Programming for Reliability

Reliability Achievement

- *Fault avoidance*
  - The software is developed in such a way that it does not contain faults
- *Fault detection*
  - The development process is organized so that faults in the software are detected and repaired before delivery to the customer
- *Fault tolerance*
  - The software is designed so that faults in the delivered software do not result in complete system failure

Fault Tolerance: Motivations

- We cannot achieve complete software reliability
- Demonstrating high reliability for safety critical applications is difficult
- How can we ensure an acceptable behavior of the system when failures occur?
- E.g., the computers of an air traffic control systems must be continuously available

Aspects of Fault Tolerance

- *Failure detection*: The system must detect that a particular state combination has resulted or will result in a system failure
- *Damage assessment*: the parts of the system state which have been affected by the failure must be detected
- *Fault recovery*: The system must restore its state to a known "safe" state
- *Fault repair*: This involves modifying the system so that the fault does not recur. For systems that need to be continuously available, replacing the faulty component is more complex.

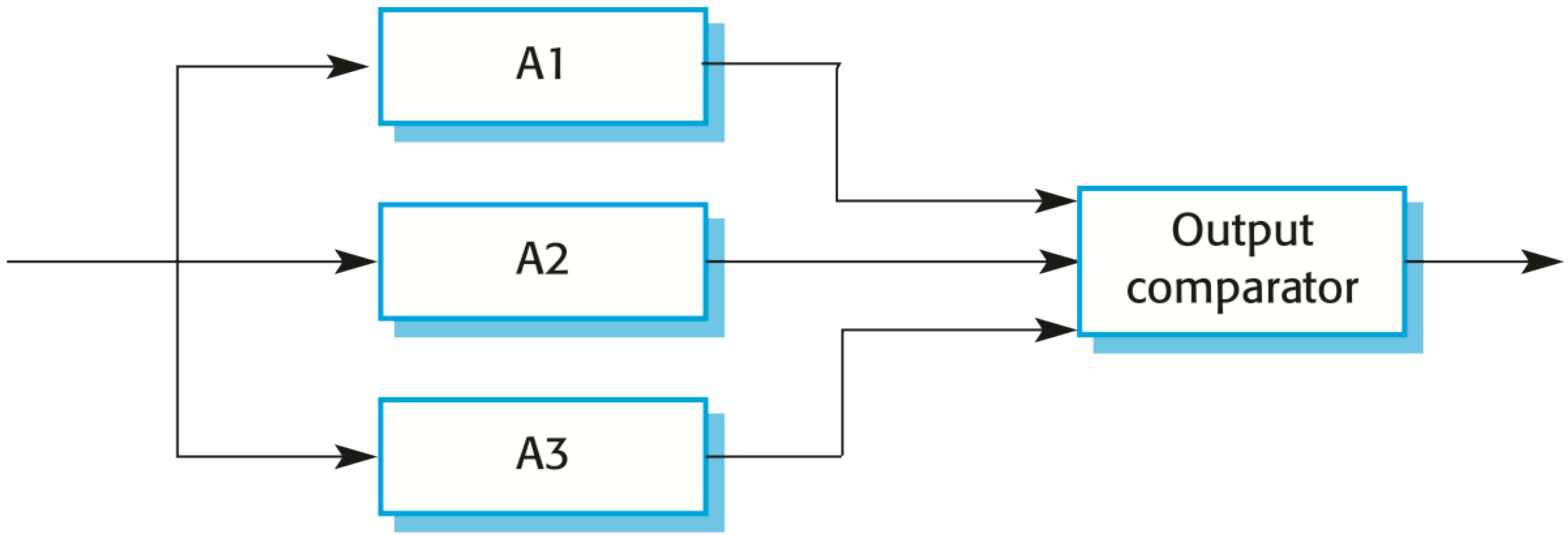
Two Main Approaches

- *Fault-tolerant architectures*: Explicit support for fault tolerance (problem detection, recovery)
- *Defensive Programming*: No specific architecture. But redundant code to check system state after modification. If inconsistencies are detected, state is restored to a known correct state.

