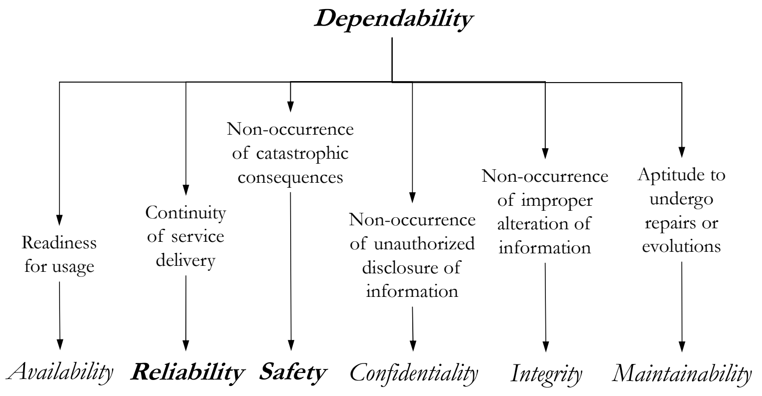
2. Dependability issues

Credits to A. Burns and A. Wellings

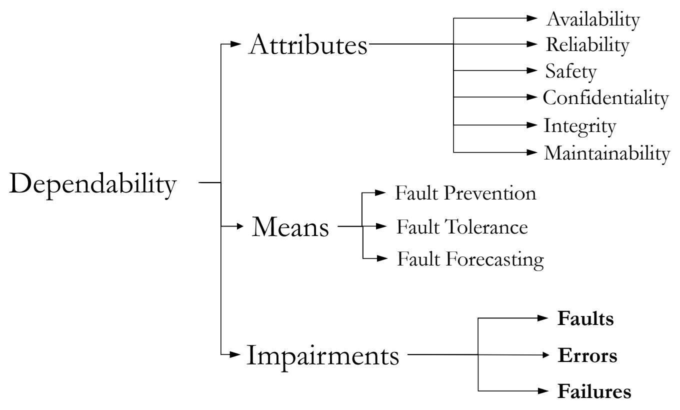
Characteristics of a RTS

- Complex and multidisciplinary
- Concurrent control of separate system components
- Interaction with special-purpose hardware
- Predictability
- Domain-specific ***dependability***
  - Reliability, safety, …
- Efficiency of implementation
  - In time, space, communication

Dependability: ramifications


Dependability
Readiness for usage
Continuity of service delivery
Non-occurrence of catastrophic consequences
Non-occurrence of unauthorized disclosure of information
Non-occurrence of improper alteration of information
Aptitude to undergo repairs or evolutions
Availability
Reliability
Safety
Confidentiality
Integrity
Maintainability

Dependability: terminology

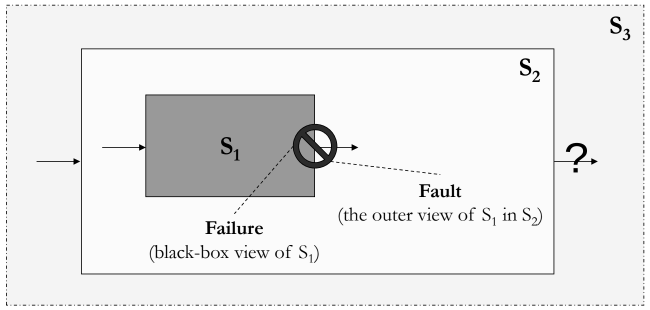


Failure and faults – 1

- A ***failure*** is when the behavior of a system deviates from what is specified for it
- Failures result from unexpected problems internal to the system which eventually manifest themselves in the system's external behavior
- These problems are called ***errors*** and their mechanical or algorithmic or conceptual cause are termed ***faults***
  - Errors are states of the system
  - Faults are what causes the error to exist
- Systems are composed of components which are themselves systems: hierarchically therefore

Failure} → {Fault → Error → Failure} → {Fault

Failure and faults – 2



Safety – 1

- General definition
  - Safety : freedom from conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or environmental harm
- Most systems which have an element of risk associated with their use are therefore unsafe by definition!
- A mishap is an unplanned event or series of events that can result in unacceptable effect
- Safety is expressed as the probability that conditions which can lead to mishaps do not occur regardless of whether the intended function is performed
- How does that relate to reliability?

Safety – 2

- A paradox
  - Measures taken to increase the likelihood of a weapon firing when required may increase the possibility of its accidental detonation
  - Aiming at better reliability may decrease safety
- In many respects the only safe airplane is one that never takes off
  - Which however is not very reliable
  - Aiming at greater safety may decrease reliability
- As with reliability, to ensure the safety requirements of an embedded system, system safety analysis must be performed throughout all stages of its development

Reliability

- The ***reliability*** of a system is a measure of the success with which it conforms to the specified behavior (its continuity of service)
  - May vary with time
- Very solid metrics exist for hardware components
  - Electronic components are observed to fail at a constant rate
- Reliability at time t for those components is modeled by
  - $R(t) = Ge^{-\lambda t}$
    where G is a component-specific constant and λ is the sum of the failure rates of all its constituent components
- The ***mean time between failures*** (MTBF) is a commonly used metric (time to failure + time to repair)
  - For a system without redundancy MTBF = 1 / λ

Scope of discussion – 1

- We want to understand the impairment factors (faults) which affect the reliability of a system and how they can be tolerated
- Topics in scope
  - Reliability, failure and faults
  - Failure modes
  - Fault prevention and fault tolerance
  - Software static redundancy (N-version programming)
  - Software dynamic redundancy
  - The recovery block approach to software fault tolerance
    - A comparison between N-version programming and recovery blocks
  - Dynamic redundancy and exceptions

Scope of discussion – 2

- Four sources of ***faults*** that can result in system ***failure***
  - Inadequate specification
    - Not covered here
  - **Erroneous software design**
    - Covered in this segment
  - Processor failure
    - Not covered here, see B&W book
  - Interference on the communication subsystem
    - Not covered here, see B&W book

