LECTURE 01

INTRODUCTION

CE 705

Realtime Systems

Dr. Salem Al-Agtash

Prerequisites

- Basic understanding of C
- Basic understanding of Computer Architectures.
- Basic understanding of Operating Systems
- Basic understanding of Systems Programming

Course Aims

- Understanding of the broad concept of real time systems
- Understand the basic requirements of real-time systems
- Understand the implementation and analysis techniques
- Practice and hands on experience using realtime Linux and applications
- To stimulate research interest

What is a real-time system?

- A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period
  - the correctness depends not only on the logical result but also the time it was delivered
  - failure to respond is as bad as the wrong response!

Terminology

- Hard real-time — systems where it is absolutely imperative that responses occur within the required deadline. E.g. Flight control systems.
- Soft real-time — systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Data acquisition system.
- Real real-time — systems which are hard real-time and which the response times are very short. E.g. Missile guidance system.

.....

- Firm real-time — systems which are soft real-time but in which there is no benefit from late delivery of service.
- A single system may have all hard, soft and real real-time subsystems
- In reality many systems will have a cost function associated with it

Examples

TABLE 1.1. A Sampling of Hard, Firm, and Soft Real-Time Systems

System	Real-Time Classification	Explanation
Avionics weapons delivery system in which pressing a button launches an air-to-air missile	Hard	Missing the deadline to launch the missile within a specified time after pressing the button may cause the target to be missed, which will result in catastrophe
Navigation controller for an autonomous weed-killer robot	Firm	Missing a few navigation deadlines causes the robot to veer out from a planned path and damage some crops
Console hockey game	Soft	Missing even several deadlines will only degrade performance

Disciplines

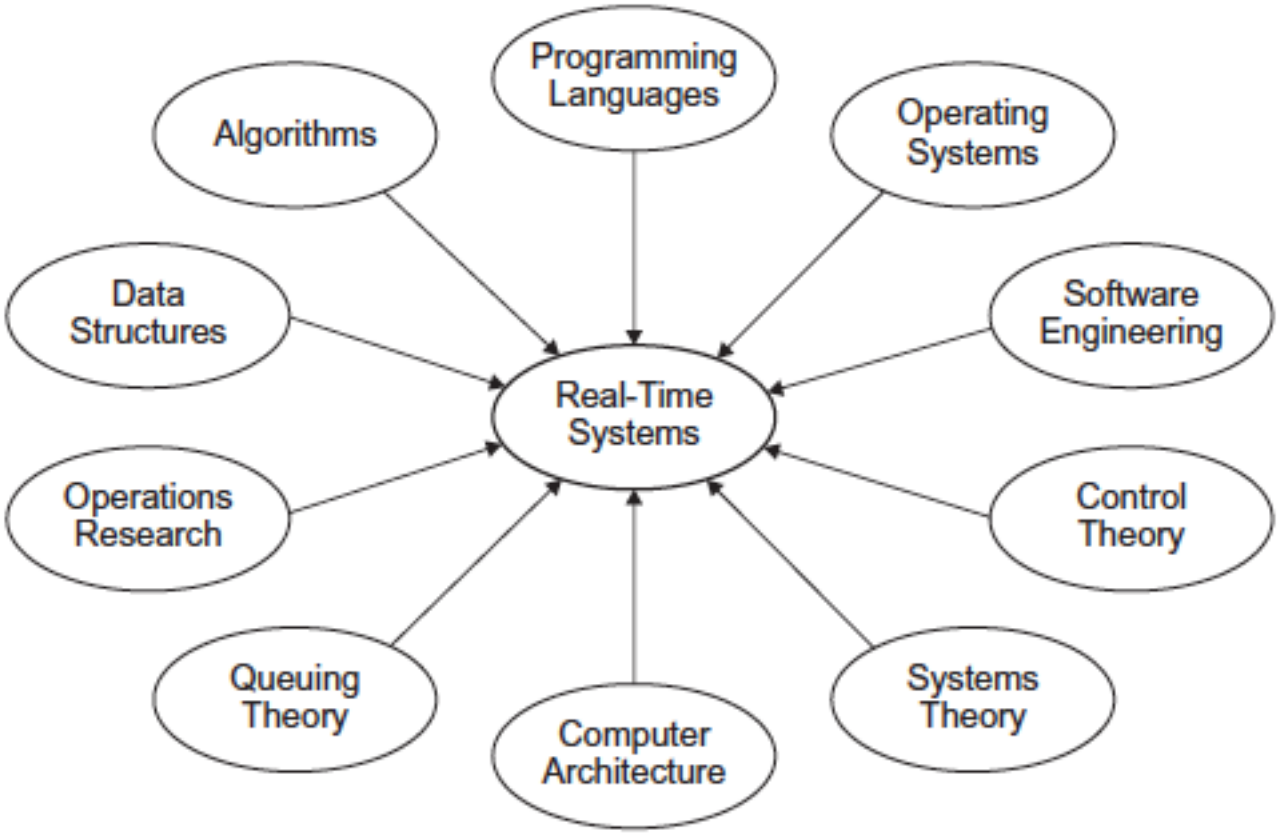


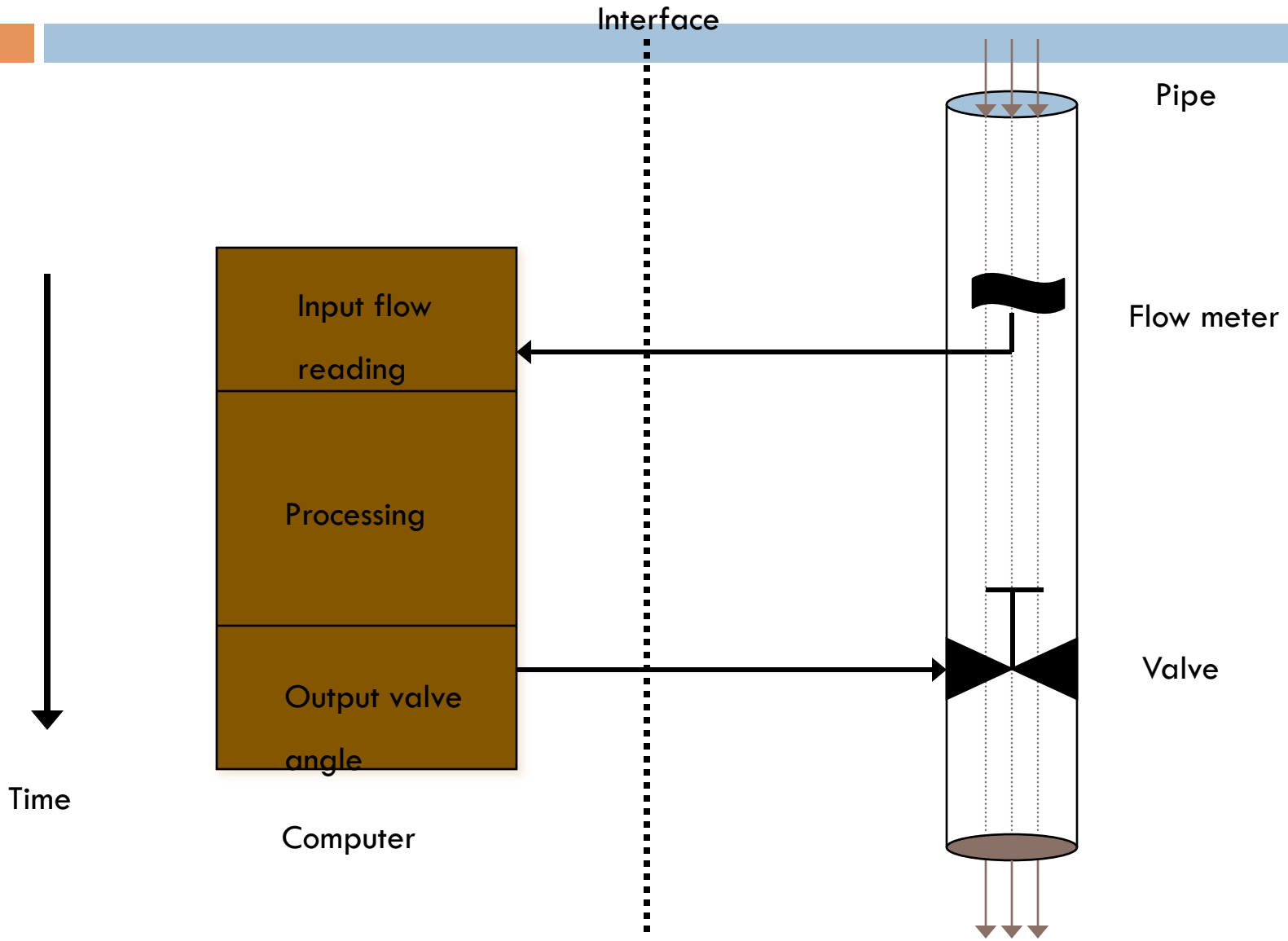
Figure 1.5. A variety of disciplines that affect real-time systems engineering.

Domains

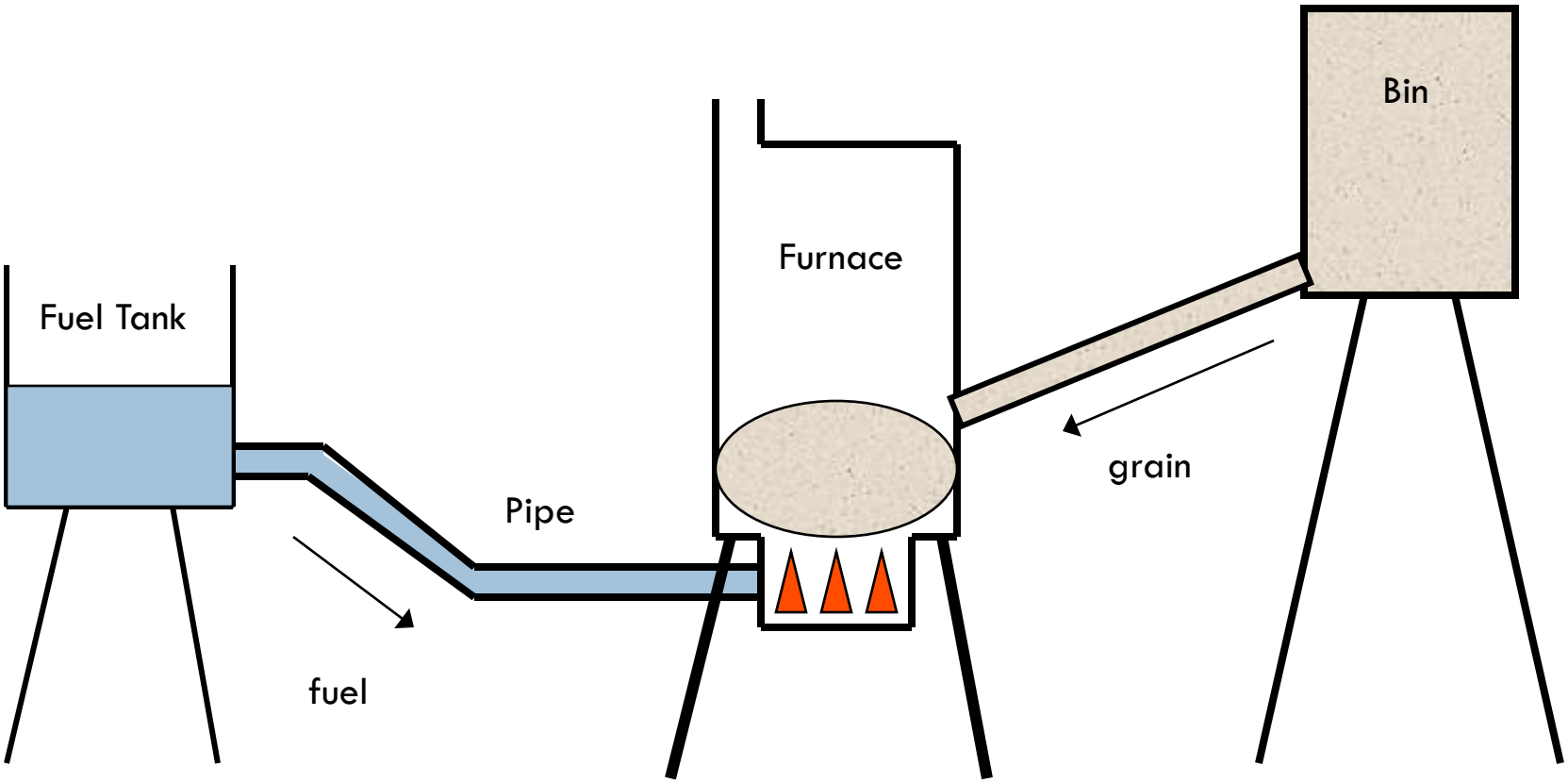
TABLE 1.4. Typical Real-Time Domains and Diverse Applications