Hardware Fault Tolerance

- *Triple-modular Redundancy*: hardware unit is replicated three (or more) times and their outputs are compared
- If one unit shows inconsistent output, it is ignored
- This approach assumes the problem results from component failures rather than design faults
- Low probability of simultaneous component failure in all hardware units
- Units may come from different manufacturers

Hardware Reliability with TMR



Sommerville

Fault Tolerant Software architectures

- The success of TMR at providing fault tolerance is based on two fundamental assumptions
  - The hardware components do not include common design faults
  - Components fail randomly and there is a low probability of simultaneous component failure
- Neither of these assumptions are true for software
  - It isn't possible simply to replicate the same component as they would have common design faults
  - Simultaneous component failure is therefore virtually inevitable
- Software systems must therefore be diverse

Design Diversity

- Different versions of the system are designed and implemented in different ways. They therefore ought to have different failure modes.
- Different approaches to design (e.g., object-oriented and function oriented)
  - Implementation in different programming languages
  - Use of different tools and development environments
  - Use of different algorithms in the implementation

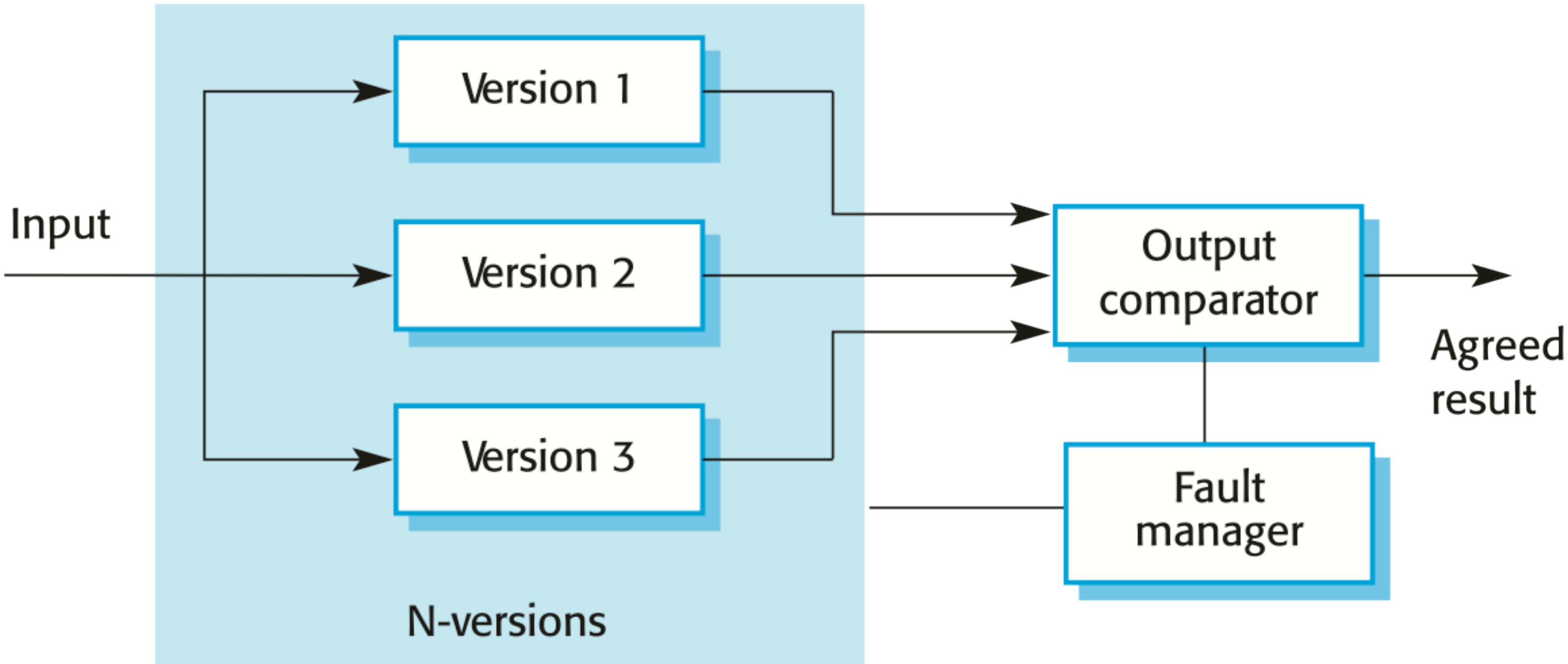
Software Analogies to TMR

- N-version programming
  – The same specification is implemented in a number of different versions by different teams. All versions compute simultaneously and the majority output is selected using a voting system..
  – This is the most commonly used approach e.g. in Airbus 320.
- Recovery blocks
  – A number of **explicitly** different versions of the same specification are written and executed in sequence
  – An acceptance test is used to select the output to be transmitted.

