Dependable software development

- Programming techniques for building dependable software systems.

Software dependability

- In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures
- Some applications, however, have very high dependability requirements and special programming techniques must be used to achieve this

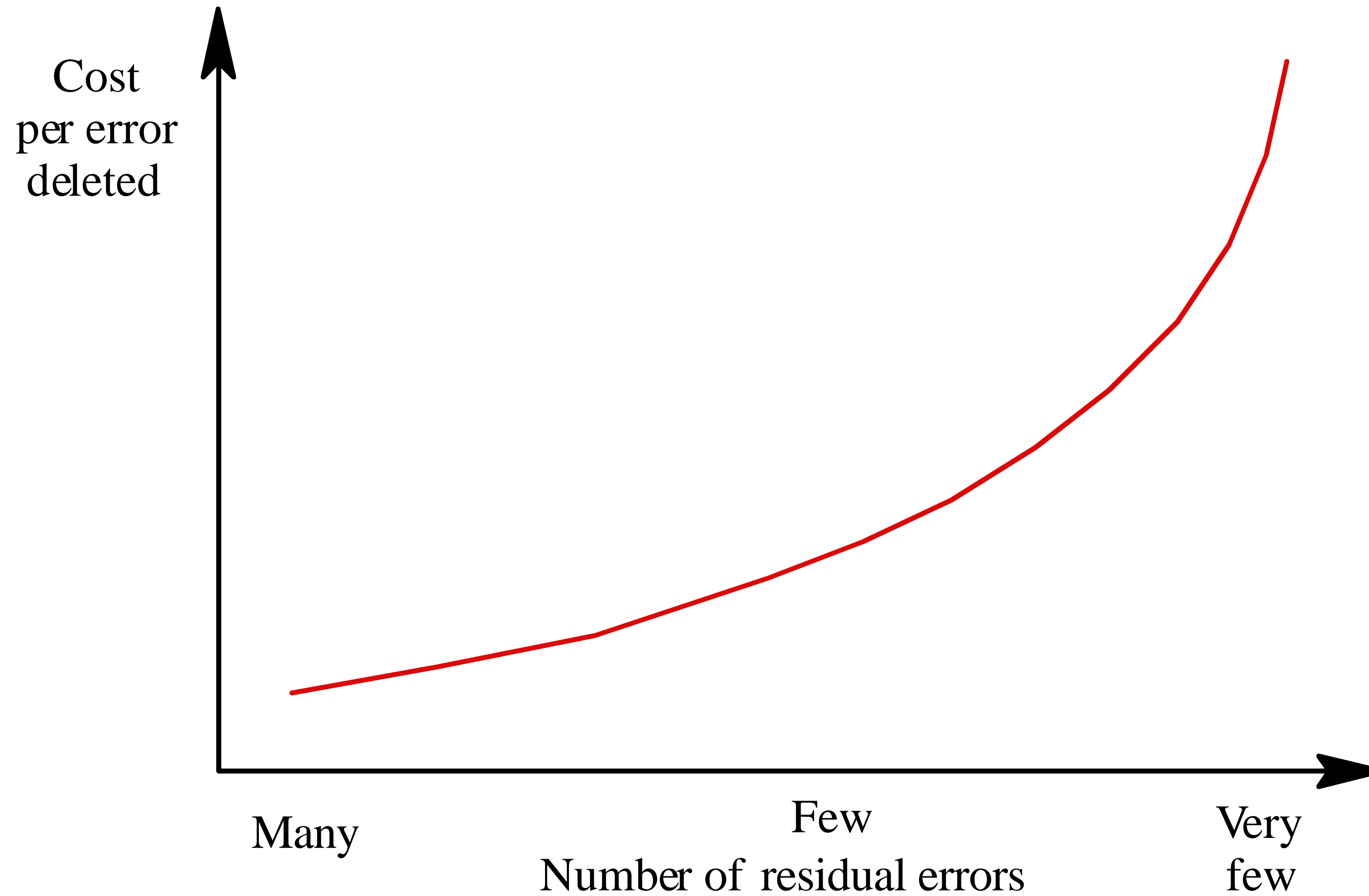
Dependability achievement

- Fault avoidance
  - The software is developed in such a way that human error is avoided and thus system faults are minimised
  - The development process is organised 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 system failure

Fault minimisation

- Current methods of software engineering now allow for the production of fault-free software.
- Fault-free software means software which conforms to its specification. It does NOT mean software which will always perform correctly as there may be specification errors.
- The cost of producing fault free software is very high. It is only cost-effective in exceptional situations. May be cheaper to accept software faults

