Real-Time Systems Programming

Summer-Semester 2002
Lecture 14
31 May 2002

Fault Removal and Fault T

Overview

1) Fault removal
2) Fault Tolerance
3) HW fault tolerance
4) SW fault tolerance

Where are we?

1) ***Fault removal***
   - ***Testing – and its limits!***
2) Fault Tolerance
3) HW fault tolerance
4) SW fault tolerance

Fault Removal

- Not an *alternative* to fault avoidance, but an *augmentation*
- ***Fault removal***: procedures for finding and removing the causes of errors
  - Automated SW analyses – as a pre-compile pass
  - Semantic style checkers
    - E.g., the automated detection of non-deterministic functional models, race conditions, etc.
  - Design reviews
  - Program verification
  - Code inspections
  - The classic: system testing, testing, ... *and more testing*

A Precaution on System Testing

- System testing can never be exhaustive and remove all potential faults
- A test normally just shows 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
  - Difficult to guarantee that the simulation is accurate

Failure of Fault Prevention Approach

- In spite of all the testing and verification techniques, hardware components will fail
- Fault prevention approach unsuccessful when
  - Frequency or duration of repair times are unacceptable, or
  - System inaccessible for maintenance and repair activities (e.g., the crewless spacecraft Voyager)
- A therefore sometimes required augmentation (*not an alternative!*) to fault prevention is ***fault tolerance***

Where are we?

1) Fault removal
2) ***Fault Tolerance***
   - ***Levels of fault tolerance***
   - ***Redundancy***
   - ***Voting – and the consistent comparison problem***
3) HW fault tolerance
4) SW fault tolerance

Levels of Fault Tolerance

- ***Full fault tolerance***
  - System continues to operate in the presence of faults
  - For a limited period
  - No significant loss of functionality or performance
- ***Graceful degradation*** (***fail soft***)
  - System continues to operate in the presence of faults
  - Accept partial degradation of functionality or performance during the presence of the fault (until recovery or repair)
- ***Fail safe***
  - System maintains its integrity
  - Accept a temporary halt in systems operation

Fault Tolerance Requirements

- Required fault tolerance level depends on application
- Most safety critical systems *require* full fault tolerance
- In practice, often settle for graceful degradation
- ***Example***: Air Traffic Control (ATC), with the levels:
  1) Full functionality within required response times
  2) Minimum functionality to maintain basic ATC
  3) Emergency functionality to provide separation among aircraft only
  4) Adjacent facility backup, used in event of catastrophic failure (e.g., earthquake)

Redundancy

- All fault-tolerant techniques rely on extra elements introduced into the system to detect & recover from faults
- ***Redundant*** components: not required in perfect system
- Often called ***protective redundancy***
  - ***Space redundancy***
  - ***Time redundancy***
  - ***Value redundancy***
- ***Aim***:
  - Minimise redundancy while maximising reliability
  - Subject to cost and size constraints of the system

Redundancy – A WARNING

- Added components increase the ***complexity*** of the overall system
  - This itself can lead to *less* reliable systems
- Inherent problems with redundancy
  - Redundancy aspects of a design are rarely exercised
  - Redundancy typically comes in when a system is already in a stress situation
  - Difficult to test – can often be simulted at best
- Redundancy can easily **lower** reliability !
  - ***Example***: first launch of the space shuttle
- ***Advice:***
  - Separate fault-tolerant components from rest of system

Voting

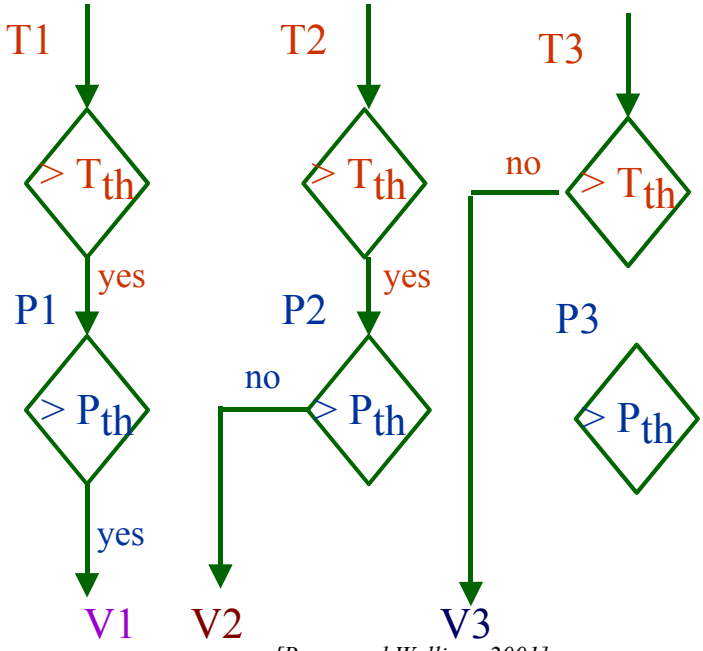
- ***1 out of 2 voting:***
  - Handles single silent failure
- ***2 out of 3 voting:***
  - Handles single consistent failure
- ***In general: m* out of *n***
  - Given *n* results, *m* of them have to agree to be accepted

