Characteristics of a RTS

- Large and complex
  - Language and OS support
  - Structuring, component-based development
- Concurrent Execution
  - Concurrent programming, synchronization
  - Real-Time Communication (e.g. CAN)
- Guaranteed response times
  - Scheduling, response time analysis
- **Extreme reliability (safety critical)**
  - **Fault tolerance and recovery**

Note that the focus of this course is on software aspects

Some facts

- 1955, 10% US weapons systems required computer software, 1980s, 80%
- 26 milions of lines of program code, Ericsson telecom system, less than 5 minutes shutdown per year -- Reseanably reliable
- E.g. 2.5 milions lines of code for industrial robots, no-stop per 60,000 hours (about 7 years) -- Highly reliable
- Typically every milion lines of code may introduce 20,000 bugs (from a study on large software systems, 1986
  - 90% may be found by testing
  - a further 200 faults may be detected in the first year of operation
  - The rest 1800 are left undetected
  - Routine maintenance may result in 200 bug fixes (with 200 new faults introduced)
- Typically 50% of the budget (money/time) for testing and bug-fixes
  - E.g. 1.2 billions $ per year for ... ...

Fault Tolerance and Recovery

- Goal
  - To understand the factors which affect the reliability of a system and techniques for fault-tolerance and recovery
- Topics
  - Reliability, failure, faults, failure modes
  - Fault prevention and fault tolerance
    - Hardware redundancy:
      - Static (e.g.TMR) and
      - dynamic (e.g. checksum)
    - Software redundancy:
      - Static: N-Version programming and
      - Dynamic redundancy: recovery block and exception handling

4 sources of faults which can result in system failure

- Inadequate specification
- Design errors in software
- Processor/hardware failure
- Interference on the communication subsystem

Reliability, Failure and Faults
(terminology)

- The reliability of a system is a measure of success with which it conforms to some authoritative specification of its behaviour
- When the behaviour of a system deviates from its specification, this is called a failure e.g. the aircraft is out of control.
- Failures result from unexpected problems or errors e.g. a deadlock internal to the system which eventually manifest themselves in the system's external behaviour
- The mechanical or algorithmic cause for errors are termed faults e.g. a "wrong" resource allocation algorithm (exception handling is needed)
- Systems are composed of components which are themselves systems: hence: fault -> error -> failure

Fault Types

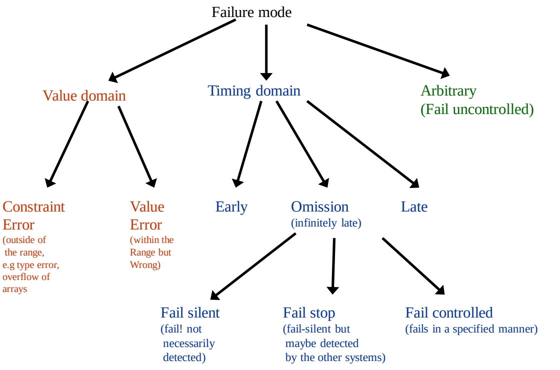
- Temporary faults occur from time to time
  - transient faults start at a particular time, remains in the system for some period and then disappears (mainly due to external changes)
    - E.g. hardware components which react to radioactivity
    - Many faults in communication systems are transient
  - Intermittent faults are transient faults that occur from time to time (mainly due to internal problems)
    - E.g. a hardware component that is heat sensitive, it works for a time, stops working, cools down and then starts to work again
- Permanent faults remain in the system until they are repaired; e.g., a broken wire or a software design error.

Approaches to Achieving Reliable Systems

- Fault prevention attempts to eliminate any possibility of faults creeping into a system before it goes operational
  - E.g. modelling, verification, testing
- Fault tolerance enables a system to continue functioning even in the presence of faults
  - Recovery
- Both approaches attempt to produce systems which have well-defined failure modes

Failure Modes

(typically)



Fault Prevention

- Two stages: fault avoidance and removal
- Fault avoidance attempts to limit the introduction of faults during system construction by:
  - use of rigorous, if not formal, specification of requirements
  - use of rigorous, if not formal, design methods
    - modelling and verification techniques
    - design reviews, code inspections and system testing
    - use of techniques of component-based design and the most reliable components within the given cost and performance constraints
  - use of languages with facilities for
    - Data abstraction and modularity
    - Concurrency, and real time

Why Fault Tolerance (1)

