatible with previous versions of PS-algol, the default action is to do nothing. Programs can still interrogate the standard boolean *interrupt* procedure to determine whether an interrupt has occurred. By replacing the default, a program could arrange to take some action, perhaps to interact with a user, and then resume from where it left off.

events(WindowChange.event)
(A misnomer for a screen change event.) Called following an attempt by the user to change the size of the screen to which the program is attached. By default the exception *WindowChange* will be raised. (The screen will however remain resolutely unchanged.)

events(Arithmetic.event)
All hardware-detected arithmetic errors such as floating-point or integer overflow, divide by zero etc. map onto this event. The default procedure, which cannot be overwritten, will terminate by raising the exception *Arithmetic.*

events(ReadI.event), events(ReadB.event),
events(ReadS.event), events(ReadR.event)
Each of these correspond to failures in a standard read procedure. They can all be replaced by the programmer with a more appropriate response. The defaults will raise the corresponding system exception, and this will carry the offending character along with it. The offending character (for example a letter where a digit was expected) is made available to the event procedure as the **string** parameter. If the event procedure is allowed to return normally, it will provide a replacement value for the failed operation.

events(OpenDatabase.event)
If a call to *open.database* fails, it will return the **pntr** yielded by this event procedure. By default, and to be compatible with previous versions of PS-algol, the event procedure returns an *error.record*. A user program could use the event procedure to return a more useful value and to take some meaningful alternative action. This does away with the need to slavishly check the result of an *open.database*. The **string** parameters are the name of the database, and an explanation for the failure, respectively. A corresponding structure for use as

an exception, is available from the standard environment. As a example of its use consider:

```
events( OpenDatabase.event ):= proc( string nam, expl ->pntr )
begin
        raise OpenDatabase( nam,expl )
        nil        !dummy
end
...
let thing= s.lookup( "thing",open.database( ... ) )
```

events(ApplyingNullproc.event)
Called when a program has attempted to apply a procedure whose body has not been defined, for example:

```
proc(); nullproc()
```

The constant default action is to terminate by raising the exception *ApplyingNullproc.*

events(StructureAccess.event)
Called by the runtime type checking system when a program acts on a false assumption about the nature of a **pntr** object by attempting to dereference a field. The event procedure is passed the actual **pntr** value together with the expected class identifier of the object. The default action, which cannot be changed, is to raise a *StructureAccess* exception.

events(VectorBounds.event)
This is another event procedure that cannot be replaced. It is due to an attempt to read or write outside the bounds of a vector. The offending index and the actual lower and upper bounds are its parameters, and these appear with the exception that the event procedure raises.

events(IliffeBounds.event)
IliffeBounds.event is called when a pair of incompatible lower and upper bounds are supplied to a vector creation expression. For example:

```
let i = 3
let v = vector i :: i - 1 of "upb < lwb not possible"
```

This event procedure raises the exception *IliffeBounds*, which is provided with the offending lower and upper bounds. It lives in a constant field.

events(PixelBounds.event)
This procedure can be replaced by a user program. It is called when a program attempts to extract a non-existent plane from a pixel. For example:

```
let pix := off and on
pix := pix( 3 )
```

The pixel itself, and the depth level that had been requested, is available as its parameters. A programmer-supplied event procedure could provide an alternative value for the failed indexing operation; the default action is to raise the *PixelBounds* exception.

events(SubImage.event)
This event occurs when an attempt is made to alias a non-existent range of planes in an image. For example:

```
let im := image 10 by 5 of on
im := im( 1|2 )
```

The offending starting plane, and the depth of the requested alias are the two integer parameters. The image parameter is the image on which the operation was being attempted. An alternative result can be supplied to replace the failed expression when the procedure is allowed to return normally. By default the exception *SubImage* is raised, but the procedure can be replaced by a user program.

events(Limit1.event), events(Limit2.event)
These are caused by failed attempts to alias images using the **limit** construct:

```
let im := image 10 by 5 of on
im := limit im at 20,20                  !Limit1.event
im := limit im to 100 by 100 at 1,1      !Limit2.event
```