Domain	Applications
Aerospace	Flight control
	Navigation
	Pilot interface
Civilian	Automotive systems
	Elevator control
	Traffic light control
Industrial	Automated inspection
	Robotic assembly line
	Welding control
Medical	Intensive care monitors
	Magnetic resonance imaging
	Remote surgery
Multimedia	Console games
	Home theaters
	Simulators

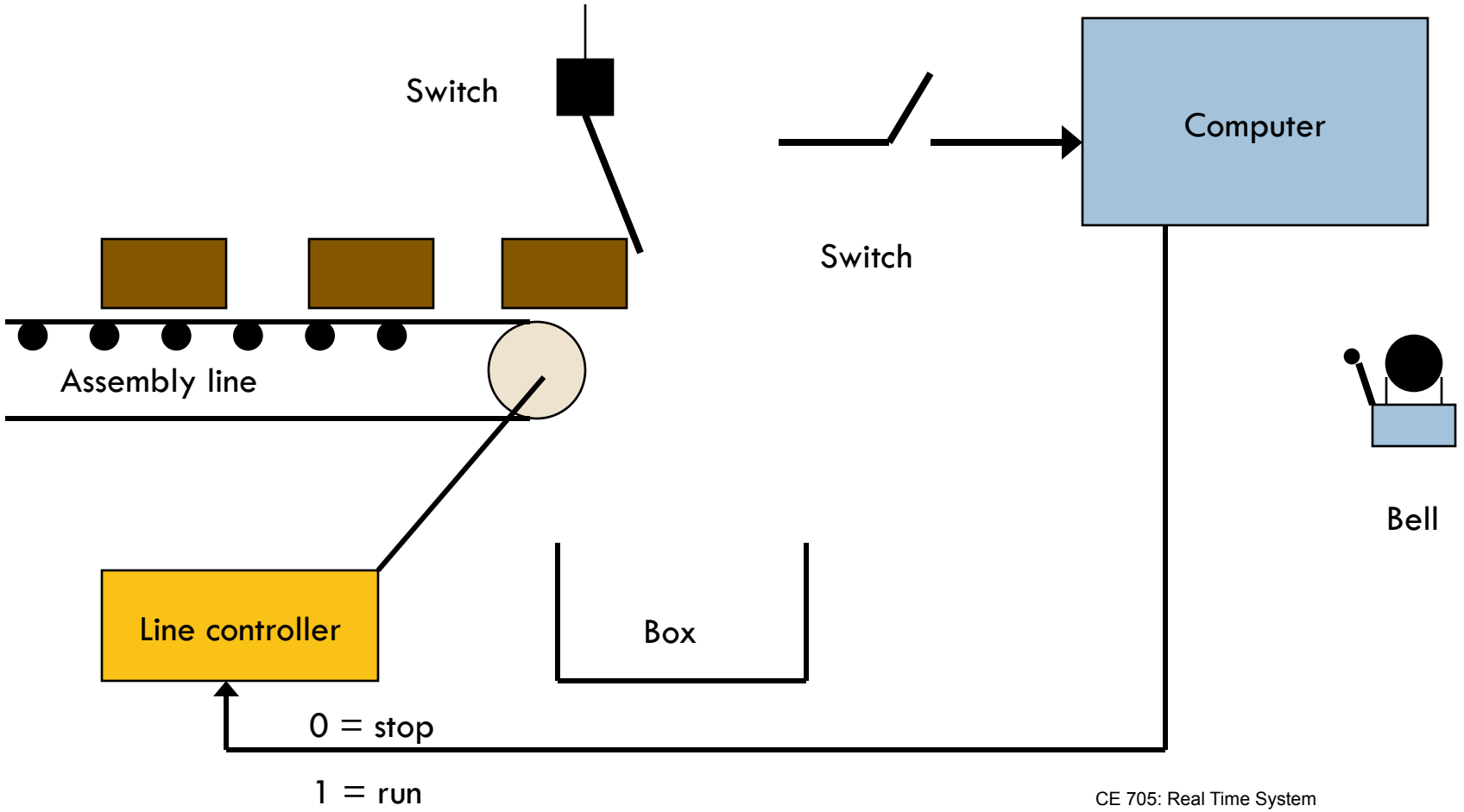
A simple fluid control system



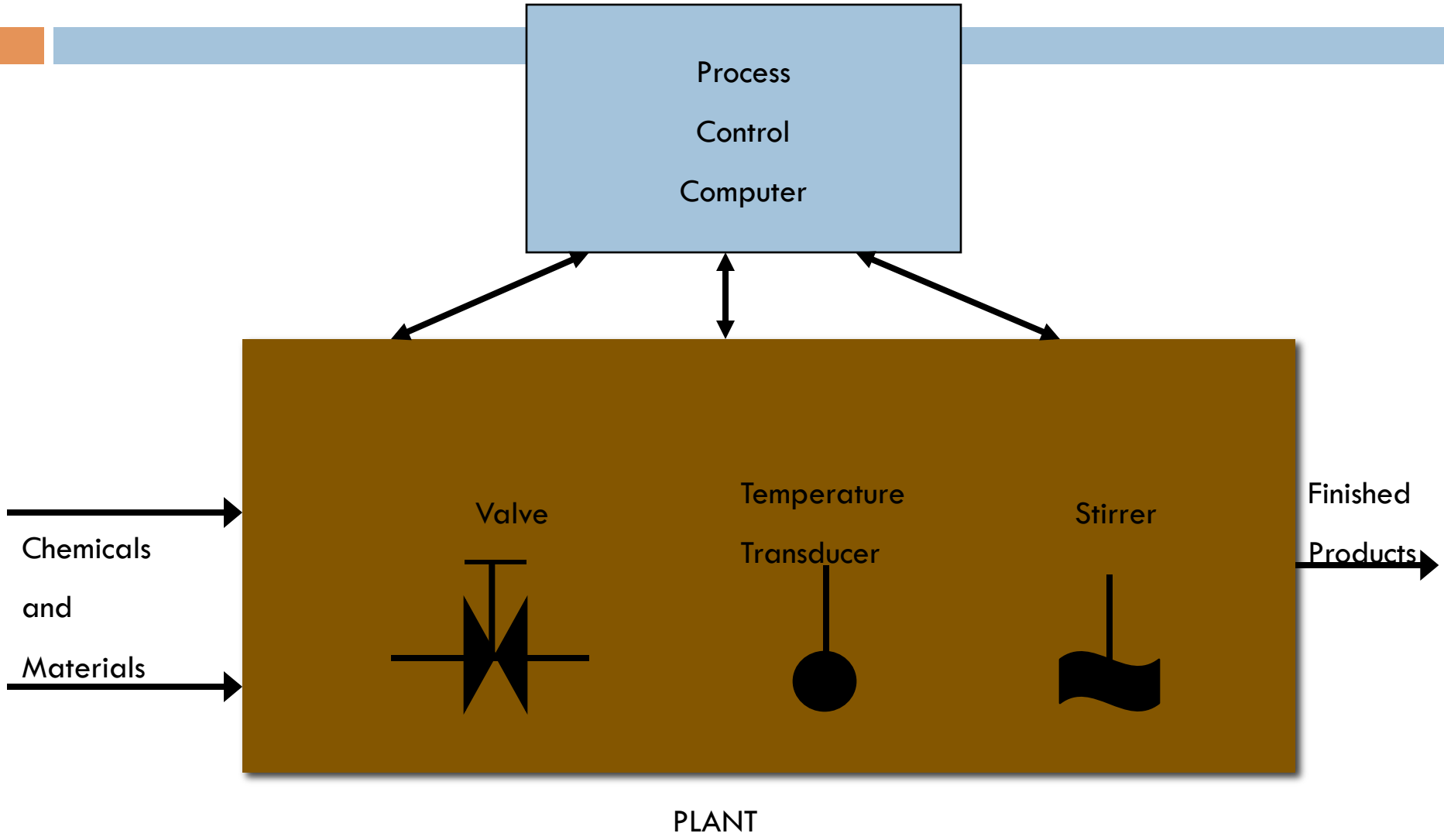
A Grain-Roasting Plant



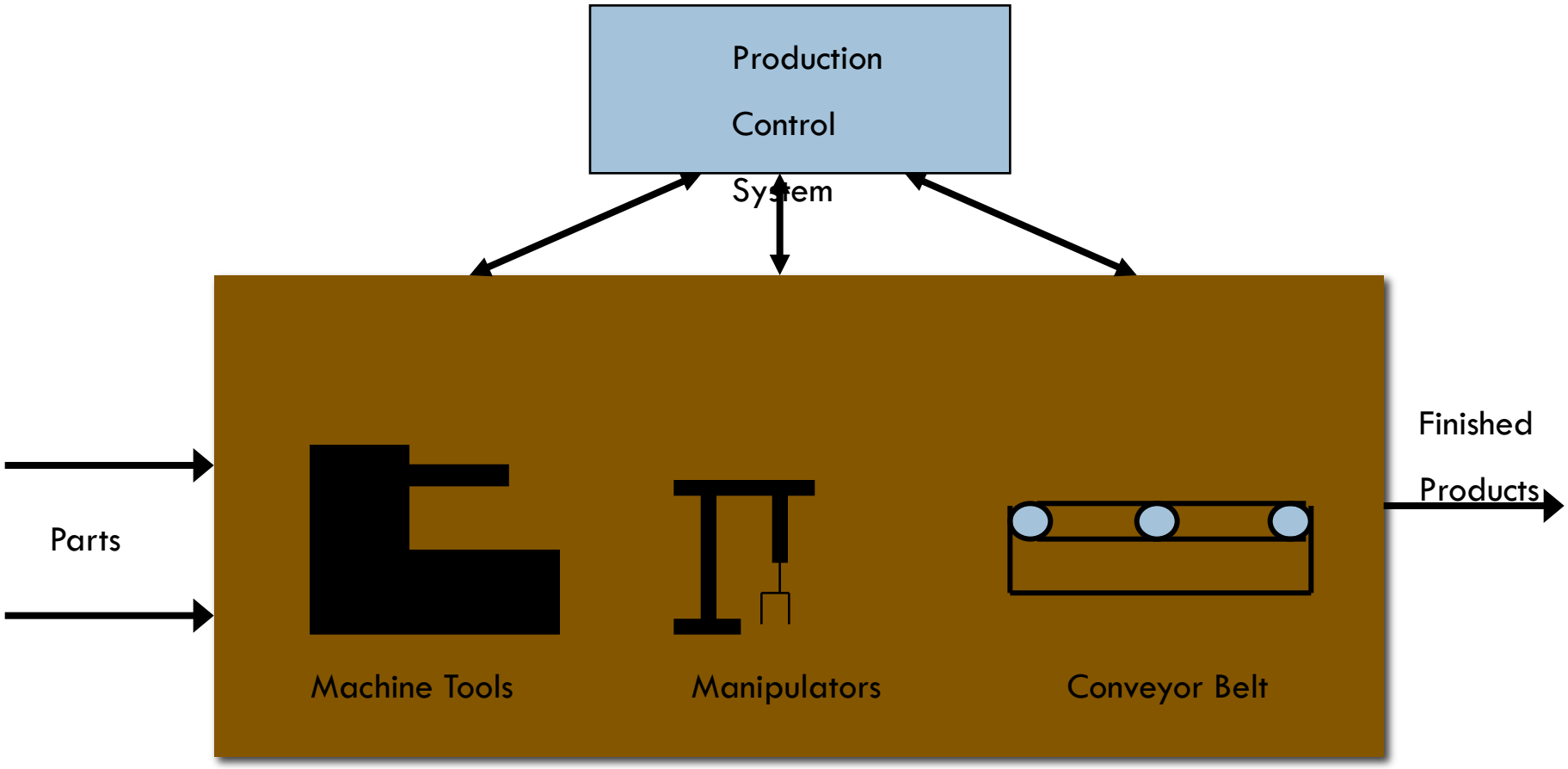
A Packing Station



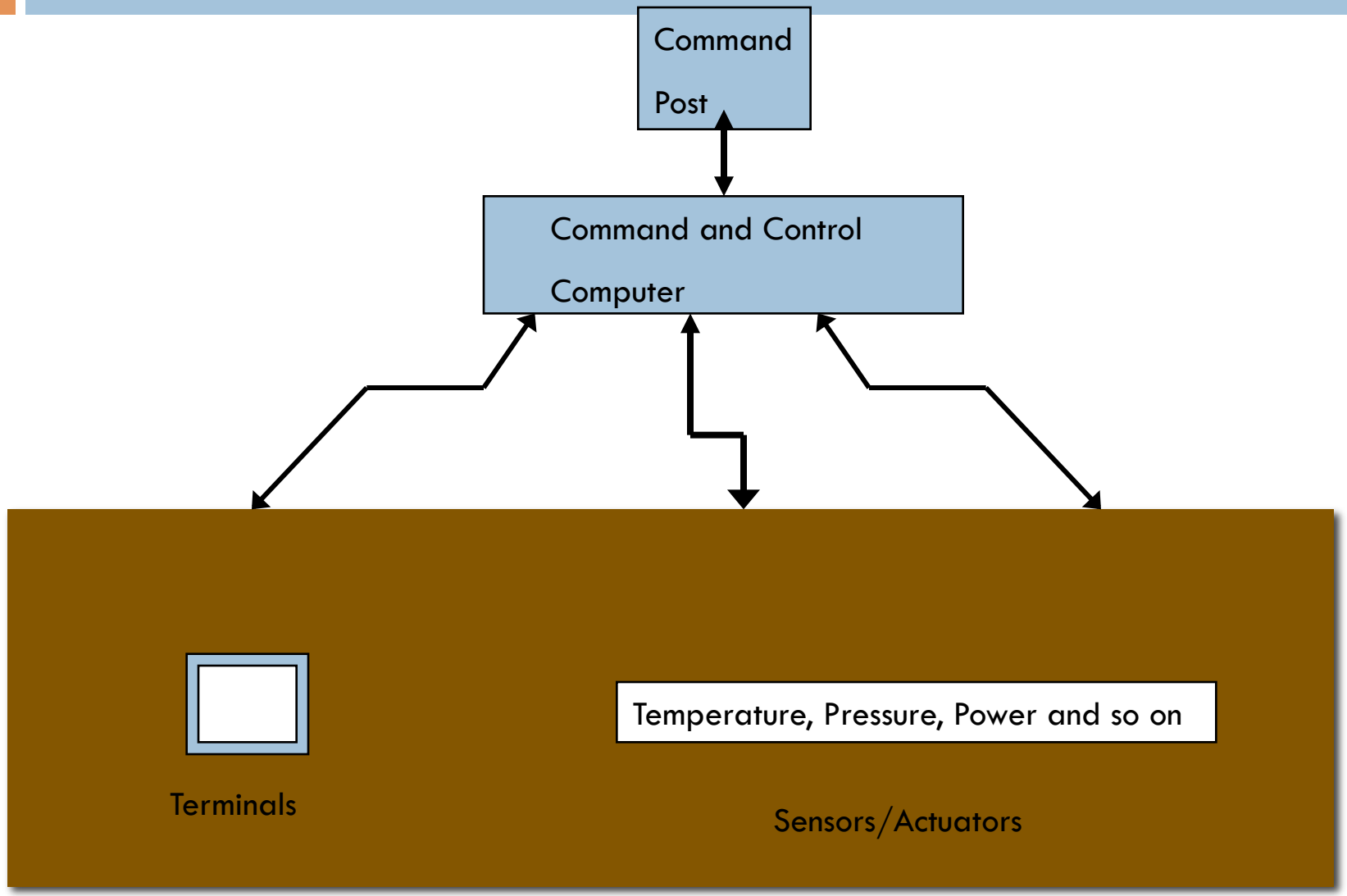
A Process Control System



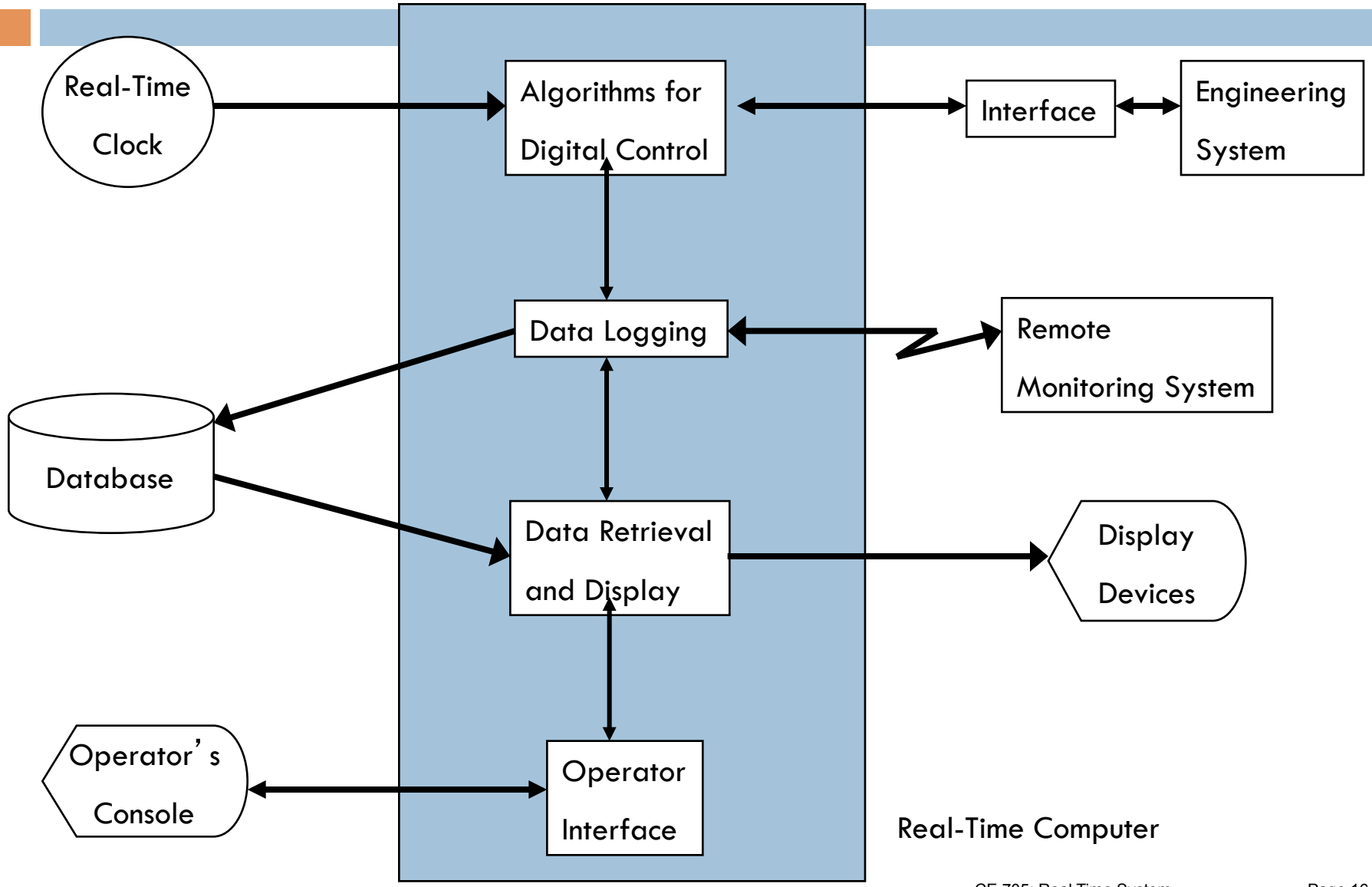
A Production Control System



A Command and Control System



A Typical Realtime System



Characteristics of a RTS

- Large and complex — vary from a few hundred lines of assembler or C to 20 million lines of Ada estimated for the Space Station Freedom
