TDDD07 – Real-time Systems

Lecture 9: Dependability and Design

Simin Nadjm-Tehrani

Real-time Systems Laboratory

Department of Computer and information Science

LINKÖPING UNIVERSITY

Dependability topics

Lectures 7 - 9 cover theory and practical examples

- Basic notions of dependability and redundancy in fault-tolerant systems
- Fault tolerance:
  - Relating faults/redundancy to distributed systems from lectures 4-6
  - Relating timing and fault tolerance

Lecture 8: Adds industrial perspective

Lecture 9: Fault prevention and design aspects

Treatment of faults

- Recall: four approaches for treating faults in dependable systems

- This lecture:
  1. Fault prevention
  2. Fault removal
  3. Fault tolerance
  4. Fault forecasting

Reading Material

- Ch. 15.1-15.4, 15.7 B&W or Ch. 5.2-5.3 in Carlsson et al.
- Section 5.1 and 5.3 of article by Avezienis et al. 2004
- Platform-independent design: Huang et al. 2003
- UML-MARTE: Weissnegger et al. 2015

Why dependability-by-design is important (and hard)

LINKÖPING UNIVERSITY

System Requirements

- Functional requirements
  - Describe the main objectives of the system, referred to as "correct service" earlier
- Extra-functional requirements
  - Cover other requirements not relating to main function, in particular dependability
    - acceptable frequency and severity of service failures
  - Also called non-functional properties (NFP), e.g. Timeliness, availability, energy efficiency
- Let's start with one that we studied in this course…