- In spite of fault avoidance, design errors in both hardware and software components will exist
- System testing can never be exhaustive and remove all potential faults:
  - A test can only be used to show the presence of faults, not their absence.
  - It is sometimes impossible to test under realistic conditions
  - most tests are done with the system in simulation mode and it is difficult to guarantee that the simulation is accurate
  - Errors that have been introduced at the requirements stage of the system's development may not manifest themselves until the system goes operational

Why Fault Tolerance (2)

- In spite of all the testing and verification techniques, hardware components will fail; the fault prevention approach will therefore be unsuccessful when
  - either the frequency or duration of repair times are unacceptable, or
  - the system is inaccessible for maintenance and repair activities, e.g. the crewless spacecraft
- Alternative is **Fault Tolerance**

Fault Tolerance

(levels depending on the application)

- Full Fault Tolerance — the system continues to operate in the presence of faults, (maybe only) for a limited period, with no significant loss of functionality or performance
  - Most safety critical systems require full fault tolerance, however in practice many settle for graceful degradation
- Graceful Degradation (fail soft) — the system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair
  - ABS in a modern car: even a sensor is broken, the brake should continue to work.
- Fail Safe — the system maintains its integrity while accepting a temporary halt in its operation
  - A310 Airbus's control computers on detecting an error on landing, restore the system to a safe state and then shut down. Safe state: both wings with the same settings

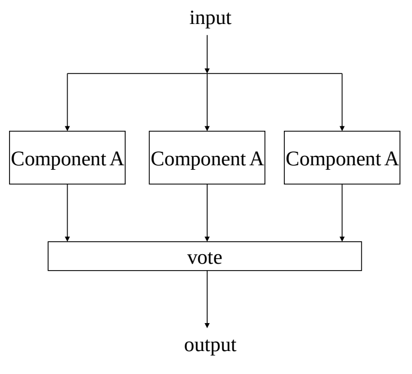
Fault tolerance mainly by *redundancy*

- All fault-tolerant techniques rely on extra elements introduced into the system to detect & recover from faults
- Components are redundant as they are not required in a perfect system, often called protective redundancy
  - Aim: minimise redundancy while maximising reliability, subject to the cost and size constraints of the system
  - Warning: the added components inevitably increase the complexity of the overall system; it itself can lead to less reliable systems
    - It is advisable to separate out the fault-tolerant components from the rest of the system

Hardware Fault Tolerance

- Two types: static (or masking) and dynamic redundancy:
  - Static: redundant components are used inside a system to hide the effects of faults; e.g. Triple Modular Redundancy
    - TMR — 3 identical subcomponents and majority voting circuits; the outputs are compared and if one differs from the other two that output is masked out
    - Assumes the fault is not common (such as a design error) but is either transient or due to component deterioration
    - To mask faults from more than one component requires NMR
  - Dynamic: redundancy supplied inside a component which indicates that the output is in error; provides an error detection facility; recovery must be provided by another component
    - E.g. communications checksums and memory parity bits

TMR



Software Fault Tolerance

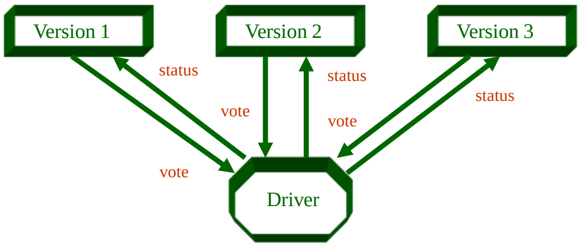
- Static: N-Version programming
- Dynamic: Detection and Recovery
  - Backward error recovery: Recovery blocks:
  - Forward error recovery: Exceptions

Static Software Redundancy

N-Version Programming

- Design diversity
  - The independent generation of N (N > 2) functionally equivalent programs from the same initial specification
    - No interactions between groups
  - The programs execute concurrently with the same inputs and their results are compared by a driver process
    - Invoking each of the versions
    - Waiting for the versions to complete
    - Comparing and acting on the results (terminate one or more versions)
- The results (VOTES) should be identical, if different the consensus result, assuming there is one, is taken to be correct
- E.g. Boeing 777 flight control system, a single Ada program was produced but 3 different processors, and 3 different compilers were used to obtain diversity

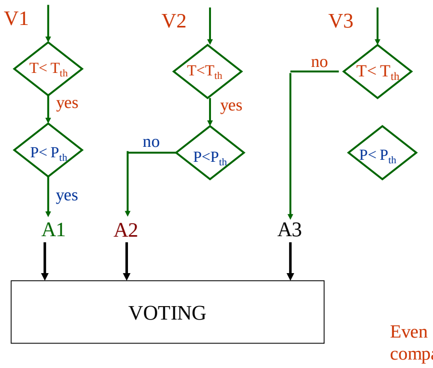
N-Version Programming



Problems with Vote Comparison

- How often the comparison should take place?
  - Certainly not every instruction, performance penalties
  - Too large granularity may produce a wide divergence in results
- To what extent can votes be compared?
  - Text or integer arithmetic will produce identical results
  - Real numbers => different values
    - Need inexact voting techniques

