

# *Characteristics of a RTS*



- Large and complex
- Concurrent control of separate system components
- Facilities to interact with special purpose hardware
- Guaranteed response times
- **Extreme reliability**
- Efficient implementation

# *Reliability and Fault Tolerance*

- Goal
  - To understand the factors which affect the reliability of a system and how software design faults can be tolerated.
- Topics
  - Reliability, failure and faults
  - Failure modes
  - Fault prevention and fault tolerance
  - N-Version programming
  - Software dynamic redundancy
  - The recovery block approach to software fault tolerance
  - A comparison between n-version programming and recovery blocks
  - Dynamic redundancy and exceptions
  - Safety, reliability and dependability

# Scope



Four sources of faults which can result in system failure:

- Inadequate specification — not covered in this course
- **Design errors in software — covered now**
- Processor failure — not covered in course, see book
- Interference on the communication subsystem — not covered in course, see book

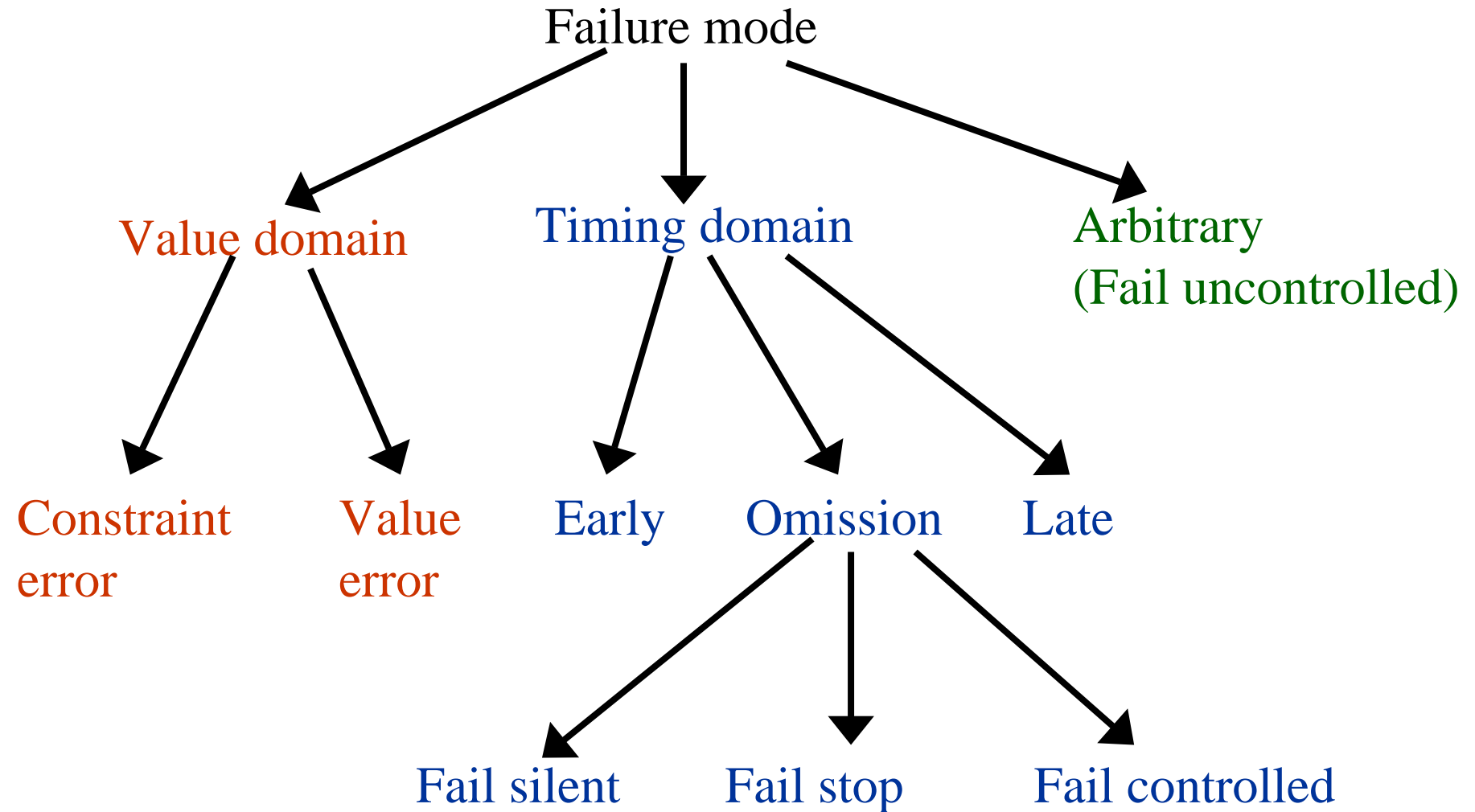
# *Reliability, Failure and Faults*

- The reliability of a system is a measure of the success with which it conforms to some authoritative specification of its behaviour
- When the behaviour of a system deviates from that which is specified for it, this is called a failure
- Failures result from unexpected problems internal to the system which eventually manifest themselves in the system's external behaviour
- These problems are called errors and their mechanical or algorithmic cause are termed faults
- Systems are composed of components which are themselves systems: hence
  - > failure -> fault -> error -> failure -> fault

# *Fault Types*

- A transient fault starts at a particular time, remains in the system for some period and then disappears
- E.g. hardware components which have an adverse reaction to radioactivity
- Many faults in communication systems are transient
- Permanent faults remain in the system until they are repaired; e.g., a broken wire or a software design error.
- Intermittent faults are transient faults that occur from time to time
- E.g. a hardware component that is heat sensitive, it works for a time, stops working, cools down and then starts to work again

# Failure Modes



# *Approaches to Achieving Reliable Systems*



- **Fault prevention** attempts to eliminate any possibility of faults creeping into a system before it goes operational
- **Fault tolerance** enables a system to continue functioning even in the presence of faults
- Both approaches attempt to produce systems which have well-defined failure modes

