Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

Training

ANSAwise - Implementing Robust ANSAware Applications

Chris Mayers

Abstract

Organizations deploying distributed applications will require them to be robust.

However distributed applications have to cope with more failure conditions than non-distributed applications.

This module of the ANSAware Simple Bank Exercise explains how ANSAware exceptions can be used to detect and handle these failure conditions. Participants then modify the Simple Bank example to report application-specific error messages, and override the default ANSAware relocation policy.

[Note: this module does not discuss robustness in the sense of dependability features such as replication, since ANSAware does not have general features of this kind.]

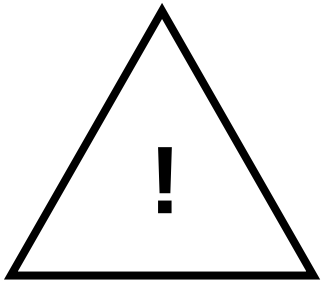
APM.1587.01 **Approved** 2nd October 1995

Briefing Note

Distribution:
Supersedes:
Superseded by:

ANSAware 4.1

Implementing Robust ANSAware Applications



In this session

- **Explain how to handle errors that are detected by the ANSAware infrastructure**
- **Explain the difference between ANSAware exceptions and ANSA terminations**
- **Explain how to use ANSAware exceptions safely**

Operations and Terminations

- **The ANSA Computational Model allows operations with multiple terminations**
- **Remember the operations on the BankAccount:**

  Credit (amount : Integer) : (newBalance : Integer)

  Debit (amount : Integer) : (newBalance : Integer)

  -- but the balance may be insufficient!

  Debit (amount : Integer) : () -> InsufficientFunds
- **There is one Debit operation with two possible outcomes; two separate terminations**
- **ANSAware does not support terminations**

Terminations and Exceptions

- ANSAware IDL declares an operation with this syntax...

```
... OPERATION [ arguments ] RETURNS [ results ]
```

- ... it only allows a single set of results
- ANSAware supports exceptions rather than terminations
- Exceptions are handled in PREPC language rather than IDL

What is an exception?

- An exception is an error condition detected by the ANSAware infrastructure
  - for example, `IllegalOperation` indicates that an operation was invoked on an interface which it does not support
- Exceptions just have a name; no arguments
  - they are an invocation status value
- Exceptions are predefined; you cannot declare new ones
- Exceptions are modelled on C signals
  - unlike C signals, they operate predictably in the presence of concurrency
  - C has no other means of passing back errors from the run-time system

PREPC - Exception syntax

- The ordinary syntax is...

```
! { results } <- ifref$OpName ( args )
```

- ... but you can also state how exceptions are to be handled

```
! { results } <- ifref$OpName ( args ) exception-list
```

- The `exception-list` has the following syntax...

```
Continue statuslist Abort statuslist Signal statuslist
```

- ... it allows you to give a list of exceptions to Continue, Abort, or Signal

Handling with Continue, Abort, or Signal

- **Continue**
  - carries on executing
- **Abort**
  - program is aborted: an error message with the exception details is printed
- **Signal**
  - A signal function (an exception handler) is called...
  - ...with status, invocation arguments, and invocation results as parameters
  - based on these, the exception handler can take appropriate action
  - ...for example, to print out a specific error message

ANSAware signal functions and C signal handlers

- **Although modelled on C signal handlers, ANSAware signal functions are different**
  - **they integrate with the ANSAware infrastructure and distribution mechanisms**
- **C signal handlers may interfere with ANSAware**
  - **Do not use C signal handlers in ANSAware programs**

PREPC - Example exception list

- The following exception list will
  - Continue on ok
  - Abort on everything else

```
!{ results } <- ifref$OpName ( args ) Continue ok Abort *
```

- Use * to indicate "any other status values"

```
! { res } <- ifref$OpName( args ) \
                              Continue ok \
                              Signal   bindFailure \
                              Abort    *
```

- The exception-list is optional
  - PREPC default behaviour if no exception list given

Signal function declaration

- **The name of the signal is determined by the names of the interface and the operation**
  - **If operation-call is as follows...**

```
! DECLARE {ifref} : IfName CLIENT

...

! { results } <- ifref$OpName( args ) Continue ok Signal *
```

  - **... then signal-function must be declared as**

```
Signal_IfName_OpName(status, arg1,...argN, res1,...resN)
```

PREPC pseudo-operations

- PREPC $Import and $Export statements are not operations...
  - they are not operations declared in IDL
  - they are pseudo-operations
- ... but they can have exceptions
  - handled with a PREPC exception list
  - with a signal function if needed
- The names of the signal function are fixed
  - they are `Signal_Prepc_Import` and `Signal_Prepc_Export`, respectively

Returning from the Signal Function

- The signal function can return one of three return-values
  - `ExceptionContinue`
  - `ExceptionAbort`
  - `ExceptionRetry`
- Code generated by PREPC will check this return-value
  - to see how to proceed on returning from the Signal function