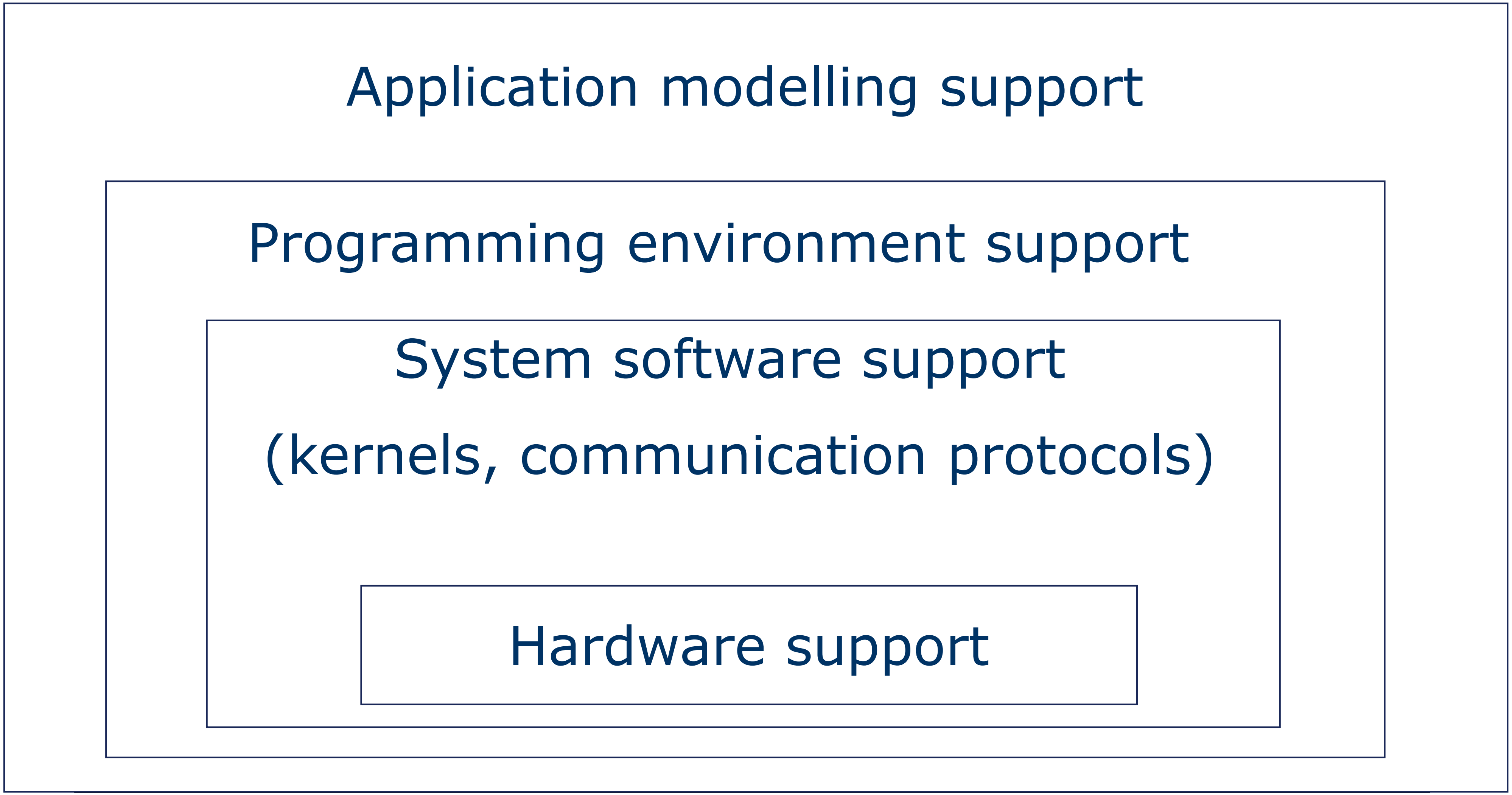
Design for timeliness

Basic approach:

- define end-to-end deadlines
- define deadlines for individual tasks
- ascertain (worst case) execution/communication time for each task/message
- document assumptions/restrictions