# *Fault Prevention*

- Two stages: **fault avoidance** and **fault removal**
- Fault avoidance attempts to limit the introduction of faults during system construction by:
  - use of the most **reliable components** within the given cost and performance constraints
  - use of **thoroughly-refined techniques** for interconnection of components and assembly of subsystems
  - packaging the hardware to **screen out** expected forms of **interference**.
  - **rigorous**, if not formal, specification of **requirements**
  - use of **proven design methodologies**
  - use of languages with facilities for **data abstraction and modularity**
  - use of **software engineering environments** to help manipulate software components and thereby manage complexity



# *Fault Removal*

- In spite of fault avoidance, design errors in both hardware and software components will exist
- **Fault removal**: procedures for finding and removing the causes of errors; e.g. design reviews, program verification, code inspections and system testing
- System testing can never be exhaustive and remove all potential faults
  - A test can only be used to show the presence of faults, not their absence.
  - It is sometimes impossible to test under realistic conditions
  - most tests are done with the system in simulation mode and it is difficult to guarantee that the simulation is accurate
  - Errors that have been introduced at the requirements stage of the system's development may not manifest themselves until the system goes operational

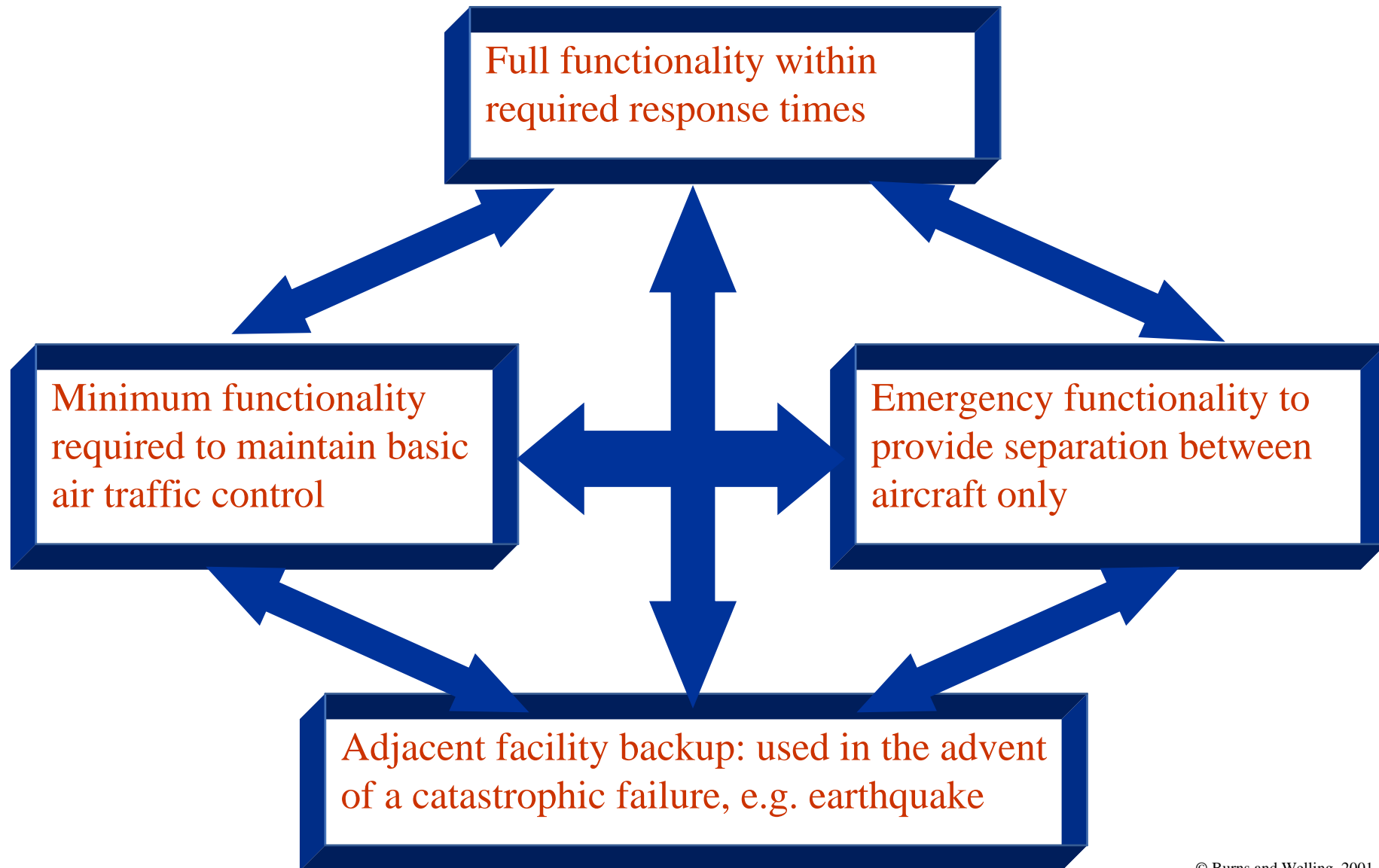
# *Failure of Fault Prevention Approach*

- In spite of all the testing and verification techniques, hardware components will fail; the fault prevention approach will therefore be unsuccessful when
  - either the frequency or duration of repair times are unacceptable, or
  - the system is inaccessible for maintenance and repair activities
- An extreme example of the latter is the crewless spacecraft Voyager
- Alternative is **Fault Tolerance**

# *Levels of Fault Tolerance*

- **Full Fault Tolerance** — the system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance
- **Graceful Degradation (fail soft)** — the system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair
- **Fail Safe** — the system maintains its integrity while accepting a temporary halt in its operation
- The level of fault tolerance required will depend on the application
- Most safety critical systems require full fault tolerance, however in practice many settle for graceful degradation

# Graceful Degradation in an ATC System



# Redundancy

- All fault-tolerant techniques rely on extra elements introduced into the system to detect & recover from faults
- Components are redundant as they are not required in a perfect system
- Often called **protective redundancy**
- Aim: minimise redundancy while maximising reliability, subject to the cost and size constraints of the system
- Warning: the added components inevitably increase the complexity of the overall system
- This itself can lead to less reliable systems
- E.g., first launch of the space shuttle
- It is advisable to separate out the fault-tolerant components from the rest of the system

