Real-Time Kernels & Systems

Academic Year 2020/2021
Master Degree in Computer Science
Department of Mathematics
University of Padua
Tullio Vardanega, tullio.vardanega@unipd.it

Lecture topics

Bibliography
- J. Liu, "Real-Time Systems", Prentice Hall, 2000
- A. Burns and A. Wellings, "Analysable Real-Time Systems" Amazon Books, 2016
- State-of-the-art literature

1. Introduction
2. Scheduling basics
3. Fixed-priority scheduling
   a. Basic workload models
   b. Task interactions and blocking
   c. Exercises and further model extensions
4. Implementation issues
   a. Programming real-time systems (in Ada)
   b. A look under the hood
5. Distributed systems
   a. Worst case with offsets
   b. End-to-end analysis
6. Execution-time analysis
7. Multicore systems
   a. Initial reckoning
   b. Seeking the lost optimality
   c. Global resource sharing
8. Mixed-criticality systems
9. Predictable parallel programming

Protocol

- **Learning outcomes**
  - Realize the existence and the needs of software systems whose response time is critical to their use and consequently to their design
  - Understand the principles, methods, techniques and technology required to develop them so as to guarantee predictability
- **Instructional methods**
  - Web resources in complement to slide desks and presentations
  - Reciprocal feedback
    - Instructor to student, about incremental (self-)assignments
    - Student to instructor, about the progression of learning
  - Interaction
    - Posts in Moodle's billboard for this class, for all that must be public
    - Email otherwise

1. Introduction

Where we make some initial acquaintance with what real-time systems are, and why they came about, and then take a first look at their abstract concept

Initial intuition /1

- **Real-time system /1**
  - An aggregate of computers, I/O devices and *application-specific software*, characterized by
    - Continuous interaction with the external environment
    - To control it after mission-specific goals
    - Capturing the variations of the environment state and reacting to in a timely fashion
  - Comprised of system activities subject to timing constraints
    - Reactivity, accuracy, duration, completion, responsiveness: all dimensions of ***timeliness***
  - System activities inherently *concurrent* and increasingly *parallel*
  - The satisfaction of all system constraints must be proved

Concurrency vs. parallelism

- Concurrent programming allows using multiple logical threads of control to reflect *cohesively* the collaborative structure of the solution
  - I do "my" bit, you do "yours"; our respective waiting is not wasteful
  - Threads form the architecture: they are long-lived
- Parallel programming promotes a divide-and-conquer logic to solve a problem, with multiple threads that work *independently* on the problem space
  - I work as fast as I can and know nothing about you
  - Threads are mindful of throughput: they are short-lived

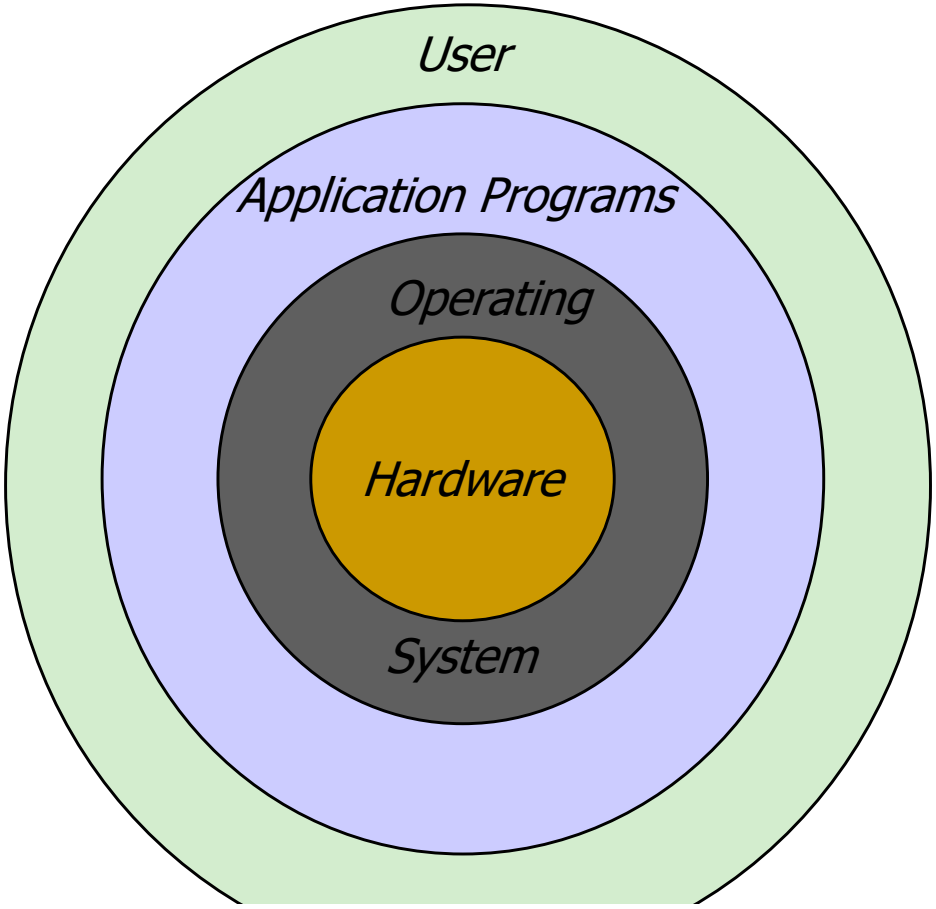
Initial intuition /2

- **Real-time system /2**
  - Operational correctness does not solely depend on the logical result but also on the time at which the result is produced
    - The computed response has an application-specific utility
    - Correctness is defined in the value domain and in the time domain
    - A logically-correct response produced later than due may be bad
- **Embedded system**
  - The computer and its software are fully immersed in an *engineering system* comprised of the external environment subject to its control
    - Defined in terms of physical attributes that can be interacted with

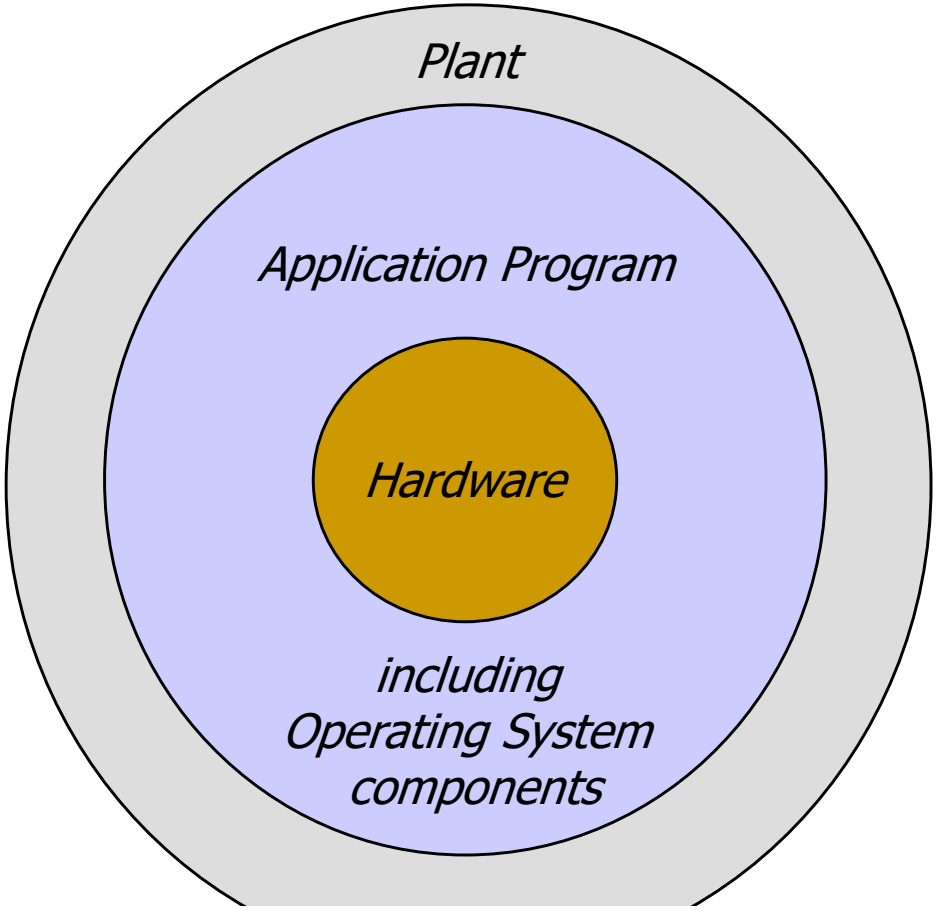
Embedded systems vs. CPS

- **Cyber-physical systems** (CPS) are the new frontier in real-time systems research: what is new in them?
  - Traditional embedded systems are *closed*
    - Their interaction with the environment is bounded to selected parameters only, and the system operation varies solely within a fixed set of modes toward given mission objectives
  - Cyber-physical systems are intrinsically *open*
    - The environment that forms part of the system is highly dynamic, has broad confines and cannot be reduced to few bounded parameters
    - The functional needs of the system may vary rapidly over time

