What is Cyber-physical Computing: Basic Concepts and Application Examples

A tale of Interactive Complexity

in Systems that Interact with the Physical World… and with People!

Early History of CPS: The Beginnings

- NSF Workshop on Cyber-Physical Systems, October 16-17, 2006, Austin, TX.
- National Meeting on Beyond SCADA: **Networked Embedded Control** for Cyber Physical Systems, November 8-9, 2006, Pittsburgh, PA.
- National Workshop on **High-Confidence Software** Platforms for Cyber-Physical Systems (HCSP-CPS), November 30 - December 1, 2006, Alexandria, VA.
- NSF Industry Round-Table on Cyber-Physical Systems, May 17, 2007, Arlington, VA.
- Joint Workshop On High-Confidence **Medical Devices**, Software, and Systems (HCMDSS) and Medical Device Plug-and-Play (MD PnP) Interoperability, June 25-27, 2007, Boston, MA.
- National Workshop on **Composable Systems** Technologies for High-Confidence Cyber-Physical Systems, July 9-10, 2007, Arlington, VA.
- National Workshop on High-Confidence **Automotive** Cyber-Physical Systems, April 3-4, 2008, Troy, MI.
- CPSWeek, April 21-24, 2008, St. Louis, MO.
- CPS Summit, April 25, 2008, St. Louis, MO: NSF Announces new CPS Initiative
- The First International Workshop on Cyber-Physical Systems, International Conference on Distributed Computing Systems (ICDCS), June 20, 2008, Beijing, CHINA.
- Workshop on CPS Applications in Smart **Power** Systems, Raleigh, NC, 2011

Early History of CPS: The Beginnings

- NSF Workshop on Cyber-Physical Systems, October 16-17, 2006, Austin, TX.
- National Meeting on Beyond SCADA: **Networked Embedded Control** for Cyber Physical Systems, November 8-9, 2006, Pittsburgh, PA.
- National Workshop on **High-Confidence Software** Platforms for Cyber-Physical Systems (HCSP-CPS), November 30 - December 1, 2006, Alexandria, VA.
- NSF Industry Round-Table on Cyber-Physical Systems, May 17, 2007, Arlington, VA.
- Joint Workshop On High-Confidence **Medical Devices**, Software, and Systems (HCMDSS) and Medical Device Plug-and-Play (MD PnP) Interoperability, June 25-27, 2007, Boston, MA.
- National Workshop on **Composable Systems** Technologies for High-Confidence Cyber-Physical Systems, July 9-10, 2007, Arlington, VA.
- National Workshop on High-Confidence **Automotive** Cyber-Physical Systems, April 3-4, 2008, Troy, MI.
- CPSWeek, April 21-24, 2008, St. Louis, MO.
- CPS Summit, April 25, 2008, St. Louis, MO: NSF Announces new CPS Initiative
- The First International Workshop on Cyber-Physical Systems, International Conference on Distributed Computing Systems (ICDCS), June 20, 2008, Beijing, CHINA.
- Workshop on CPS Applications in Smart **Power** Systems, Raleigh, NC, 2011

