Lecture 12 – Fault Tolerant Design (Sommerville Ch. 18)

Karl R. Wilcox
K.R.Wilcox@reading.ac.uk

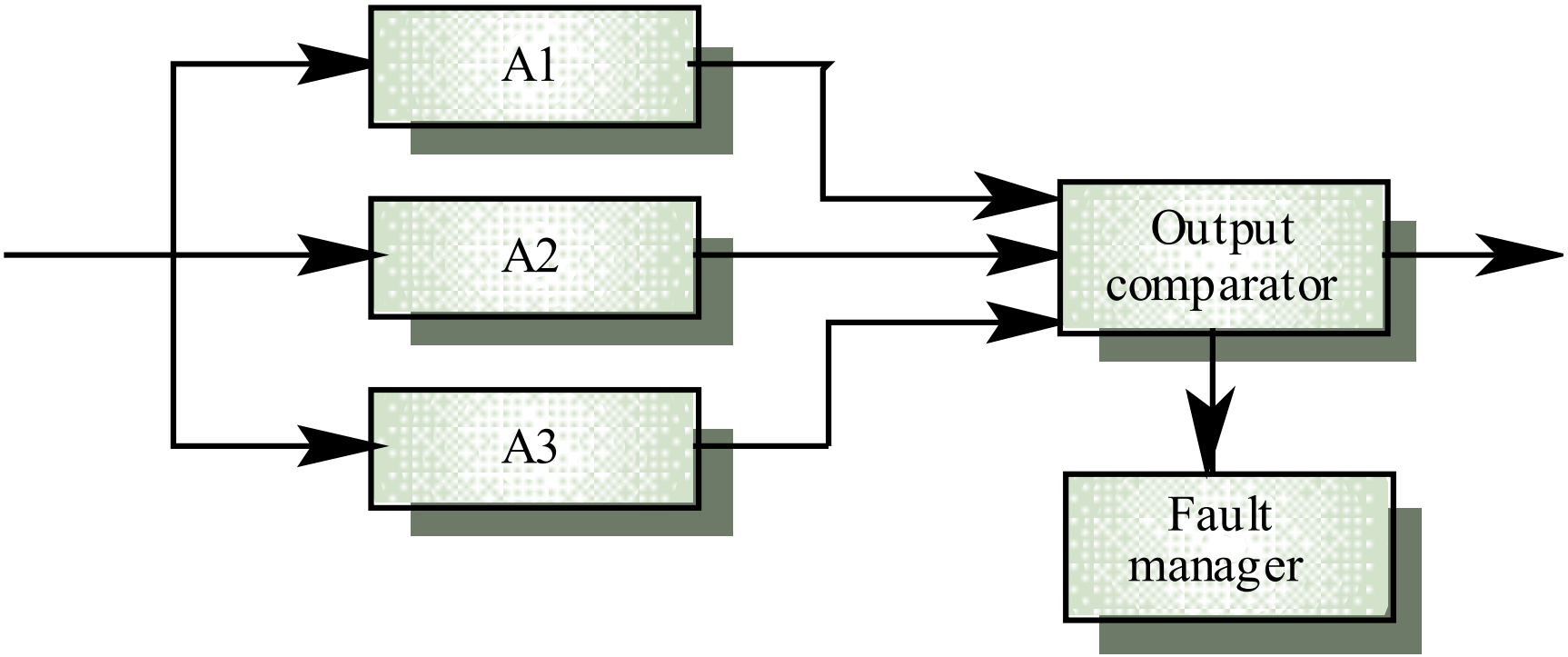
Fault tolerant architecture

- 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