# *Hardware Fault Tolerance*

- Two types: **static** (or **masking**) and **dynamic** redundancy
- **Static**: redundant components are used inside a system to hide the effects of faults; e.g. Triple Modular Redundancy
- **TMR** — 3 identical subcomponents and majority voting circuits; the outputs are compared and if one differs from the other two that output is masked out
- Assumes the fault is not common (such as a design error) but is either transient or due to component deterioration
- To mask faults from more than one component requires **NMR**
- **Dynamic**: redundancy supplied inside a component which indicates that the output is in error; provides an error detection facility; recovery must be provided by another component
- E.g. communications checksums and memory parity bits

# *Software Fault Tolerance*



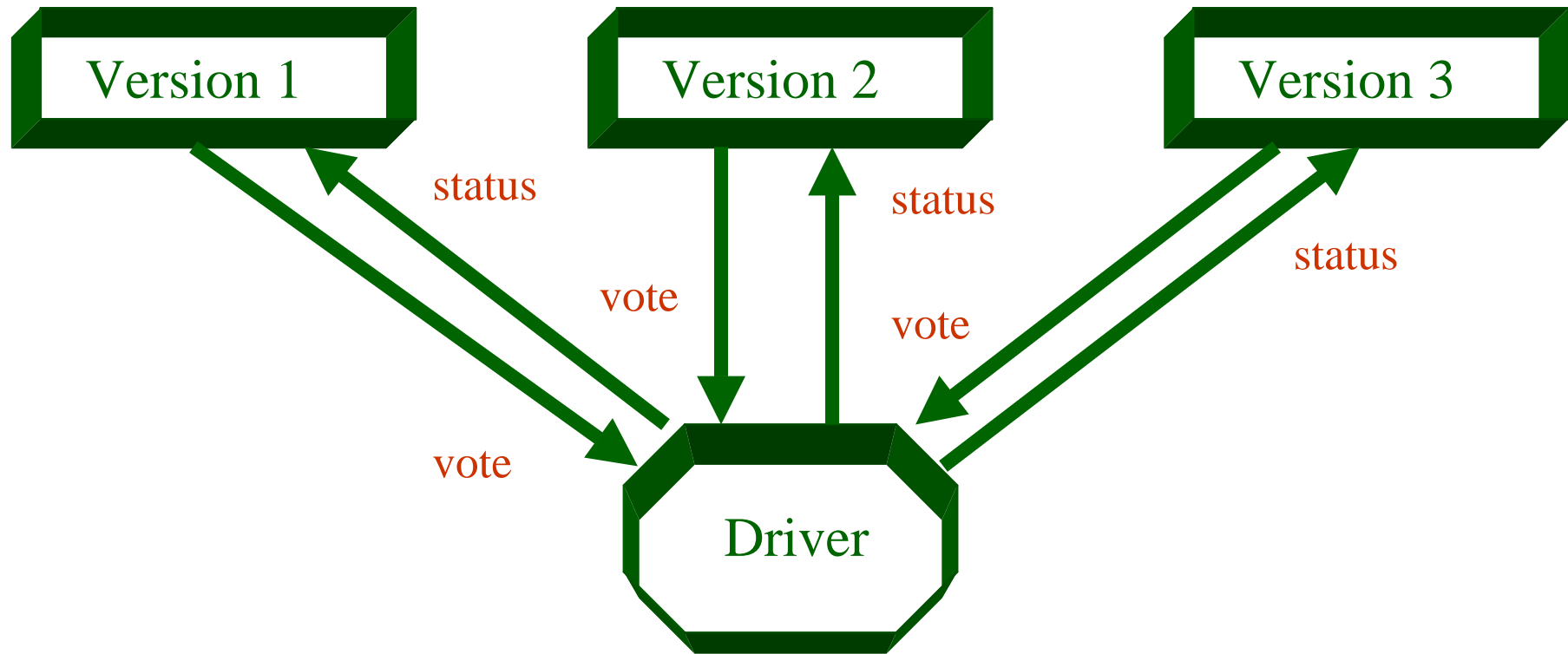
- Used for detecting design errors
- Static — **N-Version programming**
- Dynamic
  - Detection and Recovery
  - **Recovery blocks: backward error recovery**
  - **Exceptions: forward error recovery**

# *N-Version Programming*

- **Design diversity**
- The independent generation of  $N$  ( $N > 2$ ) functionally equivalent programs from the same initial specification
- **No interactions between groups**
- The programs execute concurrently with the same inputs and their results are compared by a driver process
- The results (VOTES) should be identical, if different the consensus result, assuming there is one, is taken to be correct