- Concurrent control of separate system components — devices operate in parallel in the real-world; better to model this parallelism by concurrent entities in the program

- Facilities to interact with special purpose hardware — need to be able to program devices in a reliable and abstract way
- Extreme reliability and safe — RT systems typically control the environment in which they operate; failure to control can result in loss of life, damage to environment or economic loss

- Guaranteed response times — we need to be able to predict with confidence the worst case response times for systems; efficiency is important but predictability is essential

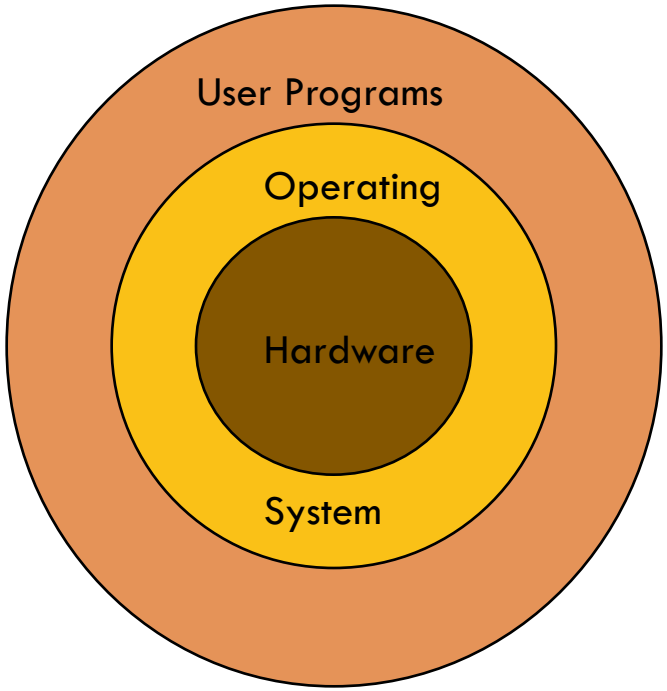
Real-time Programming Languages

- Assembly languages
- Sequential systems implementation languages — e.g. Coral 66, Jovial, C.
- Normally require operating system support.
- High-level concurrent languages: Ada, Chill, Modula-2, Mesa, Java.

