Issues in Programming Language Design for Embedded RT Systems

- Reliability and Fault Tolerance
- Exceptions and Exception Handling

Rajesh Gupta
University of California, San Diego
ES Characteristics

• Complexity in function (and in size)
  ▶ High reliability and safety
    – failure has severe life, environmental and economic consequences
• Guaranteed response times
  – predictability is important; sometimes more so than efficiency.
  – Facilities to specify time and delays.
• Facilities to interact with special purpose hardware
  – need to program devices in a reliable and abstract way
• Concurrent control of separate components
  – devices operate in parallel
Outline

• Failure, fault, error; Fault types
• How to handle failures?
  – avoidance/prevention, tolerance
• Redundancy in software
  – Static versus Dynamic
• Static Redundancy
  – NVP
• Dynamic Redundancy
  – Four steps
• Recovery Blocks
Reliability & Fault Tolerance

• A failure occurs when the behavior deviates from the one specified.
  – Result of unexpected problems internal to the system that manifest at the interfaces.
  – These problems are called errors and their source as a fault.
    • Failure $\rightarrow$ Fault $\rightarrow$ Error $\rightarrow$ Failure $\rightarrow$ Fault $\rightarrow$ Error $\rightarrow$...
    – Faults can be transient, permanent or intermittent.
• An internal state which is not specified is an error

What are the sources of faults and factors that ensure fault tolerance?
• Failure Sources
  – specification errors
  – (software) implementation errors
  – hardware failure -- processor,
  – hardware failure -- communication interference.
Fault Types

• A transient fault shows up for a limited duration
  – e.g., hardware components that have adverse reaction to an environment (e.g., radioactivity)
  – many faults in the communication systems are transient
• A permanent fault needs to be repaired before removed
  – a broken wire, software design error
• Intermittent faults are transient faults that reoccur over time
  – e.g., a heat-sensitive hardware component.
Failure Modes

Failure Mode

Value Domain
- Constraint Error
- Value Error
  
Timing Domain
- Early
- Omission
- Late

Arbitrary (uncontrolled failure)

Fail silent
Fail stop
Fail controlled

Easy to recognize

System produces correct value both in time and value domains, until it fails. The only failure possible is an omission failure.

Same as fail silent but allows other systems to detect that it has entered into the fail-silent state.
Two way to handle failures

- Fault prevention - avoid possibility of faults
  - Two stages:
    - fault avoidance: limit the introduction of faults during system construction (reliable components, composition methods, requirements specification, CASE environments)
    - fault removal: procedures for finding and removing causes of errors (reviews, verification, testing)
  - neither stages are completely error free, and may not always be possible.
  - So, we needs methods to enable tolerance to faults
- Fault tolerance - continue functioning
  - Full fault tolerance -- full function (even if for limited time)
  - Graceful degradation or fail soft -- partial degradation in functionality
  - Fail safe -- maintain integrity while accepting temporary halt in operations
Example: ATC Operation w/ 3 levels of graceful degradation

- Full functionality within required response time
- Minimum functionality required to maintain basic air traffic control
- Emergency functionality to ensure separation between aircrafts
- Adjacent facility backup used in the advent of a catastrophic failure

A system may provide several degrees of graceful degradation, to ensure high availability.

On a A310, slat and flap control computers, on error in landing, restore the system to a safe state – symmetric setting – and then shutdown.
Redundancy

• Fault-tolerance often achieved using extra components
  – to detect and recover from faults
  – protective redundancy
  – added components increase system complexity
    • E.g., additional synchronization requirements
  – goal is to minimize redundancy while maximizing fault tolerance

• Static or dynamic redundancy in hardware
  – static: using redundant components (e.g., triple modular redundancy or TMR with majority voting)
    • useful in case of transient error or due to degradation
  – dynamic: redundancy within a component with external recovery (error detection and correction), e.g., checksums

• Software Fault Tolerance
  – static: N-version programming
  – dynamic: detection and (forw or backw) error recovery
N-Version Programming

- Built off TMR/NMR in hardware
- Consists of *independent* generation of N (>2) functionally equivalent programs from same initial specifications
  - Design Diversity, Different Programming Language, Methods..
- Programs execute concurrently, results are arrived at by consensus (majority voting).
- Questions
  - How are results compared? How is voting conducted?
- NVP depends upon
  - good initial specification, independence of effort, abundance of effort.
- NVP can be taken further
  - compiling, processing, ...
NVP

- Controlled by a driver process
  - invokes each of the versions
  - waiting for the versions to complete
  - comparing and acting on the results
- Problem: assumes programs run to completion!
  - So the versions must actually interact (with the driver program)
    - Comparison Points: points in the versions when programs must communicate their votes to the driver process
    - Defines granularity of the fault tolerance
  - How the versions communicate and synchronize depend upon the programming language used, its model of concurrency
Interaction of Drivers and Versions

- This interaction is specified as consisting of 3 components
  - **Comparison vectors**
    - Data structures that represent the outputs (or votes) produced by the versions plus any attributes
      - E.g., in an ATC, values to be compared are aircraft positions, an attribute may be the associated radar reading (recent or calculated)
  - **Comparison status indicators**
    - From driver to versions: indicate the actions that each version must perform as a result of driver’s comparison
      - E.g., continuation, termination, continuation after changing a comparison vector (to a majority value)
  - **Comparison points**
    - Define the granularity of fault tolerance

▶ If different programming languages have been used for versions, then a RTOS will typically provide the means of communication and synchronization
Vote Comparison in NVP

- Efficiency of vote comparison is critical

- Complicated by comparison procedure
  - Not all results are single numeric values
  - The “consistent comparison problem”
    - When using “thresholds” for comparison the errors can stack up, resulting different execution paths in all versions.

