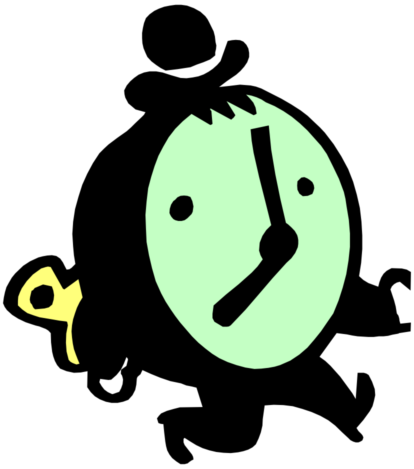
Real-Time Systems Programming

Summer-Semester 2002
Lecture 15
6 June 2002



Exceptions

The 5-Minute Review Session

1) When is *preventive maintenance* useful?
2) How can we classify *failures*?
3) What is a principal limitation of using *testing* for system validation?
4) What is the *consistent comparison problem*?
5) What are prerequisites for a successful application of *N-version programming*?
6) What is *backward error recovery*? What is the *domino effect*?

Overview

1) Example of a Fail-Safe System
2) Exceptions
   - Exception handling in older real-time languages
   - Modern exception handling
   - Resumption vs. termination

Example of a Fail-Safe System: VOTRICS

- Train Signalling System developed by Alcatel
- An industrial example of applying design diversity in a safety-critical RT environment
- Objective of train signalling system:
  - Collect data about the state of the tracks in a train station – current position and movements of trains, position of points
  - Set signals and shift points such that trains can move safely through the station according to a given time table

VOTRICS cont.

- VOTRICS is partitioned into two independent subsystems
- First system:
  - Accepts commands from operators
  - Collects data from tracks
  - Calculates intended positions of signals and points
  - Uses a standard programming paradigm
  - Uses a TMR architecture to tolerate single HW fault

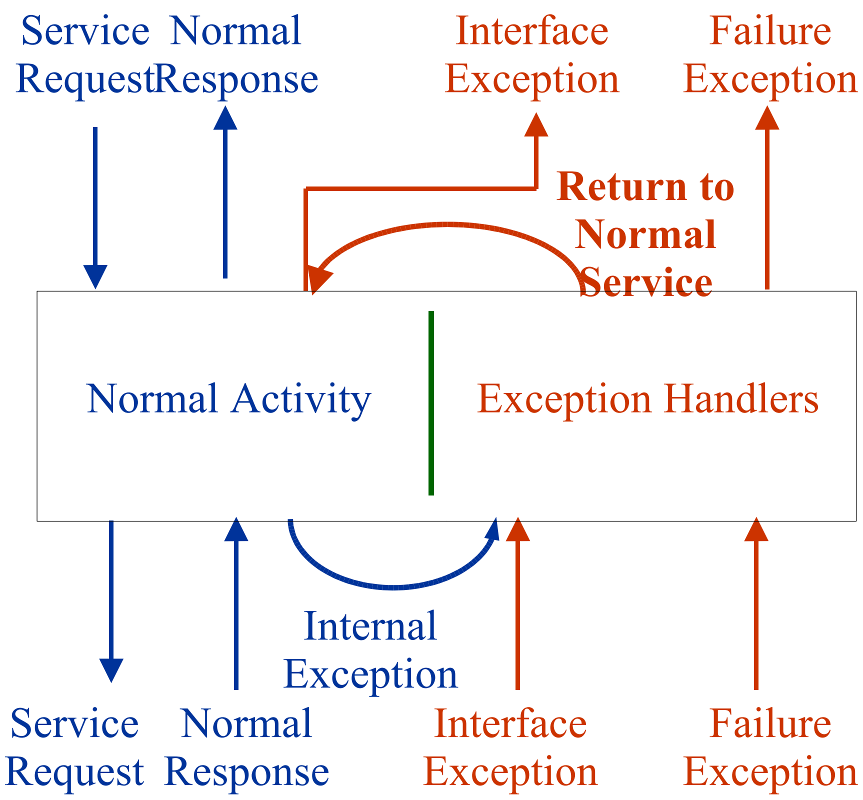
VOTRICS cont.