Fault removal costs



Fault-free software development

- Needs a precise (preferably formal) specification.
- Requires an organizational committment to quality.
- Information hiding and encapsulation in software design is essential
- A programming language with strict typing and run-time checking should be used
- Error-prone constructs should be avoided
- Dependable and repeatable development process

Structured programming

- First discussed in the 1970's
- Programming without gotos
- While loops and if statements as the only control statements.
- Top-down design.
- Important because it promoted thought and discussion about programming
- Leads to programs that are easier to read and understand

Error-prone constructs

- Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons
- Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change
- Dynamic memory allocation
  - Run-time allocation can cause memory overflow
- Parallelism
  - Can result in subtle timing errors because of unforseen interaction between parallel processes

Error-prone constructs

- Recursion
  - Errors in recursion can cause memory overflow
- Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand. they are comparable to goto statements.
- Inheritance
  - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding
- These constructs don’t have to be avoided but they must be used with great care.

Information hiding

- Information should only be exposed to those parts of the program which need to access it. This involves the creation of objects or abstract data types which maintain state and operations on that state
- This avoids faults for three reasons:
  - the probability of accidental corruption of information
  - the information is surrounded by ‘firewalls’ so that problems are less likely to spread to other parts of the program
  - as all information is localised, the programmer is less likely to make errors and reviewers are more likely to find errors

A queue specification in Java

```
interface Queue {

        public void put (Object o) ;
        public void remove (Object o) ;
        public int size () ;

} //Queue
```

Signal declaration in Java

```
class Signal {

    public final int red = 1 ;
    public final int amber = 2 ;
    public final int green = 3 ;

    ... other declarations here ...
}
```

Reliable software processes

- To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process
- A well-defined repeatable process is one that does not depend entirely on individual skills; rather can be enacted by different people
- For fault minimisation, it is clear that the proces activities should include significant verification and validation

Process validation activities

- Requirements inspections
- Requirements management
- Model checking
- Design and code inspection
- Static analysis
- Test planning and management
- Configuration management is also essential

Fault tolerance

- In critical situations, software systems must be fault tolerant. Fault tolerance is required where there are high availability requirements or where system failure costs are very high..
- Fault tolerance means that the system can continue in operation in spite of software failure
- Even if the system seems to be fault-free, it must also be fault tolerant as there may be specification errors or the validation may be incorrect

Fault tolerance actions

- Fault detection
  - The system must detect that a fault (an incorrect system state) has occurred.
- Damage assessment
  - The parts of the system state affected by the fault must be detected.
- Fault recovery
  - The system must restore its state to a known safe state.
- Fault repair
  - The system may be modified to prevent recurrence of the fault. As many software faults are transitory, this is often unnecessary.

Approaches to fault tolerance

- Defensive programming
  - Programmers assume that there may be faults in the code of the system and incorporate redundant code to check the state after modifications to ensure that it is consistent.
  - Fault-tolerant architectures
  - Hardware and software system architectures that support hardware and software redundancy and a fault tolerance controller that detects problems and supports fault recovery
  - These are complementary rather than opposing techniques

Exception management

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs 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.

Exceptions in Java

```
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
```

Programming with exceptions

- Exceptions can be used as a normal programming technique and not just as a way of recovering from faults
- Consider the example of a temperature control system for a refrigeration unit

A temperature controller

- 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 exceptions as a normal programming technique

Freezer controller (Java)

```
class FreezerController  {
        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 InterrupedException {
        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
```

Fault detection

- Languages such as Java and Ada have a strict type system that allows many errors to be trapped at compile-time
- However, some classes of error can only be discovered at run-time
- Fault detection involves detecting an erroneous system state and throwing an exception to manage the detected fault

Fault detection

- Preventative fault detection
  - The fault detetcion mechanism is initiated before the state change is committed. If an erroneous state is detected, the change is not made
- Retrospective fault detection
  - The fault detection mechanism is initiated after the system state has been changed. Used when a incorrect sequence of correct actions leads to an erroneous state ot when preventative fault detection involves too much overhead

Type system extension

- Preventative fault detection really involves extending the type system by including additional constraints as part of the type definition
- These constraints are implemented by defining basic operations within a class definition

```
class PositiveEvenInteger {
        int val = 0 ;

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

        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
```