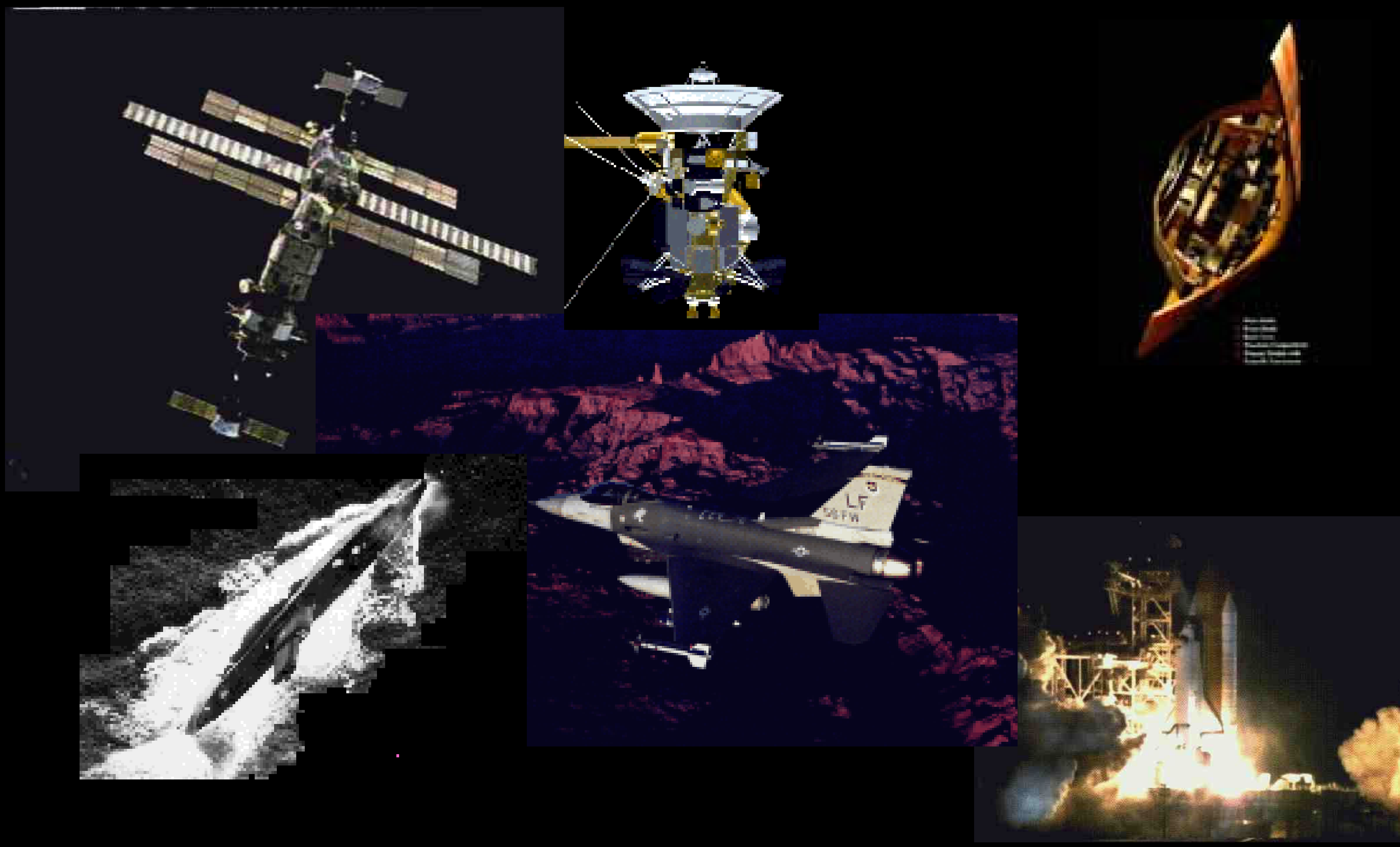
Technologies

Early History of CPS: The Beginnings

- NSF Workshop on Cyber-Physical Systems, October 16-17, 2006, Austin, TX.
- National Meeting on Beyond SCADA: **Networked Embedded Control** for Cyber Physical Systems, November 8-9, 2006, Pittsburgh, PA.
- National Workshop on **High-Confidence Software** Platforms for Cyber-Physical Systems (HCSP-CPS), November 30 - December 1, 2006, Alexandria, VA.
- NSF Industry Round-Table on Cyber-Physical Systems, May 17, 2007, Arlington, VA.
- Joint Workshop On High-Confidence **Medical Devices**, Software, and Systems (HCMDSS) and Medical Device Plug-and-Play (MD PnP) Interoperability, June 25-27, 2007, Boston, MA.
- National Workshop on **Composable Systems** Technologies for High-Confidence Cyber-Physical Systems, July 9-10, 2007, Arlington, VA.
- National Workshop on High-Confidence **Automotive** Cyber-Physical Systems, April 3-4, 2008, Troy, MI.
- CPSWeek, April 21-24, 2008, St. Louis, MO.
- CPS Summit, April 25, 2008, St. Louis, MO: NSF Announces new CPS Initiative
- The First International Workshop on Cyber-Physical Systems, International Conference on Distributed Computing Systems (ICDCS), June 20, 2008, Beijing, CHINA.
- Workshop on CPS Applications in Smart **Power** Systems, Raleigh, NC, 2011