- The second system, the "safety bag":
  - Monitors safety of the state of the station
  - Has access to RT data base and intended outputs of 1^{st} system
  - Dynamically evaluates safety predicates derived from the "rule book" of the railway authority
  - Based on expert-system technology
  - Also implemented on TMR HW architecture

VOTRICS cont.

- The two systems exhibit a substantial degree of independence
- Used different specifications as starting point
  - Operational requirements vs. safety rules
- Used different implementation approach
  - Standard programming vs. expert system
- System has been operational in different railway stations for a number of years, no unsafe state has been detected

An Ideal Fault-Tolerant Component



Dynamic Redundancy and Exceptions

- ***Exception***: can be defined as the occurrence of an error
- ***Raising*** (or ***signalling*** or ***throwing***) the exception: Bringing an exception to the attention of the invoker of the operation which caused the exception
- ***Handling*** (or ***catching***) the exception: The invoker's response

Exceptions for Error Recovery

- Exception handling is a forward error recovery mechanism
  - No roll back to a previous state
  - Instead control is passed to the handler so that recovery procedures can be initiated
- However, the exception handling facility can be used to provide backward error recovery
  - Can implement recovery blocks using exceptions

Exceptions

Exception handling can be used to:

- cope with ***abnormal conditions*** arising in the environment
  - The original purpose of exceptions
- enable program ***design faults*** to be tolerated
- provide a ***general-purpose error-detection and recovery*** facility

Requirements for Exception Handling

- ***R1 (Simplicity)***: Should be simple to understand and use
- ***R2(Unobtrusiveness)***: Exception handling code should not obscure understanding of the software
  - *R1 and R2 crucial for designing reliable systems!*
- ***R3 (Efficiency)***: Run-time overheads should be incurred only when handling an exception
  - *This may be relaxed, e.g. if speed of recovery is critical*
- ***R4 (Uniformity)***: Uniform treatment of exceptions detected both by the environment and by the program
- ***R5 (Recovery)***: It should allow recovery actions

Traditional Exception Handling

- In the following, we will discuss
  - Unusal returns (the C classic)
  - Forced branches (Assembly)
  - The non-local goto
  - Procedure variables

Exceptional Returns

- The classic: encoding exceptions as ***unusual return value*** or ***error return***
- ***Example***: ***C/POSIX***

```
if (function(params) == AN_ERROR) {
                 – error handling code
} else {
          -- normal return code
};
```

- R1 (Simplicity): ☺
- R2 (Unobtrusiveness): ☹
- R3 (Efficiency): 😐
- R4 (Uniformity): ☹
- R5 (Recovery): ☺

Forced Branch

- Typical approach in assembly languages
- ***Skip return:***
  - Instruction following the subroutine call is skipped to indicate the presence/absence of an error
  - Return address (program counter) is incremented by the length of a simple jump instruction
  - Can permit more than one exceptional return by accordingly manipulating the PC

```
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp
   DEVICE_NOT_ENABLED
# normal processing
```

- R1 (Simplicity): 😐
- R2 (Unobtrusiveness): 😐
- R3 (Efficiency): ☺
- R4 (Uniformity): ☹
- R5 (Recovery): ☺

Non-Local Goto

- A high-level language version of a forced branch which uses label variables
- ***Example***: non-local `goto` of ***RTL/2*** [Barnes 1976]

```
svc data rrerr
  label erl; % a label variable %
enddata

proc WhereErrorIsDetected();
  ... goto erl; ...
endproc;

proc Caller();
  ... WhereErrorIsDetected(); ...
endproc;
```

```
proc main();
  ...
restart:
  ...
  erl := restart;
  ...
  Caller();
  ...
end proc;
```

Non-Local Goto

- Control flow is broken
  - Is therefore best used for unrecoverable errors
- This type of goto is more than just a jump
  - Implies an abnormal return from a procedure
- The stack must be unwound
  - Until the environment restored is that of the procedure containing the declaration of the label
- R1 (Simplicity): ☹
- R2 (Unobtrusiveness): ☹
- R3 (Efficiency): ☺
- R4 (Uniformity): ☺
- R5 (Recovery): ☺