- Prove/show that implementation satisfies requirements

So, what is so hard about this?

Layers of design



Fault prevention/removal

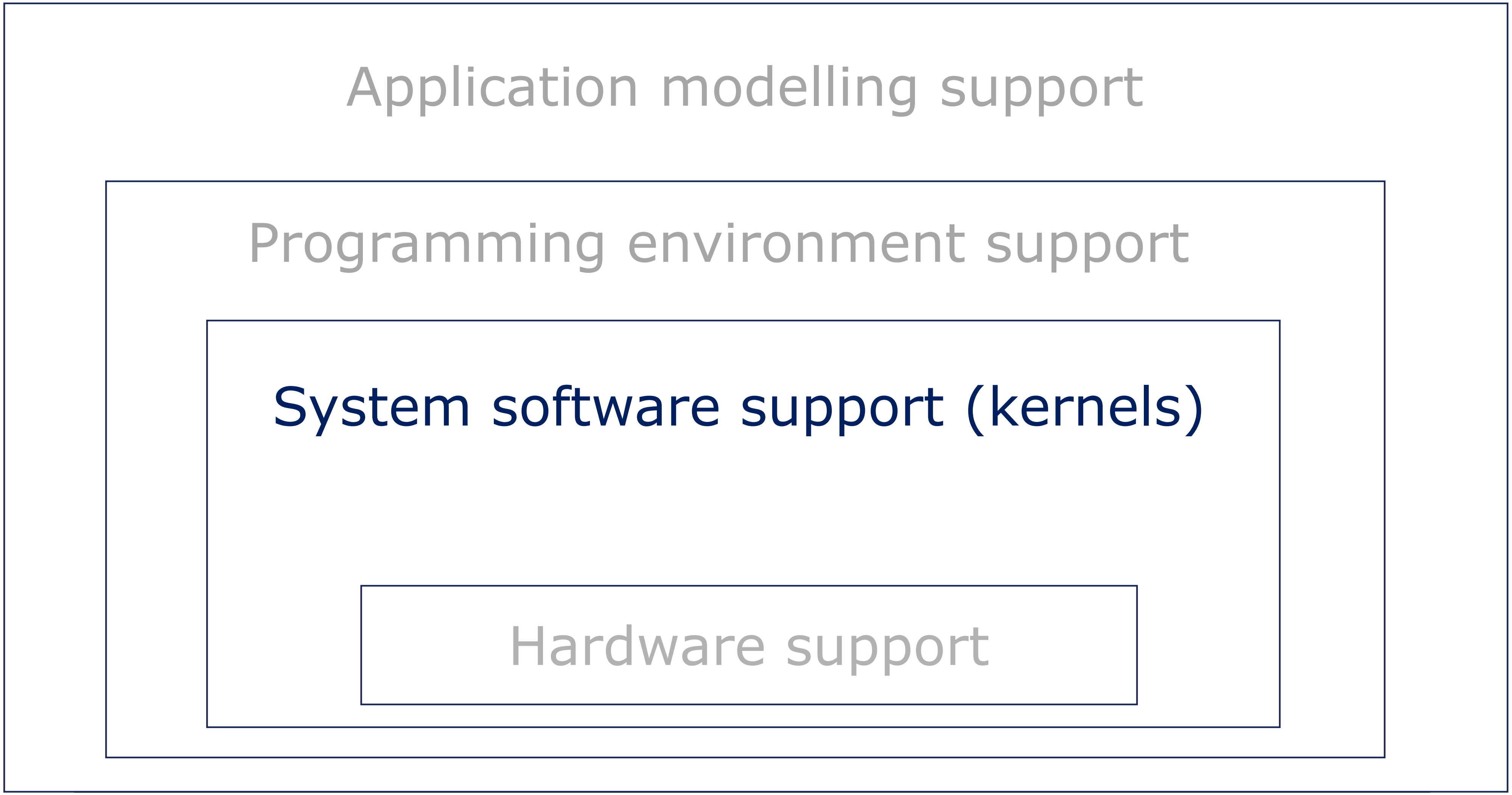
Where should we look to find faults prior to operation?