Applications

Original Focus: Mission-critical Systems



Building Timely, Predictable, Reliable Systems

Two Classical Challenges

- ***Establish Functional Correctness:*** How to build functionally correct systems from possibly flawed components?
- ***Establish Temporal Correctness:*** What are the analytic foundations for robust timing guarantees in highly dynamic, time-critical software systems?

Rate of Innovation and Development Time Issues

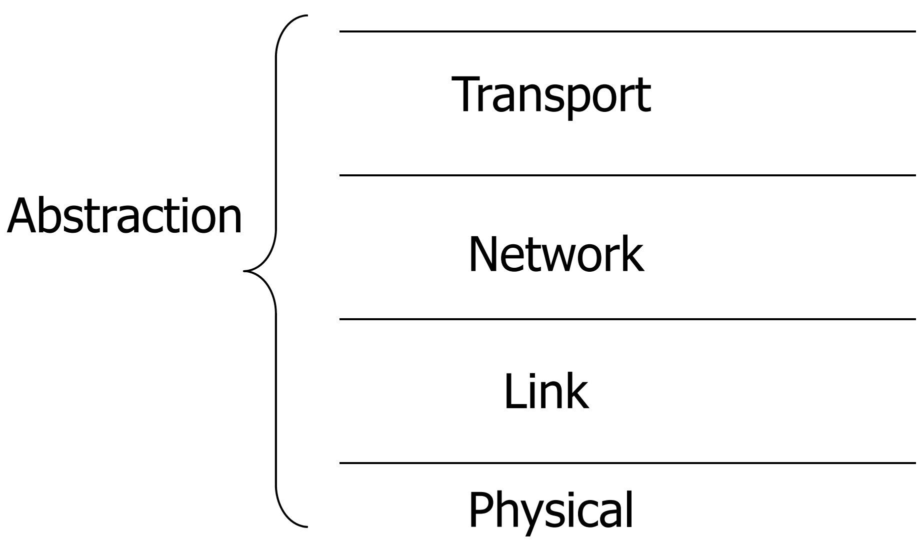
- Early in the 20th century products had a 20-30 year life-span before new "versions" were developed
- At present, a product is obsolete in 2-3 years at most
  - No time to discover and "debug" all possible problems
  - New problems introduced in new versions
  - Component reuse generates additional problems

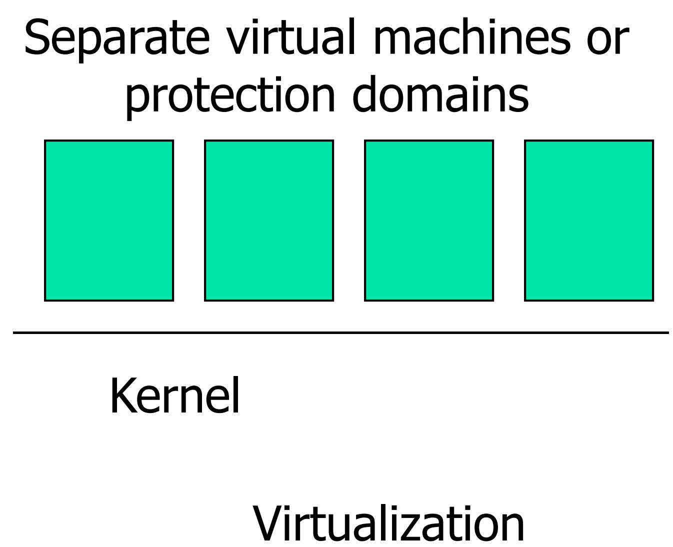
Software: Increasingly the Primary Cause of System Failure