# *N-Version Programming*

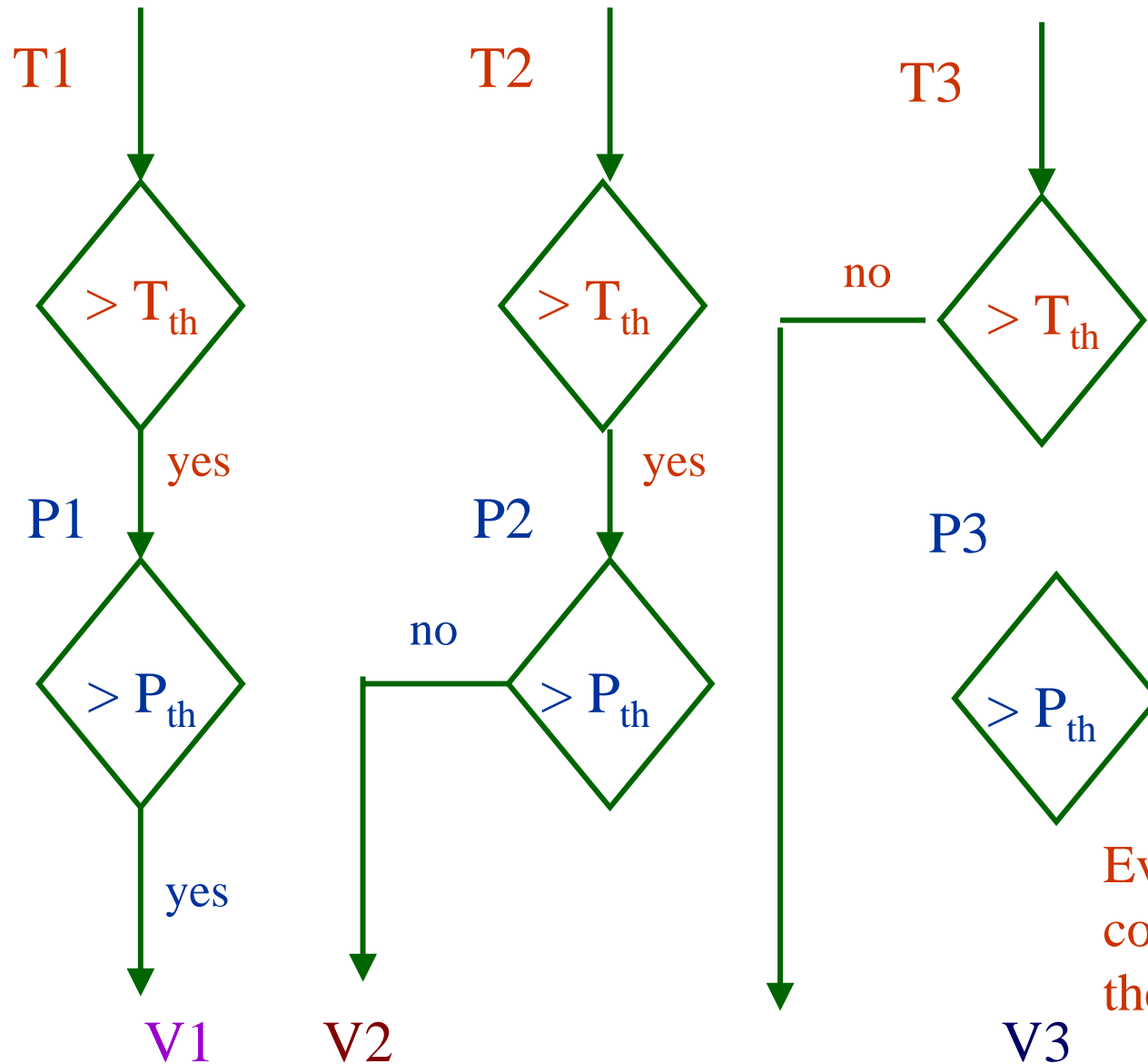


# *Vote Comparison*



- To what extent can votes be compared?
- Text or integer arithmetic will produce identical results
- Real numbers => different values
- Need inexact voting techniques

# Consistent Comparison Problem



Each version will produce a different but correct result

Even if use inexact comparison techniques, the problem occurs

# *N-version programming depends on*

- **Initial specification** — The majority of software faults stem from inadequate specification? A specification error will manifest itself in all N versions of the implementation
- **Independence of effort** — Experiments produce conflicting results. Where part of a specification is complex, this leads to a lack of understanding of the requirements. If these requirements also refer to rarely occurring input data, common design errors may not be caught during system testing
- **Adequate budget** — The predominant cost is software. A 3-version system will triple the budget requirement and cause problems of maintenance. Would a more reliable system be produced if the resources potentially available for constructing an N-versions were instead used to produce a single version?

*military versus civil avionics industry*

# *Software Dynamic Redundancy*

## Four phases

- **error detection** — no fault tolerance scheme can be utilised until the associated error is detected
- **damage confinement and assessment** — to what extent has the system been corrupted? The delay between a fault occurring and the detection of the error means erroneous information could have spread throughout the system
- **error recovery** — techniques should aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality)
- **fault treatment and continued service** — an error is a symptom of a fault; although damage repaired, the fault may still exist

# *Error Detection*



- **Environmental detection**
  - hardware — e.g. illegal instruction
  - O.S/RTS — null pointer
- **Application detection**
  - Replication checks
  - Timing checks
  - Reversal checks
  - Coding checks
  - Reasonableness checks
  - Structural checks
  - Dynamic reasonableness check

# *Damage Confinement and Assessment*

- Damage assessment is closely related to damage confinement techniques used
- Damage confinement is concerned with structuring the system so as to minimise the damage caused by a faulty component (also known as **firewalling**)
- **Modular decomposition** provides static damage confinement; allows data to flow through well-define pathways
- **Atomic actions** provides dynamic damage confinement; they are used to move the system from one consistent state to another

# *Error Recovery*

- Probably the most important phase of any fault-tolerance technique
- Two approaches: **forward** and **backward**
- **Forward error recovery** continues from an erroneous state by making selective corrections to the system state
- This includes making safe the controlled environment which may be hazardous or damaged because of the failure
- It is system specific and depends on accurate predictions of the location and cause of errors (i.e, damage assessment)
- Examples: redundant pointers in data structures and the use of self-correcting codes such as Hamming Codes