PositiveEvenInteger

Damage assessment

- Analyse system state to judge the extent of corruption caused by a system failure
- 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

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. If no response after a certain time, a problem is assumed

```
class RobustArray {
	// Checks that all the objects in an array of objects
	// conform to some defined constraint
	boolean [] checkState ;
	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 () ;
	} //assessDamage
} // RobustArray
```

Java class with damage assessment

Fault recovery

- Forward recovery
    - Apply repairs to a corrupted system state
- Backward recovery
    - Restore the system state to a 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 filestore may be rebuilt if a sufficient number of pointers are uncorrupted
  - Often used for database and filesystem repair

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
- Periodic checkpoints allow system to 'roll-back' to a correct state

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

```
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
```

Safe sort procedure (Java)

Key points

- Fault tolerant software can continue in execution in the presence of software faults
- Fault tolerance requires failure detection, damage assessment, recovery and repair
- Defensive programming is an approach to fault tolerance that relies on the inclusion of redundant checks in a program
- Exception handling facilities simplify the process of defensive programming

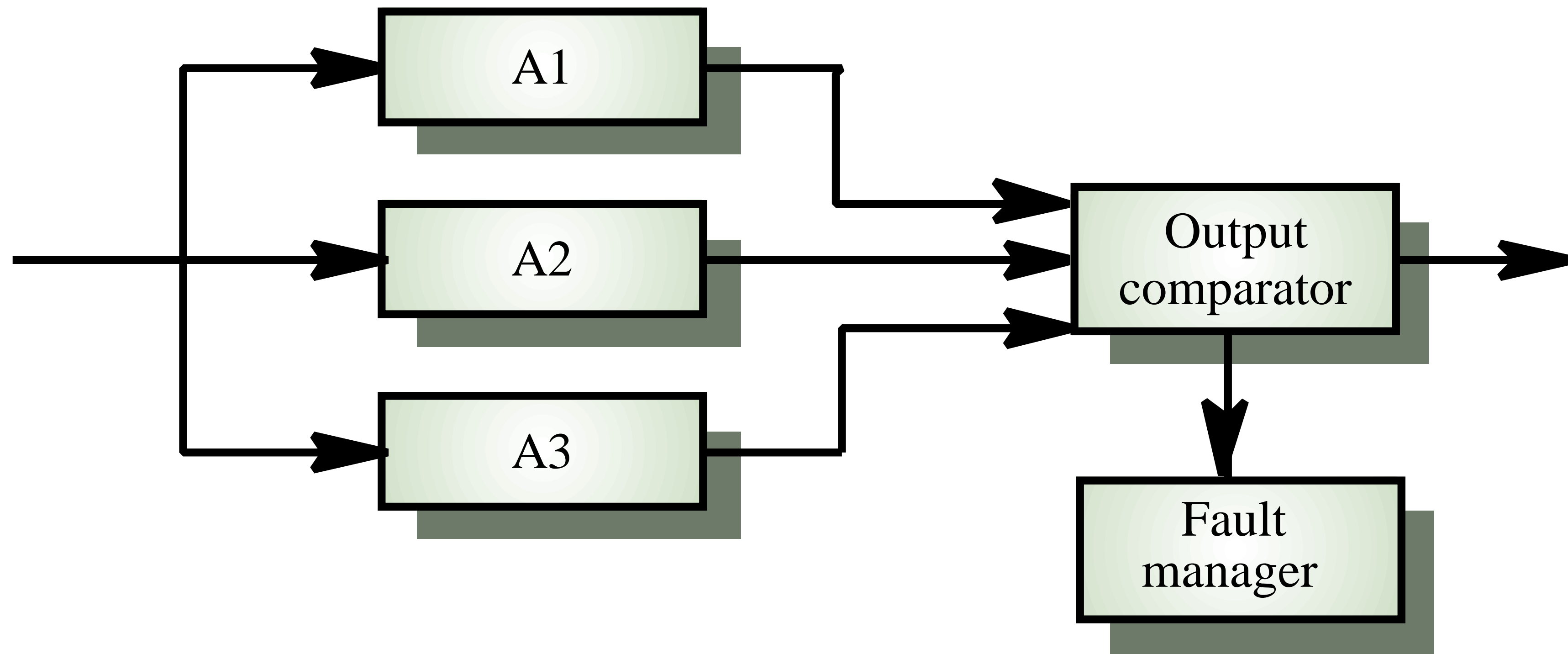
Fault tolerant architecture

- Defensive programming cannot cope with faults that involve interactions between the hardware and the software
- Misunderstandings of the requirements may mean that checks and the associated code are incorrect
- Where systems have high availability requirements, a specific architecture designed to support fault tolerance may be required.
- This must tolerate both hardware and software failure