- Arbitrary component interactions unconstrained by physical laws of nature (algorithms can do anything)
  - Potential for high interactive complexity
- Fast error propagation (at computing device speed)
  - Potential for tight coupling
- Software that interacts with the physical world is buggy!

Typical Isolation Techniques

- Abstraction
- Separation of concerns





Abstraction → Specialization

- Complexity
  - → More levels of abstraction
    - → Narrower specialization
      - → More details are "abstracted away"
        - → Myopic view. Less knowledge of possible adverse interactions
          - → More potential for interaction or incompatibility errors

The Curse of Component Re-use

The Ariane 5 Explosion

- On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff (0.5 Billion Dollars)
- Cause of problem?
  - An inertial reference software component.
    - Not needed during flight. Should be stopped before takeoff but is allowed to operate for up to 50 additional seconds
    - Component was designed for Ariane 4. Ariane 5 was a faster system. Velocity variable overflowed.
    - Overflow causes an exception and crashes the software

Interactive Complexity Bugs
Tesla Autopilot Crash

Example 1: Interactive Complexity in Distributed Protocols

- Interactive complexity means:
  - Simple individually insignificant failures interact to compound into system failures, or even…
  - Sets of correctly operating components interact to produce a system failure
    - Example:
      - Shortest hop routing
      - Adaptive rate control

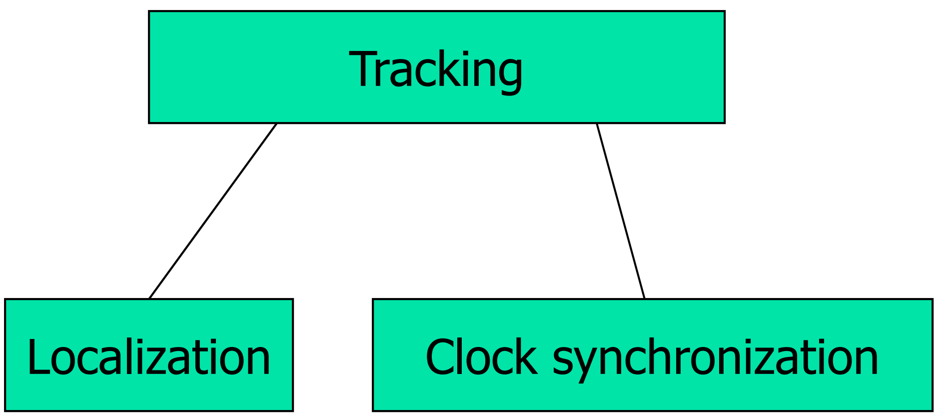
Example 1:

- Shortest hop routing
  - Find shorter path (fewer hops that are longer)
- Long wireless hops → poor channel quality
- Adaptive rate control
  - Reduce transmission rate to improve quality
- Reduced transmission rate