Two sequential thresholding lead to different execution paths in all three versions.

The problem will reappear even when using inexact comparison (just have to be near a threshold value).

And what happens when there are multiple solutions?
Dynamic fault tolerance in software

• Dynamic redundancy kicks in only when an error is detected.
• Four phases
  1. Error detection:
     fault tolerance techniques effective only when an error is detected
  2. Damage assessment and containment:
     to what extent the “damage” has spread because of the delay between a fault and its manifestation/detection?
  3. Error recovery:
     techniques to reach from a corrupted to a safe state
  4. Fault treatment and continued service:
     error correction.
1 Error Detection

- **Environmental detection**
  - **hardware** (illegal instruction) or **OS/RTS** (null pointer)

- **Application detection**: checks that can be detected by the application
  - **replication** checks (as in NVP)
  - **timing** checks: watchdog timers (which must be reset by sw); deadline misses reported by the environment
  - **reversal** checks: with 1-1 mapping between input and output
  - **coding** checks using redundant data (e.g., checksum)
  - **reasonablesness** checks: on state, on value of an object, checked through assertions, type mechanisms
  - **structural** checks: integrity of data objects (e.g., #elements in a list), redundant pointers, status information
  - **dynamic reasonableness** checks: e.g., correlation between consecutive outputs.
2 Damage Assessment & Containment

• Necessary due to the delay between fault and error
• Goal of containment is to minimize damage caused by a faulty component
  – “firewalling”
• Assessment closely related to containment techniques used
• Two techniques: modular decomposition, atomic actions
• Modular decomposition provides static damage containment
  – allows data to flow through
  – however, the static structure of the software system is lost at run-time
• Atomic actions provide dynamic damage containment
  – an activity is atomic if there are no interactions between the activity and the system for the duration of the activity
  – allow moving system from one consistent state to another.
3 Error Recovery

• **Forward or Backward**
• **Forward**: continues from an erroneous state by making **selective corrections** to the system state
  – includes making safe the controlled environment which may be hazardous or damaged because of failure
  – system specific and depends upon accurate predictions
  – e.g., redundant pointers in data structures, self-correcting codes
• **Backward**: relies on restoring the system to a previous **safe state** and executing an alternative section of the program
  – safe functionality but different algorithm
  – the point to which a process is restored is called a **recovery point** and the act of establishing it is called **checkpointing**.
  – BER can be used to recover from unanticipated faults including design errors.
  – State restoration is not always possible in (real-time) embedded systems.
Domino Effect in Concurrent Processes

- With interacting concurrent processes Backward Error Recovery may get complex

- Clearly, recovery points must be designed so that an error detected in one process will not result in a total rollback of all the interacting processes
  - Recovery lines.
4 Fault Treatment and Continued Service

- Even with recovery, the error may recur. Need to eradicate the fault from the system.
- Automatic treatment of faults is very application specific.
- Make some assumptions. For instance:
  - all faults are transient
- Fault treatment in two stages
  - Fault location
  - System repair
- Fault location
  - use error detection techniques to trace a fault to a component (hardware or software)
  - System repair
    - sometimes it has to be done while the system is in operation.
Recovery Blocks

• A language level support for **backward error recovery**
  – blocks in the normal programming language sense, but
  – at the entrance to the block is an automatic recovery point and
  – at the exit an **acceptance test** to test that the system is in an acceptable state
  – 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
  – repeat this process with alternative modules
  – if all fail, recovery must take place at a higher level

• In terms of four phases of software fault tolerance (Slide #14)
  – Error detection <-> acceptance test
  – Damage assessment <-> not needed due to BER
  – Fault treatment <-> stand-by spare code
Syntax of Recovery Blocks

- 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).
- Several experimental implementations
  - S. Srivastava, “Concurrent Pascal with backward error recovery: Implementation”, Software Practice and Experience

```
ensure <acceptance test>
by
  <primary module>
else by
  <alternative module>
else by
  <alternative module>
...
else by
  <alternative module>
else error
```
Example: Solution to Differential Eqn.

- Explicit Kutta method for fast (but inaccurate solutions when equations are stiff)
- Implicit Kutta method is more expensive but accurate

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

```plaintext
ensure rounding_err_has_acceptable_tolerance
by
  ensure sensible_value
by
  Explicit Kutta Method
else by
  Predictor-Corrector K-step Method
else error
else by
  ensure sensible_value
by
  Implicit Kutta Method
else by
  Variable Order K-Step Method
else error
else error
```
The Acceptance Test

• Provides error detection mechanism which enables the redundancy in the system
  – clearly, it is crucial to the effectiveness of the recovery blocks
• Tradeoff between comprehensive acceptance tests and keeping overhead to a minimum
  – so that the fault-free execution is not affected
• Acceptance test is not necessarily correctness
  – allows graceful degradation
NVP versus RB

• NVP is static where as RB is dynamic redundancy
• Both have design overheads
  – alternative algorithms
  – NVP requires a driver
  – RB requires an acceptance test
• Runtime overheads
  – NV requires more resources
  – RB requires establishing recovery points
• Both susceptible to errors in requirements
• Error detection
  – vote comparison (NVP) versus acceptance test (RB)
• Atomicity requirement
  – NV vote before it outputs to the environment, RB must output only following the passing of the acceptance test.
Exceptions

• An exception as the occurrence of an error
• Exception handling
  – bringing exception to attention and responding to it
  – a forward error recovery mechanism by passing control to the handler
  – exception handing can be used for backward error recovery
    • see Garg/Gupta paper from CASES 2001
Ideal Fault-Tolerant Components

Next Lecture: Exception Handling.