Both are supplied with the image on which the **limit** was being attempted. *Limit1.event* is provided with the offending x and y offsets. The x and y are also the first two integer parameters of *Limit2.event*, in addition the requested dimensions are made available. By default, the corresponding terminating exceptions *Limit1* and *Limit2* are raised, and these carry the procedure arguments with them as part of the excepted value. A user program might decide instead to resume processing by supplying an alternative image value as the result of the failed expression.

events(SubString.event)
Caused by acting on the false assumption that a string is longer than it really is. For example:

```
let s := "too short"
s := s( 5|20 )
```

The string being operated on, the start position in the original, and the length of the requested string, are its parameters. The default, but replaceable, action is to raise the system exception *SubString*. When the event procedure is allowed to return, an alternative string, to replace the failed string expression, must be provided.

events(StringTooLong.event)
There is an implementation limit on the size of a string, if *s* is a string that is close to this limit, and we attempt:

```
s := s ++ s
```

then this event procedure is activated. The participating strings are received as parameters. These are propagated with the exception *StringTooLong* in the default procedure. A user program might instead supply a procedure that displays a warning before providing an alternative result to the concatenation.

events(Decode.event)
This procedure is activated following an attempt to decode the empty string:

```
let i = decode( "" )
```

By default the exception *Decode* is raised, thus terminating the procedure. If allowed to resume, its integer value replaces the result of the failed decode operation.

events(CursorTip.event)
There are two circumstances when a call of the standard graphics function *cursor.tip* could initiate the *CursorTip.event* procedure. *Cursor.tip* expects a pair of integers wrapped in a *point.strc* structure. *CursorTip.event* is called if a **pntr** to a different structure had been supplied, or if the otherwise valid coordinate pair is outside the dimensions of *cursor*. The parameter to the event procedure is the **pntr** which had been passed to *cursor.tip*. A user program can replace the default, which otherwise raises the exception *CursorTip*.

events(Menu.event)
This event procedure is in a constant field of the events structure and so cannot be replaced. It is called following an unrecoverable error in the use of the *menu* function: *menu* may have been called with the actions and events vectors having mismatched bounds, or *screen* may be too small to accommodate the menu. The event procedure terminates by raising the exception *Menu*. The string in the excepted value structure (and argument to the event procedure itself) is a description of the problem.

events(StringToTile.event)
There are three possible causes of the event that invokes this procedure: an empty string passed to the *string.to.tile* function, a failure by *string.to.tile* to open the font database, or failure to find the specified font. It is not permitted to resume this event and so the default (and non-replaceable) event procedure terminates with the exception *StringToTile*. The string associated with this exception describes the reason for the failure.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics"; The Association for Computing Machines, 11 West 42nd St., New York, NY 10036; University Computing, Vol. 8, NO. 2, (Summer 1986) - see PPRR-23-86.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

In Preparation

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : A DBMS based on the functional data model", to be submitted.

Atkinson, M.P. & Buneman, O.P.
"Database programming languages design", submitted to ACM Computing Surveys - see PPRR-17-85.

Theses

The following Ph.D. theses have been produced by member of the group and are available from:

The Secretary,
Persistent Programming Group,
University of Glasgow,
Department of Computing Science,
Glasgow G12 8QQ,
Scotland.

W.P. Cockshott
Orthogonal Persistent, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 28th October 1986.

Copies of documents in this list may be obtained by writing to:

The Secretary,
The Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00

PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-86	PS-algol Reference Manual - third edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00

In Preparation

Some Applications Programmed in a Persistent Language - Cooper, R.L. (ed).

PS-algol Applications Programming - Cooper, R.L.

A Compilation Technique for a Block Retention Language - Cockshott, W.P. and Davie, A.J.T.

Thoughts on Concurrency - Wai, F.

Concurrency in Persistent Programming Languages - Krablin, G.K.

Providing Database Interfaces Within A Persistent Environment - Atkinson, M.P. & Cooper, R.L.