Historical snapshots

- Hardware design
  - 1970´s Dedicated hardware
  - 1980´s Microcomputers & ASICS
  - 1990´s High performance Microcomputers, FPGAs, MEMs
  - 2000 ’s SoCware, Multicore

- Earlier predictable hardware is replaced with components that are complex to analyse (caches, pipelines, inter-connects, accelerators)

Layers of design



Historical snapshots

- OS Scheduling principles
  - 1970´s Fixed priority scheduling
  - 1980´s Multiprocessor, Dynamic
  - 1990´s Incorporating shared resources
  - 2000´s Load variations, Multicore scheduling
- OS interfaces to optimise memory management e.g. prefetching instructions to boost performance, Virtualisers/orchestrators in cloud platforms

Layers of design

Application modelling support

Programming environment support

System software support (kernels)

Hardware support

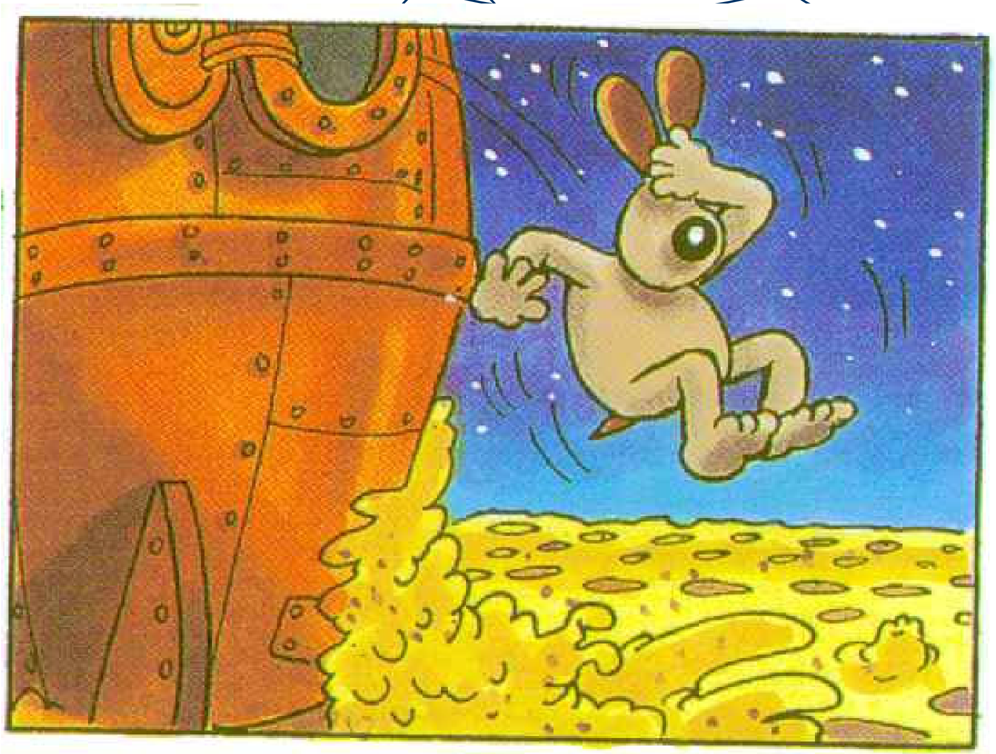
Historical snapshots

- Programming environments
  - 1970´s ”High” level programming
  - 1980´s Real-time specific: Ada
  - 1990´s OO languages, languages with formal semantics
  - 2000´s Software libraries (reuse), AUTOSAR components in automotive

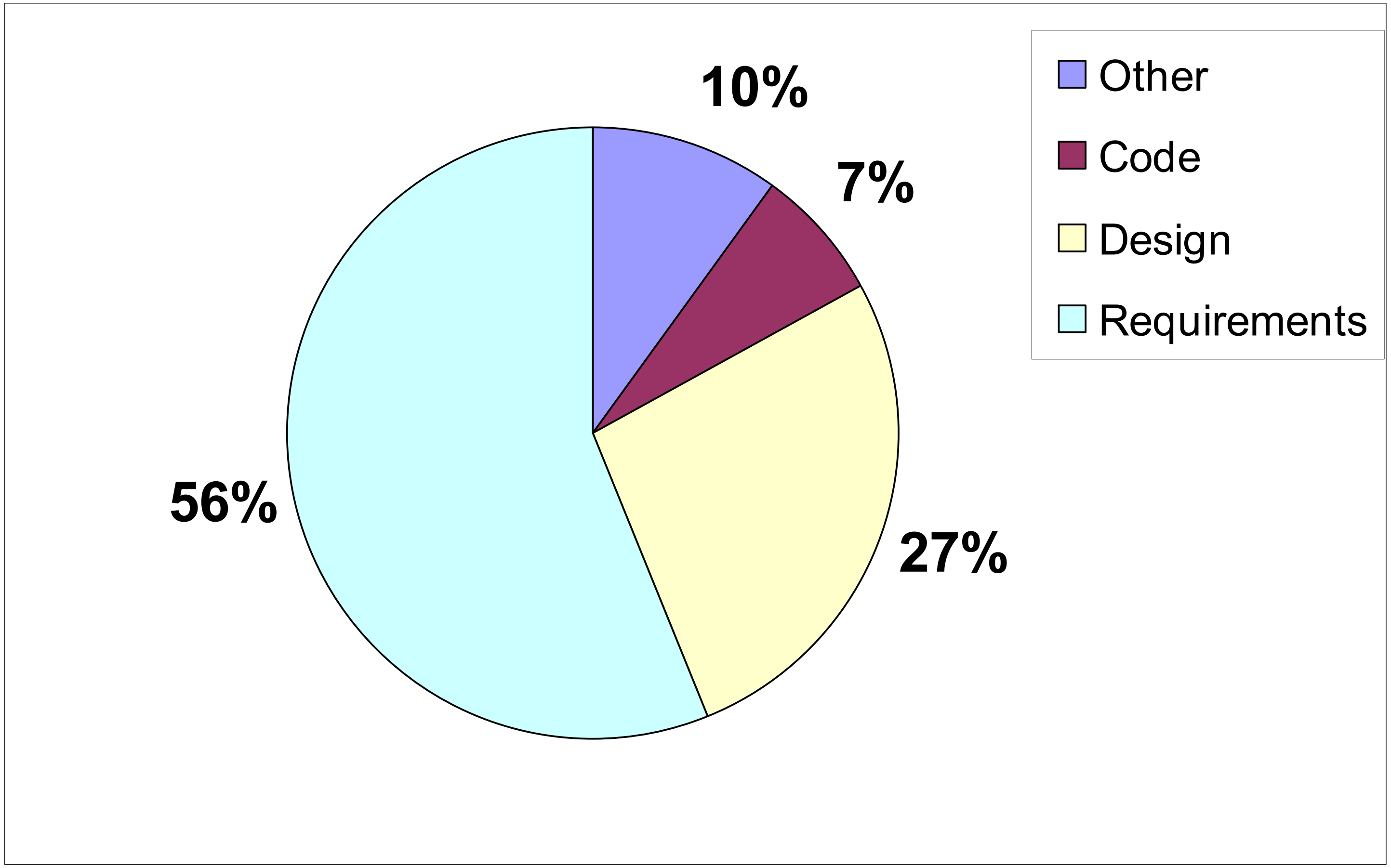
- Today: Data-driven, machine learning!