Embedded system /1

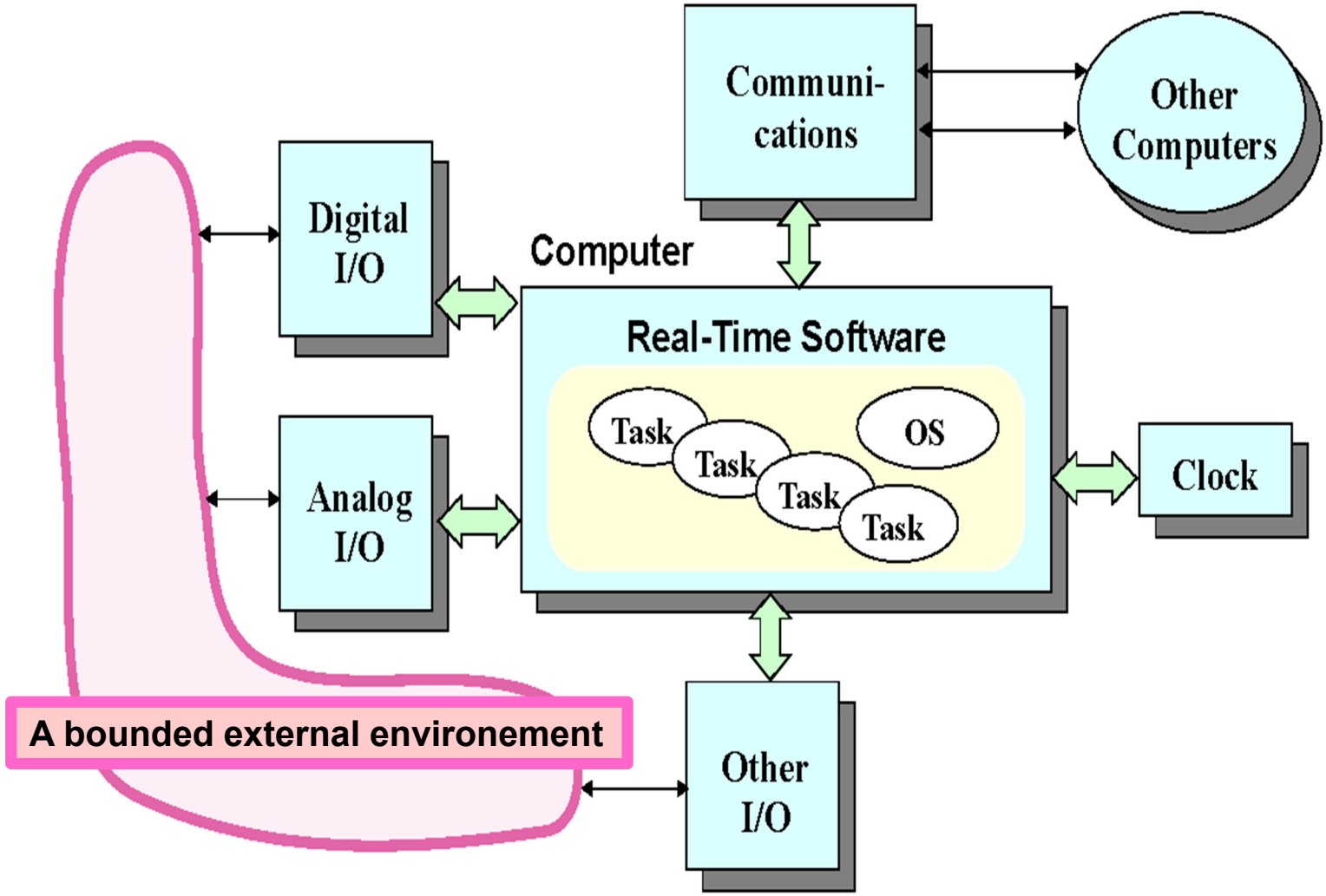


Typical General-Purpose Computing Configuration



Typical Embedded Computing Configuration

Embedded system /2



Cyber-physical system

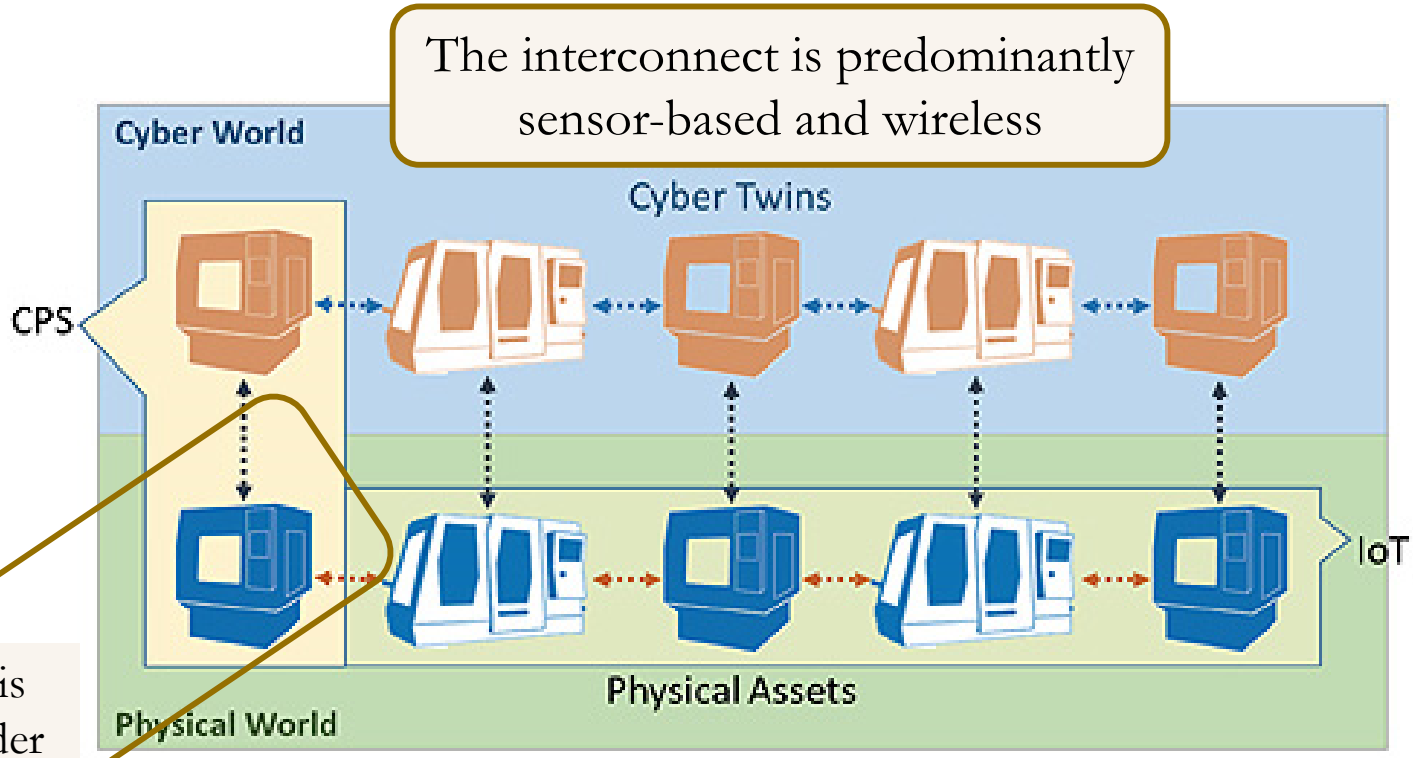


Image credits to www.designwordonline.com

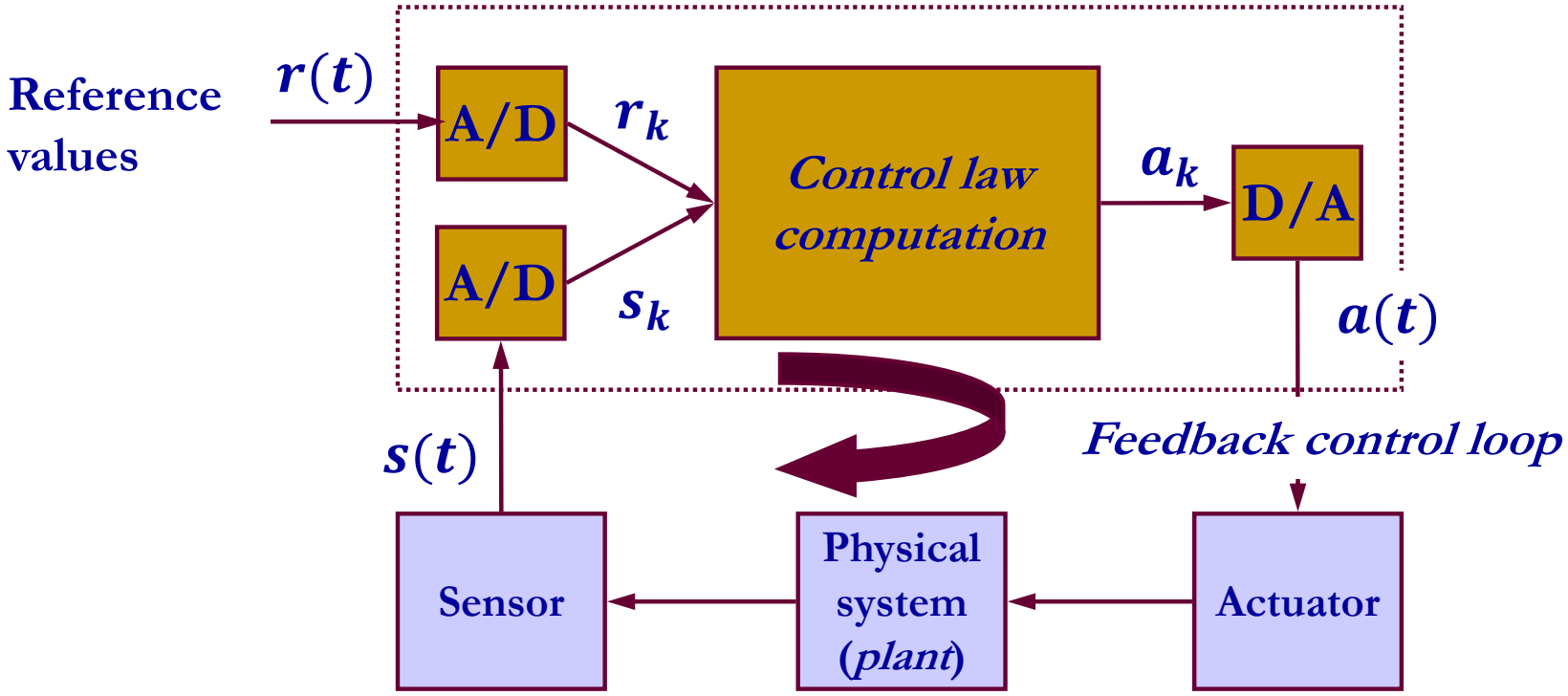
The boundaries of this "world" are much wider and more dynamic than for traditional embedded systems