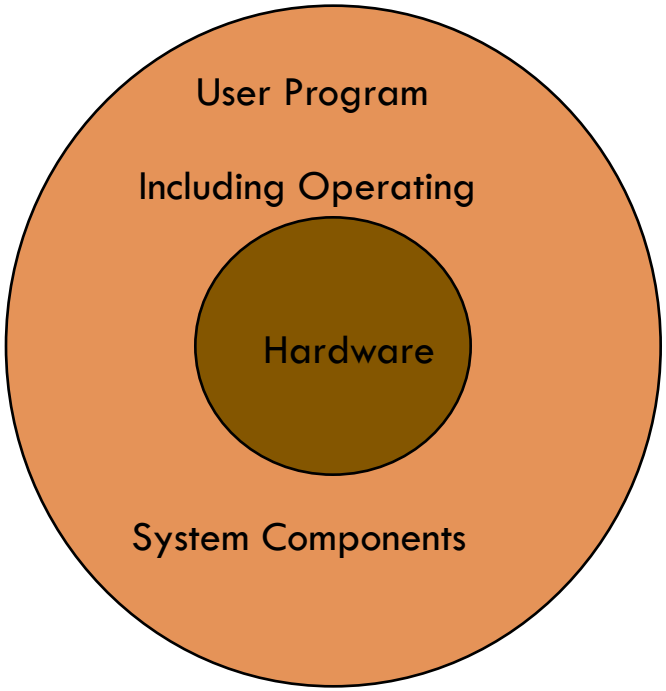
.....

- No operating system support!
  - Java/Real-Time Java
  - C and Real-Time POSIX
  - Ada 95
  - Also Modula-1 for device driving

Real-Time Languages and OSs



Typical OS Configuration



Typical Embedded Configuration

Typical applications

- Aerospace
  - Navigation systems, automatic landing systems, flight attitude controls, engine controls, space exploration (e.g., the Mars Pathfinder).
- Automotive
  - Fuel injection control, passenger environmental controls, anti-lock braking systems, air bag controls, GPS mapping.
- Children's Toys
  - Nintendo's "Game Boy", Mattel's "My Interactive Pooh", Tiger Electronics' "Furby".

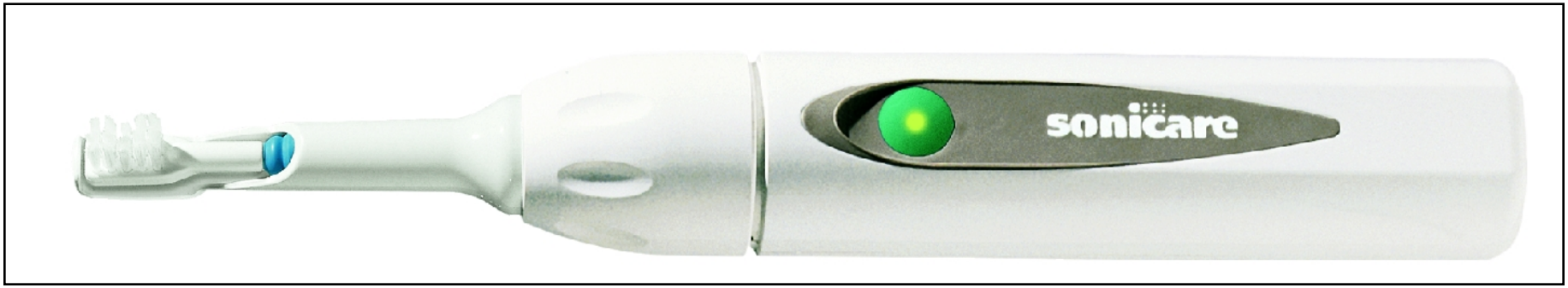
Typical applications

- Communications
  - Satellites; network routers, switches, hubs.
- Computer Peripherals
  - Printers, scanners, keyboards, displays, modems, hard disk drives, CD-ROM drives.
- Home
  - Dishwashers, microwave ovens, VCRs, televisions, stereos, fire/security alarm systems, lawn sprinkler controls, thermostats, cameras, clock radios, answering machines.

Typical applications

- Industrial
  - Elevator controls, surveillance systems, robots.
- Instrumentation
  - Data collection, oscilloscopes, signal generators, signal analyzers, power supplies.
- Medical
  - Imaging systems (e.g., XRAY, MRI, and ultrasound), patient monitors, heart pacers.
- Office Automation
  - FAX machines, copiers, telephones, cash registers.

Product: Sonicare Plus toothbrush. Microprocessor: 8-bit Zilog Z8.

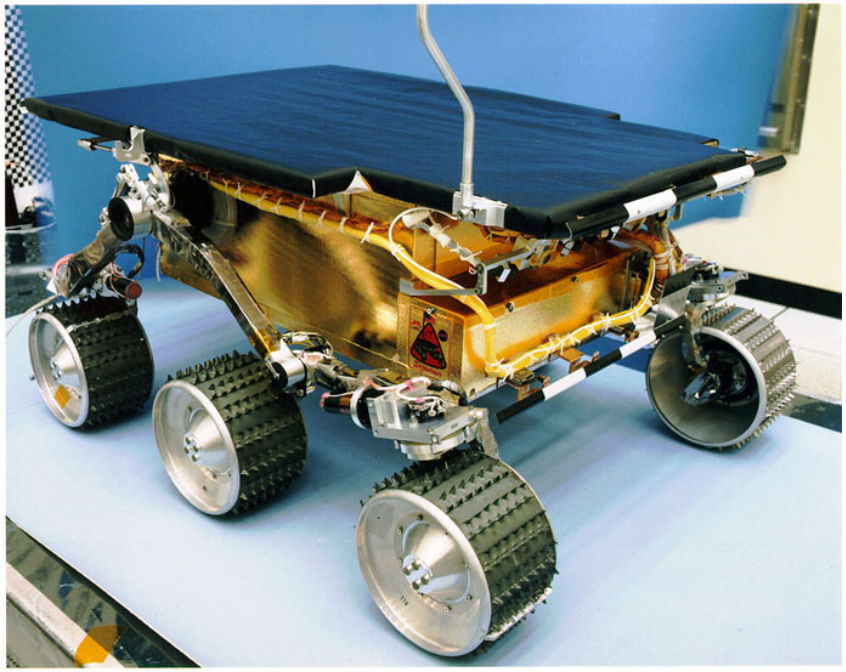


Miele dishwashers

Product: Miele dishwashers.

Microprocessor: 8-bit Motorola 68HC05.

NASA's Mars Sojourner Rover



Product: NASA's Mars Sojourner Rover.

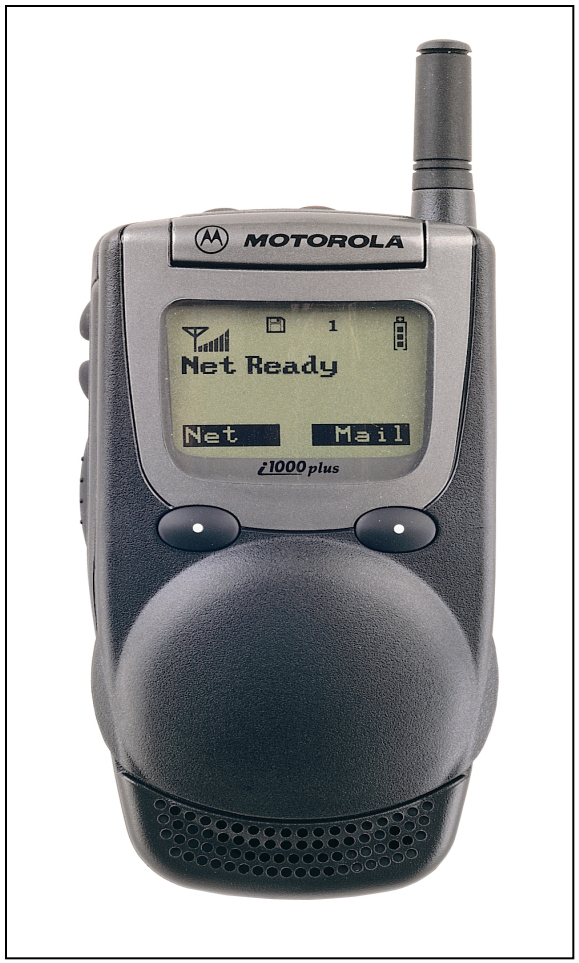
Microprocessor:
8-bit Intel 80C85.

Garmin StreetPilot GPS Receiver

Product: Garmin StreetPilot GPS Receiver.

Microprocessor: 16-bit.

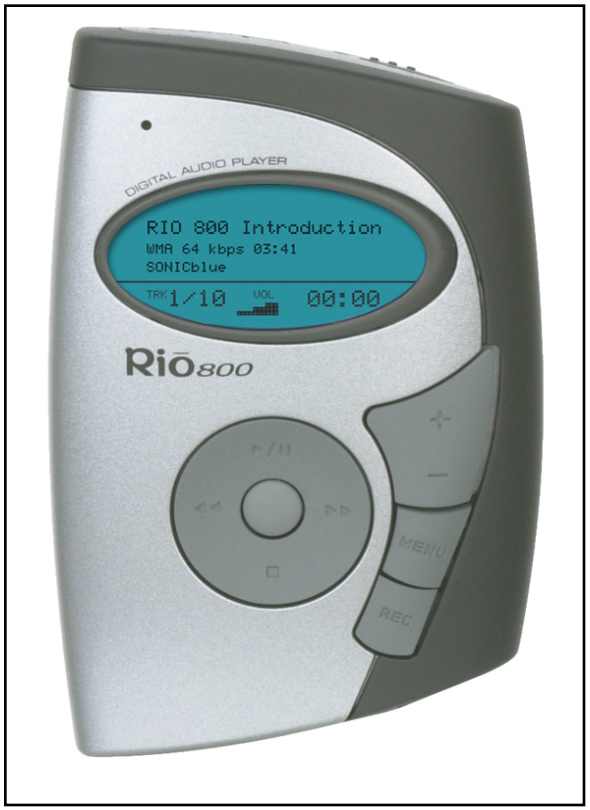
Motorola i1000plus iDEN Multi-Service Digital Phone



Product: Motorola i1000plus iDEN Multi-Service Digital Phone.

Microprocessor: Motorola 32-bit MCORE.

Rio 800 MP3 Player



Product: Rio 800 MP3 Player.

Microprocessor: 32-bit RISC.