Dependability means – 1

- ***Fault prevention*** attempts to eliminate any possibility of faults creeping into a system before it goes operational
  - Fault avoidance
  - Fault removal
- ***Fault tolerance*** enables a system to continue functioning even in the presence of faults
  - Hardware / software fault tolerance
  - Static / dynamic fault tolerance
- Both approaches attempt to produces systems which have well-defined failure modes
- *Fault forecasting* is of no consequence here

Dependability means – 2

- *Fault prevention* techniques base on
  - Quality control
  - Robust engineering of components
    - However, cost penalty for engineering reliability into components through reduced failure rate
- *Fault tolerance* techniques base on
  - Use and management of redundant components
    - Made possible by microprocessor technology as weight, volume and power requirements associated with redundant hardware decrease

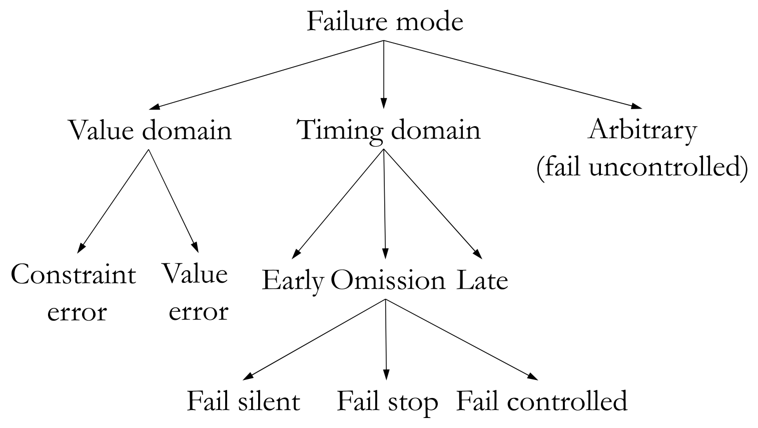
Fault types

- ***Permanent faults*** remain in the system until they are repaired
  - E.g., a broken wire or a software design error
- ***Transient faults*** start at a particular time, remain in the system for some period and then disappear
  - E.g., HW components with adverse reaction to radioactivity
    - Only fails when exposed
  - Many faults in communication systems are transient (e.g., congestion)
- ***Intermittent faults*** are transient faults that occur from time to time
  - E.g., a HW component that is heat sensitive, it works for a time, stops working, cools down and then starts to work again

Software faults

- Colloquially called "bugs"
  - Bohr-bugs: consistently reproducible and identifiable
    - Pun on Bohr's atom model
    - E.g., a division by zero, an out-of-bound access to an array
  - Heisen-bugs: extremely difficult or impossible to reproduce exactly
    - Pun on Heisenberg's uncertainty principle of quantum mechanics
    - E.g., a race condition, …
- Software doesn't deteriorate with age
  - It is either correct or incorrect
  - But its faults can remain dormant for long periods so that errors are not activated

Failure modes



Fault prevention: fault avoidance

- *Fault avoidance* attempts to limit the introduction of faults during system construction by
  - Use of the most reliable components within the given cost and performance constraints
  - Use of thoroughly-refined techniques for interconnection of components and assembly of subsystems
  - Packaging the hardware to screen out expected forms of interference
  - Rigorous, if not formal, specification of requirements
  - Use of proven design methodologies
  - Use of languages with facilities for data abstraction and modularity
  - Use of software engineering environments to help manipulate software components and thereby manage complexity

Fault prevention: fault removal

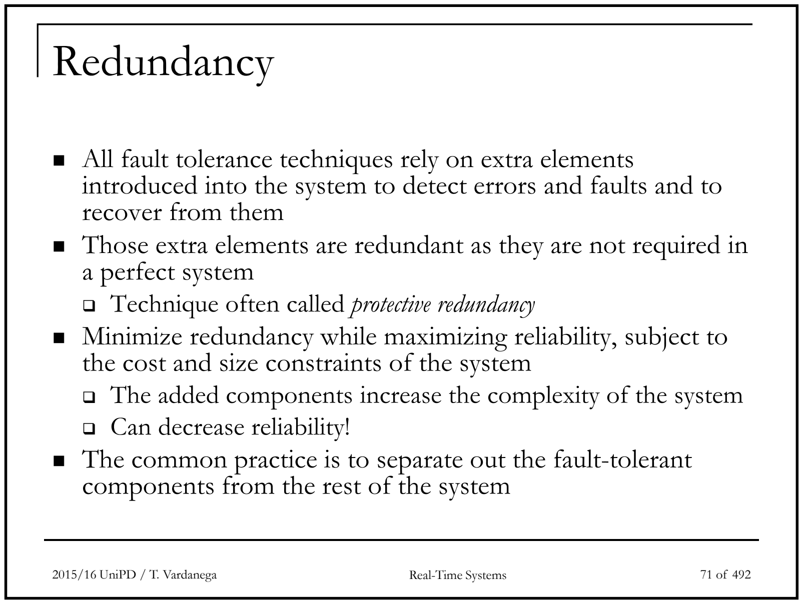
- In spite of fault avoidance, design faults may still inject errors in both hardware and software components
- *Fault removal* uses procedures for finding errors and removing their causes
  - E.g., design reviews, program verification, code inspection, system testing
- System testing can never be exhaustive 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 introduced at the requirements stage of the system development may not manifest themselves until the system goes operational

Limits of fault prevention

- In spite of all the testing and verification techniques, hardware components will certainly decay and fail
  - Even if all software design faults were removed
- The fault prevention approach will therefore be unsuccessful when
  - The frequency of failure or the duration of repair times are unacceptable (too high, too long)
  - The system is inaccessible for maintenance and repair activities
    - An extreme example of such system is Voyager, the crewless spacecraft currently 10 billions km from the sun!
- In those cases *fault tolerance* is the necessary complement