Exact vs. Inexact Voting

- ***Exact voting***:
  - Bit-by-bit comparison of the results
  - Can be used with exact results: Bools, strings, ints
- ***Inexact voting***:
  - Results are considered identical if they are within an (application specific) interval
  - Required with inexact results: Floats, analog values
- *Most embedded RT systems have to use inexact voting!*

Consistent Comparison Problem

- Inexact voting does not help if internal control flow of redundant copies internally depends on inexact values
- ***Example***: process control system that bases decisions on temperatures and pressures
- All redundant controllers have valid data, just below or above the thresholds



[Burns and Wellings 2001]

Where are we?

1) Fault removal
2) Fault Tolerance
3) ***HW fault tolerance***
4) SW fault tolerance

Hardware Fault Tolerance

- ***Static redundancy***
  - Redundancy inside a system to hide effects of faults
  - ***Example***: Redundant flight control unit
- ***Dynamic redundancy***
  - Redundancy supplied inside a component which indicates that the output is in error
  - Provides an ***error detection*** facility
  - Recovery must often be provided by another component
  - ***Examples***:
    - Parity bits
    - Communications checksums (Cyclic Redundancy Check – CRC) – may be *error-correcting* or not

HW Fault Tolerance – A WARNING

- HW fault tolerance:
  - Classical means employed by designers of safety critical applications
- The (*often implicit*) assumption:
  - Faults are not common to redundant components (***common mode faults***)
  - Faults are either transient or due to component deterioration
- However, with increased design complexity, hw faults are becoming more likely to be *design faults*
- *Design faults are not eliminated by replication !*

Degrees of Redundancy

- The required degree of redundancy depends on
  - the type of fault to be tolerated
  - the number of faults to be tolerated
- Let n be the degree of redundancy; to tolerate k faults, we need
  - $n = k + 1$ for **fail-silent** faults
  - $n = 2k + 1$ for **fail-consistent** faults
  - $n = 3k + 1$ for **malicious** (**Byzantine**) faults
- Typical: $n = 3$, ***Triple Mode Redundancy*** (***TMR***)
- Use of ***voting*** to derive end result

Where are we?

1) Fault removal
2) Fault Tolerance
3) HW fault tolerance
4) ***SW fault tolerance***
   - ***N-Version programming***
   - ***SW dynamic redundancy***
   - ***Forward and backward error recovery***
   - ***Recovery blocks***

Software Fault Tolerance

- Used for detecting design faults
- *Static SW fault tolerance:*
  - N-Version programming
- *Dynamic SW fault tolerance:*
  - Detection and Recovery
  - *Recovery blocks:* backward error recovery
  - *Exceptions:* forward error recovery

N-Version Programming

> ***"When the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways [...] If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the acuracy of them all."***
>
> Babbage, 1837

- Independent generation of N (N > 2) functionally equivalent programs
  - Use same initial specification
  - No interactions between groups
- The programs execute concurrently
  - Use same inputs
  - Compare results by a driver process
- A voting process produces the final result

Assumptions of N-Version Programming

- A program can be specified in a manner that is
  - complete
  - consistent
  - unambiguous
- Faults in different versions are not correlated
- Both assumptions are not inherently valid !

N-version programming depends on ...

- Quality of the initial *specification*
  - Software faults are likely to stem from an inadequate specification
  - A specification error will manifest itself in all N versions of the implementation
- Level of *abstraction* of the initial specification
  - *Too high:* unambiguity may result
  - *Too low:* design is too constrained
- The *budget*
  - Can we afford a 3x increase in sw development costs?
  - 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?