Digital control

$$u_k = u_{k-2} + \alpha e_k + \beta e_{k-1} + \gamma e_{k-2}$$

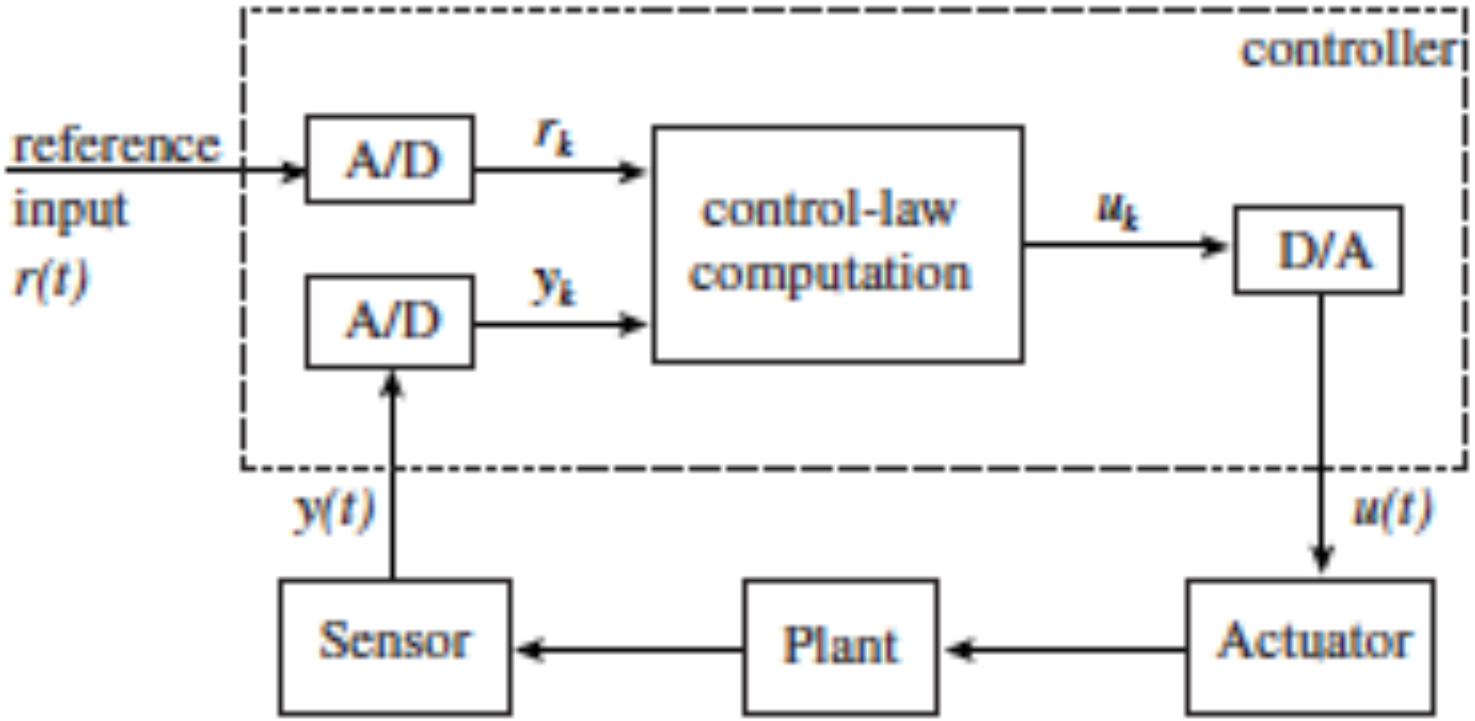


FIGURE 1–1 A digital controller.

Sampling rate

Typical "Control Law" in feedback control system

Set timer to interrupt periodically with period T;

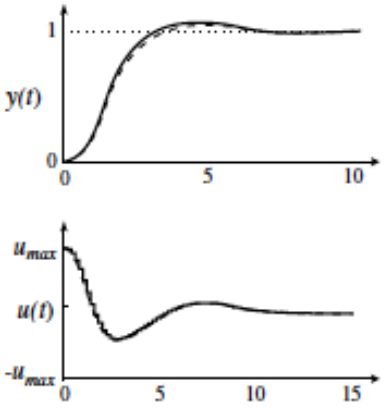
at each timer interrupt, do

do analog-to-digital conversion to get y ;

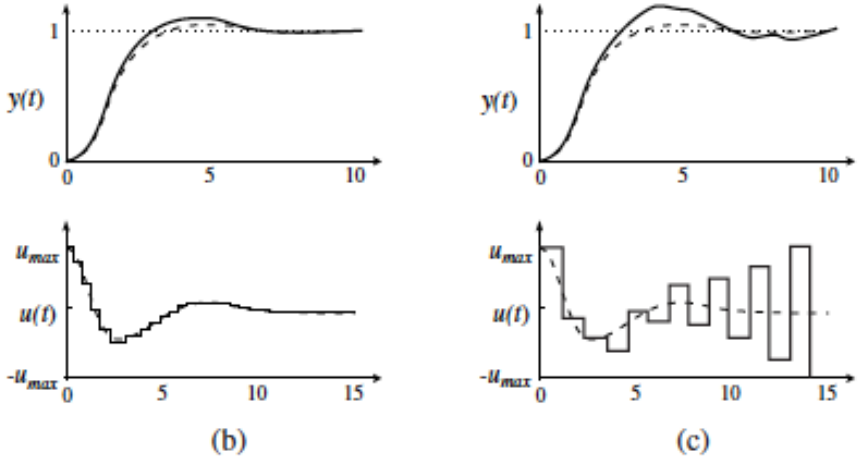
compute control output u ;

output u and do digital-to-analog conversion;

end do;



(a)



(b) (c)

FIGURE 1–2 Effect of sampling period.

More complex control law

Set timer to interrupt periodically with period T ;

at each clock interrupt, do

- sample and digitize sensor readings to get measured values;
- compute control output from measured and state-variable values;
- convert control output to analog form;
- estimate and update plant parameters;
- compute and update state variables;

end do;

State update

- Deadbeat controller

$$u_k = \alpha \sum_{i=0}^{k} (r_i - y_i) + \sum_{i=0}^{k} \beta_i x_i$$

- Kalman filter

$$\bar{x}_k = \bar{x}_{k-1} + K_k(y_k - \bar{x}_{k-1})$$

$$K_k = \frac{P_k}{\sigma_k^2 + P_k}$$

$$P_k = E[(\bar{x}_k - x)^2] = (1 - K_{k-1})P_{k-1}$$

$$\mathbf{y}_k = \mathbf{A}\mathbf{x}_k + \mathbf{e}_k$$

$$\bar{\mathbf{x}}_k = \bar{\mathbf{x}}_{k-1} + \mathbf{K}_k(\mathbf{y}_k - \mathbf{A}\bar{\mathbf{x}}_{k-1})$$

High-level controls – flight control

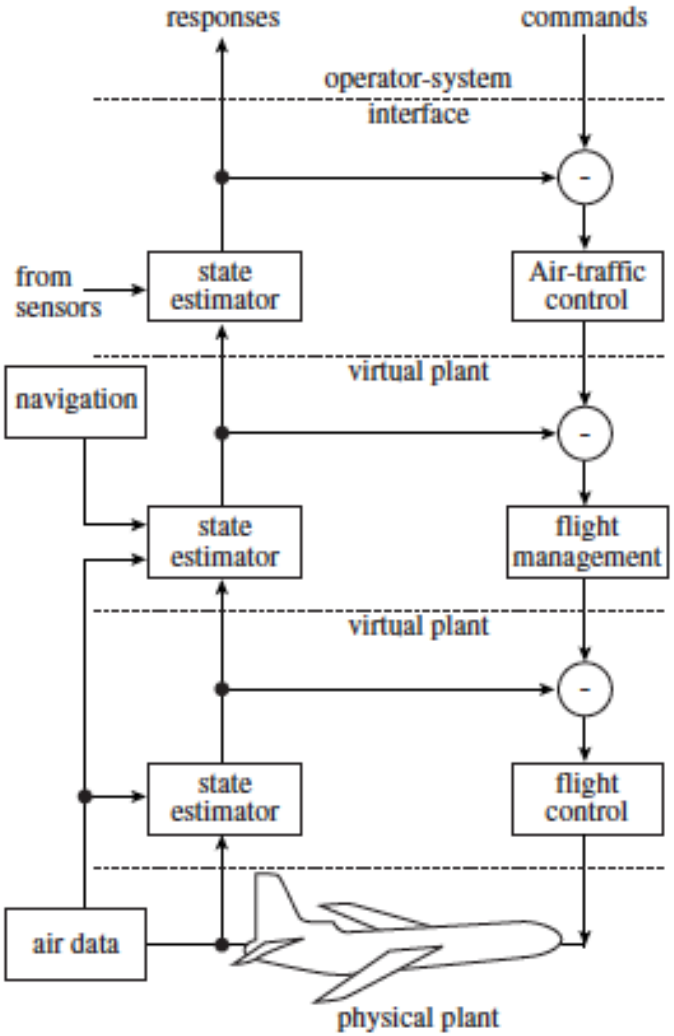


FIGURE 1–4 Air traffic/flight control hierarchy.