Cybernetics: now and then

- Born in 1948 as *the science of control systems*, prerequisite to the **automation of control**
  - From Greek's **κυβερνητης** (Latin's "gubernator"), steersman
    - *Sensing* the external (physical) environment
    - *Computing* the distance from the expected status
    - *Actuating* devices to effect the system or the environment, so as to reduce that distance
  - Every control action performed on the external environment causes (positive or negative) *feedback*
  - The goal of cybernetics is to calibrate control actions so that the system objective is reached with bounded feedback

Automation of control /1

- A digital system comprised of sensors and actuators



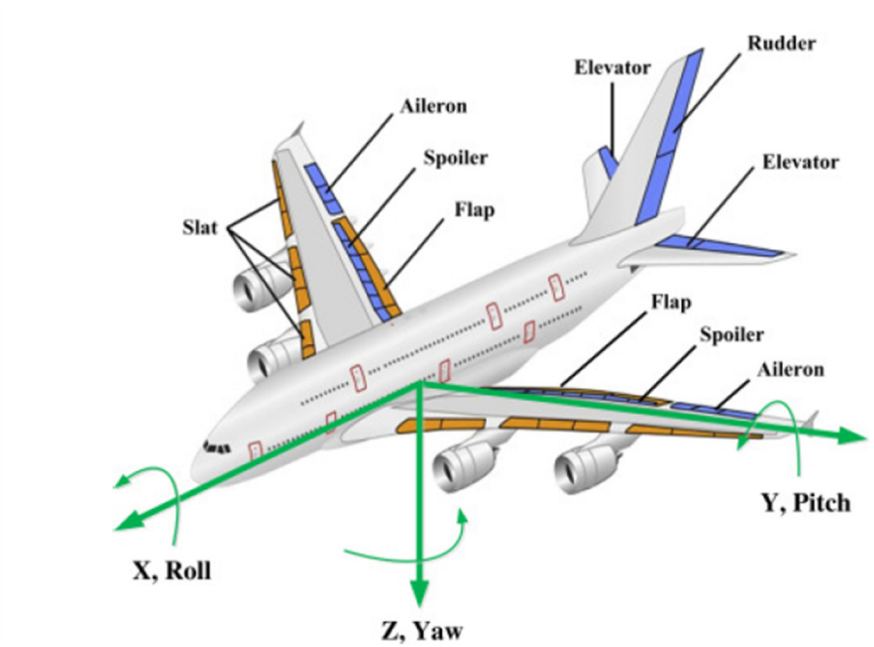
Gradient over time

$$a_k = a_{k-2} + \alpha(r_k - s_k) + \beta(r_{k-1} - s_{k-1}) + \gamma(r_{k-2} - s_{k-2})$$

Automation of control /2

- Factors of influence
  - Quality of response (***responsiveness***)
    - Sensor sampling is typically periodic with period T
      - For the convenience of control theory
    - Actuator commanding is produced at the time of the next sampling
      - As part of feedback control mathematics
    - System stability degrades with the width of the sampling period
  - Plant ***capacity***
    - Good-quality control reduces oscillations
    - A system that needs to react rapidly to environmental changes and is capable of it within $rise\ time\ R$ requires higher frequency of actuation and thus faster sampling → hence shorter T
    - A rule-of-thumb R/T ratio normally ranges [10 .. 20]

Automation of control /3



Any three-dimensional rotation can be described as a sequence of $roll\ (x)$, $pitch\ (y)$, $yaw\ (z)$ rotations (Euler angles)

- Complex systems must support multiple distinct periods T_i
  - A **harmonic** relation among all T_i does help
    - This removes the need for concurrency of execution in the relevant computations
    - But it causes coupling between possibly unrelated control actions which is a poor architectural choice
  - There may be diverse components of speed
    - *Forward, side slip, altitude*
  - As well as diverse components of rotation
    - *Roll, pitch, yaw*
  - Each of them requires separate control activities each performed at a specific rate

Automation of control /4

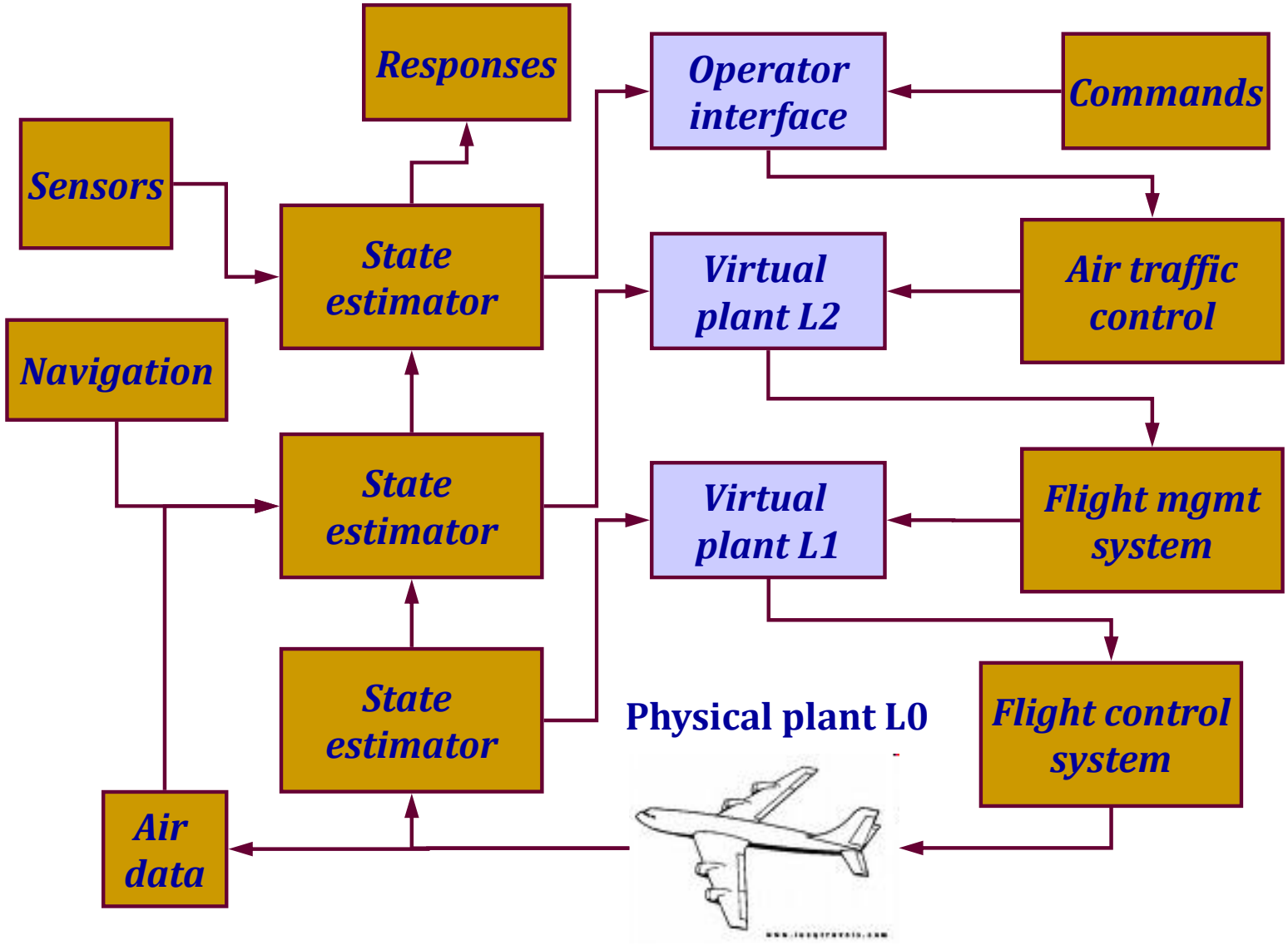
(Artificially) harmonic multi-rate system