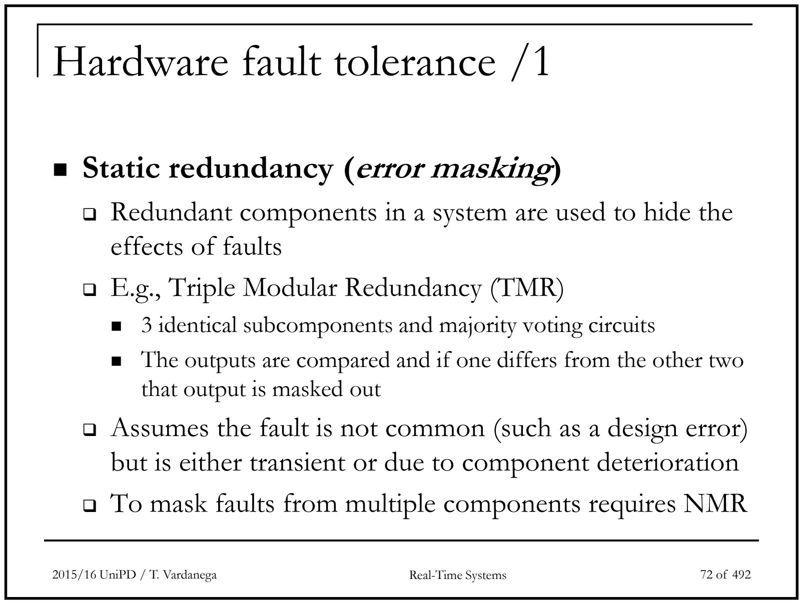
Levels of fault tolerance

- Full fault tolerance
  - The system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance
- Graceful degradation (fail soft)
  - The system continues to operate in the face of errors, accepting partial degradation of functionality/performance during recovery or repair
- Fail safe
  - The system maintains its integrity while accepting a temporary halt in its operation (which must be fail silent or fail stop or fail controlled)
- The level of fault tolerance required will depend on the domain of application
- Most safety-critical systems require full fault tolerance, however in practice many settle for graceful degradation



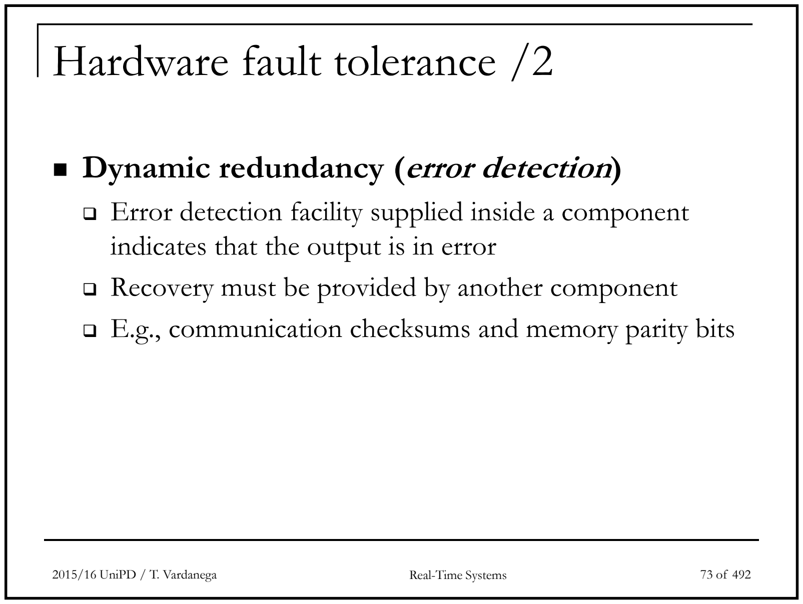
Redundancy

- All fault tolerance techniques rely on extra elements introduced into the system to detect errors and faults and to recover from them
- Those extra elements are redundant as they are not required in a perfect system
  - Technique often called *protective redundancy*
- Minimize redundancy while maximizing reliability, subject to the cost and size constraints of the system
  - The added components increase the complexity of the system
  - Can decrease reliability!
- The common practice is to separate out the fault-tolerant components from the rest of the system



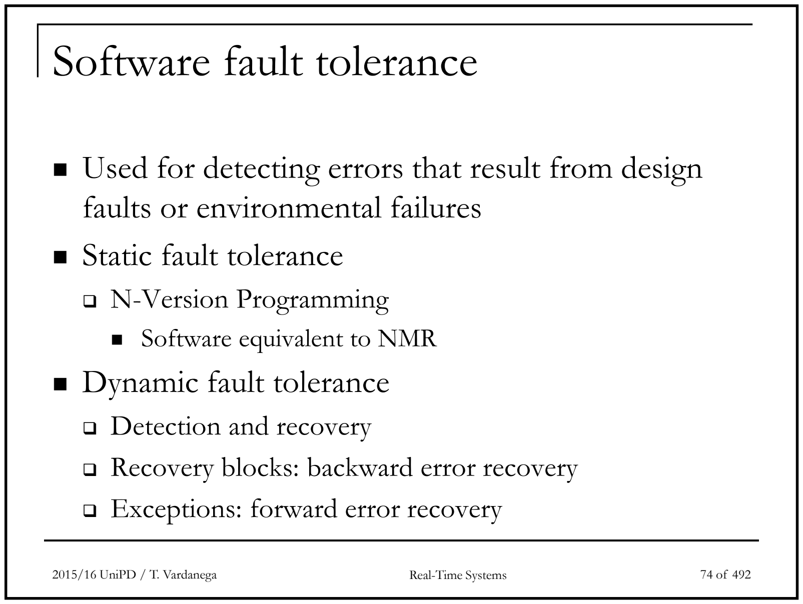
Hardware fault tolerance /1

- **Static redundancy (*error masking*)**
  - Redundant components in a system are used 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 multiple components requires NMR



Hardware fault tolerance /2

- **Dynamic redundancy (*error detection*)**
  - Error detection facility supplied inside a component indicates that the output is in error
  - Recovery must be provided by another component
  - E.g., communication checksums and memory parity bits