N-version Programming

- Using a common specification, the software system is implemented in a number of *different versions by different teams*
- Versions are executed in parallel
- Outputs are compared using a *voting system* and inconsistent outputs are rejected
- At least three versions should be available
- *Assumption*: it is unlikely different teams will make the same design or programming errors
- However, there is some empirical evidence that teams commonly misinterpret specifications in the same way and use the same / similar algorithms in their systems

N-version Programming

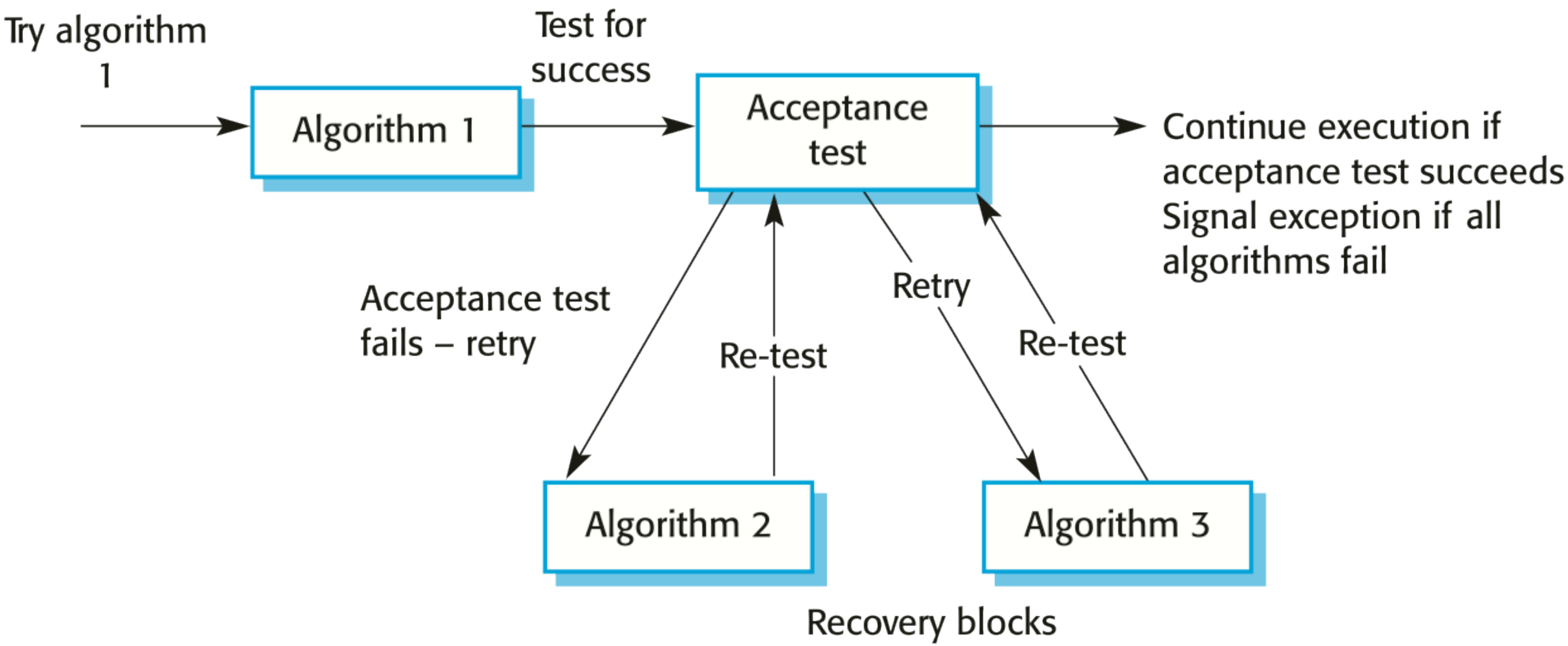


Sommerville

Recovery Blocks

- Finer grain approach to fault tolerance
- Each program component includes a test to check if the component has executed successfully
- It includes alternative code to back-up and repeat the computation with another algorithm (versions) if the test detects a failure
- Versions are executed in sequence.
- The output which conforms to an “acceptance test” is selected.
- Reduce probability of common errors as different algorithms MUST be used for each *recovery block*
- The weakness in this system is writing an appropriate acceptance test.