- 180 Hz cycle
  - Check all sensor data and select sources to sample
  - Reconfigure system in case of read error
- 90 Hz cycle
  - Perform control law for pitch, roll, yaw (internal loop)
  - Command actuators
  - Perform sanity check
- 30 Hz cycle
  - Perform control law for pitch, roll, yaw (external loop) and integration
- 30 Hz cycle
  - Capture operator keyboard input and choice of operation model
  - Normalize sensor data and transform coordinates; update reference data
- *Can you figure how those activities can progress together?*

Automation of control /5

- Command and control systems are often organized in a hierarchical fashion
  - At the lowest level we place the digital control systems that operate on the physical environment
  - At the highest level we place the interface with the human operator
    - The output of higher-level controllers becomes a reference value $\boldsymbol{r}(t)$ for lower-level controllers
  - The more composite the hierarchy the more complex the interdependence in the logic and timing of operation

Example: hierarchical control system



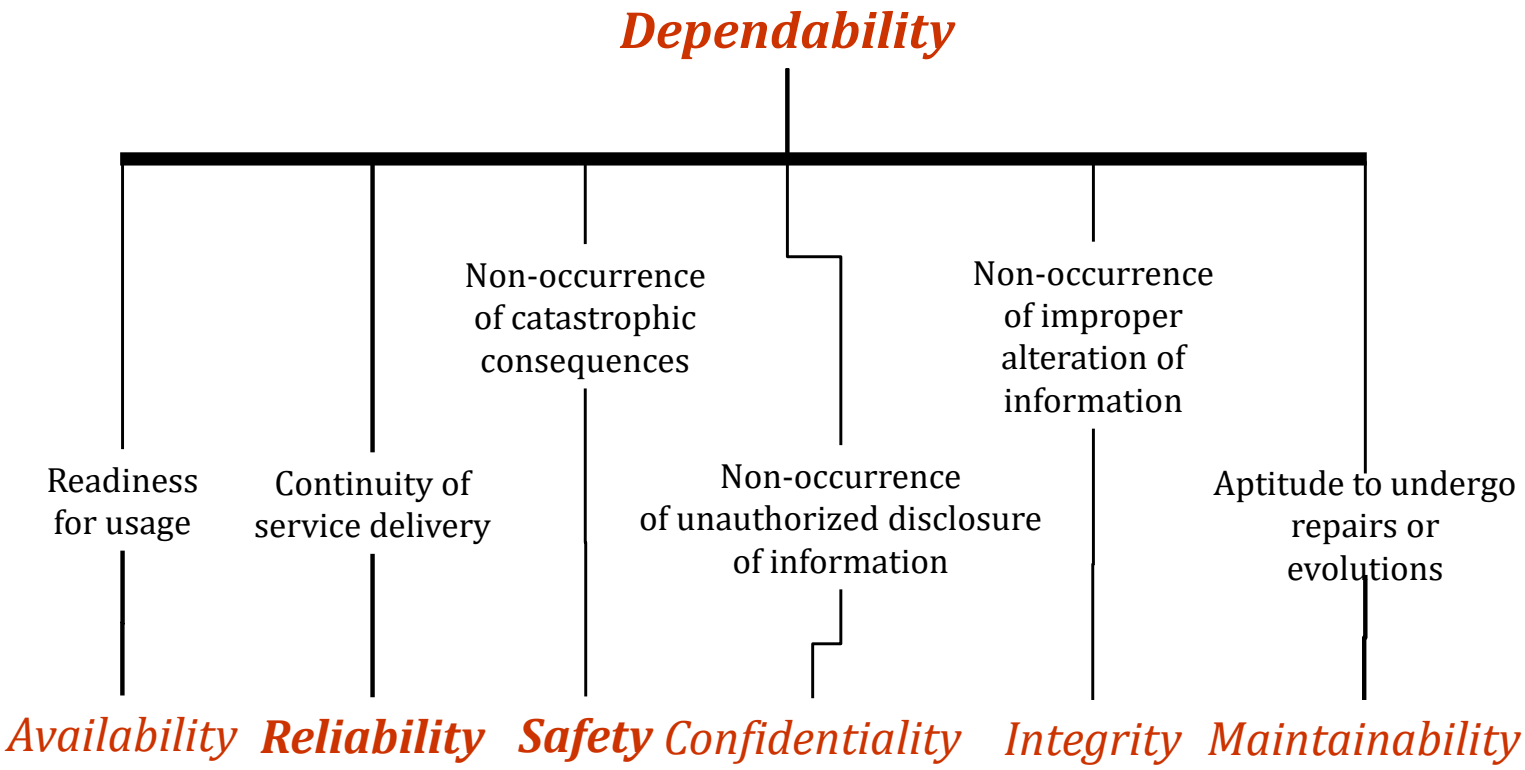
Application requirements

- A control system consists of (distributed) resources governed by a *real-time operating system*, RTOS
- The system design overall must meet stringent ***reliability*** requirements
  - Measured in terms of *maximum acceptable probability of failure*
  - For example: 10^{-9}/hour of flight for the Airbus A-3X0 control system
    - One failure allowed in 10^9 hours of flight ($> 114k$ years!)

RTS key characteristics /1

- **Complexity**
  - In algorithms, mostly because of the need to apply discrete control over analog and continuous physical phenomena
  - In development, mostly owing to more demanding verification and validation processes
- **Heterogeneity** of components and of processing activities
  - Multi-disciplinary engineering (spanning control, SW, and system)
- Extreme **variability** in size and scope
  - From tiny and pervasive (nanodevices) to very large (aircraft, plant)
  - In all cases, finite in computational resources
- Proven ***dependability***

Dependability attributes



RTS key characteristics /2

- Must respond to events triggered by the external environment as well as by the passing of time
  - Double nature: *event-driven* and *time-driven*
- Continuity of operation
  - A real-time embedded system must be capable of operating without (constant) human supervision
  - Nearly no keyboard-based interaction!
- Software architecture inherently concurrent and increasingly parallel
- Must be temporally ***predictable***
  - Need for static (off-line) verification of correct temporal behaviour
  - How does that relate to ***determinism***?

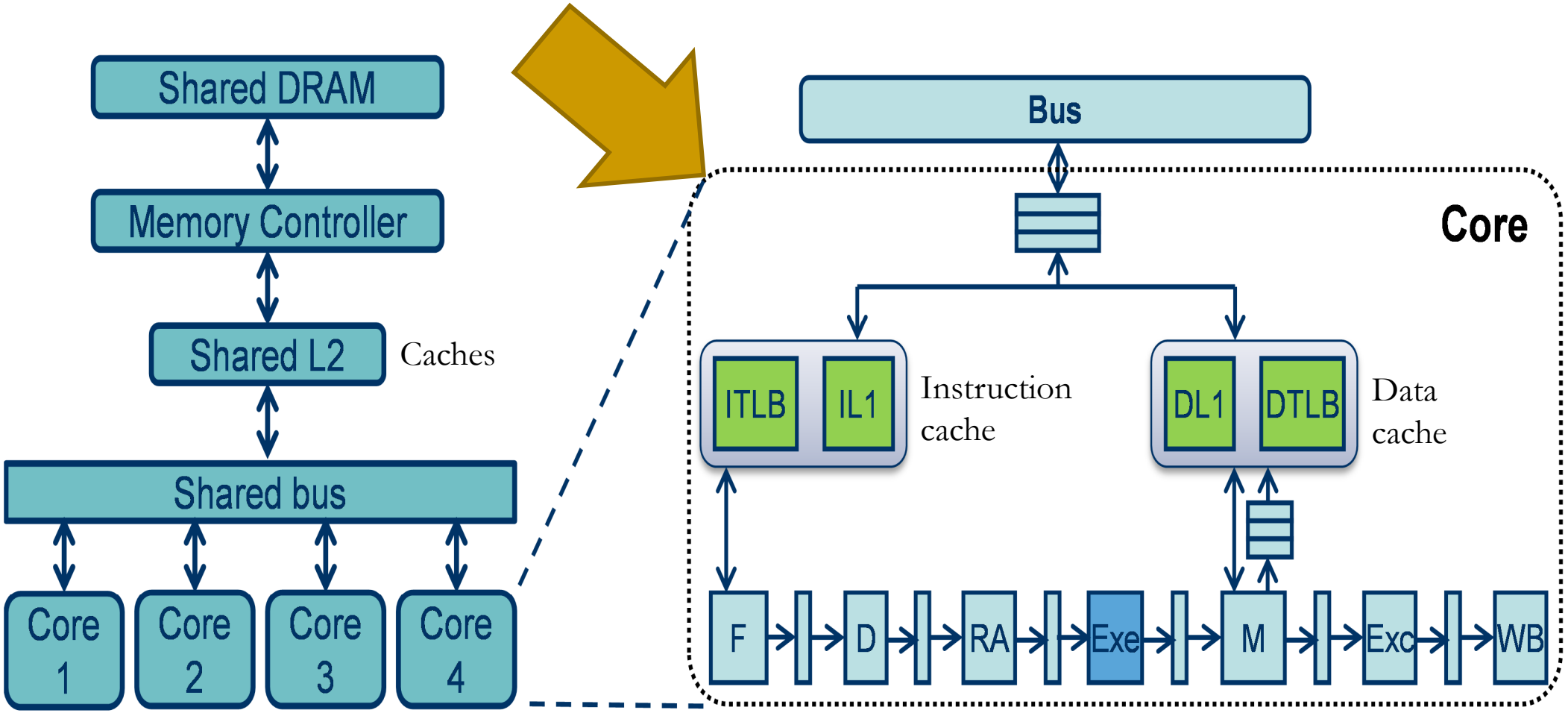
Predictability vs. determinism

- *Predictability* (what can be established a priori) may be regarded as a continuum
  - Its highest end is full a-priori knowledge
    - Which allows for deterministic reasoning, and yields absolute certainty
  - Its lowest end is total absence of a-priori knowledge
    - In the style of "See what happens …"
- Seeking predictability implies reasoning about kinds and degrees of uncertainty, pursuing an acceptable balance
  - We must reason conservatively
    - Considering *worst-case* conditions, which may be difficult to capture, as very rarely we have full a-priori knowledge of the factors of influence
    - At the same time, we must avoid exceedingly pessimistic reasoning