Software Dynamic Redundancy

Four phases:

- ***Error detection***
- ***Damage confinement and assessment***
  - To what extent has the system been corrupted?
  - The delay between a fault occurring and the detection of the error could allow fault propagation
- ***Error recovery***
  - Transformation of 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

- ***Environmental detection***
  - Hardware: e.g. illegal instruction
  - OS/RTS: null pointer, array index out of bounds
- ***Application detection***
  - Replication checks
  - Timing checks
  - Reversal checks
    - If i/o-function is isomorphic
  - Coding checks
    - E.g. checksums
  - Reasonableness checks
    - Structure
    - Values – static (absolute) or dynamic (relative)

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 minimise the damage caused by a faulty component (also known as ***firewalling***)
- ***Modular decomposition***
  - Provides ***static*** damage confinement
  - Allows data to flow through well-define pathways
- ***Atomic actions***
  - Provides ***dynamic*** damage confinement
  - Move the system from one consistent state to another
- Other protection mechanisms – e.g., access permissions

Error Recovery

- Probably the most important phase of any fault-tolerance technique
- ***Forward error recovery (FER)***
  - Continues from an erroneous state by making selective corrections to the system state
- ***Backward error recovery (BER)***
  - Restores system to a safe state previous to the one where the error occurred

Forward Error Recovery

- FER includes making safe the controlled environment which may be hazardous or damaged because of the failure
- Is **system specific** and depends on accurate predictions of the location and cause of errors (i.e, damage assessment)
- ***Examples***:
  - Redundant pointers in data structures
  - Use of self-correcting codes such as Hamming Codes
- There may be a trade off between error detection and error correction capabilities
  - ***Example***: CRCs

Backward Error Recovery

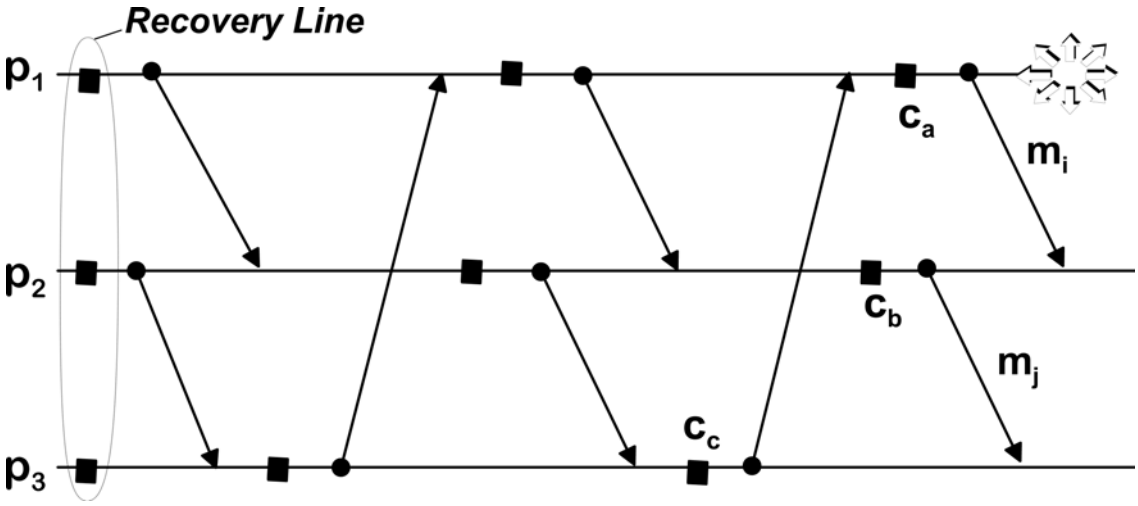
- BER restores the system to a previous safe state
- ***Static BER***
  - Executes the same program again
  - Handles transient faults
  - Not application specific
- ***Dynamic BER***
  - Executes an alternative section of the program
  - Handle permanent faults
  - This has the same functionality but uses a different algorithm and therefore (*hopefully*) no fault
  - Compare with *N*-Version Programming – also regarding the limitations of this approach

Backward Error Recovery

- ***Recovery point***: Point to which a process is restored
- ***Checkpointing***: Establishing a recovery point
- ***Advantage***:
  - Erroneous state is cleared and it does not rely on finding the location or cause of the fault
- BER can, therefore, be used to recover from unanticipated faults including design errors, if an alternative code is executed
- ***Disadvantage***:
  - Cannot undo errors in the environment!
- ***Incremental checkpointing*** can reduce overhead

The Domino Effect

- With concurrent processes that interact with each other, BER is more complex
- Recovery points insufficient – must establish ***recovery lines***



[Veríssimo and Rodrigues 2001]

Fault Treatment and Continued Service

- Error recovery returned the system to an error-free state; however, the error may recur
- The final phase of F.T. 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 recurring faults

Fault Treatment and Continued Service

- Fault treatment can be divided into 2 stages:
  - ***Fault location***
  - ***System repair***
- Error detection techniques can help to trace the fault to a component.
- *Hardware:*
  - Component replacement
- Software fault:
  - Corrected version of the code
- In non-stop applications it will be necessary to modify the program while it is executing!