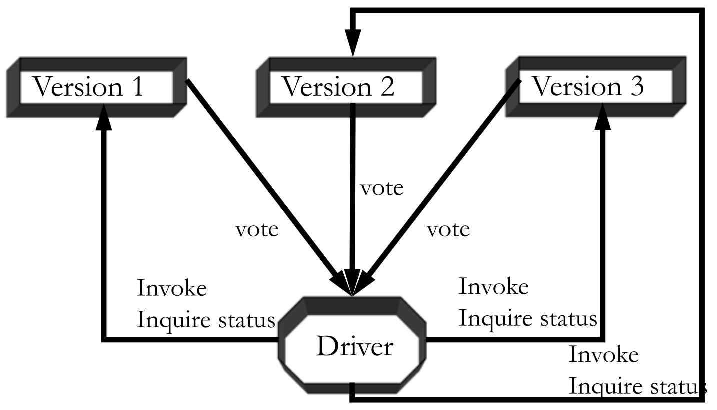
Software fault tolerance

- Used for detecting errors that result from design faults or environmental failures
- Static fault tolerance
  - N-Version Programming
    - Software equivalent to NMR
- Dynamic fault tolerance
  - Detection and recovery
  - Recovery blocks: backward error recovery
  - Exceptions: forward error recovery

N-version programming – 1

- Design diversity
  - The independent generation of N (N > 2) functionally equivalent programs from the same initial specification
  - No interactions between development groups ⇦
  - The programs execute concurrently with the same inputs and their results are compared by a driver process
  - The results (assimilated to votes) should be identical
  - If they are not the consensus result – assuming there is one – is taken to be correct

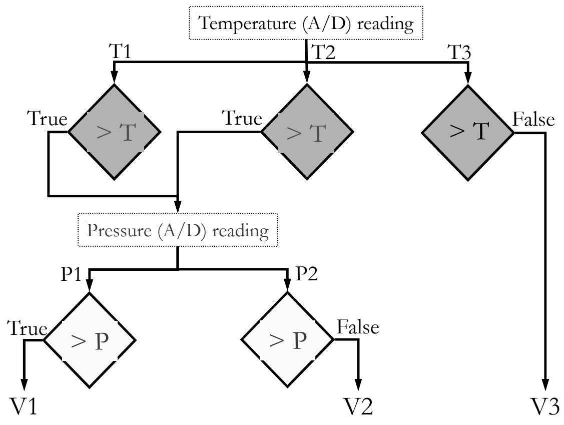
N-version programming – 2



Vote comparison

- To what extent can votes be compared?
  - Far from obvious
- Text or integer or Boolean arithmetic will produce identical results
  - Can vote on equality
- Real numbers will produce different values
  - Need inexact voting techniques
- User defined types outside of numerics will need their own equality
  - E.g., limited types in Ada

Consistent comparison problem



Each version will produce a different result, but correct within finite-precision arithmetic

Even using inexact voting, the problem occurs when the values are close to the decision threshold

N-version programming – 3

- Initial specification
  - The majority of software faults stem from *inadequate specification*
  - A specification error will manifest itself in all N versions of the implementation
- Independence of effort
  - Experiments produce conflicting results
  - A complex part of a specification leads to lack of understanding of the requirements
  - If poorly specified requirements also refer to rarely occurring input data, common design errors may not be caught during system testing
- Adequate budget
  - The predominant cost in some real-time embedded systems is software
  - A 3-version system will triple the budget requirement and complicate maintenance
  - Would a more reliable system be produced if the resources potentially available for constructing an N-versions were instead used to produce a better single version?

Software dynamic redundancy

- *Error detection*
  - No fault tolerance scheme can be utilized until the associated error is detected
- *Damage confinement and assessment*
  - To what extent has the system been corrupted?
  - The delay between fault occurrence and error detection means that erroneous information could have spread throughout the system
- ***Error recovery***
  - ER 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/manifestation of a fault
  - Although the damage is repaired the fault may still exist

Error detection

- Environmental detection
  - Hardware
    - E.g., illegal instruction
  - OS / run-time support
    - E.g., null pointer, out of bound address
- Application detection
  - Replication checks
  - Timing checks
  - Reversal checks
  - Coding checks
  - Reasonableness checks
  - Structural checks
  - Dynamic reasonableness check

Damage confinement and assessment

- Damage assessment is closely related to damage confinement techniques used
- Damage confinement is concerned with structuring the system so as to minimize the damage caused by a faulty component (a.k.a. *firewalling*)
- Modular decomposition provides static damage confinement
  - Allows data to flow through well-defined pathways
    - This needs a strongly typed language
- Atomic actions provides dynamic damage confinement
  - They are used to progress the system from one consistent state to another

Forward error recovery

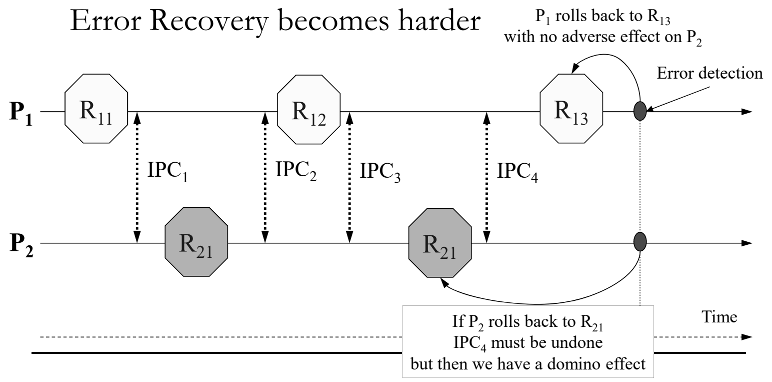
- FER continues on from an erroneous state by making selective corrections to the system state
- This includes making safe the controlled environment after it may have become hazardous or damaged because of the activation of the error
- It is system specific and depends on accurate predictions of the location (where to look), cause of errors (how to tell) and damage assessment
- Examples
  - Redundant pointers in data structures
  - Use of self-correcting codes such as Hamming Codes

Backward error recovery

- 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 and therefore no same fault
  - As in N-Version Programming
- The point to which a process is restored is called a ***recovery point*** and the act of establishing it is termed ***check-pointing***
  - The recovery point contains a trustworthy system state
  - The erroneous state is cleared and no attempt is made at finding the location or cause of the fault
  - Can therefore be used to recover from unanticipated faults including design errors
  - But it cannot undo errors in the environment!