Recovery Blocks



Sommerville

Discussion

- Different teams can make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place.
- N-version programming gives increased confidence though, but not absolute confidence
- Both presented approaches to fault tolerance assume that the *specifications are correct*
- They both require a *fault-tolerant controller* which will ensure that the steps involved in tolerating faults are executed
- That fault-tolerant controller may fail …

Defensive Programming

- Assume there may be undetected faults and inconsistencies
- Does not require a fault-tolerant controller
- Do not assume correct specifications
- *Redundant code* is incorporated to prevent incorrect state changes and check system state after modification
- If inconsistent, state change is retracted or restored to known state
- One common approach to fault tolerance

Failure Prevention

- One approach is to use *state assertions* to check whether certain constraints are fulfilled
- Logical predicates over the state variables (state invariant in UML terms)
- This predicate is checked before an assignment is made to a state variable
- If an *anomalous value* for the variable would result from the assignment, an error has occurred
- In most programming languages it is up to the programmer to include *explicit assertion checks*
- Can be simplified if all assignments to state variables are always implemented as operations (methods) on objects – the assertion code is part of the operation

Example: Even Number Class

```
class PositiveEvenInteger {
        int val = 0 ;

public void assign (int n) throws NumericException
        {
                if (n < 0 | n%2 == 1)
                        throw new NumericException ();
                else
                        val = n ;
        } // assign

        int toInteger ()
        {
                return val ;
        } //to Integer

        boolean equals (PositiveEvenInteger n)
        {
                return (val == n.val) ;
        } // equals

} //PositiveEven
```

Discussion

- *Failure prevention* avoids the problems related to damage assessment and recovery (next)
- But it involves significant *overhead* (copies of state variables) and for systems where performance is important this may not be applicable
- *Retrospective fault detection* may be a more adequate alternative in some cases: *Damage assessment* and *Recovery*

Damage Assessment

- Analyze system state, after a state change, to judge the *extent of corruption*
- Must assess what parts of the state space have been affected by the failure
- Generally based on 'validity functions' which can be applied to the state elements to assess if their value is within an allowed range
- If damage is identified, an *exception* is signaled and a *repair* mechanism is used to recover from the damage

Java Implementation

- Objects to be checked are instantiations of a class that implements the interface:

```
Interface CheckableObject {
  Public boolean check();
 }
```

- Each class implements its own check method
- When the state as a whole is checked, dynamic binding is used to ensure that the appropriate check function is executed

Example Damage Assessment (java)

```
class RobustArray {
    // Checks that all the objects in an array of objects
    // conform to some defined constraint
    private boolean [] checkState ;
    private CheckableObject [] theRobustArray ;

    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } //RobustArray
    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;

        for (int i= 0; i <this.theRobustArray.length ; i ++)
        {
            if (! theRobustArray [i].check ())
                {
                    checkState [i] = true ;
                    hasBeenDamaged = true ;
                }
            else
                checkState [i] = false ;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException (checkState) ;
    } //assessDamage
} // RobustArray
```

Exception Handling

- *Exception*: User error, hardware failure, software failure
- *Exception handling*: Mechanism by which a system treats an exception
  - User Error: meaningful error message
- In OO systems: Exceptions usually associated with violations of pre-conditions, post-conditions, and/or class invariants
- Using normal control constructs (if statements) to detect exceptions in a sequence of nested procedure calls needs many additional statements to be added to the program and adds a significant timing overhead.
- Some languages have built-in mechanisms for exceptions e.g., Java, C++)

Exception Handlers

- Some programming languages include facilities to detect and handle exceptions (Ada, C++, Java)
- An exception is signaled and control in the program is transferred to an exception handler, I.e., a segment of code that deals with this exceptional situation (e.g., *catch block* in Java)
- Exceptions are often handled by catch block in a calling unit higher up the call sequence, as the units called often do not know what to do when an exception is detected

Java Exception Handling

- Keyword `throw` means raise an exception. It can only be used in a `try` block or a function (indirectly) called from it. Handler is indicated by the keyword `catch`.
- The `try` block wraps the code that may throw an exception and the code that should not execute in this case
- Exceptions are defined as classes so may inherit properties from other exception classes. There is a pre-defined *Exception* class in Java. All exceptions are defined as a subclass of *Exception*
- When possible, exceptions are completely handled in the block where they arise rather than propagated for handling. But this is not often the case

Example: SensorFailureException

```
class SensorFailureException extends Exception {
   SensorFailureException (String msg) {
     super (msg) ;
     Alarm.activate (msg) ;
   }
} // SensorFailureException

class Sensor {
   int readVal () throws SensorFailureException {
   try {
     int theValue = DeviceIO.readInteger () ;
     if (theValue < 0)
        throw new SensorFailureException ("Sensor failure") ;
     return theValue ;
   }
   catch (deviceIOException e)
     { throw new SensorFailureException (" Sensor read error ") ; }
   } // readVal
} // Sensor
```