Consistent Comparison Problem



Each version will produce a different but correct result

Even if use inexact comparison techniques, the problem occurs

N-version programming depends on

- Initial specification — The majority of software faults stem from inadequate specification? A specification error will manifest itself in all N versions of the implementation
  - We need to assume the assumption: no error in the specification
- Independence of effort — Experiments produce conflicting results
  - It is very rare that different versions can find identical faults.
  - More recent studies: a 3-version system is 5 to 9 times more reliable than a single version system of high-quality.
- Adequate budget — The predominant cost is software. A 3-version system will triple the budget requirement and cause problems of maintenance.
  - Would a more reliable system be produced if the resources potentially available for constructing an N-versions were instead used to produce a single version?

Dynamic Software Redundancy

Software Dynamic Redundancy

Four phases

- error detection — no fault tolerance scheme can be utilised until the associated error is detected
  - damage confinement and assessment — to what extent has the system been corrupted? The delay between a fault occurring and the detection of the error means erroneous information could have spread throughout the system
- error recovery — techniques should aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality)
  - fault treatment and continued service — an error is a symptom of a fault; although damage repaired, the fault may still exist

Error Detection

- Platform detection (by the execution environment where the program runs)
  - hardware — protection violation, arithmetic overflow
  - OS/RTS — array bound error, null pointer, value out of range
- Application detection
  - Timing checks (e.g. watch dog timer)
  - Coding checks (checksums, memory parity bits)
  - Reasonableness checks (assertions?)
  - Dynamic reasonableness check (new output should not be too different from the previous one)

Error Recovery

- Probably the most important phase of any fault-tolerance technique
- Two approaches: forward and backward recovery

Forward error recovery (FER)

- FER relies on continue from an erroneous state by making selective corrections to the system state
  - This includes making the controlled environment safe, which may be damaged because of the failure
  - It is system specific and depends on accurate predictions of the location and cause of errors (i.e, damage assessment)
    - E.g. error code in UNIX for system calls

Backward Error Recovery (BER)

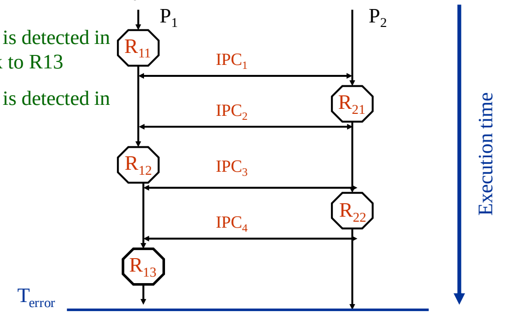
- BER relies on restoring the system to a previous safe state and executing an alternative section of the program
  - This has the same functionality but uses a different algorithm (c.f. N-Version Programming) and therefore "no fault"
  - The point to which a process is restored is called a recovery point and the act of establishing it is termed checkpointing (saving appropriate system state)
- Advantage: the erroneous state is cleared and it does not rely on finding the location or cause of the fault
- Disadvantage: it cannot undo errors in the environment!

The Domino Effect

- With concurrent processes that interact with each other, BER is more complex Consider:

If the error is detected in P1 rollback to R13

If the error is detected in P2 ?



Fault Treatment and Continued Service

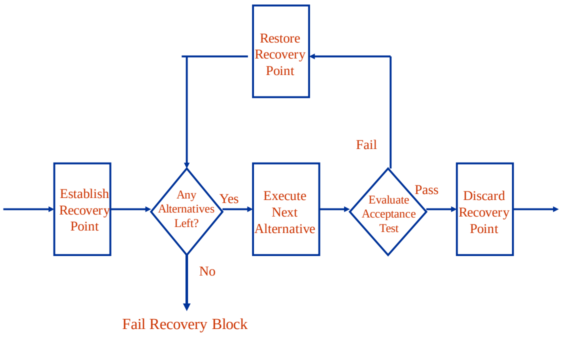
- ER returned the system to an error-free state; however, the error may recur; the final phase of F.T. is to remove the fault from the system
  - The automatic (on-line) treatment of faults is difficult and system specific
  - Often, assume that all faults are transient, and error recovery techniques can cope with recurring faults
- Fault treatment can be divided into 2 stages: fault location and system repair
  - Error detection techniques can help to trace the fault to a component. For hardware the component can be replaced
  - A software fault can be removed in a new version of the code
- In non-stop applications it will be necessary to modify the program while it is executing, e.g. Erlang allows "on-line upgrading of module"