  → longer transmission range

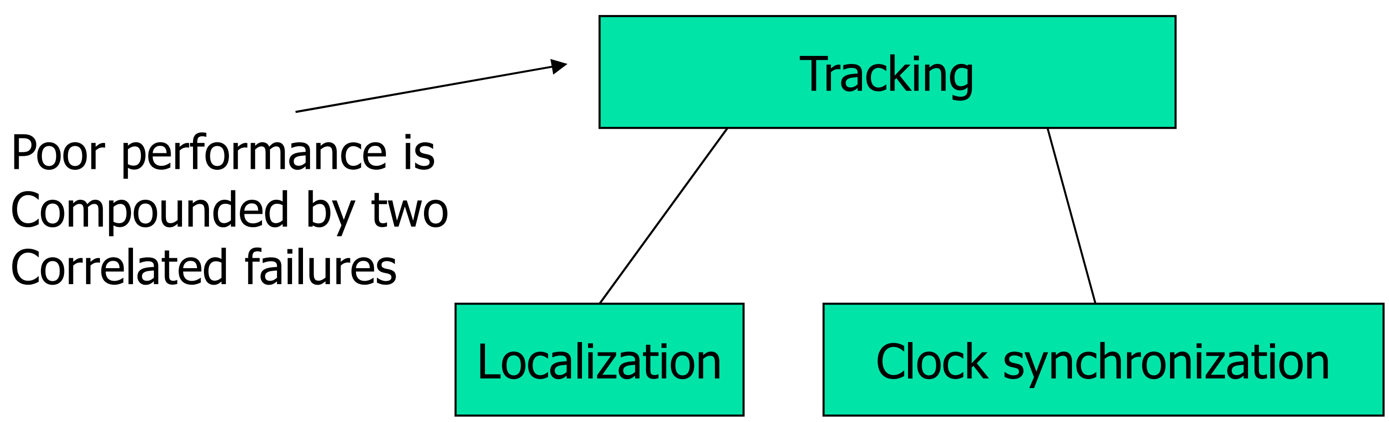
Example 2:
Correlated failure modes between "independent components"

- Localization (determining a node's location) fails in a correlated manner with failure to synchronize clocks. Why?
  - Note: None of the two components uses the other



Example 2:
Correlated failure modes between "independent components"

- Localization (determining a node's location) fails in a correlated manner with failure to synchronize clocks. Why?
  - Note: None of the two components uses the other
- Answer: communication problems. Both subsystems rely on distributed protocols



Example 3:
More on hidden interactions

- Magnetic tracking system operates perfectly in calm weather but fails under strong wind conditions. Why?
    - Wind should not change magnetic sensor reading

Example 3:
More on hidden interactions

- Magnetic tracking system operates perfectly in calm weather but fails under strong wind conditions. Why?
  - Wind should not change magnetic sensor reading
- Explanation
  - Wind caused node antenna to vibrate
  - Moving (metal) antenna caused a lot of noise on the magnetic sensor
  - Noise filter adapted noise threshold to remove background noise (and in this case the signal too)

Example 4: Three Mile Island Nuclear Reactor Failure

Core temperature and pressure continue to build up

Coolant pressure relief valve opens to reduce pressure

Core overheating triggers emergency shutdown

Pressure drops. Valve is stuck open. Coolant boils off. Core temperature rises. Reaction resumes.

Valve failure indicator light turns on but is occluded by repair tag on another device

Core is flooded with water

Failure to open valves

Water at very high temperature oxydizes metal fuel rod coating (rusting)

Open emergency feed-water pumps from emergency tank to cool coolant

Heat exchange stops between primary and secondary cooling Systems. Primary overheats.

Hydrogen is released eventually leading to explosion

Stop secondary coolant flow and turbine

False alarm of minor secondary system coolant leakage through seal

The Fukushima Reactor Failure?

- In April 2011, Japan was hit with an Earthquake followed by a Tsunami. This led to a series of events that ultimately caused a level-7 meltdown in the Fukushima Nuclear Reactor. Can you show the chain of events that led to the meltdown?

Ensuring Software Correctness

- The physical world has no “reset” button
  - When failures occur, they can be costly!
- Must reduce:
  - Interactive complexity
    - Unexpected interactions between seemingly correct components
  - Coupling
    - Fast propagation of effects of failure to other system components

Designing Complex Systems
(Example: Air-traffic control)

- Reduce interactive complexity
  - Air traffic is restricted to non-intersecting "corridors" that separate flight paths in the sky
- Reduce coupling
  - Separate aircraft by a substantial distance to reduce cascaded failure effects (think: multiple-car pile-ups in freeway accidents)

Interaction Examples

- ??