Another Example

- System that controls a freezer and keeps temperature within a specified range
- Switches a refrigerant pump on and off
- Sets of an alarm is the maximum allowed temperature is exceeded
- Uses external objects of type `Pump`, `TempDial`, `TempSensor`, `Alarm`

Example: FreezerController (Java)

```
class FreezerController  extends Thread {
    Sensor tempSensor = new Sensor () ;
    Dial tempDial = new Dial () ;
    float freezerTemp = tempSensor.readVal () ;
    final float dangerTemp = (float) -18.0 ;
    final long coolingTime = (long) 200000.0 ;
    public void run ( )  throws FreezerTooHotException, InterruptedException {
    try {
            Pump.switchIt (Pump.on) ;
            do { if (freezerTemp > tempDial.setting ())
                            if (Pump.status == Pump.off)
                            {     Pump.switchIt (Pump.on) ;
                                  Thread.sleep (coolingTime) ;
                            }
                    else
                            if (Pump.status == Pump.on)
                                   Pump.switchIt (Pump.off) ;
                    if (freezerTemp > dangerTemp)
                            throw new FreezerTooHotException () ;
                    freezerTemp = tempSensor.readVal () ;
            } while (true) ;
    }  // try block
    catch (FreezerTooHotException f)
    {       Alarm.activate ( ) ;   }
    catch (InterruptedException e)
    {       System.out.println ("Thread exception") ;
            throw  new InterruptedException ( ) ;
    }
    } //run
} // FreezerController
```

Other Damage Assessment Techniques

- *Checksums* are used for damage assessment in data transmission
- *Redundant pointers* can be used to check the integrity of data structures
- *Watch dog timers* can check for non-terminating processes in concurrent systems. If no response after a certain time, a problem is assumed

Fault Recovery

- Forward recovery
  - Apply “repairs” to a corrupted system state
- Backward recovery
  - Restore the system state to a previous, known safe state
- Forward recovery is usually application specific
  - domain knowledge is required to compute possible state corrections
- Backward error recovery is simpler. Details of a safe state are maintained and this replaces the corrupted system state

Forward Recovery

- Corruption of data coding
  - Error coding techniques which add redundancy to coded data can be used for repairing data corrupted during transmission
- Redundant pointers
  - When redundant pointers are included in data structures (e.g. two-way lists), a corrupted list or file store may be rebuilt if a sufficient number of pointers are uncorrupted
  - Often used for database and file system repair
- Sometimes, a simple approach is possible:
  - Reinitialize system, acquire new operating context (e.g., re-reading the sensors), bring to *safe* state

Backward Recovery

- *Transactions* are a frequently used method of backward recovery. Changes are not applied until computation is complete. If an error occurs, the system is left in the state preceding the transaction
- E.g., database systems, changes made during transactions are not immediately incorporated in the database (committed), database updated after transaction is completed
- Periodic *checkpoints* allow system to 'roll-back' to a correct state – restore to a correct state from a *copy*

Example: Safe Sort Procedure

- Sort operation monitors its own execution and assesses if the sort has been correctly executed
- Maintains a *copy* of its input so that if an error occurs, the input is not corrupted
- Based on identifying and handling *exceptions*
- Possible in this case as ‘valid’ sort is known. However, in many cases it is difficult to write *validity checks*

Backward Recovery Code (Java)

```
class SafeSort {
    static void sort ( int [] intarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length];

        // copy the input array

        for (int i = 0; i < intarray.length ; i++)
            copy [i] = intarray [i] ;
        try {
            Sort.bubblesort (intarray, intarray.length, order) ;
            if (order == Sort.ascending)
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i] > intarray [i+1])
                        throw new SortError () ;
            else
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i+1] > intarray [i])
                        throw new SortError () ;
        } // try block
        catch (SortError e )
        {
            for (int i = 0; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("Array not sorted") ;
        } //catch
    } // sort
} // SafeSort
```

Conclusions

- Many programming techniques to make the code more reliable and more robust
- All of these techniques have a cost, in terms of development effort and system performance
- Should be used with discretion
- Some technical issues:
  - backward recovery difficult to implement in concurrent, distributed systems, incompatible with systems that have had real-time deadlines