Meeting real-time requirements

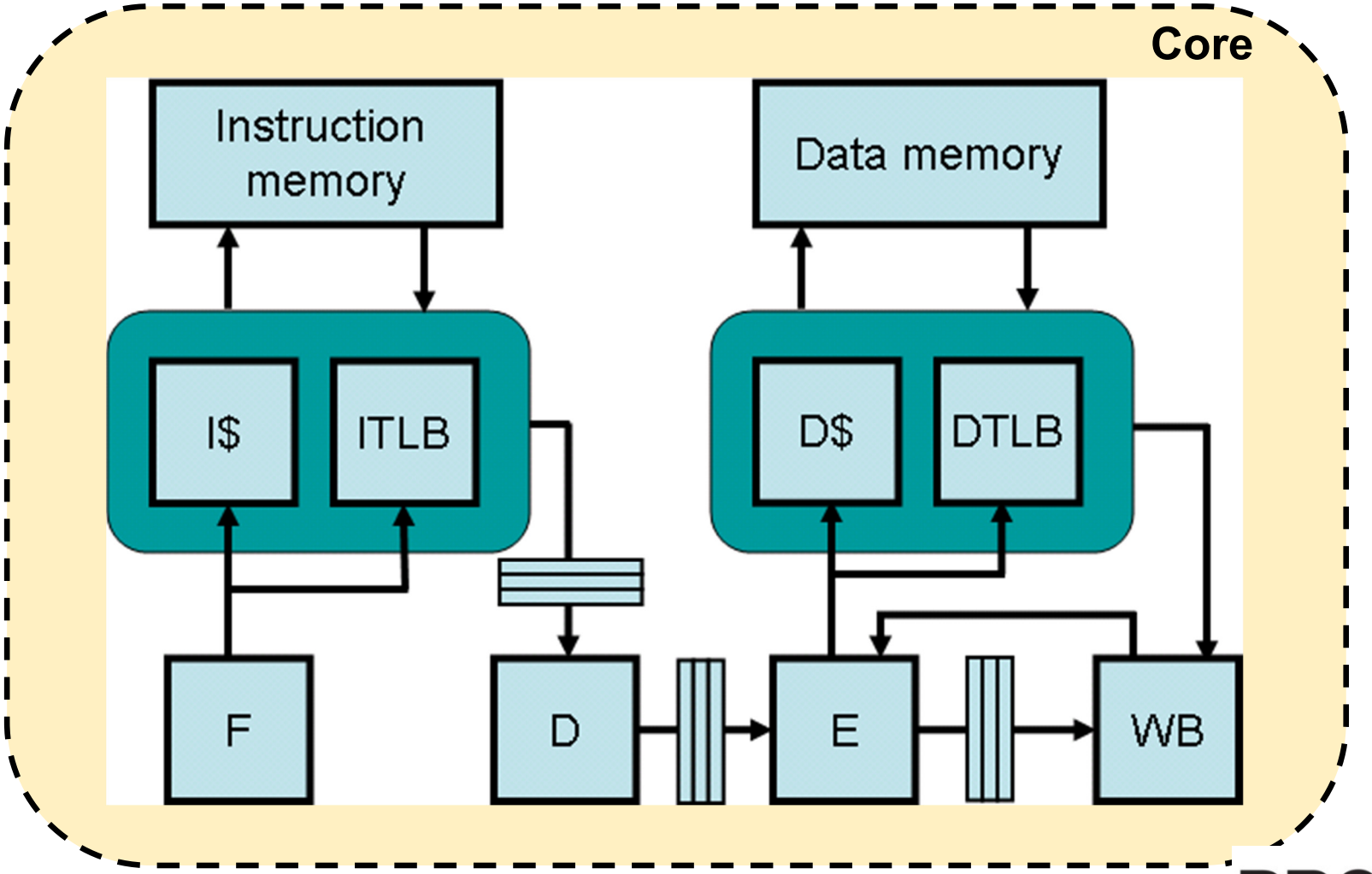
- Minimizing the application tasks' average response time may matter to general-purpose computing; it does **not** to RTS!
  - Real-time computing is **not** equivalent to fast computing
- Given real-time requirements and a HW/SW implementation, how can one show that those requirements are met?
  - Testing and simulation are **not** sufficient (obviously!)
  - Maiden flight of space shuttle, 12 April 1981: there was a 1/67 probability of a *transient overload* occurring at system initialization
  - It never showed up in testing; it did at launch
- The answer to that requires seeking system-level ***predictability***
  - This requires understanding the *worst case*
  - Which in turn requires understanding how execution works …

Understanding the processor /1



Courtesy of PROXIMA

Understanding the processor /2



Courtesy of PROXIMA

Definitions /1

- ***Task***
  - Concurrent unit of functional architecture
  - Issues one job at a time, until completion, to perform actual work
    - One such task is said to be *recurrent*
    - The tasks in a real-time system are typically recurrent: think of the body a task as an *endless loop*
- ***Job***
  - Called into execution by a task, following a given law of activation
  - Unit of work that competes for scheduler-controlled execution
    - Think of a job as a top-level (*always terminating*) procedure, which does the recurrent work of the issuing task …
  - Needs physical and logical *resources* to execute

Definitions /2

- ***Release time***
  - When it occurs, a job becomes eligible for execution
    - The corresponding trigger is called ***release event***
    - There may be some temporal delay between the arrival of the release event and when the scheduler recognizes the job as ready
  - May be set at some *offset* from system start time
    - For example to avoid congestion on access to the CPU
    - The offset of the *first* job of task τ to the system start time is named ***phase***, φ, and it is one of the attributes of τ

Definitions /3

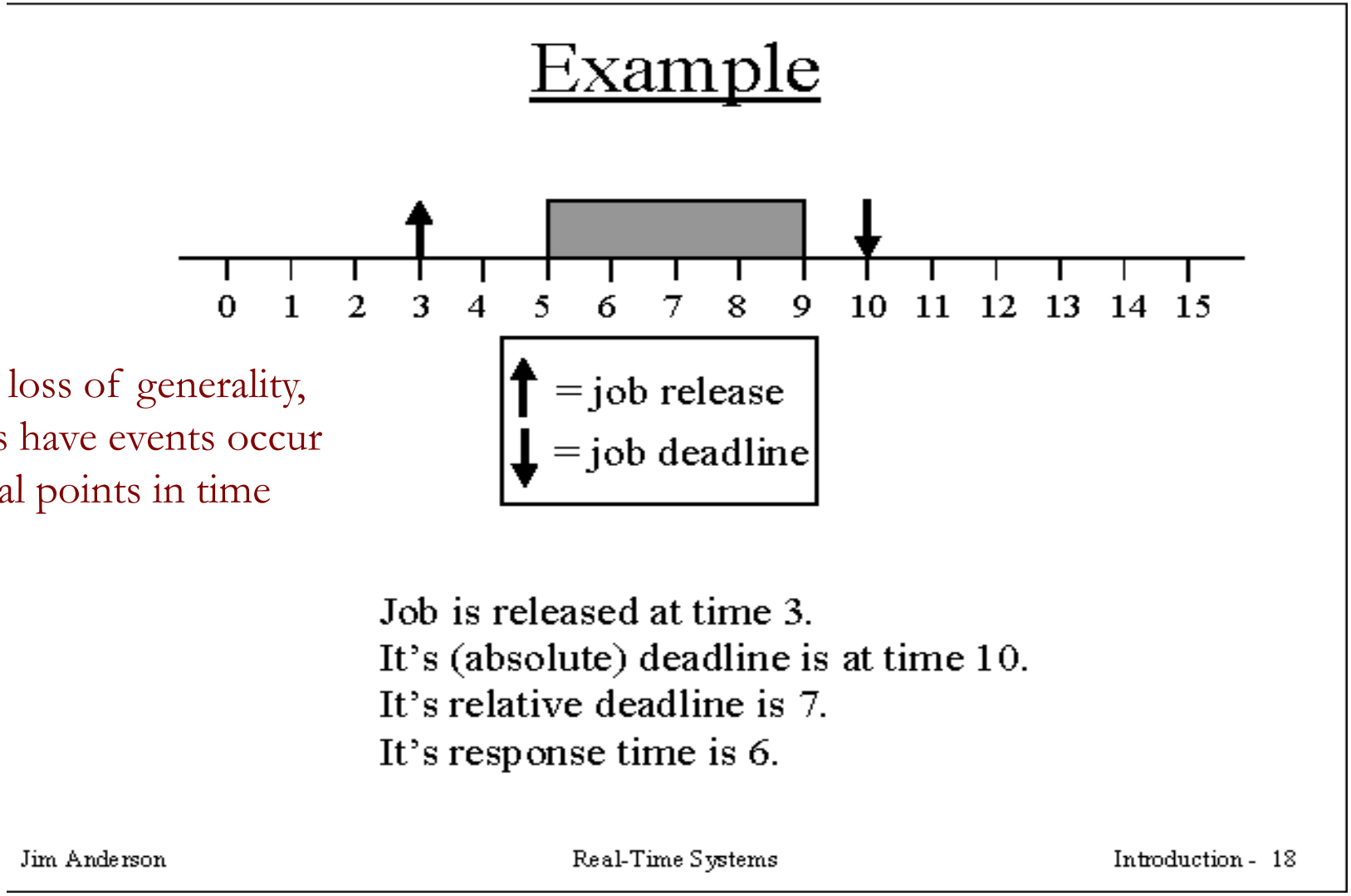
- ***Jitter***
  - Variability in the release time or in the time of input (data freshness) or output (stability of control)
- ***Inter-arrival time***
  - Separation between the release time of successive jobs which are not strictly periodic
    - Job is *sporadic* if a guaranteed minimum such value exists
    - Job is *aperiodic* otherwise
- ***Execution time, C***
  - For any job J_i, C_i may vary between a *best-case* (BCET) C_i^b and a *worst-case* (WCET) C_i^w