Interaction Examples

- Function calls
- Resource sharing
  - One module crashes → overwrites memory of another → second "unrelated" module crashes (analogy to physical proximity and correlated damage)
  - One module is overloaded → another starves
- Timing and synchronization constraints
  - Precedence constraints (one module must execute before another)
  - Exclusion constraints (cannot operate at the same time)
- Assumptions
  - I thought you submitted our project report?
  - No, I thought you did?

Question: How to Build Reliable Software?

- Common approaches:
  - Tracing, source level debugging
  - Simulation/emulation
  - Network error status reporting
  - Log and replay
- Hard to catch all bugs.

Candidate Approach: Formal Methods

- Express safety properties (e.g., task A will never miss its deadline)
- Prove that safety properties hold
  - If proof fails, counter example is presented (a sequence of events that leads to failure)
- Problem:
  - Proofs require axioms. Axioms may make incorrect assumptions (e.g., circular sensing range)
  - Interactions must be explicitly modeled. Failure to model interactions (e.g., between wind and magnetic sensor) may overlook some failure modes.

Living with Buggy Systems

- If errors cannot be avoided (even using formal methods), we must design systems to tolerate them
  - Architectures for "living with bugs"
  - Fast diagnosis and recovery
  - Issues
    - Problem must be observable (or else cannot diagnose)
    - Observation must be in time so that recovery is possible (observing that you forgot your parachute *after* you jump will not help you)
    - Systems with highly auto-correlated state on long time-scales will likely take long to recover

Simplicity to Conquer Complexity
Lui Sha

- Elements of a good design
  - Simple safety core
  - Complex enhanced mission functionality
  - Formal proof of core correctness
  - Well formed dependency (core may use but will not depend on any other components)

Example: A Sorting Exercise

- Sorting:
  - Bubble sort: easy to write but slower, $O(n^2)$
  - Quick sort: faster, $O(n \log(n))$, but more complicated to write
- Joe remembers how to do bubble sort but is not perfectly sure of quick sort (has a 50% chance of getting it right).
- Joe is asked to write a sorting routine:
  - Correct and fast: A
  - Correct but slow: B
  - Incorrect: F

Example: A Sorting Exercise

- Sorting:
  - Bubble sort: easy to write but slower, $O(n^2)$
  - Quick sort: faster, $O(n \log(n))$, but more complicated to write
- Joe remembers how to do bubble sort but is not perfectly sure of quick sort (has a 50% chance of getting it right).
- Joe is asked to write a sorting routine:
  - Correct and fast: A
  - Correct but slow: B
  - Incorrect: F

What is Joe's optimal strategy?