Real time commands and control

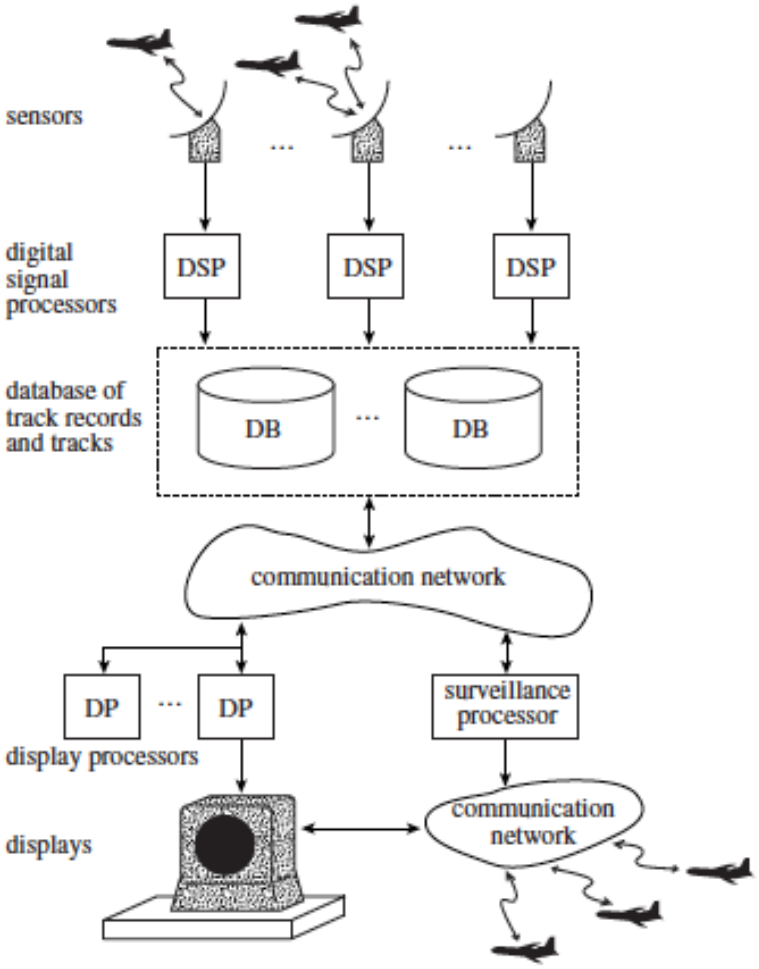


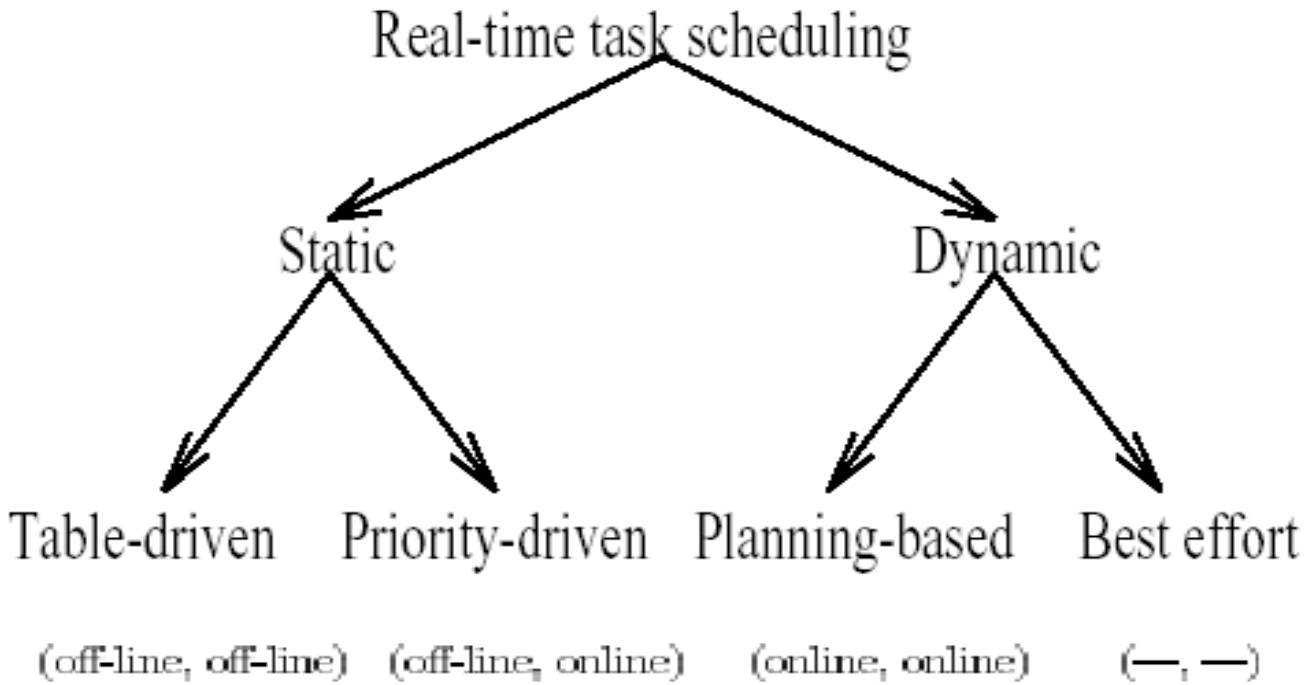
FIGURE 1–5 An architecture of air traffic control system.

Real-Time Systems - Issues

- Resource Management (RM) Issues
  - Scheduling, Fault-tolerance, Resource reclaiming, Communication
- Architectural Issues
  - Computing subsystem, Communication subsystem, I/O subsystem
- Software Issues
  - Requirements, specification, and verification, Real-time languages, Real-time databases

Real-time Scheduling Paradigms – RM Issue

- Allocate time slots for tasks onto processor(s).
- [i.e., Where and When a given task executes]
- Objective: **predictably meeting task deadlines.**
- (schedulability check, schedule construction)



Preemptive vs Non-preemptive scheduling

- **Preemptive Scheduling**
  - Task execution is preempted and resumed later
  - Preemption occurs to execute higher priority task.
  - Offers higher schedulability
  - Involves higher scheduling overhead due to context switching

- **Non-preemptive Scheduling**
  - Once a task starts executing, it completes its full execution
  - Offers lower schedulability
  - Less overhead due to less context switching

Optimal scheduling

- A **static scheduling** algorithm is said to be optimal if, for any set of tasks, it always produces a feasible schedule (i.e., a schedule that satisfies the constraints of the tasks) whenever any other algorithm can do so.

- A **dynamic scheduling** algorithm is said to be optimal if it always produces a feasible schedule whenever a static algorithm with complete prior knowledge of all the possible tasks can do so.

- Static scheduling is used for scheduling periodic tasks, whereas dynamic scheduling is used to schedule both periodic and aperiodic tasks.

Architectural Issues

- Predictability in: Instruction execution time, Memory access, Context switching, Interrupt handling.
- RT systems usually avoid caches and superscalar features.
- Support for error handling (self-checking circuitry, voters, system monitors).
- Support for fast and reliable communication (routing, priority handling, buffer and timer management).

.....

- Support for scheduling algorithms (fast preemptability, priority queues).
- Support for RTOS (multiple contexts, memory management, garbage collection, interrupt handling, clock synchronization).
- Support for RT language features (language constructs for estimating worst-case execution time of tasks).

Requirement, Specification, Verification

- **Functional requirements :** Operation of the system and their effects.
- **Non-Functional requirements :** e.g., timing constraints.
- F & NF requirements must be precisely defined and together used to construct the specification of the system.

.....

- A **specification** is a mathematical statement of the properties to be exhibited by a system. It is abstracted such that
  - it can be checked for conformity against the requirement.
  - its properties can be examined independently of the way in which it will be implemented.

- The usual approaches for specifying computing system behavior entail enumerating events or actions that the system participates in and describing orders in which they can occur. It is not well understood how to extend such approaches for real-time systems.

Real-time Languages

- Support for the management of time
  - Language constructs for expressing timing constraint, keeping track of resource utilization.
- Schedulability analysis
  - Aid compile-time schedulability check.
- Reusable real-time software modules
  - Object-oriented methodology.
- Support for distributed programming and fault-tolerance

Real-time Databases

Conventional database systems

- Disk-based.
- Use transaction logging and two-phase locking protocols to ensure transaction *atomicity* and *serializability*.
- These characteristics preserve data integrity, but they also result in relatively slow and unpredictable response times.

Real-time database system, issues include:

- transaction scheduling to meet deadlines.
- explicit semantics for specifying timing and other constraints.
- checking the database system's ability of meeting transaction deadlines during application initialization.

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

Decomposition and Abstraction

- Decomposition — the systematic breakdown of a complex system into smaller and smaller parts until components are isolated that can be understood and engineered by individuals and small groups
  TOP DOWN DESIGN

- Abstraction — Allows detailed consideration of components to be postponed yet enables the essential part of the component to be specified
  BOTTOM UP DESIGN

Modules

- A collection of logically related objects and operations
- Encapsulation — the technique of isolating a system function within a module with a precise specification of the interface
  - information hiding
  - separate compilation
  - abstract data types
- How should large systems be decomposed into modules?

The answer to this is at the heart of all Software Engineering!

Fault sources

Four sources of faults which can result in system failure:

- Inadequate specification
- Design errors in software
- Processor failure
- Interference on the communication subsystem

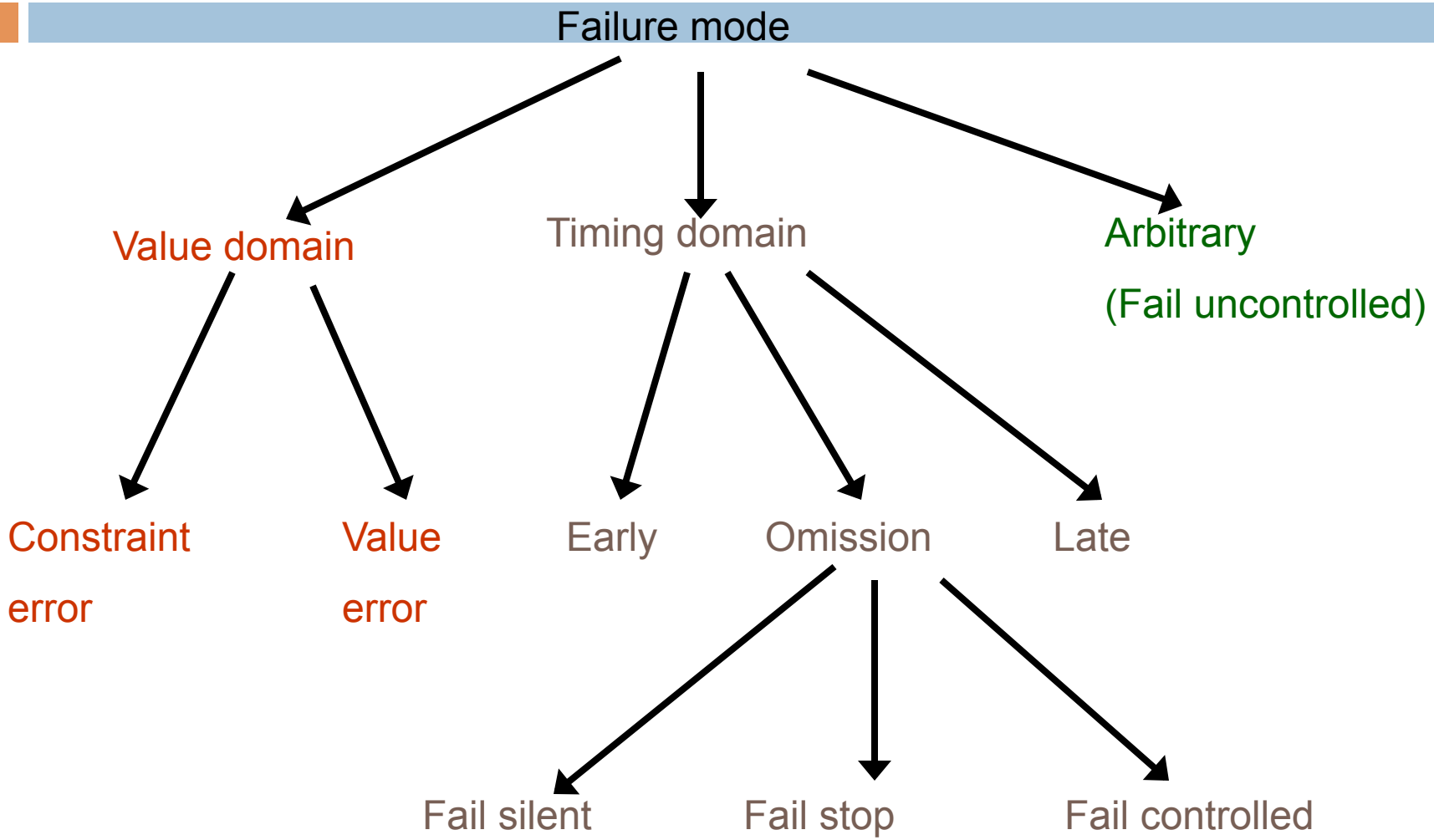
Reliability, Failure and Faults

- 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 produces 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

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
- 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