Communicating between invocation and signal function

- The thread that invoked the operation may need to know whether an exception occurred
  - the invocation may have decided to Continue...
  - ... and the invoking thread has to tidy up afterwards
- But signal functions cannot access the data of the calling thread
- Special macro-definitions are provided for this
  - `thread_setExceptionCode(ansa_Cardinal code)`
  - `ansa_Cardinal thread_getExceptionCode()`
- `thread_setExceptionCode` sets a value, stored in the current thread's data record
  - this can be examined when the operation which generated the exception has completed

Signal Function - Basic approach


```
/* signal handler */
Signal_Foo_Op( ... )
{
   thread_setExceptionCode( 1 );
   return ExceptionContinue;
}
body()
{
!  {} <- foo$Op() Continue ok Signal *
   if( thread_ExceptionCode() == 1 )
      /* exception has been raised */

}
```

Signal Function - Example


```
...
! {access_result} <- bank_ifref$Access(account_number, pin) \
                     Continue ok Signal *


   /* Check that no exception occurred on Access operation. */
   if ( thread_getExceptionCode() == 1 )
   {
      bank_state == BANK_OUT_OF_SERVICE
      return( OpFailure );
   }

   /* check whether access operation succeeded... etc */
```

More complex communications

- To communicate more than an exception code between signal function and calling thread:
  - `thread_setExceptionState(ansa_StatePtr state)`
  - `ansa_StatePtr thread_getExceptionState()`
- Can pass a pointer to any sort of data structure
  - but must coerce it to be an `ansa_StatePtr`

Initializing the exception state

- The exception code and exception state are always initialized
  - initialized with `thread_setExceptionCode (0)`...
  - `...` and `threadSetExceptionState ((ansa_StatePtr) 0)`
  - this is done by PREPC-generated code before each invocation that could generate an exception
- So applications should not initialize them

Exceptions - Default behaviour

- The default behaviour (if no exception-list is given) is roughly...

```
!  { results } <- ifref$op ( arguments ) \
      Continue ok \
      Signal transmitTimeout,invalidNonce,illegalOperation, \
             illegalInterface,abnormalReturn \
      Abort *
```

- ...but the ANSAware default signal function will be called if any of the "Signal" status-values occur
  - it will try to *relocate* the interface
  - the default signal function is called `Signal_binder_relocate()`
  - it is provided by the infrastructure

Relocation in the default signal function

- `Signal_binder_relocate()` tries to relocate the interface
  - by contacting any locator services it knows about
- If the interface is registered with the Trader
  - it will have information about Trader's `Relocator` interface, and will call it
  - this will check whether it knows a new location for the faulty interface...
  - ... it may have moved

Effect of relocation attempt

- If relocation succeeds
  - the interface-reference will be updated to a new one...
  - ...`ExceptionRetry` is returned from the default signal function
  - ... the ANSAware infrastructure will retry the invocation
- If relocation fails
  - otherwise, `ExceptionAbort` is returned from the default signal handler...
  - ... which causes the program to be aborted with the message

```
Abort: Prepc.Relocate: 1282 (transmitTimeout)
```

- In either case, nothing happens to the faulty service itself
  - This is relocation, not migration
  - There is no support in ANSAware yet for migrating a service

Using relocation from a signal function

- If an exception list is given, the default signal function will never be called
  - so relocation will never happen
- To use relocation with your own signal function
  - just call the `Signal_binder_relocate` function from your signal function

Exercises

- **Modify the Simple Bank teller program to print out an appropriate error message when relocation fails**
  - **to inform the users that the bank service is not running**
- **Modify the teller program so it does not attempt relocation and prints out the error message immediately**
  - **rather than waiting for the timeout**
- **Modify the teller program error messages for the Access operation**
  - **so the same error message is given for Invalid PIN and No Such Account**

Your notes

Exceptions - Summary

- ANSAware does not support terminations
  - use an ENUMERATION as a discriminator as one of the results, instead
- ANSAware supports exceptions for error conditions detected by the infrastructure
  - can rely on default behaviour
  - can use signal functions to handle specific error conditions
  - can use relocation in case an interface has moved

Exceptions (invocation status values)/1

- **Here are some common causes of exceptions**

`bindFailure`

- **a PREPC Trader pseudo-operation (Import, for example) has failed**
- **the Trader's checkpoint file is corrupt**

`abnormalReturn`

- **the server operation returned `Unsucessful Invocation` as its value**

Exceptions (invocation status values)/2

`transmitFailure`

- the Trader is itself not running (on `Trader.Lookup`)
- the Trader machine is unreachable (on `Trader.Lookup`)
- the client is built for the wrong Trader address (on `Trader.Lookup`)

`transmitTimeout`

- the Trader has a stale offer; the server is not running (on `Prepc.Relocate`)
- the Trader is itself not running (on `Prepc.Relocate`)
- the client is configured for the wrong Trader address (on `Prepc.Relocate`)
- the Trader is built for the wrong address (on `Prepc.Relocate`)

Exceptions - more information

- For more information on exceptions, see Chapter 3 of *Application Programming in ANSAware*
- For a complete list of exceptions (invocation status values) and their causes, also see Chapter 3 of *Application Programming in ANSAware*
- To briefly review the Computational Model, see Chapter 2 of *Application Programming in ANSAware*