# *Backward Error Recovery (BER)*

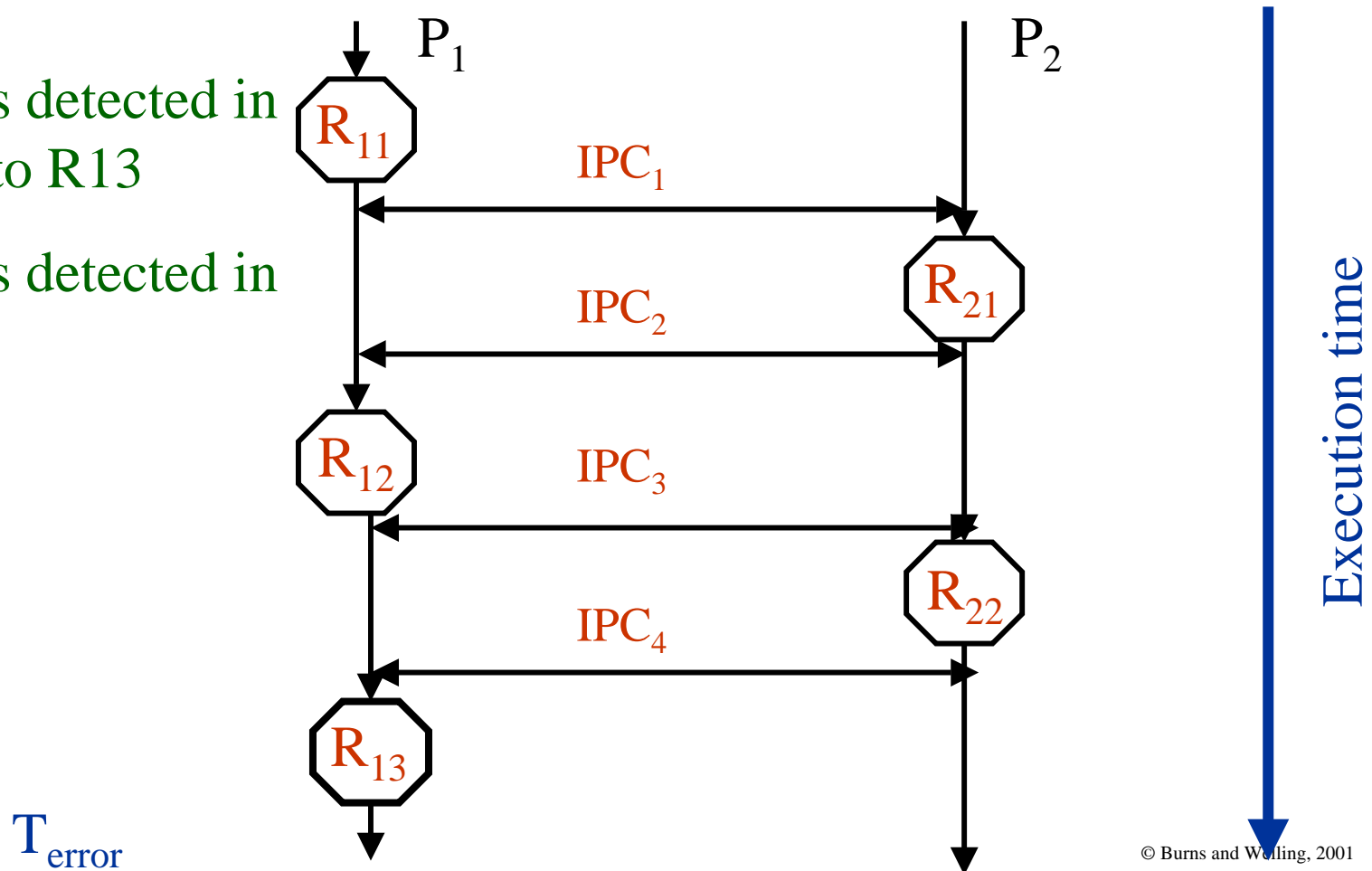
- **BER** relies on restoring the system to a previous safe state and executing an alternative section of the program
- This has the same functionality but uses a different algorithm (c.f. N-Version Programming) and therefore no fault
- The point to which a process is restored is called a **recovery point** and the act of establishing it is termed **checkpointing** (saving appropriate system state)
- Advantage: the erroneous state is cleared and it does not rely on finding the location or cause of the fault
- BER can, therefore, be used to recover from unanticipated faults including design errors
- Disadvantage: it cannot undo errors in the environment!

# The Domino Effect

- With concurrent processes that interact with each other, BER is more complex Consider:

If the error is detected in P1 rollback to R13

If the error is detected in P2 ?



# *Fault Treatment and Continued Service*

- ER returned the system to an error-free state; however, the error may recur; the final phase of F.T. is to eradicate the fault from the system
- The automatic treatment of faults is difficult and system specific
- Some systems assume all faults are transient; others that error recovery techniques can cope with recurring faults
- Fault treatment can be divided into 2 stages: **fault location** and **system repair**
- Error detection techniques can help to trace the fault to a component. For, hardware the component can be replaced
- A software fault can be removed in a new version of the code
- In non-stop applications it will be necessary to modify the program while it is executing!