Procedure Variables

- An ***error procedure variable*** allows to return control to the point where the error originated
  - Can be used for recoverable errors
- ***Example*** in RTL/2:

```
svc data rrerr;
   label erl;
   proc(int) erp;
enddata;
proc recover(int);
  ...
endproc;

proc WhereErrorIsDetected();
  ...
  if recoverable then erp(n)
  else goto erl end;
 ...
endproc;
```

```
proc Caller();
  ...
  WhereErrorIsDetected();
  ...
endproc;

proc main();
  ...
  erl := fail;
  erp := recover;
  ...
  Caller();
  ...
fail:
  ...
end proc
```

Procedure Variables – Assessment

- Again, programs can become very difficult to understand
- R1 (Simplicity): ☹
- R2 (Unobtrusiveness): ☹
- R3 (Efficiency): ☺
- R4 (Uniformity): ☺
- R5 (Recovery): ☺

- ***The modern approach:***
  - Introduce exception-handling facilities directly into the language – this allows ***better structuring***

Exceptions and their Representation

- Error detection can be classified according to **<u>who</u>** detects the error
  - ***Environmental*** error detection (*divide by zero*)
  - ***Application*** error detection (*assertion failure*)
- Error detection can also be classified according to **<u>when</u>** it is detected:
  - A ***synchronous*** exception is raised as an immediate result of a process attempting an inappropriate operation
  - An ***asynchronous*** exception is raised some time after the operation causing the error
    - may be raised in the process which executed the operation or in another process

Classes of Exceptions I

According to when exceptions are raised by whom, we can distinguish four types of exceptions:

1. ***Detected by the environment and raised synchronously***
   - Array bounds error, divide by zero
2. ***Detected by the application and raised synchronously***
   - The failure of a program-defined assertion check

Classes of Exceptions II

3. ***Detected by the environment and raised asynchronously***
   - An exception raised due to the failure of some health monitoring mechanism
4. ***Detected by the application and raised asynchronously***
   - one process may recognise that an error condition has occurred earlier in another process

- Asynchronous exceptions are also called ***asynchronous notifications*** or ***signals***
  - Mostly an issue with concurrent programming (⇨ *later*)

Synchronous Exceptions

There are two models for the ***declaration*** of synchronous exceptions:

- A ***constant name***
  - needs to be explicitly declared
  - *Example*: ***Ada***
- An ***object*** of a particular type
  - may or may not need to be explicitly declared
  - *Example*: ***C++***, ***Java***

Exception Declarations in Ada

- The exceptions that can be raised by the Ada run time system are declared in package **Standard**
- This package is visible to all Ada programs

```
  package Standard is
          ...
          Constraint_Error : exception;
          Program_Error : exception;
          Storage_Error : exception;
          Tasking_Error : exception;
          ...
   end Standard;
```

The Domain of an Exception Handler

- Within a program, there may be **several handlers** for a particular exception
- Associated with each handler is a ***domain***:
  - The region of computation during which, if an exception occurs, the handler will be activated
- The ***accuracy*** or ***granularity*** with which a domain can be specified will determine how precisely the source of the exception can be located

Exception Domains in Ada

- In a block structured language, like Ada, the domain is normally the ***block***
- Procedures, functions, accept statements etc. can also act as domains

```
declare
  subtype Temperature is Integer range 0 .. 100;
begin
  -- read temperature sensor
exception
  -- handler for Constraint_Error
end;
```

Exception Domains in Java

- Not all blocks can have exception handlers
- The domain of an exception handler must be explicitly indicated and the block is considered to be ***guarded***
- Java does this with a ***try-block***

```
try {
  // statements which may raise exceptions
}

catch (ExceptionType e) {
  // handler for e
}
```

Granularity of Exception Handling Domain

```
declare
  subtype Temperature is Integer range 0 .. 100;
  subtype Pressure is Integer range 0 .. 50;
  subtype Flow is Integer range 0 .. 200;
begin
  -- read temperature sensor and calculate its value
  -- read pressure sensor and calculate its value
  -- read flow sensor and calculate its value
  -- adjust temperature, pressure and flow
exception
  -- handler for Constraint_Error
end;
```