Engineers: Fool me once, shame on you – fool me twice, shame on me

Software developers: Fool me N times, who cares, this is complex and anyway no one expects software to work...



Software: where do we find faults?



[Cooling 2003]

Testing is not enough…

If a test fails, what was the cause?

- Undocumented assumptions on operational conditions, external impact?
- Wrong program code?
- Unexpected impact of OS? Scheduling?
- Virtualiser overhead?
- Hardware timing dependencies?
- Embedded test code affecting timing?

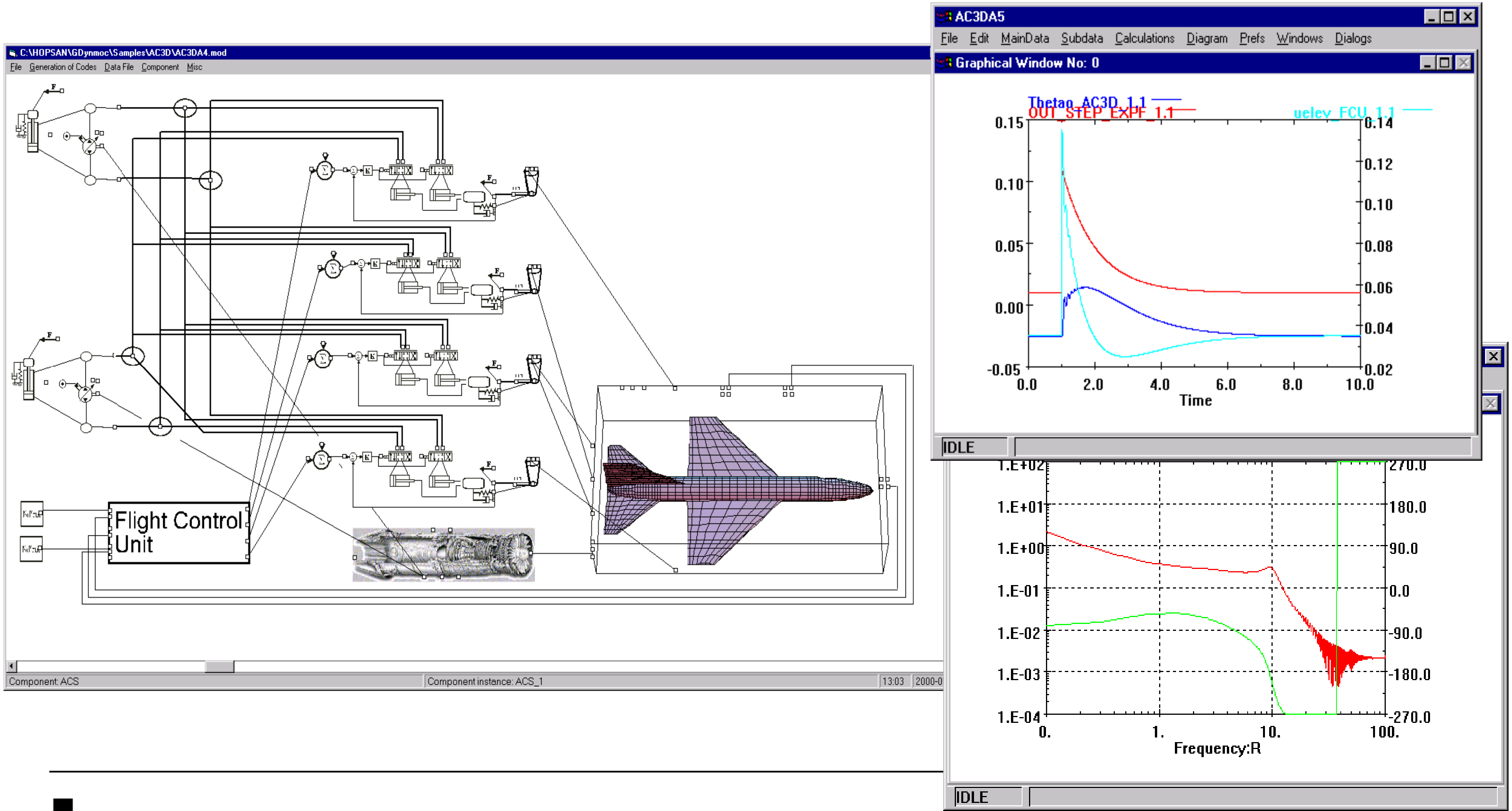
Platform-independent design

Eliminating “butterfly effect” means trying to isolate the impacts of different layers

Back to basics

- define end-to-end deadlines
  - Model the environment!
- define deadlines for individual tasks
  - Specify system decomposition!
- ascertain (worst case) execution/communication time for each task/message
  - Assume hardware/bus characteristics!
- document assumptions/restrictions
  - Model, model, model!
- Prove/show that implementation satisfies requirements
  - Analyse models, then test implementation!

Non-digital hardware



[Krus 2000]

An engineering discipline

Using mathematics can never be wrong!

Model-based development (MBD)

LINKÖPING UNIVERSITY

Model-based development

- In software-intensive systems
  - Models as “higher level” programs

- Idea: use models to analyse the design, automatically generate code from the model!

- Adequate support for modularisation: Well-tested libraries with well-defined interfaces

Layers of design

Application modelling support

Programming environment support

System software support (kernels)

Hardware support

Historical snapshots

- Mathematical modelling & analysis tools
  - 1970´s Sequential systems
  - 1980´s Concurrent/Distributed systems
  - 1990´s Timed models, Combining discrete & continuous, UML
  - 2000's Incorporation in tools: MBD (domain–specific or universal)
- Today: Models in learning-based systems, explainable AI