# *The Recovery Block approach to FT*

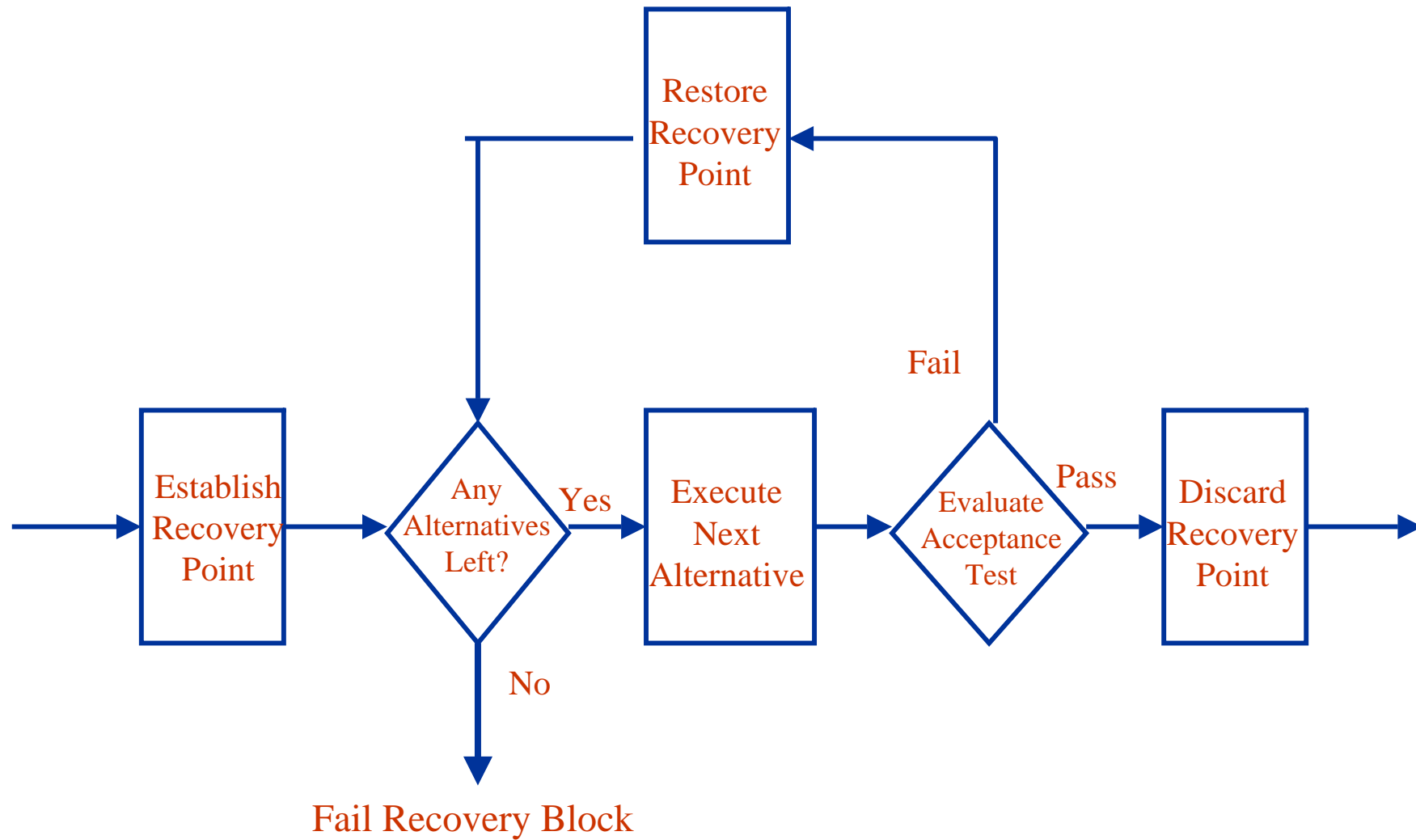
- Language support for BER
- At the entrance to a block is an **automatic recovery point** and at the exit **an acceptance test**
- The acceptance test is used to test that the system is in an acceptable state after the block's execution (primary module)
- If the acceptance test fails, the program is restored to the recovery point at the beginning of the block and an alternative module is executed
- If the alternative module also fails the acceptance test, the program is restored to the recovery point and yet another module is executed, and so on
- If all modules fail then the block fails and recovery must take place at a higher level

# Recovery Block Syntax

```
ensure <acceptance test>  
by  
    <primary module>  
else by  
    <alternative module>  
else by  
    <alternative module>  
    ...  
else by  
    <alternative module>  
else error
```

- 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 executed

# Recovery Block Mechanism



# *Example: Solution to Differential Equation*

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

- Explicit Kutta Method fast but inaccurate when equations are stiff
- Implicit Kutta Method more expensive but can deal with stiff equations
- The above will cope with all equations
- It will also potentially tolerate design errors in the Explicit Kutta Method if the acceptance test is flexible enough