Language Support for Error Recovery

Language support for BER: Recovery Block

- At the entrance to a block, design an automatic recovery point and at the exit an acceptance test
  - The acceptance test is used to test that the system is in an acceptable state after the block's execution (primary module)
  - If the acceptance test fails, the program is restored to the recovery point at the beginning of the block and an alternative module is executed
- If the alternative module also fails the acceptance test, the program is restored to the recovery point and yet another module is executed, and so on
- If all modules fail then the block fails and recovery must take place at a higher level

Recovery Block Mechanism



Recovery Block Syntax

(it may be easily programmed using "exception handling" e.g. in Ada)

```
ensure <acceptance test>
by
   <primary module>
else by
   <alternative module>
else by
   <alternative module>
   ...
else by
   <alternative module>
else error
```

- Recovery blocks can be nested
- If all alternatives in a nested recovery block fail the acceptance test, the outer level recovery point will be restored and an alternative module to that block executed

The Acceptance Test

- The acceptance test provides the error detection mechanism which enables the redundancy in the system to be exploited
  - The design of the acceptance test is crucial to the effectiveness of the RB scheme, and "completeness" to detect "all possible errors"
  - There is a trade-off between providing comprehensive acceptance tests and keeping overhead to a minimum, so that fault-free execution is not affected
- Note that the term used is acceptance not correctness; this allows a component to provide a degraded service
  - All the previously discussed error detection techniques can be used to form the acceptance test

N-Version Programming vs Recovery Blocks

- Static (NV) versus dynamic redundancy (RB)
- Design overheads — both require alternative algorithms, NV requires driver, RB requires acceptance test
- Runtime overheads — NV requires N * resources, RB requires establishing recovery points
- Diversity of design — both susceptible to errors in requirements
- Error detection — vote comparison (NV) versus acceptance test(RB)
- Atomicity — NV vote before it outputs to the environment, RB must be structured to only output after the passing of an acceptance test

Language support for FER: Exception Handling

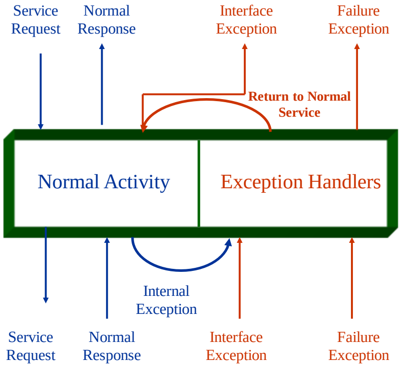
- An exception = occurrence of an error
- Exception handling is a forward error recovery mechanism, as there is no roll back to a previous state; instead control is passed to the handler so that recovery procedures can be initiated
  - However, the exception handling facility can be used to provide backward error recovery

Exceptions

Exception handling can be used to:

- cope with abnormal conditions arising in the environment,
- provide a general-purpose error-detection and recovery facility
- enable program design faults to be tolerated.

Ideal Fault-Tolerant Component



EH in "Traditional" Languages

- Unusual return value or error return from a procedure or a function.
- C supports this approach

```
if(function_call(parameters) == AN_ERROR) {
  -- error handling code
} else {
  -- normal return code
}
```

Exception Declaration and Handling in Ada (1)

- Each handler is a sequence of statements

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead : exception;
begin
  -- statements which may cause the exceptions
exception
  when E: Sensor_High | Sensor_Low =>
    -- Take some corrective action
    -- if either sensor_high or sensor_low is raised.
    -- E contains the exception occurrence
  when Sensor_Dead =>
    -- sound an alarm if the exception
    -- sensor_dead is raised
end;
```

Exception Declaration and Handling in Ada (2)

- `when & others` is used to avoid enumerating all possible exception names
- Only allowed as the last choice and stands for all exceptions not previously listed

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead: exception;
begin
  -- statements which may cause exceptions
exception
  when Sensor_High | Sensor_Low =>
    -- take some corrective action
  when E: others  =>
    Put(Exception_Name(E));
    Put_Line(" caught. Information is available is ");
    Put_Line(Exception_Information(E));
    -- sound an alarm
end;
```

"Pre-defined/Standard" Exceptions in Ada

- The exceptions that can be raised by the Ada RTS are declared in package Standard:

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

- This package is visible to all Ada programs.