- ***Which calculation caused the exception ?***
- Arithmetic overflow etc. ⇒ further difficulties

Solution 1: Decrease Block Size

```
declare
      ...
begin
      begin
           -- read temperature sensor and calculate its value
      exception
           -- handler for Constraint_Error for temperature
      end;
      begin
           -- read pressure sensor and calculate its value
      exception
           -- handler for Constraint_Error for pressure
      end;
      begin
           -- read flow sensor and calculate its value
      exception
           -- handler for Constraint_Error for flow
      end;
      -- adjust temperature, pressure and flow
exception
      -- handler for other possible exceptions
end;
```

- ***This is fairly tedious !***

Sol. 2: Handle Exceptions at Statement Level

```
-- NOT VALID Ada
declare
    ...
begin
    Read_Temperature_Sensor;
    exception -- handler for Constraint_Error;
    Read_Pressure_Sensor;
    exception -- handler for Constraint_Error;
    Read_Flow_Sensor;
    exception -- handler for Constraint_Error;
    -- adjust temperature, pressure and flow
end;
```

- ***CHILL*** has such a facility [CCITT 1980]
- R2 (Unobtrusiveness): ☹

Solution 3: Parametrized Exceptions

- Allow parameters to be passed with the exceptions
- ***Java***:
  - Exception is an **object**
  - Can contain arbitrary information
- ***Ada***:
  - Provides a **predefined procedure `Exception_Information`**
  - This returns implementation-defined details on the occurrence of the exception

Exception Handling Resolution

- ***Question***: *Which handler handles a raised exception?*
- The answer is easy if there exists a handler for the exception that is immediately associated with the block or procedure where the exception was raised
- However, this may not always be the case
  - ***Example***: an exception raised in a procedure as a result of a failed assertion involving the parameters passed to the procedure
- In these cases, the answer is not so obvious

Static Exception Handling Association

- The compiler/linker tries to establish for each exception the corresponding handler
- If no such handler can be found, report this as an error at ***compile time***
- ***CHILL***:
  - Requires that a procedure specifies which exceptions it may raise (that is, not handle locally)
  - The compiler can then check the calling context for an appropriate handler
- ***Java*** and ***C++***:
  - Allows a function to define which exceptions it can raise
  - however, does not require a handler to be available in the calling context

Dynamic Association/Exception Propagation

- Look for handlers up the chain of invokers; this is called ***propagating*** the exception
- The approach taken by ***Ada***, ***Java***, ***C++***, ***Modula 2/3***
- A problem occurs where exceptions have scope
  - An exception may be propagated outside its scope, thereby making it impossible for a handler to be found
- Most languages provide a ***catch all*** exception handler

Unhandled Exceptions

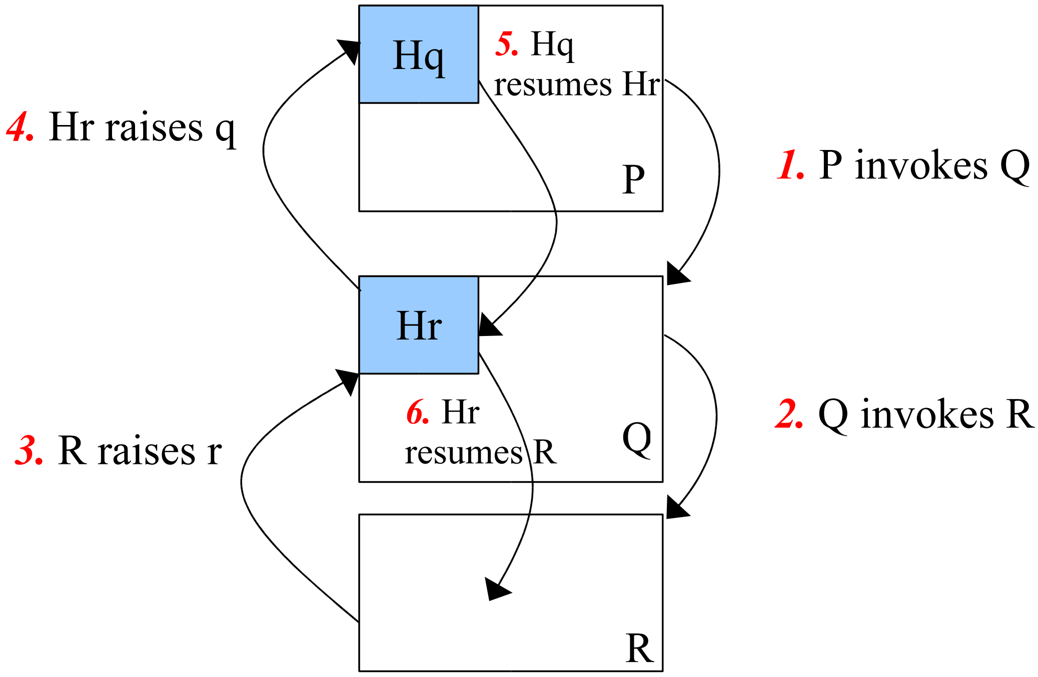
- An unhandled exception causes a sequential program to be aborted
- If the program contains more than one process and a particular process does not handle an exception it has raised, then usually that process is aborted
- ***It is not clear whether the exception should be propagated to the parent process – see later***