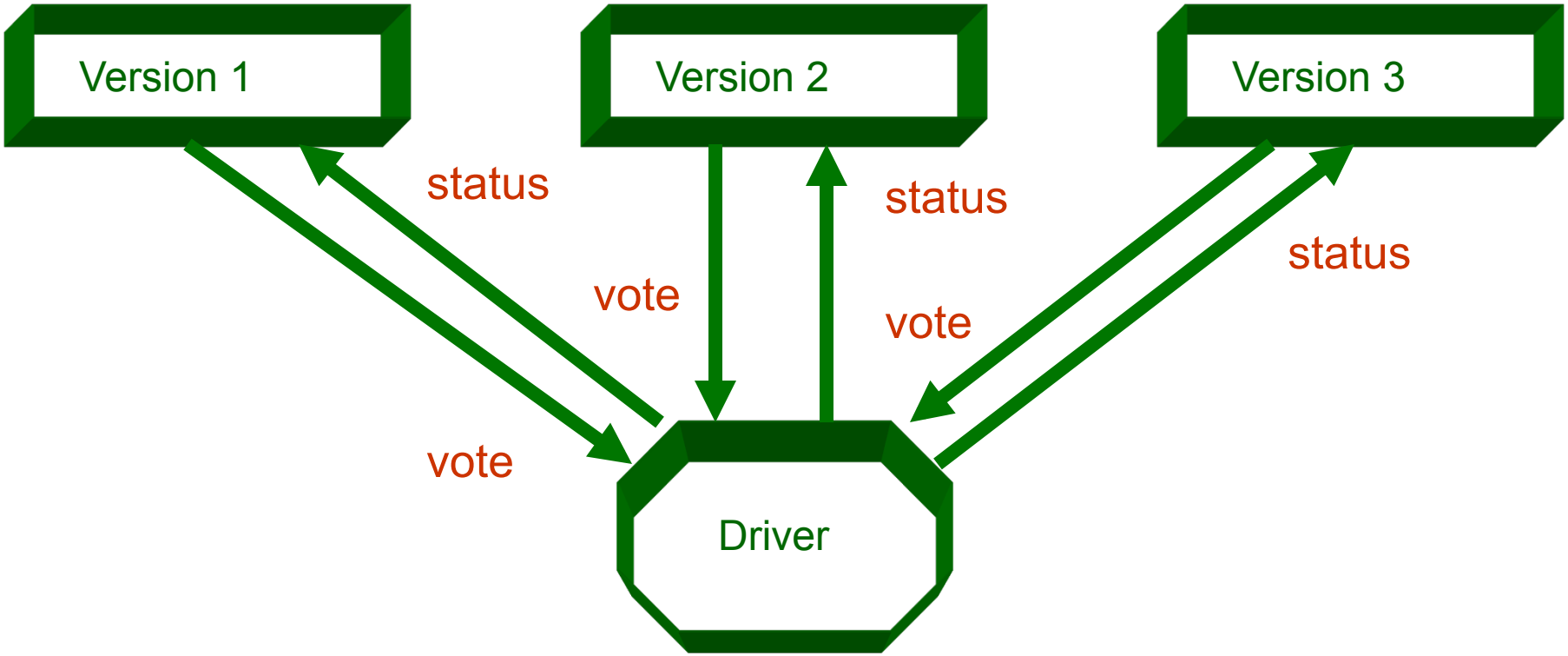
Software Fault Tolerance

- 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

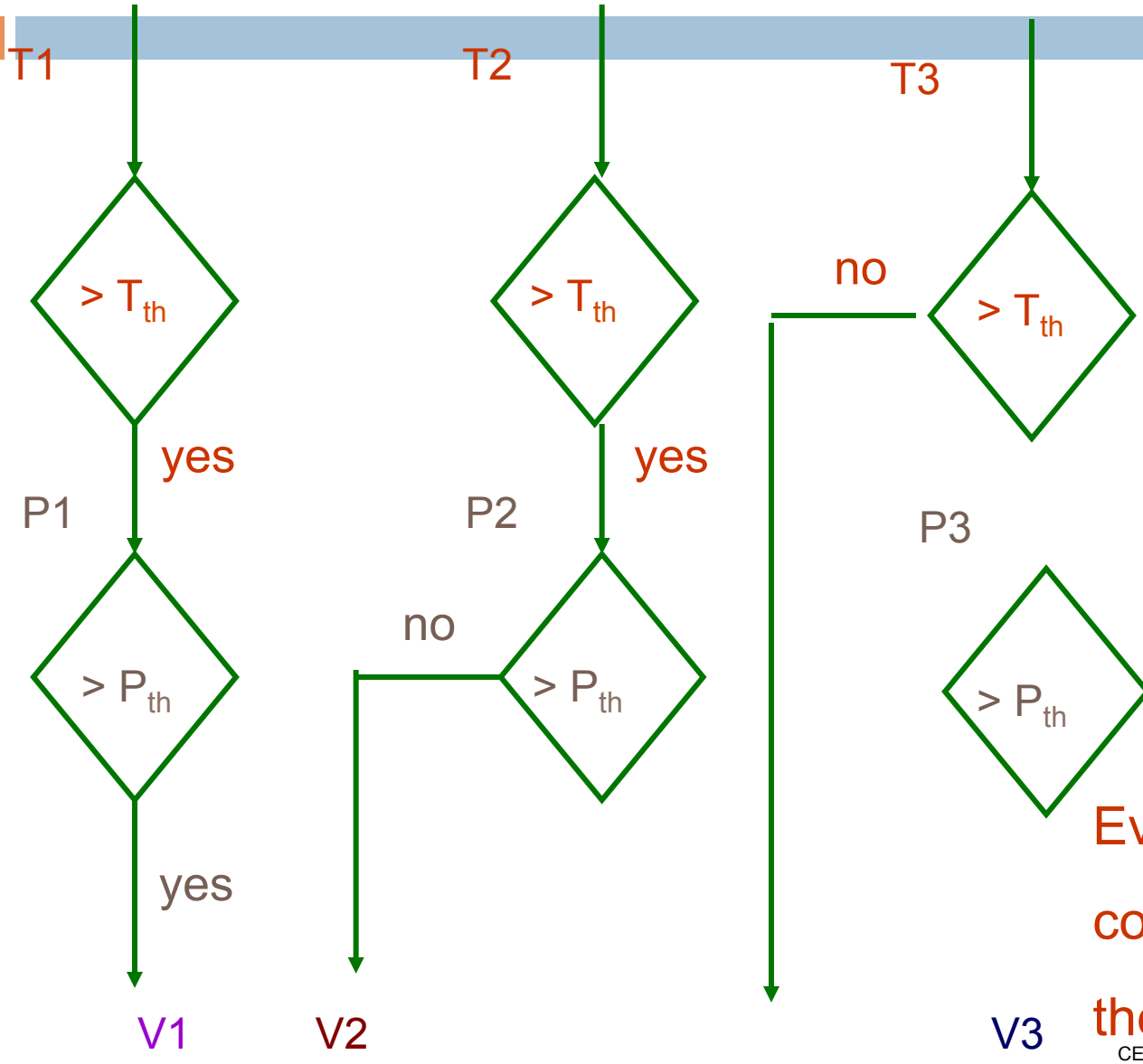
N-Version Programming



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)

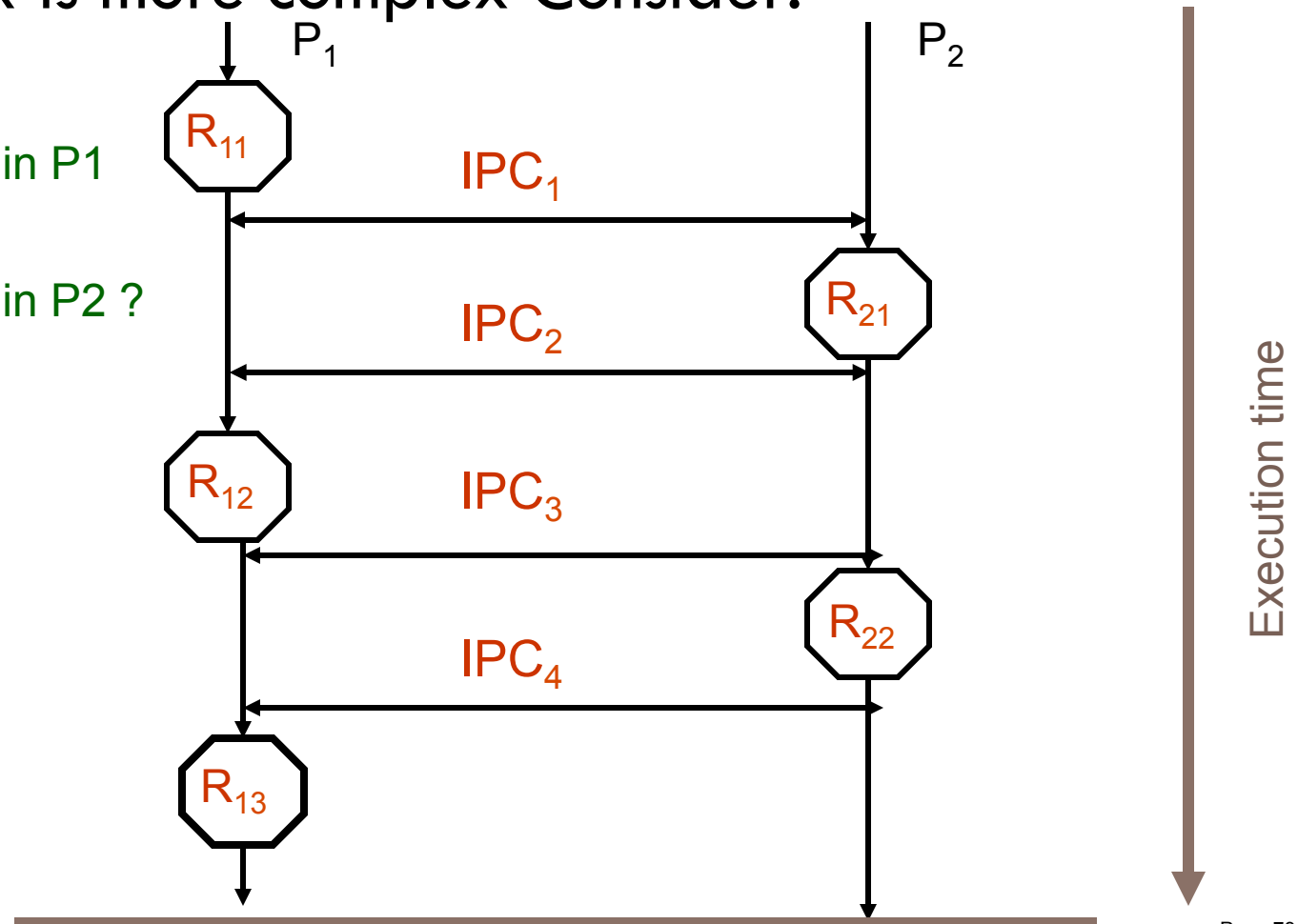
- BER relies on restoring the system to a previous safe state and executing an alternative section of the program
- This has the same functionality but uses a different algorithm (c.f. N-Version Programming) and therefore no fault
- The point to which a process is restored is called a recovery point 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