Example

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

Scope/Domain

- In a block structured language, like Ada, the domain is normally the block.

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

- Procedures, functions, accept statements etc. can also act as domains

Granularity of Domain

- Is the granularity of the block is inadequate?

```
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
  -- according to requirements
exception
  -- handler for Constraint_Error
end;
```

- The problem for the handler is to decide which calculation caused the exception to be raised
- Further difficulties arise when arithmetic overflow and underflow can occur

```
declare  -- First Solution: decrease block size
  subtype Temperature is Integer range 0 .. 100;
  subtype Pressure is Integer range 0 .. 50;
  subtype Flow is Integer range 0 .. 200;
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 according
  -- to requirements
exception  -- handler for other possible exceptions
end;
```

-- this **is** long-winded **and** tedious!
(there are other solutions, check the details in Ada)

Recovery Blocks and Exceptions

- Remember:

```
ensure  <acceptance test>
by
   <primary module>
else by
   <alternative module>
else by
   <alternative module>
...
else by
   <alternative module>
else error
```

- Error detection is provided by the acceptance test; this is simply the negation of a test which would raise an exception
- The only problem is the implementation of state saving and state restoration

Recovery Blocks in Ada

```
procedure Recovery_Block is
  Primary_Failure, Secondary_Failure,
  Tertiary_Failure: exception;
  Recovery_Block_Failure : exception;
  type Module is (Primary, Secondary, Tertiary);

  function Acceptance_Test return Boolean is
  begin
    -- code for acceptance test
  end Acceptance_Test;
```

```
procedure Primary is
begin
  -- code for primary algorithm
  if not Acceptance_Test then
    raise Primary_Failure;
  end if;
exception
  when Primary_Failure  =>
    -- forward recovery to return environment
    -- to the required state
    raise;
  when others  =>
    -- unexpected error
    -- forward recovery to return environment
    -- to the required state
    raise Primary_Failure;
end Primary;
-- similarly for Secondary and Tertiary
```

```
 begin
   Recovery_Cache.Save;
    for Try in Module loop
    begin
      case Try is
        when Primary => Primary; exit;
        when Secondary => Secondary; exit;
        when Tertiary => Tertiary;
      end case;
    exception
      when Primary_Failure =>
        Recovery_Cache.Restore;
      when Secondary_Failure =>
        Recovery_Cache.Restore;
      when Tertiary_Failure =>
        Recovery_Cache.Restore;
        raise Recovery_Block_Failure;
      when others =>
        Recovery_Cache.Restore;
        raise Recovery_Block_Failure;
    end;
  end loop;
end Recovery_Block;
```

Summary

- All exception handling models address the following issues
  - Exception representation: an exception may, or may not, be explicitly represented in a language
  - The domain of an exception handler: associated with each handler is a domain which specifies the region of computation during which, if an exception occurs, the handler will be activated
  - Exception propagation: when an exception is raised and there is no exception handler in the enclosing domain, either the exception can be propagated to the next outer level enclosing domain, or it can be considered to be a programmer error
  - Resumption or termination model: this determines the action to be taken after an exception has been handled.

Exception handling: final remark

- It is not unanimously accepted that exception handling facilities should be provided in a language
- The C and the occam2 languages, for example, have none
- To sceptics, an exception is a GOTO where the destination is undeterminable and the source is unknown!
- They can, therefore, be considered to be the antithesis of structured programming
- This is not the view taken here!

Summary

- Reliability: a measure of the success with which the system conforms to some authoritative specification of its behaviour
- Failure: When the behaviour of a system deviates from that which is specified for it, this is called a failure
  - Failures result from errors caused by faults
  - Faults can be transient, permanent or intermittent
- Fault prevention consists of fault avoidance and fault removal
- Fault tolerance involves the introduction of redundant components into a system so that faults can be detected and tolerated

Summary

- Static techniques for fault-tolerence
  - N-version programming: the independent generation of N (where N >= 2) functionally equivalent programs from the same initial specification
  - TMR: Triple Modular Redundancy
- Dynamic techniques:
  - BER: backward error recovery
  - FER: forward error recovery

Summary

- With backward error recovery, it is necessary for communicating processes to reach consistent recovery points to avoid the domino effect
- For sequential systems, the recovery block is an appropriate language concept for BER
- Although forward error recovery is system specific, exception handling has been identified as an appropriate framework for its implementation
- The concept of an ideal fault tolerant component was introduced which used exceptions