Resumption vs. Termination Model

- ***Question***: *Should the invoker of an exception continue its execution after the exception has been handled ?*
  - ***Yes*** ⇒ ***Resumption*** or ***notify model***
  - ***No*** ⇒ ***Termination*** or ***escape model***
  - ***Perhaps*** ⇒ ***Hybrid model***
    - The exception handler decides

The Resumption Model

- Can view handler as an implicit procedure which is called when the exception is raised
- ***Example***: procedures P, Q and R, handlers Hr and Hq



The Resumption Model

- The exception handler may be able to take care of the problem that caused the exception
- Resumption model may be advantageous when the exception has been raised **asynchronously** (⇨ *later*) and has little to do with the current process execution
- ***A Difficulty***: the repair of errors raised by the RTS
- ***Example***: arithmetic overflow in the middle of a sequence of complex expressions
  - Registers contain partial evaluations
  - Calling the handler overwrites these registers
- ***Pearl*** & ***Mesa*** support resumption and termination

Retry Resumption

- The strict resumption model is not easy to implement
- Alternative: ***Retry model***
  - Re-execution of the block associated with the exception handler
  - Exception handler sets flag to indicate that error has occurred
  - *Example*: ***Eiffel***
- Note that local variables of the block must not be re-initialised on a retry

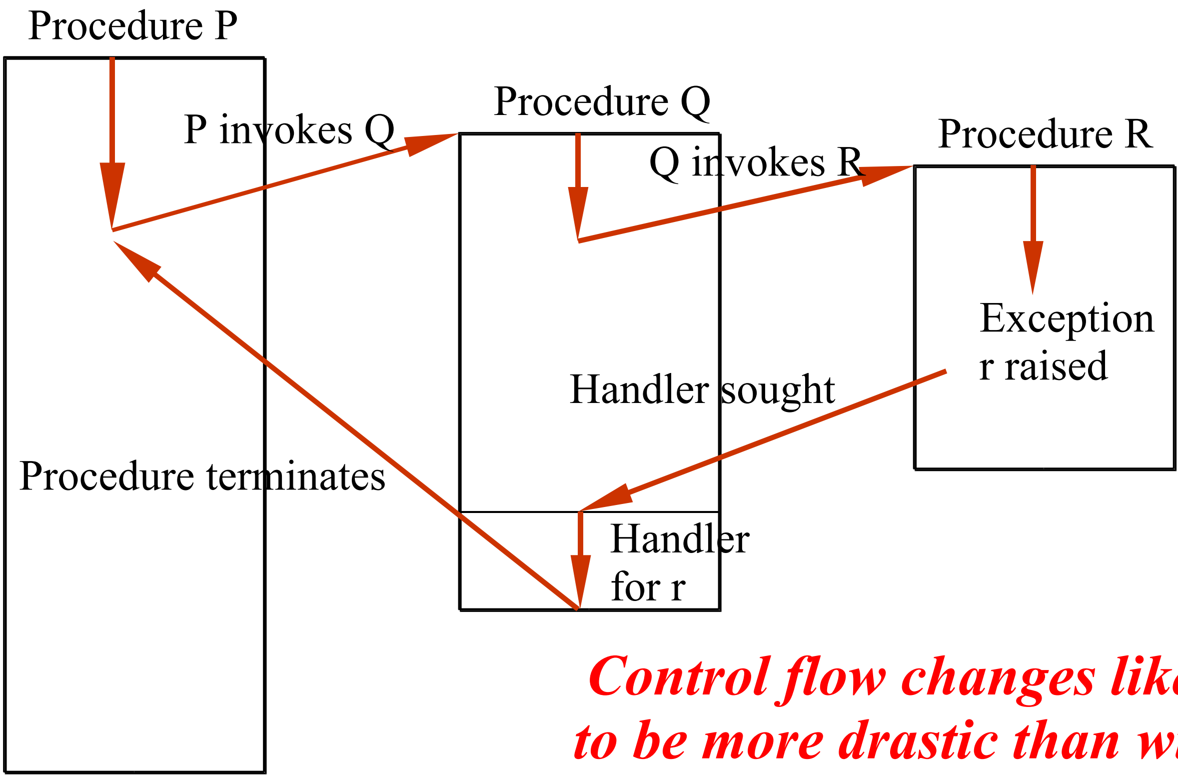
The Termination Model

- In the ***termination model***, when an exception has been raised and the handler has been called:
  - The block or procedure containing the handler is terminated
  - Control ***does not return*** to where the exception occurred
  - Control is passed to the caller (procedure domain) or to the first statement following the block (block domain)
- An invoked procedure, therefore, may terminate in one of a number of conditions:
  - the ***normal condition***, or
  - any of the possible ***exception conditions***
- ***Ada*** and ***Java*** support the termination model

Block Termination

```
declare
  subtype Temperature is Integer range 0 .. 100;
begin
  ...
  begin
    -- read temperature sensor and calculate its value,
    -- may result in an exception being raised
  exception
    -- handler for Constraint_Error for temperature,
    -- once handled this block terminates