Hardware fault tolerance

- Depends on triple-modular redundancy (TMR)
- There are three replicated identical components which receive the same input and whose outputs are compared
- If one output is different, it is ignored and component failure is assumed
- Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure

Hardware reliability with TMR



Output selection

- The output comparator is a (relatively) simple hardware unit.
- It compares its input signals and, if one is different from the others, it rejects it. Essentially, selection of the actual output depends on the majority vote.
- The output comparator is connected to a fault management unit that can either try to repair the faulty unit or take it out of service.

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

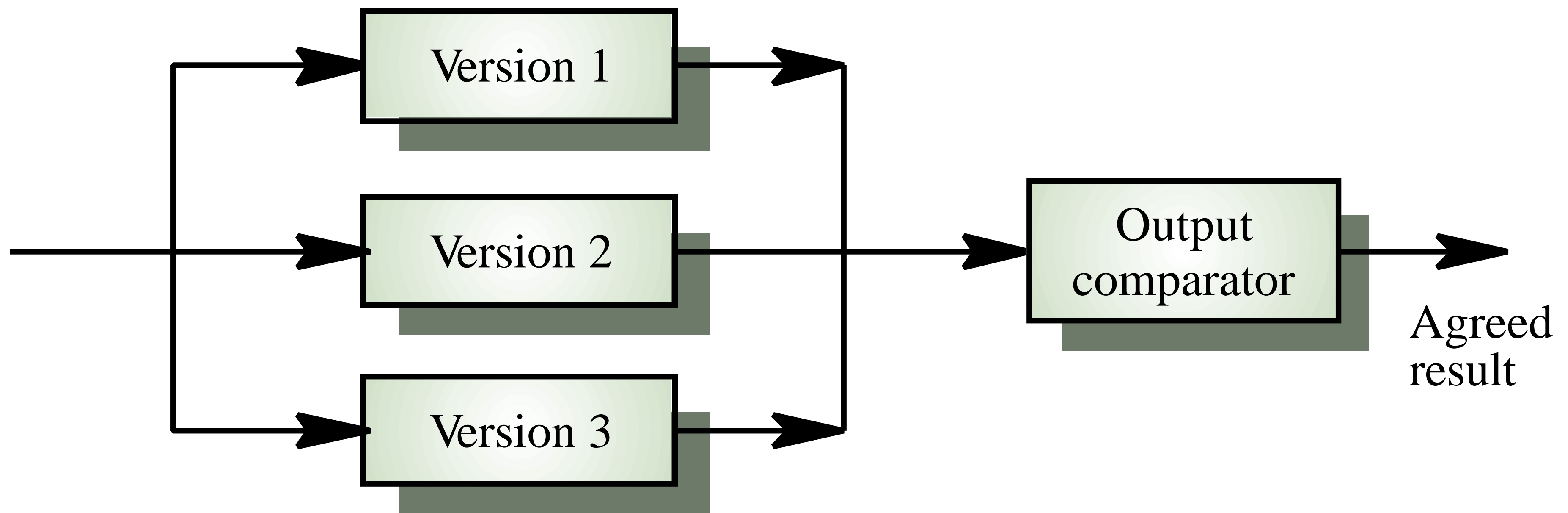
Design diversity

- 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



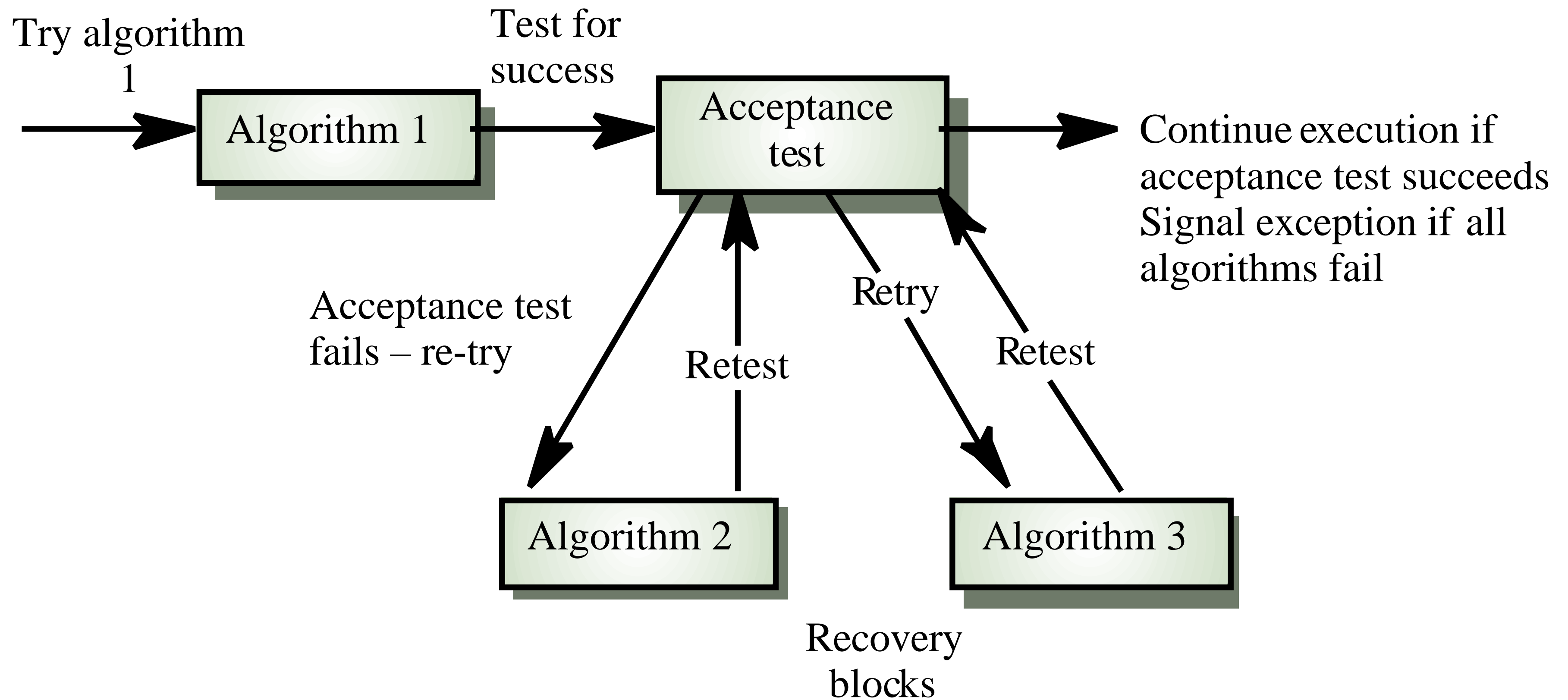
Output comparison

- As in hardware systems, the output comparator is a simple piece of software that uses a voting mechanism to select the output.
- In real-time systems, there may be a requirement that the results from the different versions are all produced within a certain time frame.

N-version programming

- The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.
- There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

Recovery blocks



Recovery blocks

- Force a different algorithm to be used for each version so they reduce the probability of common errors
- However, the design of the acceptance test is difficult as it must be independent of the computation used
- There are problems with this approach for real-time systems because of the sequential operation of the redundant versions

Problems with design diversity

- Teams are not culturally diverse so they tend to tackle problems in the same way
- Characteristic errors
  - Different teams 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.
  - Specification errors
  - If there is an error in the specification then this is reflected in all implementations
  - This can be addressed to some extent by using multiple specification representations

Specification dependency

- Both approaches to software redundancy are susceptible to specification errors. If the specification is incorrect, the system could fail
- This is also a problem with hardware but software specifications are usually more complex than hardware specifications and harder to validate
- This has been addressed in some cases by developing separate software specifications from the same user specification

Is software redundancy needed?

- Unlike hardware, software faults are not an inevitable consequence of the physical world
- Some people therefore believe that a higher level of reliability and availability can be attained by investing effort in reducing software complexity.
- Redundant software is much more complex so there is scope for a range of additional errors that affect the system reliability but are caused by the existence of the fault-tolerance controllers.

Key points

- Dependability in a system can be achieved through fault avoidance and fault tolerance
- Some programming language constructs such as gotos, recursion and pointers are inherently error-prone
- Data typing allows many potential faults to be trapped at compile time.

Key points

- Fault tolerant architectures rely on replicated hardware and software components
- The include mechanisms to detect a faulty component and to switch it out of the system
- N-version programming and recovery blocks are two different approaches to designing fault-tolerant software architectures
- Design diversity is essential for software redundancy