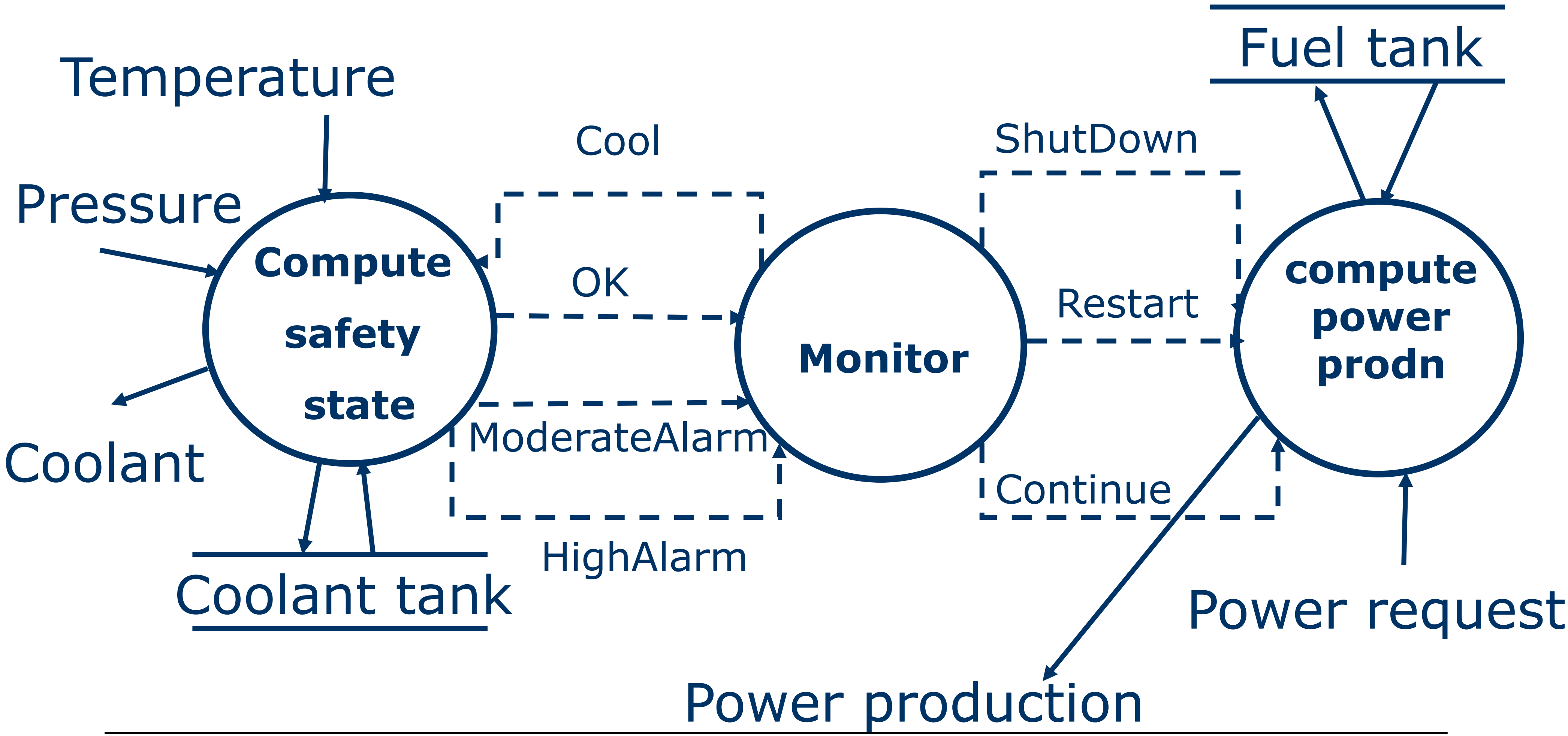
UML standard

- UML 2.0 models components with required and provided interfaces

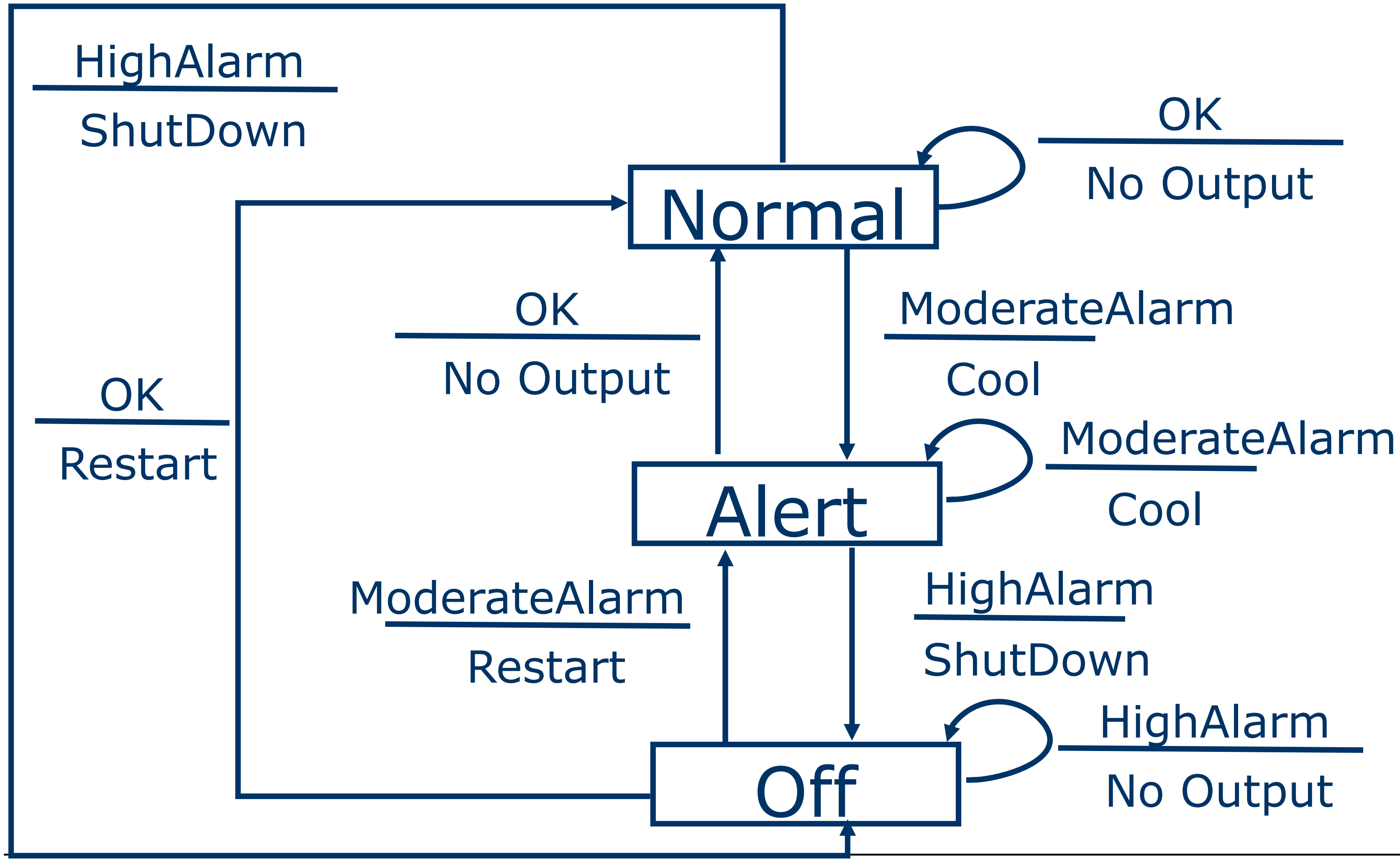
- Family of modelling techniques that are a further development of languages in early 80’s (Ward & Mellor Diagrams)
- Next two slides from an example