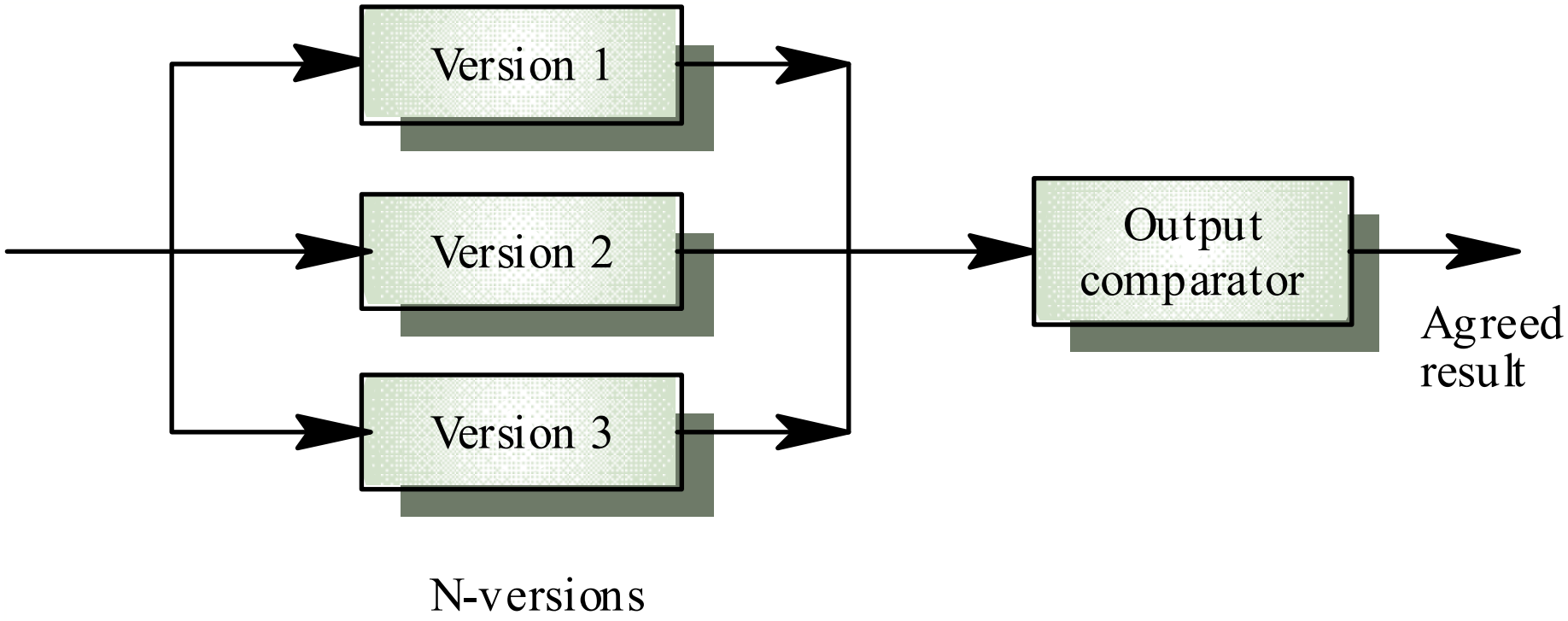
Design diversity

- 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



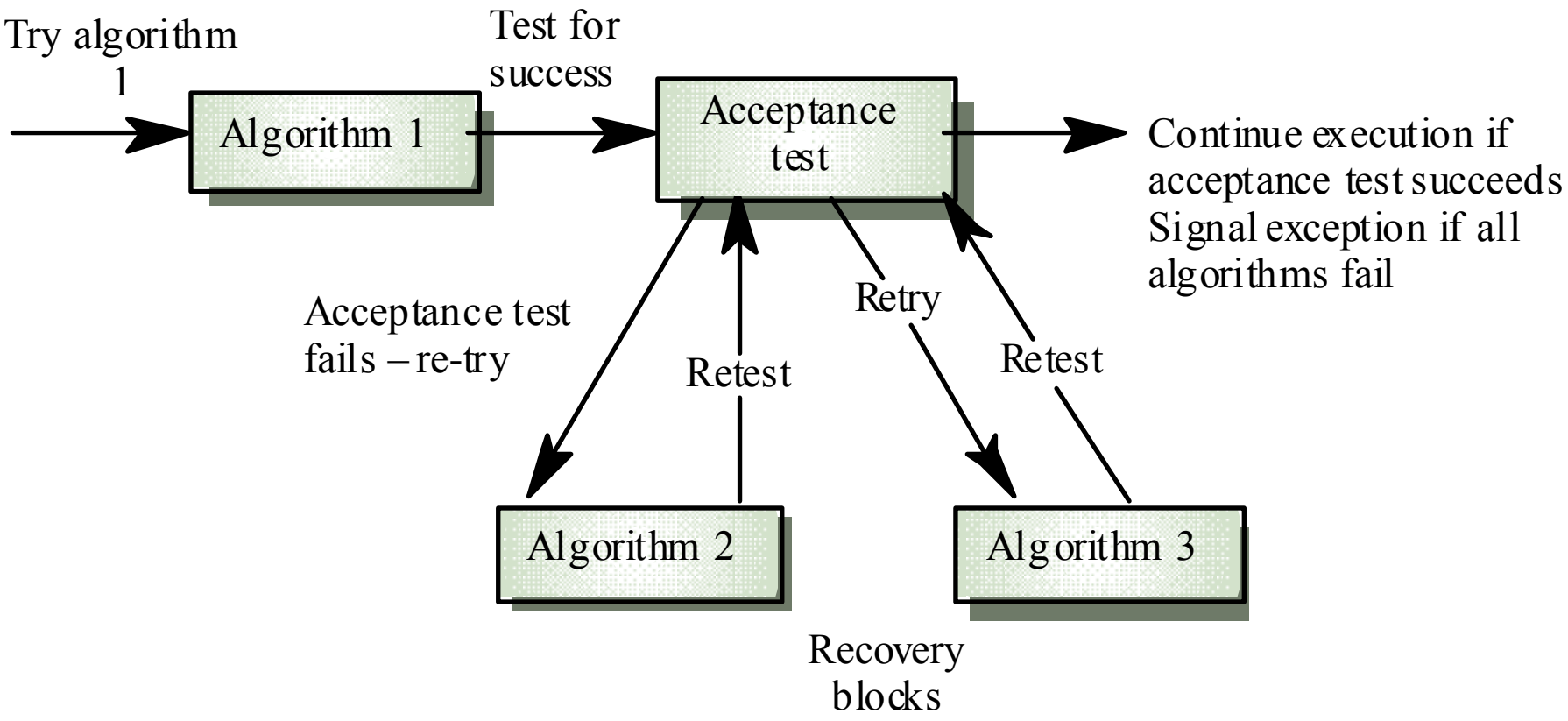
Output comparison

- 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