The domino effect

- With cooperative concurrent processes Backward Error Recovery becomes harder


P$_1$ rolls back to R$_{13}$ with no adverse effect on P$_2$
Error detection
P$_1$
R$_{11}$
R$_{12}$
R$_{13}$
IPC$_1$
IPC$_2$
IPC$_3$
IPC$_4$
P$_2$
R$_{21}$
R$_{21}$
Time
If P$_2$ rolls back to R$_{21}$ IPC$_4$ must be undone but then we have a domino effect

Fault treatment and continued service

- Error recovery returns the system to an error-free state
- The error may however recur
- The final phase of fault tolerance thus is to eradicate the fault from the system
- The automatic treatment of faults is difficult and system specific
- Some systems assume all faults are transient; others that error recovery techniques can cope with staying faults
- *Fault treatment* can be divided into 2 stages
  - Fault location and diagnosis
  - System repair
- Error detection techniques can help trace the fault to a component
  - The hardware component can be replaced
  - A software fault can be removed in a new version of the code
    - But non-stop applications shall then modify the program while executing!

Recovery blocks – 1

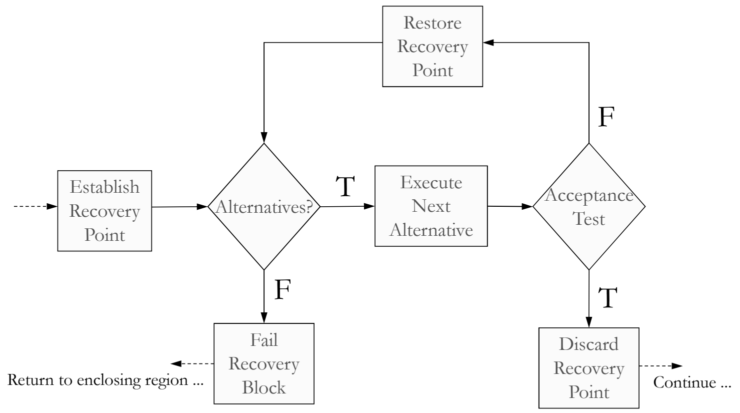
- Language support for backward error recovery
- At the entrance to a block is an automatic recovery point and at the exit an acceptance test
- The acceptance test is used to test that the system is in an acceptable state after the block's execution
  - Primary module
- If the acceptance test fails, 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 forth
- If all modules fail then the block fails and recovery must take place at a higher level

Recovery blocks – 2

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

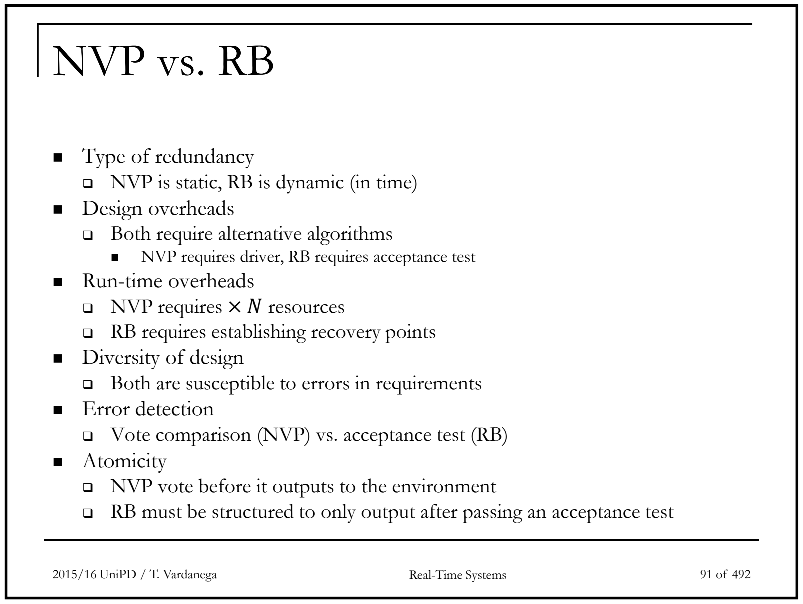
- Recovery blocks can be nested
- If all alternatives in a nested recovery block fail the acceptance test, the outer level recovery point will be restored and an alternative module to that block will be executed

Recovery blocks – 3



The acceptance test

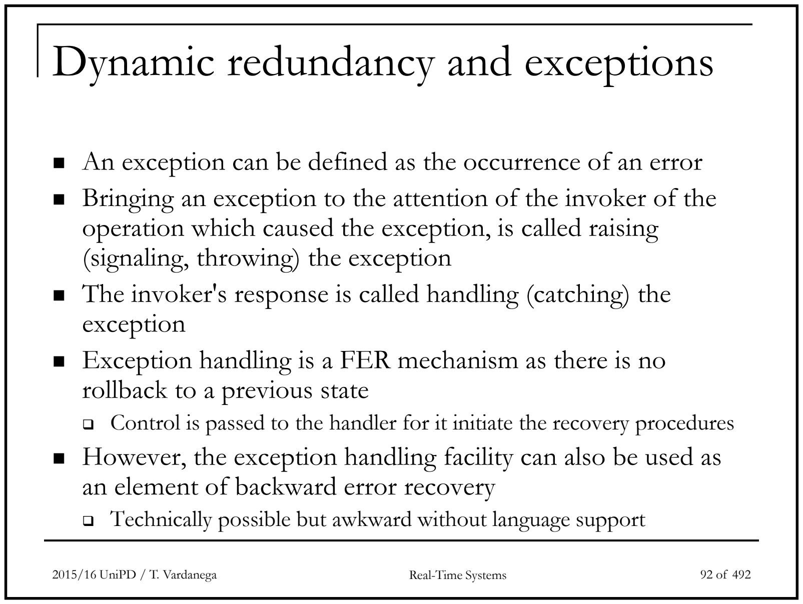
- The acceptance test provides the *error detection* mechanism which enables the redundancy in the system to be exploited
- The design of the acceptance test is crucial to the efficacy of the Recovery Block scheme
- There is a trade-off between providing comprehensive acceptance tests and keeping overhead 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 tests
- However, care must be taken as a faulty acceptance test may lead to residual errors going undetected