# *Nested Recovery Blocks*

```
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*

- The acceptance test provides the **error detection** mechanism which enables the redundancy in the system to be exploited
- The design of the acceptance test is crucial to the efficacy of the RB scheme
- There is a trade-off between providing comprehensive acceptance tests and keeping overhead to a minimum, so that fault-free execution is not affected
- Note that the term used is **acceptance not correctness**; this allows a component to provide a degraded service
- All the previously discussed error detection techniques discussed can be used to form the acceptance tests
- However, care must be taken as a faulty acceptance test may lead to residual errors going undetected

# *N-Version Programming vs Recovery Blocks*

- **Static** (NV) versus **dynamic** redundancy (RB)
- **Design overheads** — both require alternative algorithms, NV requires driver, RB requires acceptance test
- **Runtime overheads** — NV requires  $N * \text{resources}$ , RB requires establishing recovery points
- **Diversity of design** — both susceptible to errors in requirements
- **Error detection** — vote comparison (NV) versus acceptance test(RB)
- **Atomicity** — NV vote before it outputs to the environment, RB must be structure to only output following the passing of an acceptance test

# *Dynamic Redundancy and Exceptions*

- An **exception** can be defined as the occurrence of an error
- Bringing an exception to the attention of the invoker of the operation which caused the exception, is called **raising** (or **signally** or **throwing**) the exception
- The invoker's response is called **handling** (or **catching**) the exception
- Exception handling is a **forward error recovery** mechanism, as there is no roll back to a previous state; instead control is passed to the handler so that recovery procedures can be initiated
- However, the exception handling facility can be used to provide backward error recovery

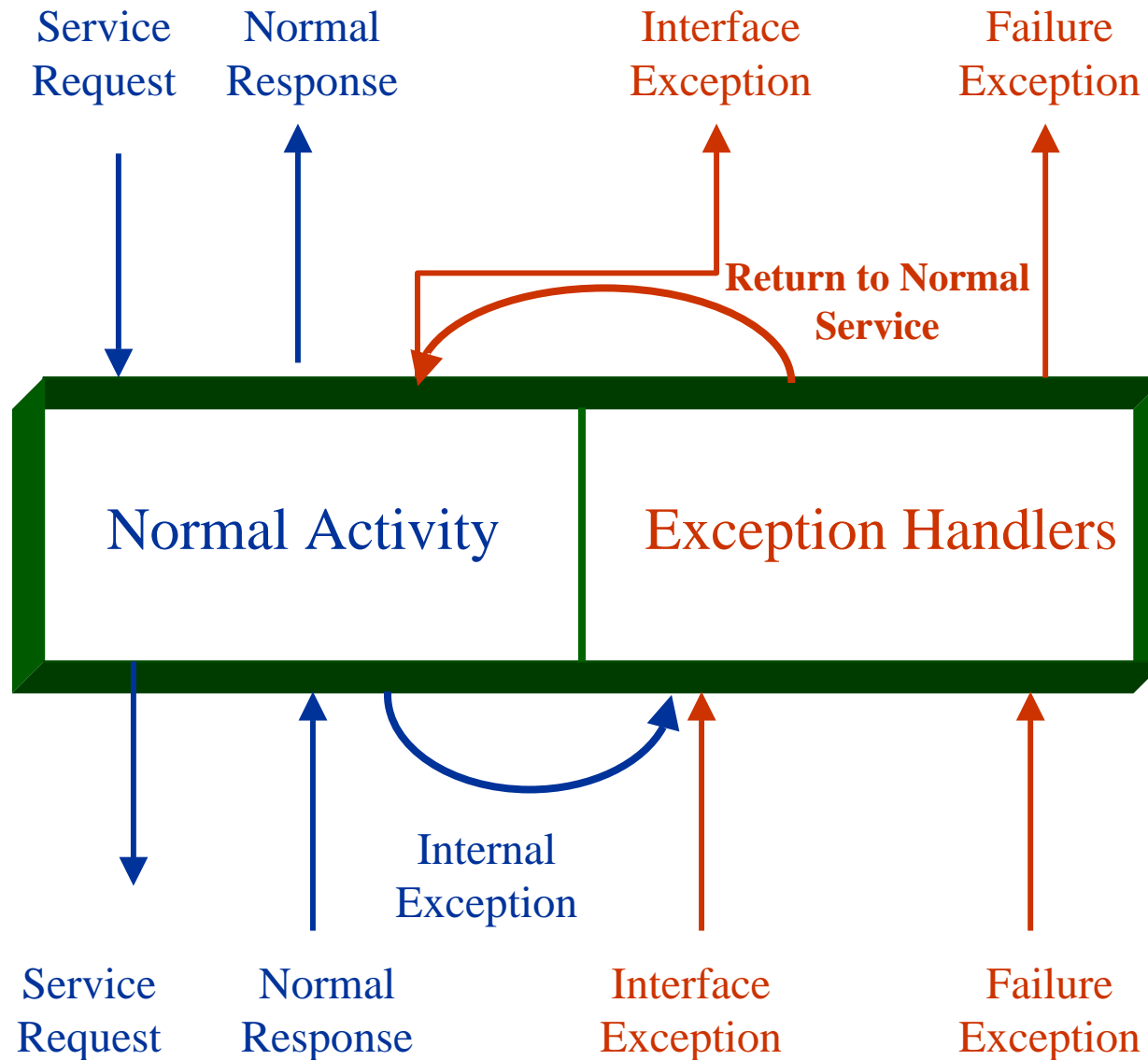
# *Exceptions*



Exception handling can be used to:

- cope with abnormal conditions arising in the environment
- enable program design faults to be tolerated
- provide a general-purpose error-detection and recovery facility

# *Ideal Fault-Tolerant Component*



# *Safety and Reliability*

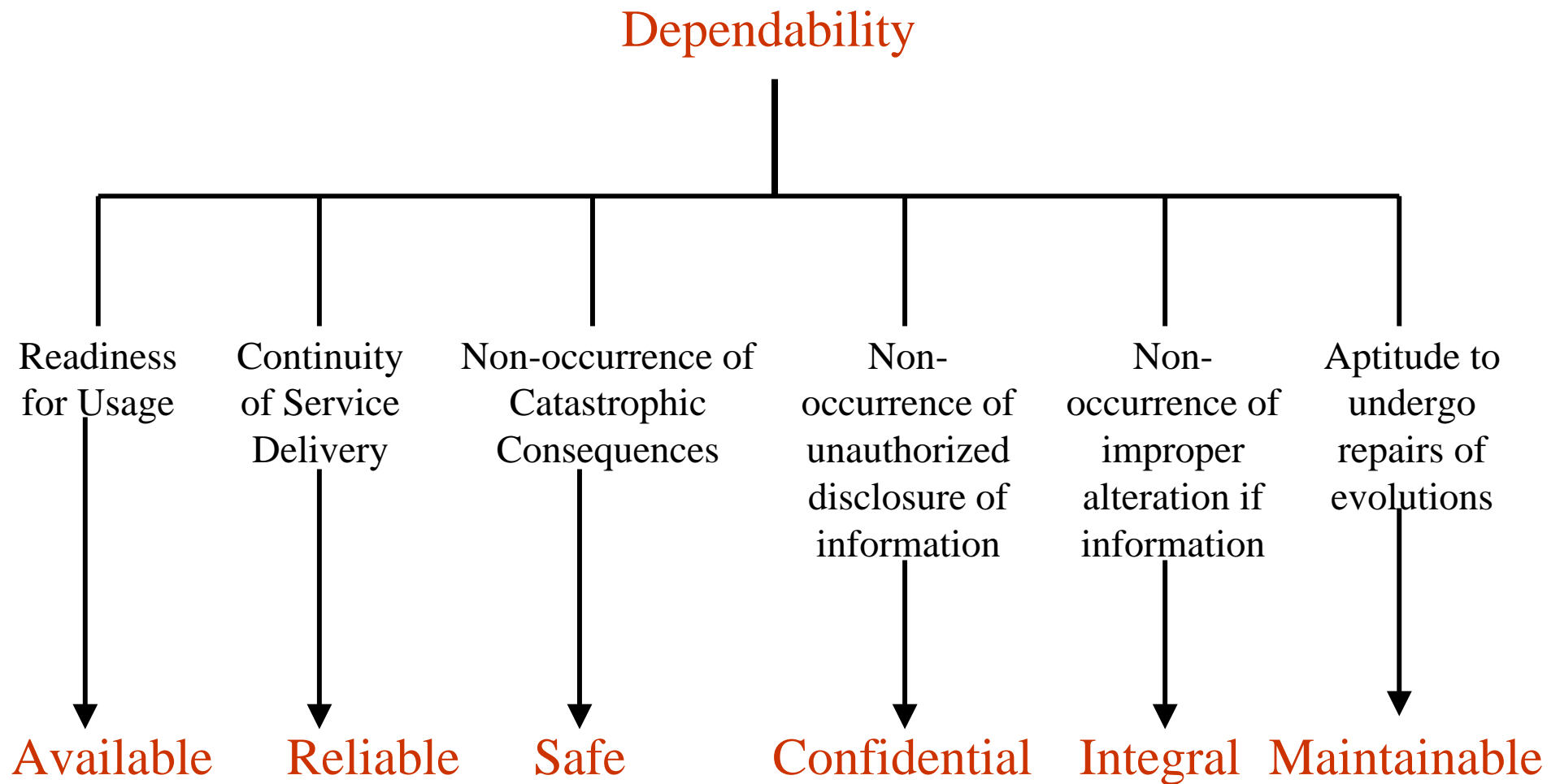
- **Safety**: freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm
  - By this definition, most systems which have an element of risk associated with their use as unsafe
