Basics

- Reusable Software Components
  - Encapsulated services, interfaces, contracts
- Safety-Critical
  - Failure may cause DEATH!
  - Medical, aviation, transport, military, etc.
- Real-Time Systems
  - Scheduling, deadlines, time-sensitive

Why reuse Software Components?

- Rapid prototyping
- Outsourcing – use third-party vendors
- Increased reliability – already been tested/used at least once
- Cheaper (all of the above reduce costs)

Increased Reliability

- One reason for reuse
- Existing code has been tested/used before
- Testing is the single most expensive and time-consuming activity during system development
- Assume we can "inherit" reliability from previous uses and eliminate testing
Bad Idea!

- Ariane 5
- Therac 25
- Changes in target environment were misunderstood or ignored
- Assumed reliability was irrelevant in new environment
- Retesting was neglected

Lessons Learned

- Need to retest reused components
- But this is expensive and time-consuming
- Can we do better?
- Framework for identifying:
  - components can be reused immediately
  - complete retesting is necessary
  - some additional testing on parts
- Still need detailed system testing, but can forgo component testing in some areas

Reuse and Exhaustive Testing

- Existing component exhaustively tested for input domain $I_0$ (ex. [5, 100])
- If new environment requires input domain $I_n$ that is contained in $I_0$ => No Testing
  - Ex. $I_n$=[10, 75] has already been tested
- If new input domain exceeds boundaries, need to retest for new input domain only
  - Ex. $I_n$=[25, 250] need to test for [100, 250]

Reuse and Statistical Evidence

- For non-exhaustive testing, we consider statistical reliability and confidence in module
- If the new environment requires a higher proven reliability, we need to re-test
- If it requires a lower reliability, we can reuse without testing
Failure Behavior

- Certain types of failure are improbable in some systems, but likely in others
- Testing can ignore improbable failure modes
- The types of failure differ in sequential programs and multitasking real-time systems

General Failure Modes

- Failure modes are defined by their effects, as perceived by the component user
- Worst Failures: Byzantine Failure
  - Arbitrary failure or two-faced behavior
- As complexity increase, types of failure to consider increase
  - Sequential Systems (Strongest Assumptions)
  - Real-Time Systems
  - Concurrent Real-Time Systems (Weakest Assumptions)

Sequential Systems

- Control Failures
  - Ex. Wrong branch of an if-then-else statement
- Value Failures
  - Ex. Assigning incorrect value to a variable
- Addressing Failures
  - Ex. Assigning value to incorrect variable
- Termination Failures
  - Ex. Infinite loops
- Input Failures
  - Ex. Receiving an erroneous value from a sensor

Variations in Sequential Systems

- In some environments the prevalence of each error mode will vary
- Ex. Using hardware with memory management units (MMU) make addressing failures improbable, so we can skip these tests
- However, moving to hardware without MMU would necessitate testing addressing failures
Real-Time Systems

- Characterized by the required **temporal correctness**
  - Execution times, deadlines, period times, etc.
- Timing Failure
  - Correct result at wrong time
  - Can be caused by changes in hardware (ex. CPU)
- Deadline Requirement
  - Can reuse component if new CPU is faster
- Response Time Requirement
  - Can reuse if new execution time is within tolerance
- Periodicity/Frequency Requirement
  - Periodic controllers run infrequently may become unstable
  - When running at higher frequencies, no need to retest

Concurrent Real-Time Systems

- All the error modes of previous systems plus more because of added synchronization
- Ordering Failures
  - Violations of precedence or mutual exclusion relations
- Synchronization Failures
  - Ordering failures plus deadlocks
- Interleaving Failures
  - Preemption in scheduling may cause unwanted side-effects with non-reentrant code and shared data

Modes of Synchronization

- Off-Line Synchronized: Separation in time
- On-Line Synchronized: Semaphores
- Deadlock failures are impossible in off-line synchronized systems
- If a component is being reused in a system using on-line synchronization, we must retest
- If the component is being reused in an off-line synchronized system (and was synchronized before), we can skip retesting

Summary

- System testing is always required when reusing components
- Some component testing can be ignored if the original testing is thorough enough to ensure confidence in new environment
  - Input domains
  - Reliability
  - Failure modes