Definitions /4

- ***Deadline***
  - The (latest) time by which a job must complete its execution
  - May be < (*constrained*), = (*implicit*), > (*arbitrary*) than the next job's release time
- ***Response time***
  - The time span between the job's release and its actual completion

- The longest admissible response time for a job j_i is termed the job's *relative deadline*, D_i
- The algebraic summation of release time and relative deadline is termed *absolute deadline*, d_i

A timeline



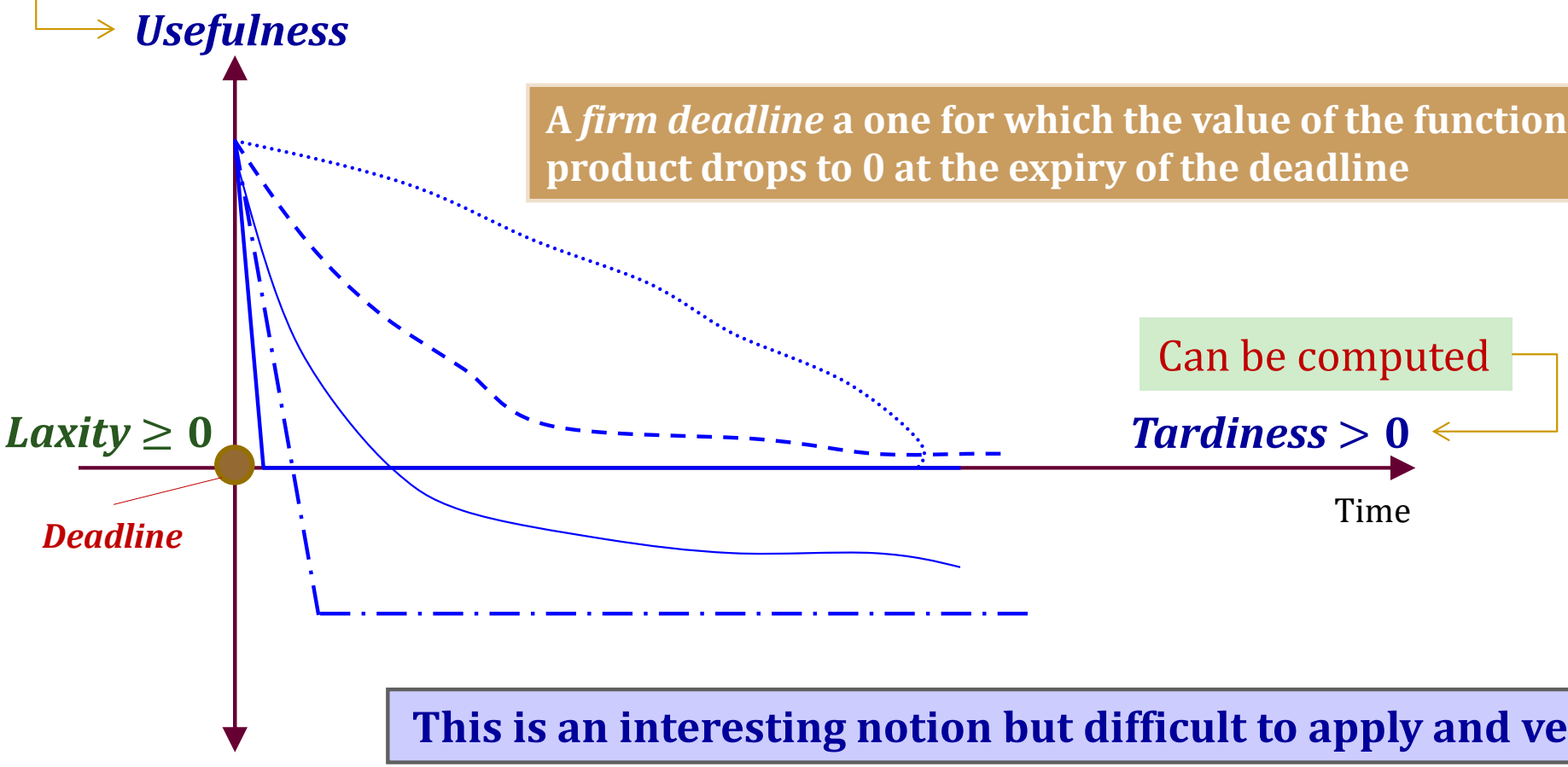
Without loss of generality, timelines have events occur at integral points in time

Definitions /5

- Deadlines are said to be **hard**
  - If the consequences of a job completing past its deadline are serious and possibly intolerable
  - Satisfaction of all deadlines must be proven off line
- Deadlines are said to be **soft**
  - If the consequences of a job occasionally completing past the assigned deadline are tolerable
  - The quantitative interpretation of "occasional" may be established in probabilistic terms or in terms of ***utility***
- Deadlines are said to be **firm**
  - When they are soft but have utility ≤ 0 past the deadline point, and therefore may cause damage if missed

Utility function

Difficult to quantify



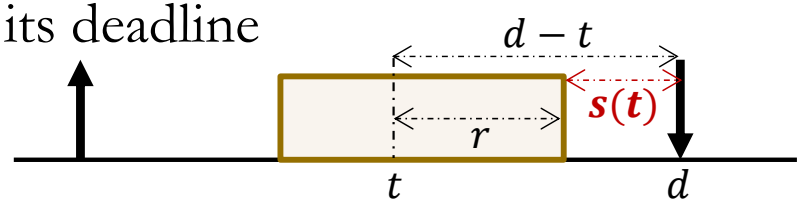
A *firm deadline* a one for which the value of the functional product drops to 0 at the expiry of the deadline

Can be computed

This is an interesting notion but difficult to apply and verify

Definitions /6

- ***Laxity*** (aka ***slack***)
  - $s(t) = (d - t) - r$ is the *slack* at time t of job J with deadline d and remaining time of execution r
    - A job with non-negative laxity meets its deadline



- ***Tardiness***
  - The span between a job's response time and its deadline
    - A job with negative laxity has tardiness
- ***Usefulness***
  - Value of (residual) utility of the job's computational product as a function of its tardiness

An initial taxonomy /1

- The prevailing classification stems from the traditional standpoint of control algorithms
- **Strictly periodic** systems
  - Harmonic multi-rate (artificially harmonized)
  - Polling for not-periodic events
- **Predominantly** (but not exclusively) **periodic** systems
  - Lower coupling
  - Better responsiveness to not-periodic events
- **Predominantly not-periodic systems** but still predictable
  - Events arrive at variable times but within bounded intervals
- **Not-periodic and unpredictable** systems
  - Another ballgame!