NVP vs. RB

- Type of redundancy
  - NVP is static, RB is dynamic (in time)
- Design overheads
  - Both require alternative algorithms
    - NVP requires driver, RB requires acceptance test
- Run-time overheads
  - NVP requires $\times N$ resources
  - RB requires establishing recovery points
- Diversity of design
  - Both are susceptible to errors in requirements
- Error detection
  - Vote comparison (NVP) vs. acceptance test (RB)
- Atomicity
  - NVP vote before it outputs to the environment
  - RB must be structured to only output after passing an acceptance test

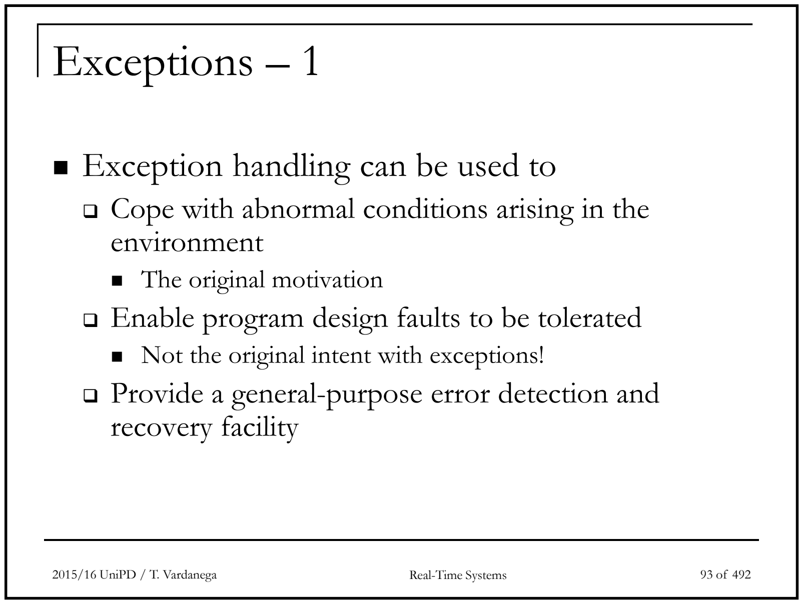
2015/16 UniPD / T. Vardanega Real-Time Systems 91 of 492



Dynamic redundancy and exceptions

- An exception can be defined as the occurrence of an error
- Bringing an exception to the attention of the invoker of the operation which caused the exception, is called raising (signaling, throwing) the exception
- The invoker's response is called handling (catching) the exception
- Exception handling is a FER mechanism as there is no rollback to a previous state
  - Control is passed to the handler for it initiate the recovery procedures
- However, the exception handling facility can also be used as an element of backward error recovery
  - Technically possible but awkward without language support

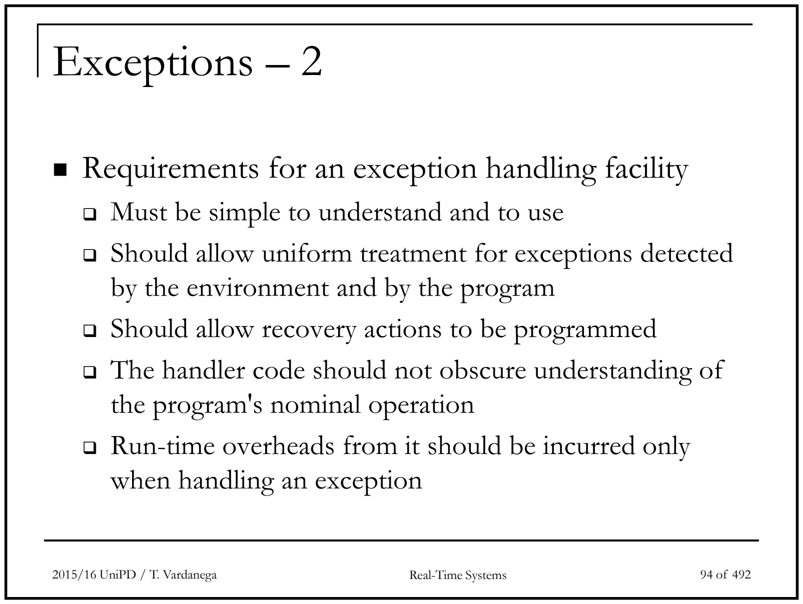
2015/16 UniPD / T. Vardanega Real-Time Systems 92 of 492



Exceptions – 1

- Exception handling can be used to
  - Cope with abnormal conditions arising in the environment
    - The original motivation
  - Enable program design faults to be tolerated
    - Not the original intent with exceptions!
  - Provide a general-purpose error detection and recovery facility

2015/16 UniPD / T. Vardanega Real-Time Systems 93 of 492



Exceptions – 2

- Requirements for an exception handling facility
  - Must be simple to understand and to use
  - Should allow uniform treatment for exceptions detected by the environment and by the program
  - Should allow recovery actions to be programmed
  - The handler code should not obscure understanding of the program's nominal operation
  - Run-time overheads from it should be incurred only when handling an exception

2015/16 UniPD / T. Vardanega Real-Time Systems 94 of 492

Exceptions – 3

- Two sources of detection
  - Environmental detection
  - Application error detection
- 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 (and need a callback mechanism)
  - Often called asynchronous notifications or signals

Exceptions – 4

- Detected by the environment and raised *synchronously*
  - E.g. array bounds *error* or divide-by-zero
- Detected by the application and raised synchronously
  - E.g. the *failure* of a program-defined assertion check
- Detected by the environment and raised *asynchronously*
  - E.g. an exception raised due to the failure of some health monitoring mechanism
- Detected by the application and raised *asynchronously*
  - E.g. one process may recognise that an error condition has occurred which can effect another process
    - Causing it to miss its deadline or to not terminate correctly