- A **mishap** is an unplanned event or series of events that can result in death, injury, etc.
- **Reliability**: a measure of the success with which a system conforms to some authoritative specification of its behaviour.
- **Safety** is the probability that conditions that can lead to mishaps do not occur whether or not the intended function is performed

# Safety



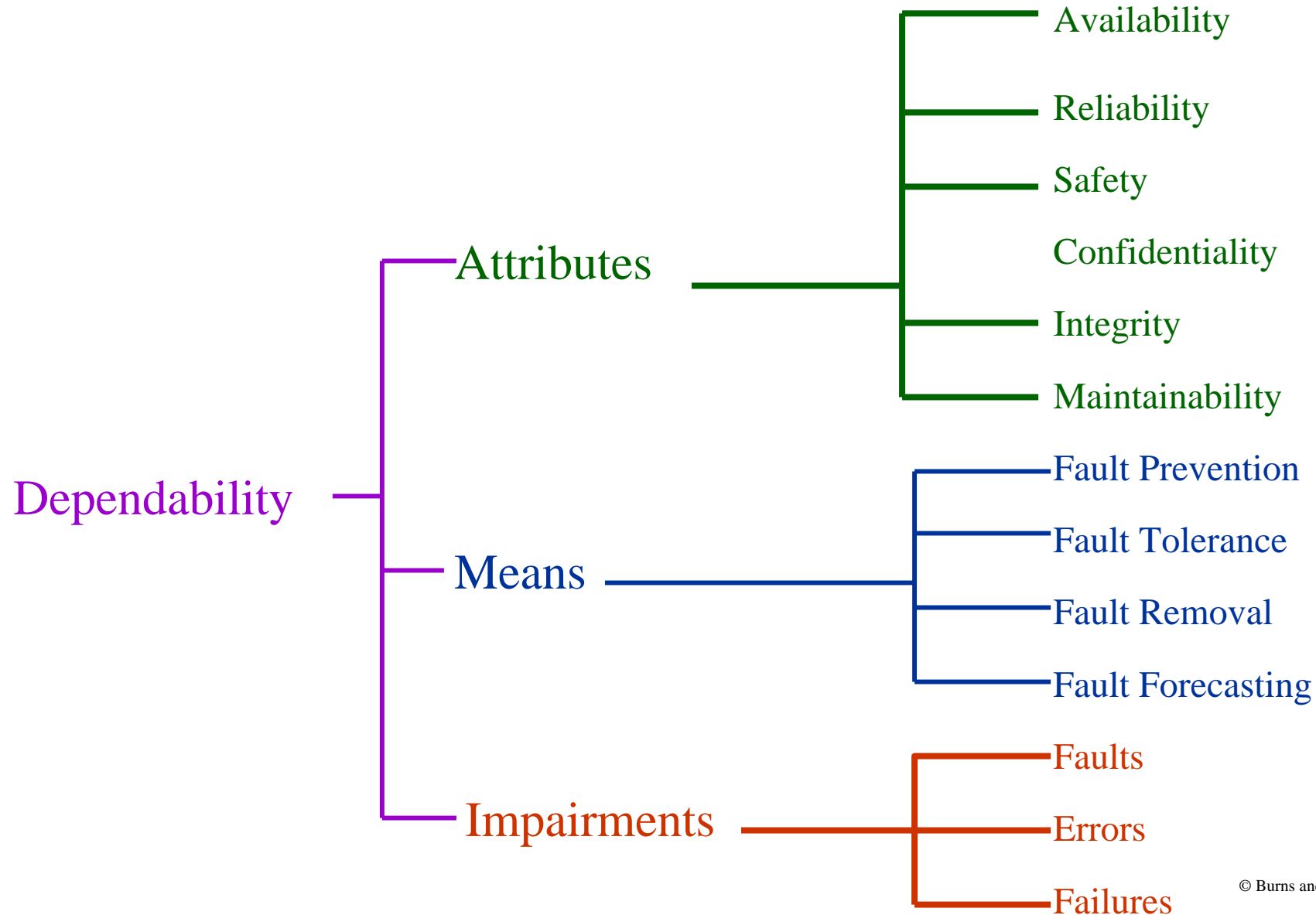
- E.g., measures which increase the likelihood of a weapon firing when required may well increase the possibility of its accidental detonation.
- In many ways, the only safe airplane is one that never takes off, however, it is not very reliable.
- As with reliability, to ensure the safety requirements of an embedded system, system safety analysis must be performed throughout all stages of its life cycle development.

# Aspects of Dependability





# Dependability Terminology



# Summary

- **Reliability**: a measure of the success with which the system conforms to some authoritative specification of its behaviour
- When the behaviour of a system deviates from that which is specified for it, this is called a **failure**
- Failures result from **faults**
- Faults can be **accidentally** or **intentionally** introduced into a system
- They can be **transient, permanent or intermittent**
- **Fault prevention** consists of **fault avoidance** and **fault removal**
- Fault tolerance involves the introduction of **redundant components** into a system so that faults can be detected and tolerated

# Summary

- **N-version programming**: the independent generation of N (where  $N \geq 2$ ) functionally equivalent programs from the same initial specification
- Based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been **developed independently** will **fail independently**
- Dynamic redundancy: error detection, damage confinement and assessment, error recovery, and fault treatment and continued service
- Atomic actions to aid damage confinement

# Summary



- With **backward error recovery**, it is necessary for communicating processes to reach consistent recovery points to avoid the domino effect
- For sequential systems, the **recovery block** is an appropriate language concept for BER
- Although **forward error recovery** is system specific, **exception handling** has been identified as an appropriate framework for its implementation
- The concept of an **ideal fault tolerant component** was introduced which used exceptions
- The notions of software **safety** and **dependability** have been introduced