Example: A Sorting Exercise

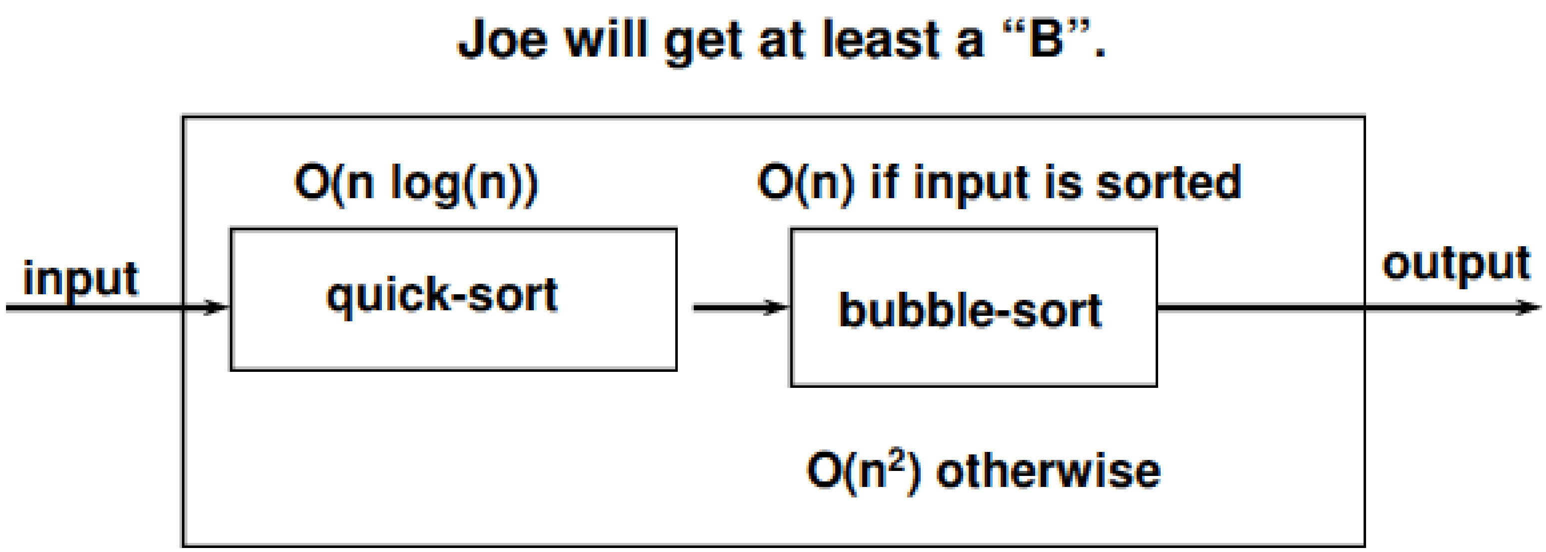
- Sorting:
  - Bubble sort: easy to write but slower, $O(n^2)$
  - Quick sort: faster, $O(n \log(n))$, but more complicated to write
- Joe remembers how to do bubble sort but is not perfectly sure of quick sort (has a 50% chance of getting it right).
- Joe is asked to write a sorting routine:
  - Correct and fast: A
  - Correct but slow: B
  - Incorrect: F

What is Joe's optimal strategy?

Critical requirement:
Must pass!

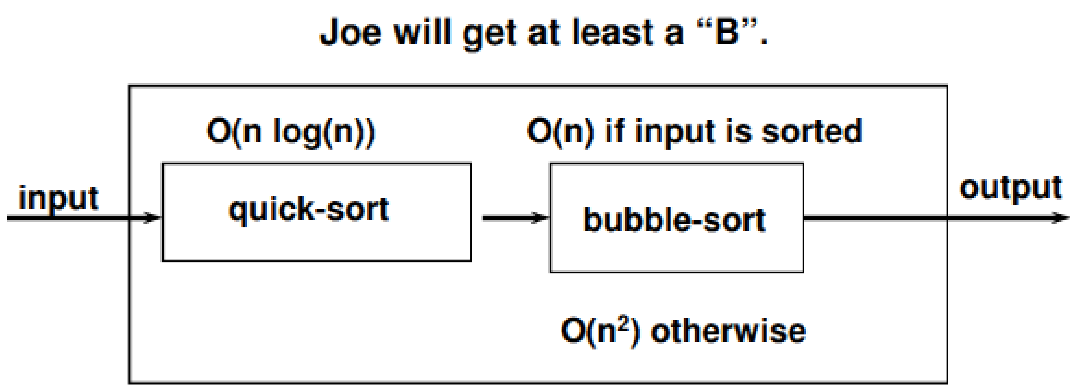
Solution

- Simplicity to "control" complexity



Solution

- Key property
  - Use complex but efficient solution in the common case
  - If the complex solution fails, catch the failure and switch to the simple (less efficient) but safe option



Asimov Laws of Robotics

- What are the implications on software system design for the robot?