Exceptions – 5

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

Exceptions – 6

- If no handler is associated with the block or procedure
  - May regard it as a programmer error to be reported at compile time
  - An exception raised in a procedure and not handled in it can only be handled within the context the procedure was called from
    - E.g., an exception raised in a procedure as a result of a failed assertion involving the parameters
- CHILL requires that a procedure specifies which exceptions it may raise (that it does not handle locally)
  - The compiler can then check the calling context for the presence of an appropriate handler
- Java allows a function to define which exceptions it can raise
  - However, unlike CHILL, it does not require a handler to be available in the calling context

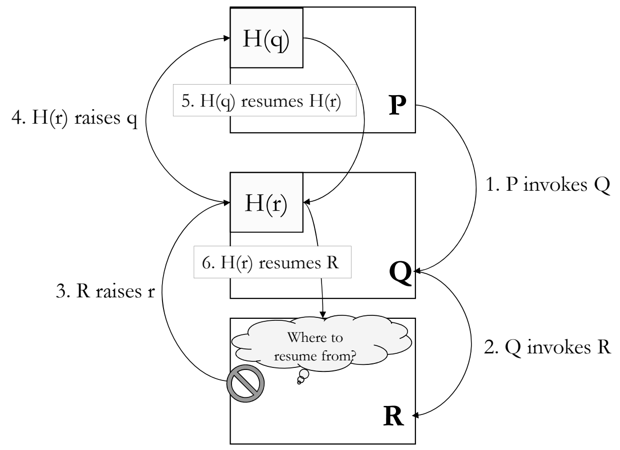
Exceptions – 7

- Otherwise look for handlers up the chain of invokers
  - This is called propagating the exception
  - The Ada and Java approach
- A problem occurs where exceptions have scope
  - An exception may thus be propagated outside its scope
  - This makes it impossible for a handler to be found
- Most languages provide a catch-all exception handler
- An unhandled exception causes a sequential program to be aborted
- If the program contains more than one process (thread) and a particular process does not handle an exception it has raised, then usually that process (thread) is aborted
  - However, it is not clear whether the exception should be propagated to the parent process

Exceptions – 8

- Should the invoker of the exception continue its execution after the exception has been handled?
- If the invoker can continue then it may be possible for the handler to cure the problem that caused the exception to be raised and for the invoker to resume as if nothing had happened
  - This is referred to as the ***resumption*** or notify model
- Instead the model where control is not returned to the invoker is called ***termination*** or escape
- Clearly it is possible to have a model in which the handler can decide whether to resume the operation which caused the exception, or to terminate the operation
  - This is called the hybrid model

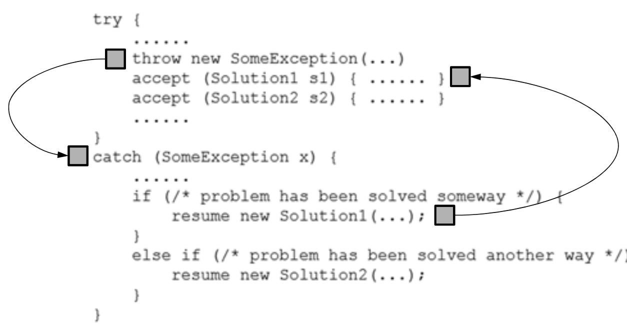
The resumption model – 1



The resumption model – 2

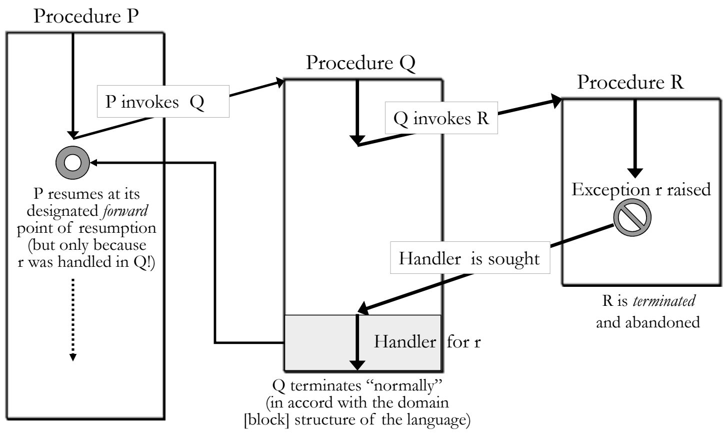
- Repairing errors raised by the run-time system is difficult
  - E.g., an arithmetic overflow in a complex expressions results in registers containing partial evaluations: calling the handler overwrites these registers
    - Some languages from the late '70 (Pearl, Mesa) support the resumption <u>and</u> termination models – **Ada and Java support the termination model**
- Implementing a strict resumption model is difficult
  - A compromise solution is to re-execute the block associated with the exception handler: that's what Eiffel does
  - In that case the local variables of the block must not be re-initialised on a retry (needs a form of non-reentrancy)
- The resumption model is useful with asynchronous exceptions when current execution is ≠ from the exception context

The resumption model – 3

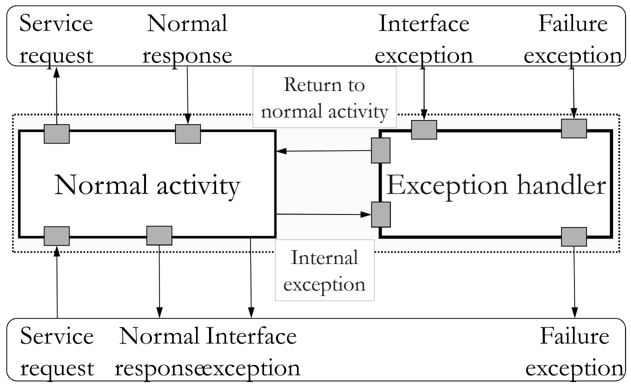


```
try {
    ......
    throw new SomeException(...)
    accept (Solution1 s1) { ...... }
    accept (Solution2 s2) { ...... }
    ......
}
catch (SomeException x) {
    ......
    if (/* problem has been solved someway */) {
        resume new Solution1(...);
    }
    else if (/* problem has been solved another way */) {
        resume new Solution2(...);
    }
}
```