The Recovery Block Approach to FT

- Recovery blocks are a language support for BER
- Block entrance: ***automatic recovery point***
- Block exit: an ***acceptance test***
  - Tests that the system is in an acceptable state after the block's execution (***primary module***)

Possible Recovery Block Syntax

```
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:
  - Restore outer level recovery point
  - Execute alternative module

Example:Differential Equation Solver

```
ensure Rounding_err_has_acceptable_tolerance
by
  Explicit Kutta Method
else by
  Implicit Kutta Method
else error
```

- Explicit Kutta Method *fast*
  - ... but inaccurate when equations are stiff
- Implicit Kutta Method more expensive
  - ... but can deal with stiff equations
- The above will cope with all equations
- It will also potentially tolerate design errors in the Explicit Kutta Method if the acceptance test is detailed enough

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 RB 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

N-Version Programming vs Recovery Blocks

- **Static** (NV) versus **dynamic** redundancy (RB)
- ***Design overheads***
  - NV requires alternative algorithms, may use alternatives for RB as well
  - NV requires voter, RB requires acceptance test
- ***Runtime overheads***
  - NV requires *N* x resources
  - RB requires establishing recovery points

N-Version Programming vs Recovery Blocks

- ***Diversity of design***
  - Both susceptible to errors in requirements
- ***Error detection***
  - Vote comparison (NV) versus acceptance test (RB)
  - RB can eliminate problems with inexact voting or inconsistent comparisons
- ***Atomicity***
  - NV vote before it outputs to the environment
  - RB must be structures to only output following the passing of an acceptance test
- ***Note***:
  - Can also use NV and RB complementarily

Summary I

- *Fault prevention* consists of *fault avoidance* and *fault removal*
- *Fault tolerance* enables system to continue functioning even in the presence of faults
- *System testing* is the (more or less) systematic attempt to find faults in a system by observing its behavior in various scenarios
  - ***Caution:*** Testing can only prove the *presence* of faults, *not their absence!*

Summary II

- *Redundancy* adds components beyond mere functionality
  - ***Caution 1:*** Redundancy adds complexity to a design
  - ***Caution 2:*** *Replicating* design faults *does not eliminate* them!
- *Voting* is the process to merge the outputs of redundant components
- Depending on the types of values voted on, voting may be *exact* or *inexact*
  - ***Caution:*** Inexact voting does not help if internal control flow depends on inexact inputs
  - The *consistent comparison problem* may introduce additional complexity

Summary III

- *N-version programming:* independent generation of functionally equivalent programs from the same initial specification
- Assumes that a program can be completely, consistently and unambiguously *specified*, and that programs which have been developed independently will *fail independently*
- Design diversity does not eliminate specification faults !
- *Dynamic redundancy:* error detection, damage confinement and assessment, error recovery, and fault treatment and continued service

Summary IV

- 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

To Go Further

- ***Dependability:***
  - ☞ [Kopetz 1997], Chapter 6
  - ☞ [Burns and Wellings 2001], Chapter 5

Problem Set 7 – Due: (Mon) 10 June 2002

Optimize *the robot you built last week, according to the following design objectives:*

1. *T* becomes minimal
2. The minimal thickness L_{min} of the crossed line for safe detection becomes minimal
3. The braking distance *B* – measured from the end of the crossed line to the center of the light sensor – becomes minimal
4. The measurement error of *D* becomes minimal
5. The measurement error of *L* becomes minimal

a) *Documentation* (overview of approach and assumptions, commented source code) **(3 pts)**

b) Functional *robot* **(2 pts)**

c) *Discussion*, for each of the 5 design objectives: **(5x2 Pts)**

- How well (quantitatively) your design fulfills the objectives
- What are the influencing factors for the fulfillment of the objectives
- What tradeoffs there are with the other design objectives

Enjoy!