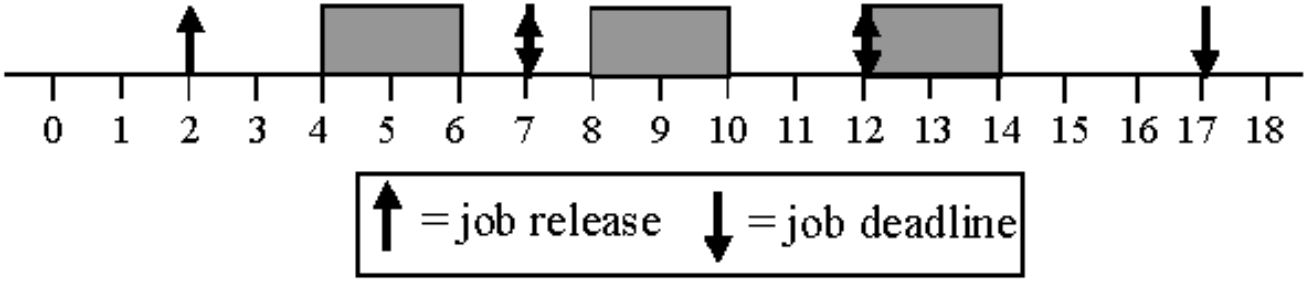
An initial taxonomy /2

- ***Periodic*** tasks
  - Their jobs become ready at regular intervals of time, *T*
  - Their arrival is synchronous to some time reference
- ***Aperiodic*** tasks
  - Recurrent but irregular
  - Their arrival cannot be anticipated (asynchronous)
- ***Sporadic*** tasks
  - Their jobs become ready at variable times but at bounded minimum distance from one another

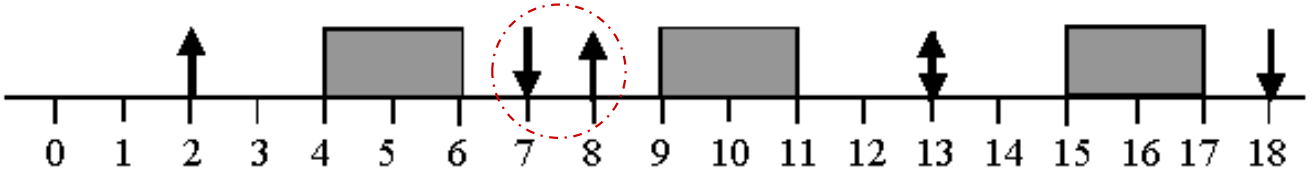
Periodic task and sporadic task

Examples

A periodic task T_i with $r_i = 2$, $p_i = 5$, $e_i = 2$, $D_i = 5$ executes like this according to the rest of the world:



According to Liu, it could execute like this:



To the rest of the world, this is a sporadic task.

Jim Anderson Real-Time Systems Introduction - 26

Abstract modelling /1

- ***Active resources*** (processor, server)
  - They "do" what they have to
    - Execute machine instructions, move data, process queries, etc.
  - Jobs must acquire them to make progress toward completion
    - Contention occurs on access to them
- Active resources have a type
  - Those of the same type can be used interchangeably by a job
  - Those of different types cannot
    - For example, processors may have different speed, which affects the rate of progress for the jobs that run on them

Abstract modelling /2

- ***Passive resources*** (memory, shared data, semaphores, …)
  - A passive resource doesn't do anything per se, but jobs *may need* it to make progress
  - They may be reused if use does not exhaust them
    - If always available in sufficient quantity to satisfy all needs, they are said to be *plentiful* and can be ignored
  - Passive resources that matter to real-time systems are those that may cause *bottlenecks*
    - Access to memory may matter more (owing to *arbitration*) than memory itself (which may be considered plentiful)

Abstract modelling /3

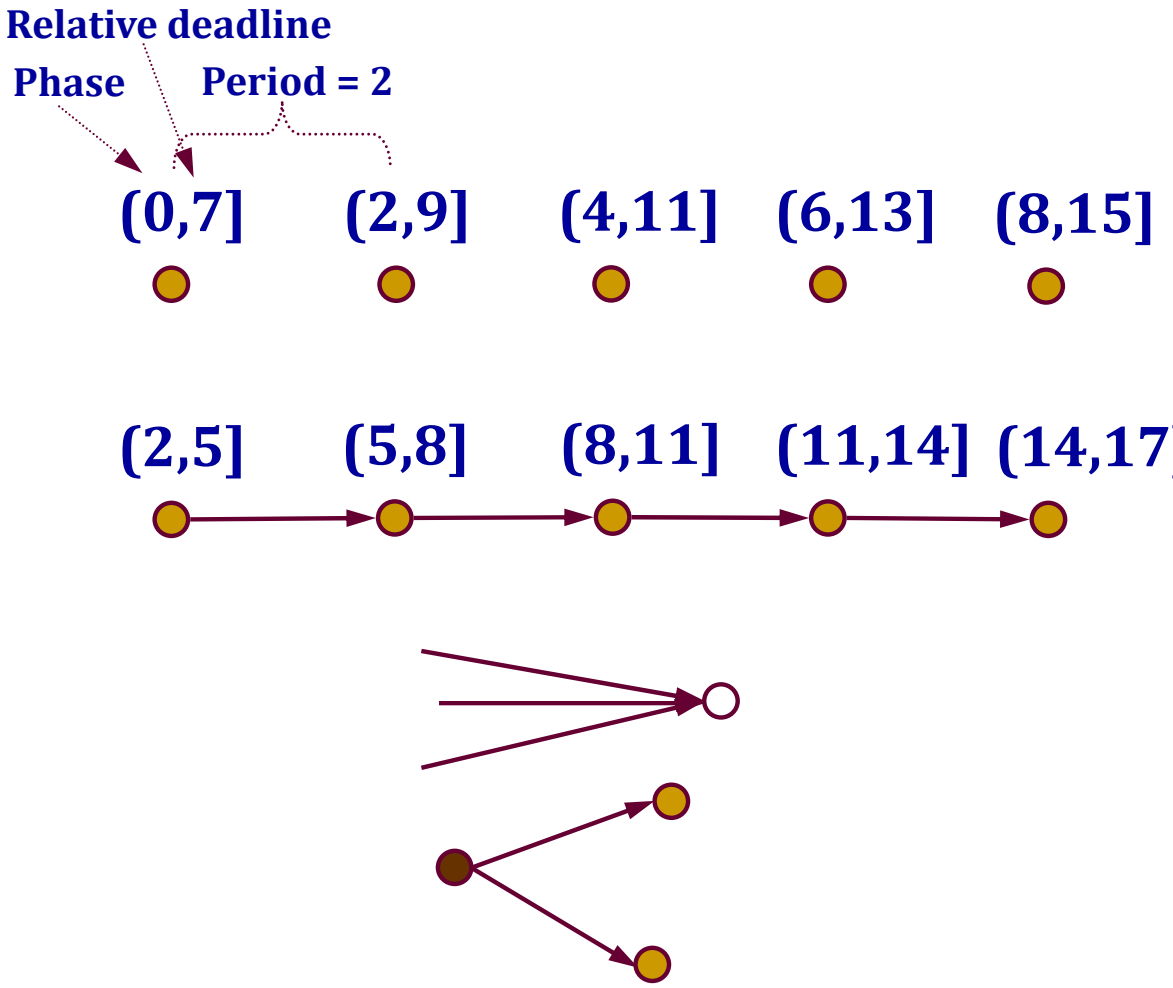
- Fixing design parameters
  - Permissibility of job preemption
    - May depend on the capabilities of the execution environment (e.g., ***non-reentrancy***) but also on the programming style
    - Preemption causes time and space overhead
  - Job ***criticality***
    - May be assimilated to a priority of execution eligibility
    - In general, it indicates which activities must be guaranteed, perhaps even to the detriment of others
  - Permissibility of resource preemption
    - Some resources are intrinsically preemptable
    - Others do not permit it

Which ones?

Abstract modelling /4

- Fixing execution parameters
  - The time that elapses between when a periodic job becomes ready and the next period p is certainly $\leq p$
  - Setting phase $\varphi > 0$ and deadline $\mathrm{d} < p$ for a job may help limit its output jitter (**why?**)
  - The jobs of a system may be independent of one another
    - Hence they can execute in any order
  - Or they may be subject to *precedence constraints*
    - As it is typically the case in collaborative architectural styles (e.g., producer – consumer)

Task precedence graphs



Independent jobs

Dependent jobs

Job of type AND (join)

Job of type OR (branch) typically followed by a join job

Types of precedence constraints

- One job's release time cannot follow that of a successor job
- ***Effective release time (ERT)***
  - For a job J_i with predecessors $\{J_{k=1,\ldots,i-1}\}$, ERT_i is the *latest* value between its own release time and the maximum effective release time of its predecessors, ERT_k, plus C_k
- One job's deadline cannot precede that of a predecessor job
- ***Effective deadline (ED)***
  - For a job J_i with successors $\{J_{k=i+1,\ldots,n}\}$, ED_i is the *earliest* value between D_i and the minimum effective deadline of its successors, ED_k, less C_k
- For single processors with preemptive scheduling, ERT and ED are the only precedence constraints of consequence

Abstract modelling /5

- ***Periodic model***
  - Comprises periodic *and* sporadic jobs
  - Accuracy of representation decreases with increasing jitter and variability of execution time
  - ***Hyperperiod*** H_S of task set $S = \{\tau_i\}, i = 1, \ldots, N$
    - Defined as LCM (least common multiple) of task periods $\{p_i\}$
  - ***Utilization***
    - For every task τ_i : defined as the ratio between execution time and period : $U_i = \frac{C_i}{p_i} \leq 1$
    - For the system (*total utilization*) : $U = \sum_i U_i \leq m$, where m is the number of CPUs ($m = 1$, for now)

Abstract modelling /6

Recall
BCET: best-case execution time
WCET: worst-case execution

- Selecting jobs for execution
  - The scheduler assigns a job to the processor resource
  - The resulting assignment is termed ***schedule***
- A schedule is ***valid*** if
  - Each processor is assigned to at most 1 job at a time
  - Each job is assigned to at most 1 processor at a time
  - No job is scheduled before its release time
  - The scheduling algorithm ensures that the amount of processor time assigned to a job is ≥ than its BCET and ≤ than its WCET
  - All precedence constraints in place among tasks as well as among resources are satisfied