  end;
  -- code here executed when block exits normally
  -- or when an exception has been raised and handled.
exception
     -- handler for other possible exceptions
end;
```

Procedure Termination



Control flow changes likely to be more drastic than with block termination !

The Hybrid Model

- The handler decides if the error is recoverable
  - ***Yes***: the handler can return a value and the semantics are the same as in the resumption model
  - ***No***: invoker is terminated
- ***Example***: Signal mechanisms of ***Mesa***
- ***Eiffel*** supports the restricted **retry** model

Exception Handling and Operating Systems

- SW will often (*not always*) run on top of an OS
- The OS will detect certain synchronous error conditions
  - Memory violation, illegal instruction, etc.
- This will usually ***terminate*** the process; however, many systems allow ***error recovery***
- ***Example***: ***signal*** mechanism in POSIX
  - Allows handlers to be called when exceptions are detected
  - Once the signal is handled, the process is resumed at the point where it was “interrupted” (***resumption model***)
- If a language supports the termination model, the run-time support system (RTSS) must catch the error and manipulate the program state accordingly

Summary I

An ***exception handling model*** has to specify:

- ***How are exception represented ?***
  - May or may not be explicitly represented in a language
- ***What is the domain of an exception handler ?***
  - What is the region of computation during which, if an exception occurs, the handler will be activated ?
- ***What if there is no exception handler in the enclosing domain ?***
  - An exception can be ***propagated*** to the next outer level enclosing domain –
  - or it can be considered to be a ***programmer error***

Summary II

- ***How to proceed after an exception has been handled ?***
  - ***Resumption model***: the invoker of the exception is resumed at the statement after the one at which the exception was invoked
  - ***Termination model***: the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure.
  - ***Hybrid model***: the handler may choose whether to resume or to terminate
- ***Can parameters be passed to the handler ?***
- If an ***OS*** is used, the OS may also use exceptions to communicate error conditions to the application