The termination model

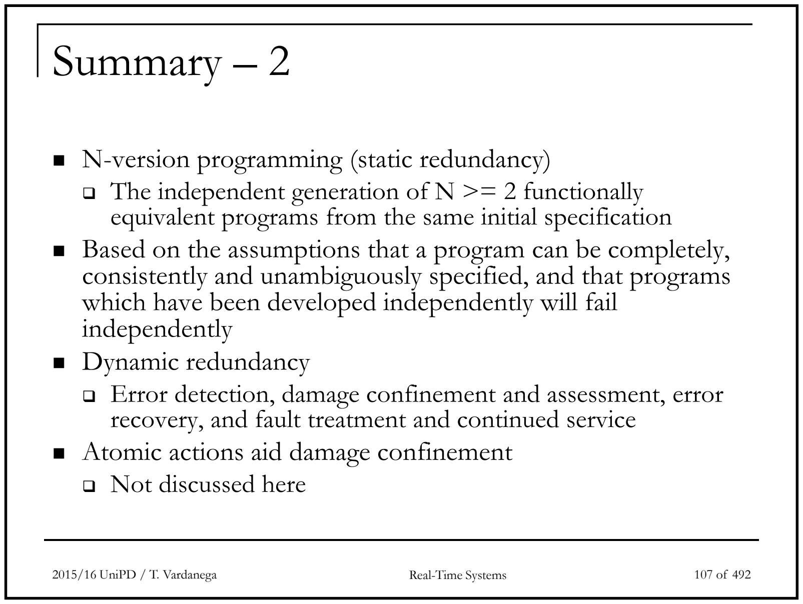

Procedure P
P invokes Q
Procedure Q
Q invokes R
Procedure R
Exception r raised
P resumes at its designated *forward* point of resumption (but only because r was handled in Q!)
Handler is sought
R is *terminated* and abandoned
Handler for r
Q terminates "normally" (in accord with the domain [block] structure of the language)

Ideal fault-tolerant component



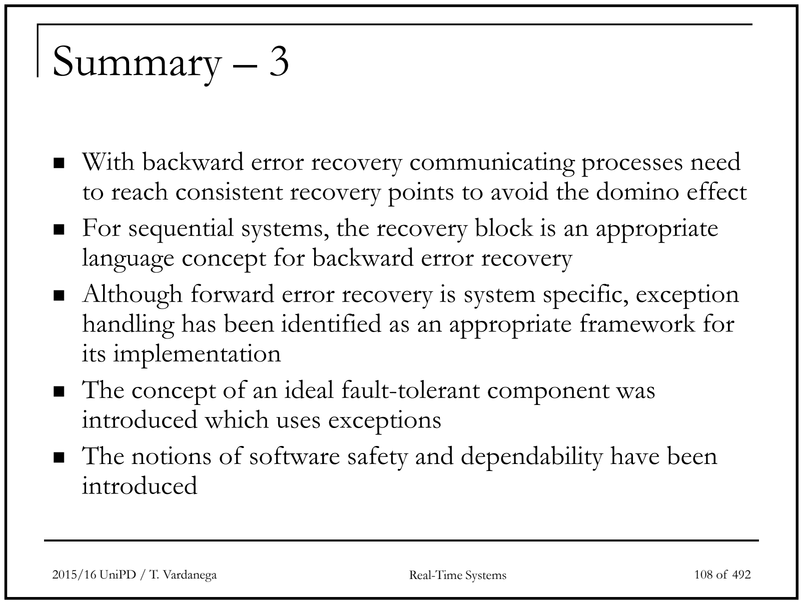
Summary – 1

- Reliability
  - A measure of the success with which the system conforms to some authoritative specification of its behavior
- When the behavior of a system deviates from that which is specified for it, this is called a failure
- Failures result from faults
- Faults can be accidentally or intentionally introduced into a system
- They 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



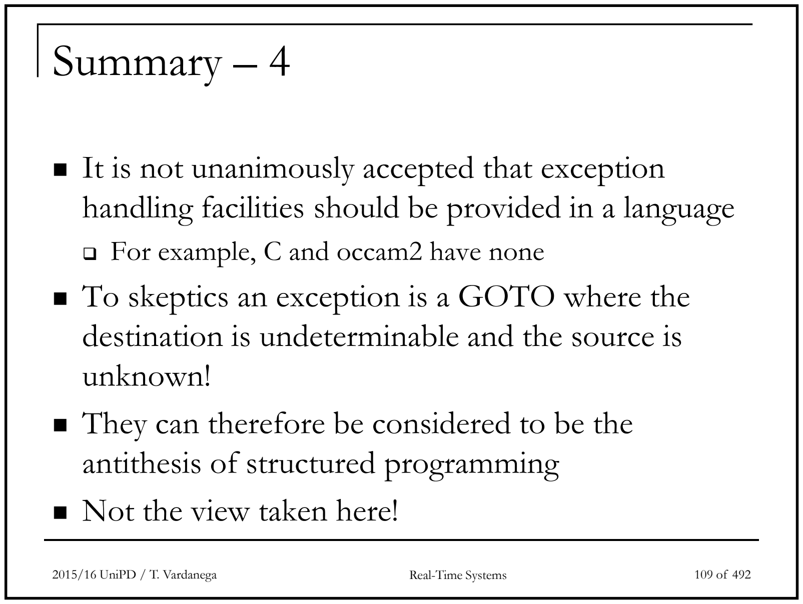
Summary – 2

- N-version programming (static redundancy)
  - The independent generation of N >= 2 functionally equivalent programs from the same initial specification
- Based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently
- Dynamic redundancy
  - Error detection, damage confinement and assessment, error recovery, and fault treatment and continued service
- Atomic actions aid damage confinement
  - Not discussed here



Summary – 3

- With backward error recovery communicating processes need to reach consistent recovery points to avoid the domino effect
- For sequential systems, the recovery block is an appropriate language concept for backward error recovery
- 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 uses exceptions
- The notions of software safety and dependability have been introduced



Summary – 4

- It is not unanimously accepted that exception handling facilities should be provided in a language
  - For example, C and occam2 have none
- To skeptics 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
- Not the view taken here!