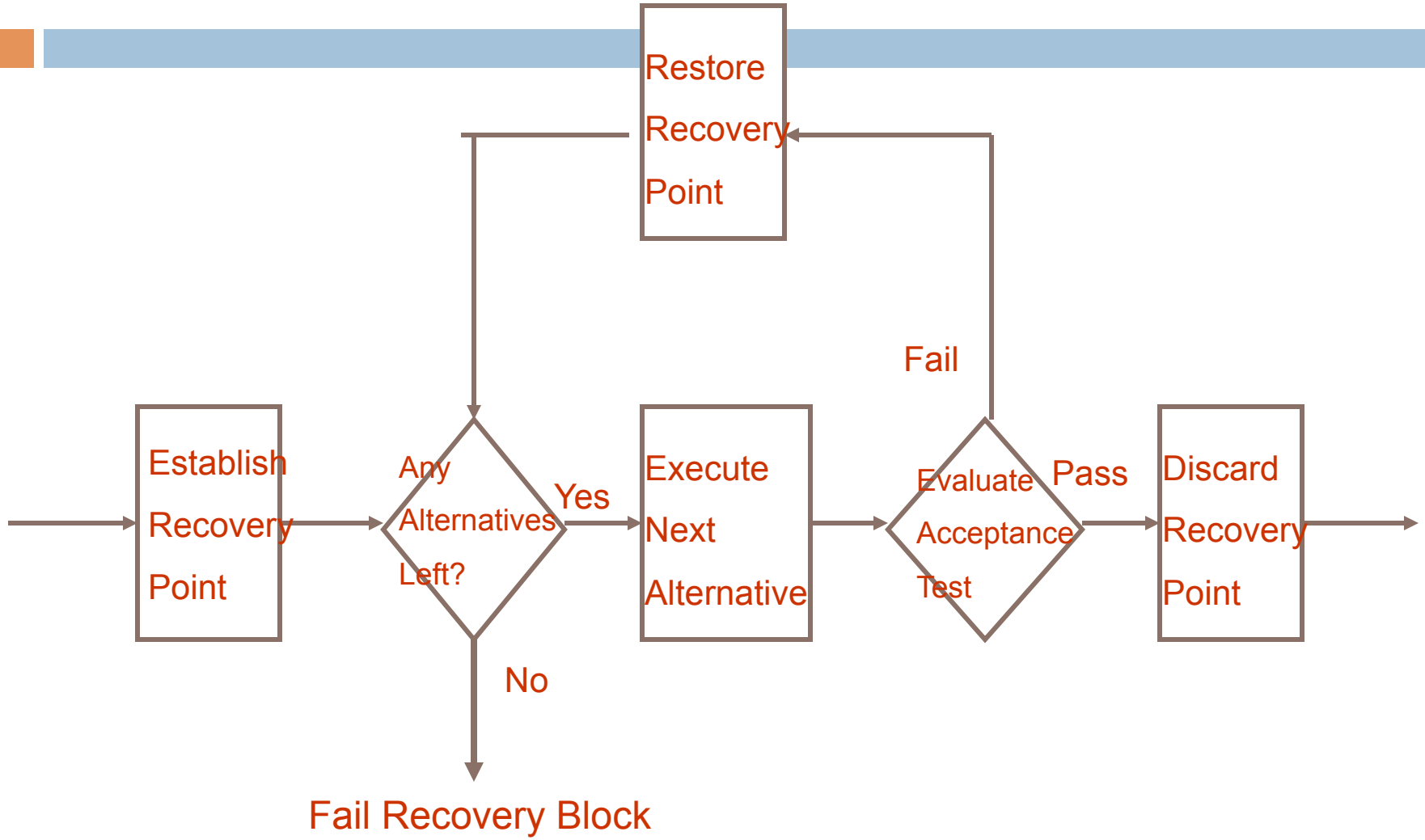
- Language support for BER
- At the entrance to a block is an automatic recovery point and at the exit an acceptance test
- The acceptance test is used to test that the system is in an acceptable state after the block's execution (primary module)
- If the acceptance test fails, the program is restored 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 * 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

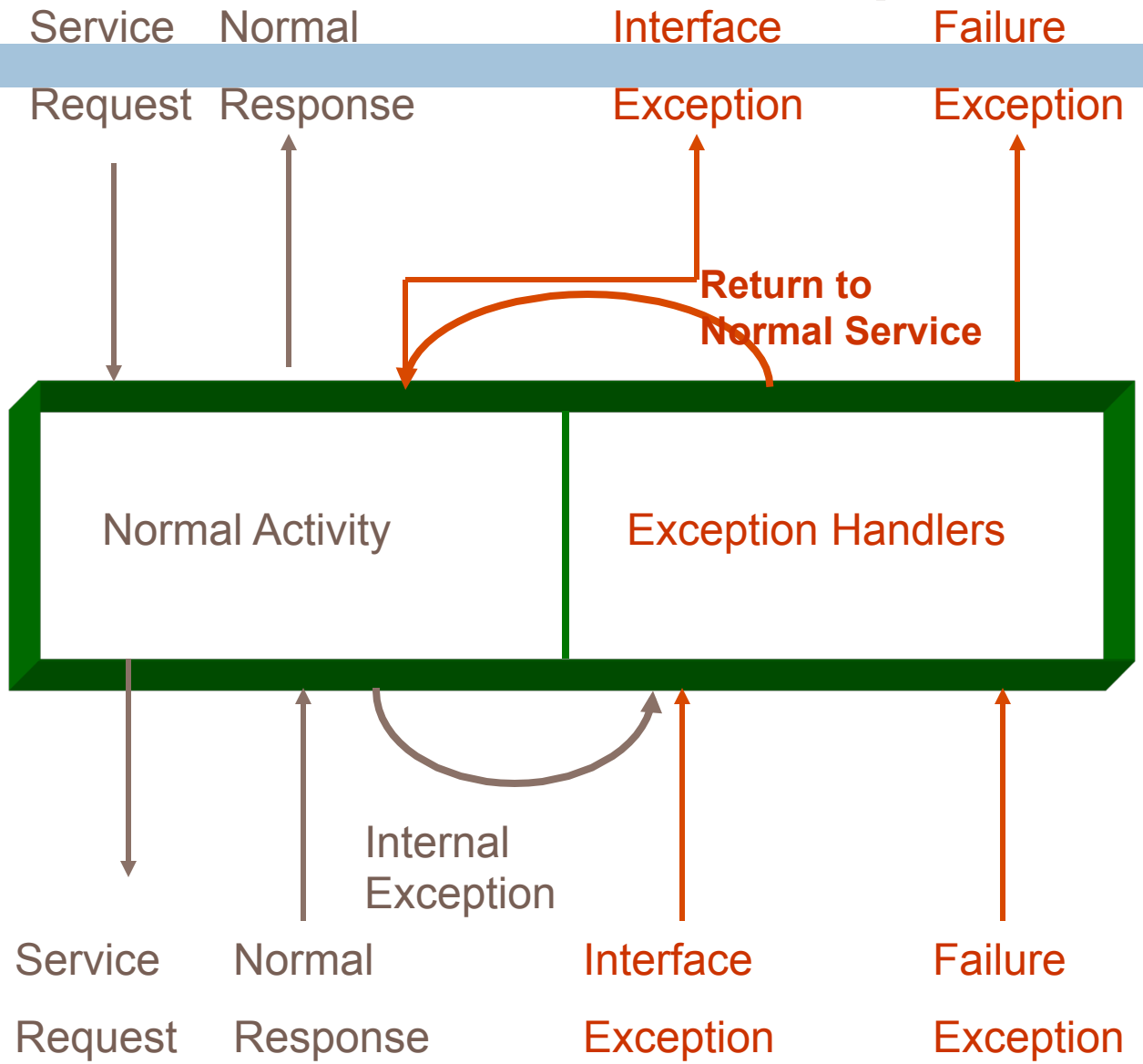
- An exception can be defined as the occurrence of an error
- Bringing an exception to the attention of the invoker of the operation which caused the exception, is called raising (or signally or throwing) the exception
- The invoker's response is called handling (or catching) the exception
- Exception 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



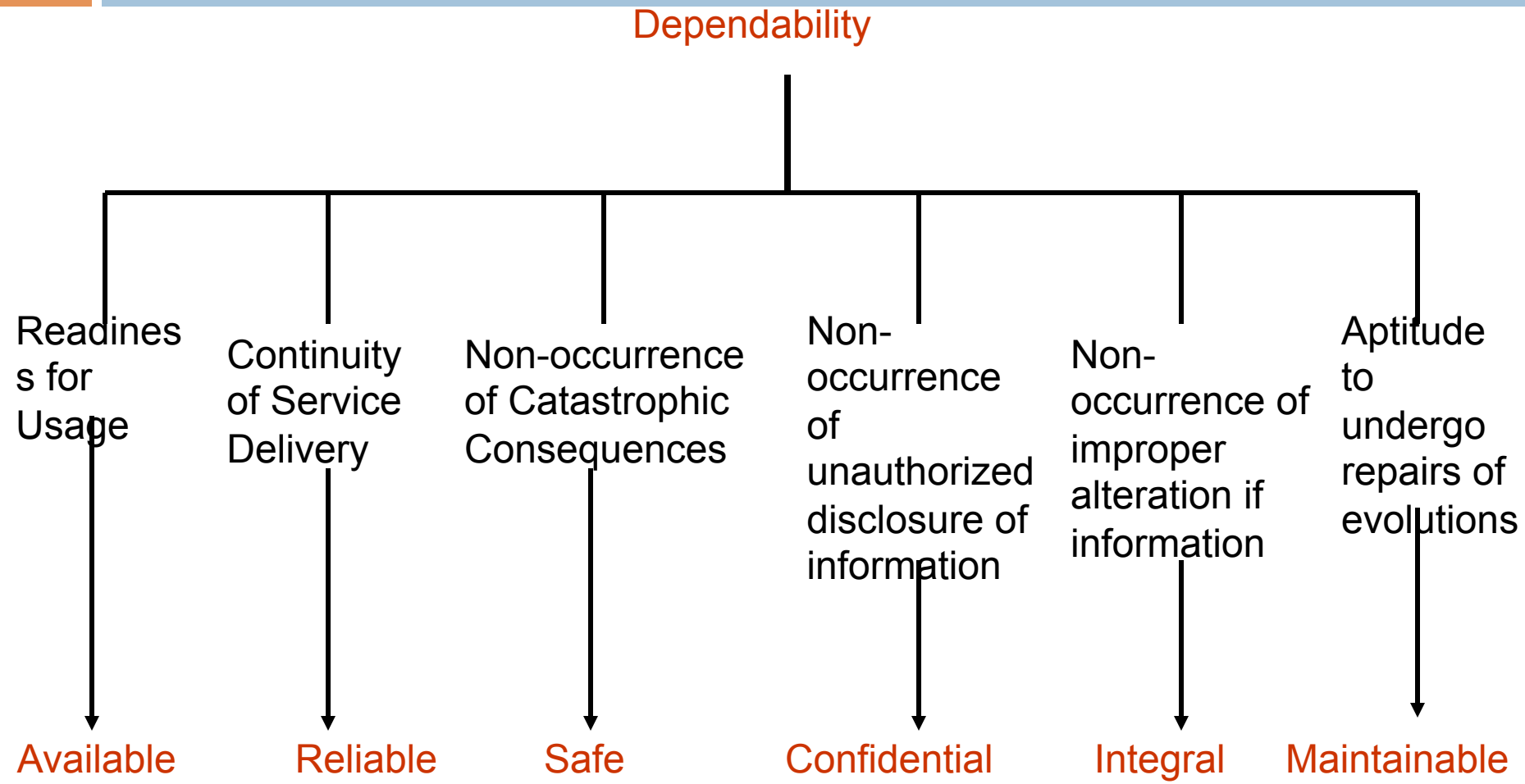
Safety and Reliability

- 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