Abstract modelling /7

- A *valid schedule* is said to be ***feasible*** if it satisfies the temporal constraints of every job
- A *job set* is said to be ***schedulable*** by a scheduling algorithm if that algorithm always produces a *valid* schedule for that problem
- A *scheduling algorithm* is ***optimal*** if it always produces a *feasible* schedule when one exists
- Actual systems may include multiple schedulers that operate in some hierarchical fashion
  - E.g., some scheduler governs access to logical resources; some other schedulers govern access to physical resources

Abstract modelling /8

- Two algorithms are of prime interests for real-time systems
  - The *scheduling algorithm*, which we should like to be *optimal*
    - Comparatively easy problem
  - The *analysis algorithm* that tests the *feasibility* of applying a scheduling algorithm to a given job set
    - Much harder problem
- The scientific community, but not always in full consistency, divides the analysis algorithms in
  - ***Feasibility tests***, which are exact (necessary and sufficient)
  - ***Schedulability tests***, which are only sufficient

Predictability of execution

- The execution of system *S* under a given scheduling algorithm *A* is *predictable* if the actual start time and the actual response time of every job in *S* vary within the bounds of the *maximal schedule* and *minimal schedule*
  - *Maximal schedule*: the schedule created by *A* under worst-case conditions for contention and execution demands
  - *Minimal schedule*: analogously for the best case
- **Theorem**: the execution of *independent* jobs with given release times under preemptive priority-driven scheduling on a single processor **is** predictable
  - Good news: this enables us to understand what "worst case" means …

Cute teasers

- Björn Brandenburg's post on SIGBED Blog
  - https://sigbed.org/2020/09/05/liu-and-layland-and-linux-a-blueprint-for-proper-real-time-tasks/
- … shows a cute way to reconcile the basic theory seen so far and simple-but-sound conveniency programming
  - Check it out
- Linux is certainly not what you would expect to find in a real-world embedded real-time system
  - Yet, lots of serious users like "convenience-development"
  - https://www.zdnet.com/article/to-infinity-and-beyond-linux-and-open-source-goes-to-mars/

Further characterization /1

	Time-Share Systems	Real-Time Systems
Capacity	High throughput	Ability to meet timing requirements: Schedulability
Responsiveness	Fast average response	Ensured worst-case latency
Overload	Fairness	Stability of critical part

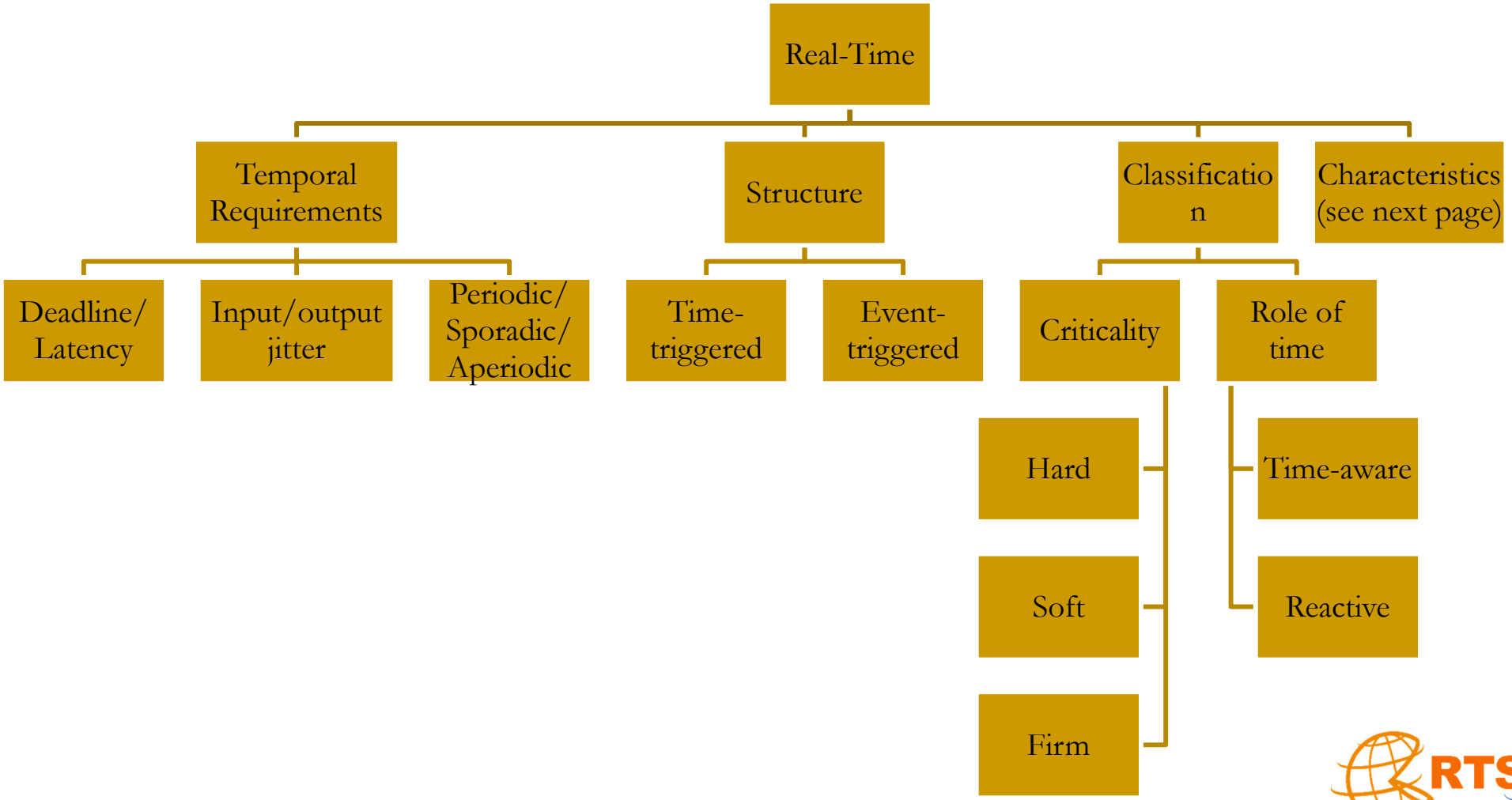
Further characterization /2

- The design and development of a RTS mind the worst case before considering the average case (if at all)
  - Improving the average case is of no use and it may even be counterproductive
    - The cache addresses the average case and therefore operates *adversarially* to the needs of real-time systems
- Stability of control prevails over fairness
  - The former concern is selective the other general
- When feasibility is proven, starvation is of no consequence
  - The non-critical part of the system may even experience starvation

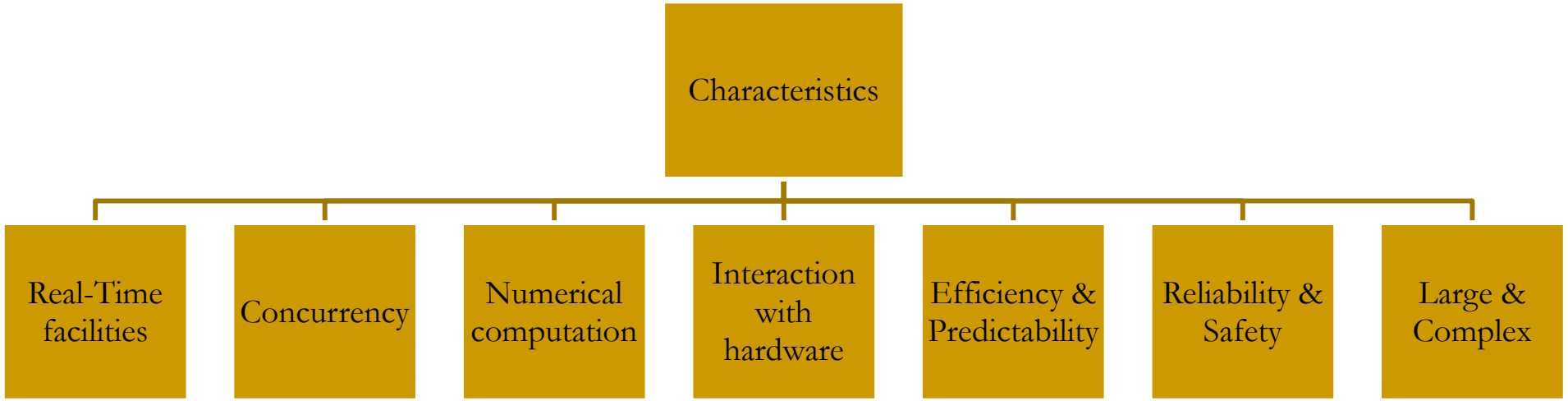
Summary /1

- From an initial intuition to a more solid definition of real-time embedded systems
- Bird's-eye survey of application requirements and key characteristics
- Taxonomy of tasks
- Abstract models to help reason in general about real-time systems

Summary /2


Real-Time
Temporal Requirements
Structure
Classification
Characteristics (see next page)
Deadline/ Latency
Input/output jitter
Periodic/ Sporadic/ Aperiodic
Time-triggered
Event-triggered
Criticality
Role of time
Hard
Soft
Firm
Time-aware
Reactive

Summary /3


Characteristics
Real-Time facilities
Concurrency
Numerical computation
Interaction with hardware
Efficiency & Predictability
Reliability & Safety
Large & Complex