[Heitmeyer and Mandrioli, Wiley, 1996]

Power plant: Functional part & safety part



Monitor state machine



Fault prevention and removal

In classical (not learning based) software

What do we want to do with models once we create them?

Use of high-level models

- Tools to represent digital hardware and software components, support for *functional analysis by*
  - Simulations

- Theory:
  - Formal verification of functional properties
  - Semi-automatic code generation

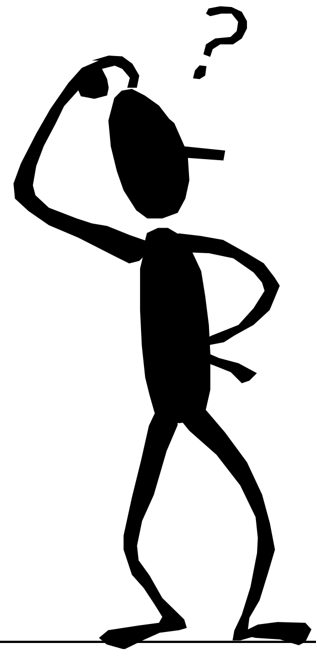
Simulations of a model

Need a unique interpretation:

- What happens in this state if that input arrives?
- The language should
  - have (standard) operational semantics to enable “execution” of the model
  - be platform-independent

Simulations

What do they show?



Formal proofs

- Can be used to **Prove** that *specific* bad things *never* happen

- **Create counterexamples,** identify (design) faults that lead to demonstrated bad things
  - debugging the design

- Can be automated, but suffer from combinatorial explosion

Abstractions and search algorithms help

- Smart data structures for efficient representation of state space
- Smart deduction engines (satisfiability checkers) that find proofs fast
- Smart abstractions of the design to capture the essential properties
  - Synchronous languages (e.g. Esterel, Lustre), used for Airbus 320 software

Historical snapshots

- Mathematical modelling & analysis tools
  - 1970´s Sequential systems
  - 1980´s Concurrent/Distributed systems
  - 1990´s Timed models, Combining discrete & continuous, UML
  - 2000’s Incorporation in tools: MBD (domain–specific or universal?)
- Today: Model of learning-based systems, explainable AI

Adding time to UML

- UML profile for Real-time and Embedded Systems (MARTE)
- Is the approach most automotive and aerospace systems use if they are modelling at all…

- Meta-models for a class of systems with timing and performance parameters

See case study in the Weissnegger et al. paper

Questions?

www.ida.liu.se/~TDDD07

LINKÖPING UNIVERSITY