Effective Creation of Reliable Microcontroller Software

M.Sc. Ari Okkonen

Managing Director, OBP Research Oy

Goals of the Approach

- Ensure the fitness of the software for its purpose
- Minimise faults found in target testing and on the field
- Minimise the effort and time-to-market

The Principles

- Keep it simple
- Encapsulate the explicit understanding of the problem
- Start with the essential functionality
- Develop the parts together with their corresponding environment simulators

Keep it Simple

- Why
  - Complexity compromises productivity
    - Try yourself: does it take longer to solve a 5000 piece puzzle than 10 puzzles of 500 pieces each?
    - (However, engineers like complex problems…)
- How
  - Separate the concerns, e.g.
    - required functionality vs. implementation techniques
    - temporal behaviour vs. data processing algorithms
    - logical correctness vs. timing accuracy
  - Avoid smart tricks - *as long as possible*

Separating the Concerns

- Different concerns, different phases
- Different responsibilities, different parts
- Different requirements, different techniques
- Different concepts, different notation

Different Concerns, Different Phases

1. Understanding the problem
   - build rough simulation, validate with specialists
2. Control (“the essence”) solution
   - model the essential control (etc.)
   - simulate with the environment model
3. Interfacing
   - model the software and the counterparts - test
   - develop and test low level interfacing separately
4. Physical challenges
   - allocate to physical units and priority levels

Different Responsibilities, Different Parts

- Develop modules that does one thing well
  - "This ensures, that a door cannot close when somebody is in the way"
- Explain the purpose of a module using at most concepts of 2 domains.
  - "This scales AD input to given integer range" - *do not mention irrelevant reason, be it temperature, pressure, or ...*

Different Requirements, Different Techniques

- Response times
  - HW, interrupt servers, pre-emptive RTX, cyclic executive
- Reliability
  - redundant space-grade HW, …, standard office solutions
- Etc.

Different Concepts, Different Notation

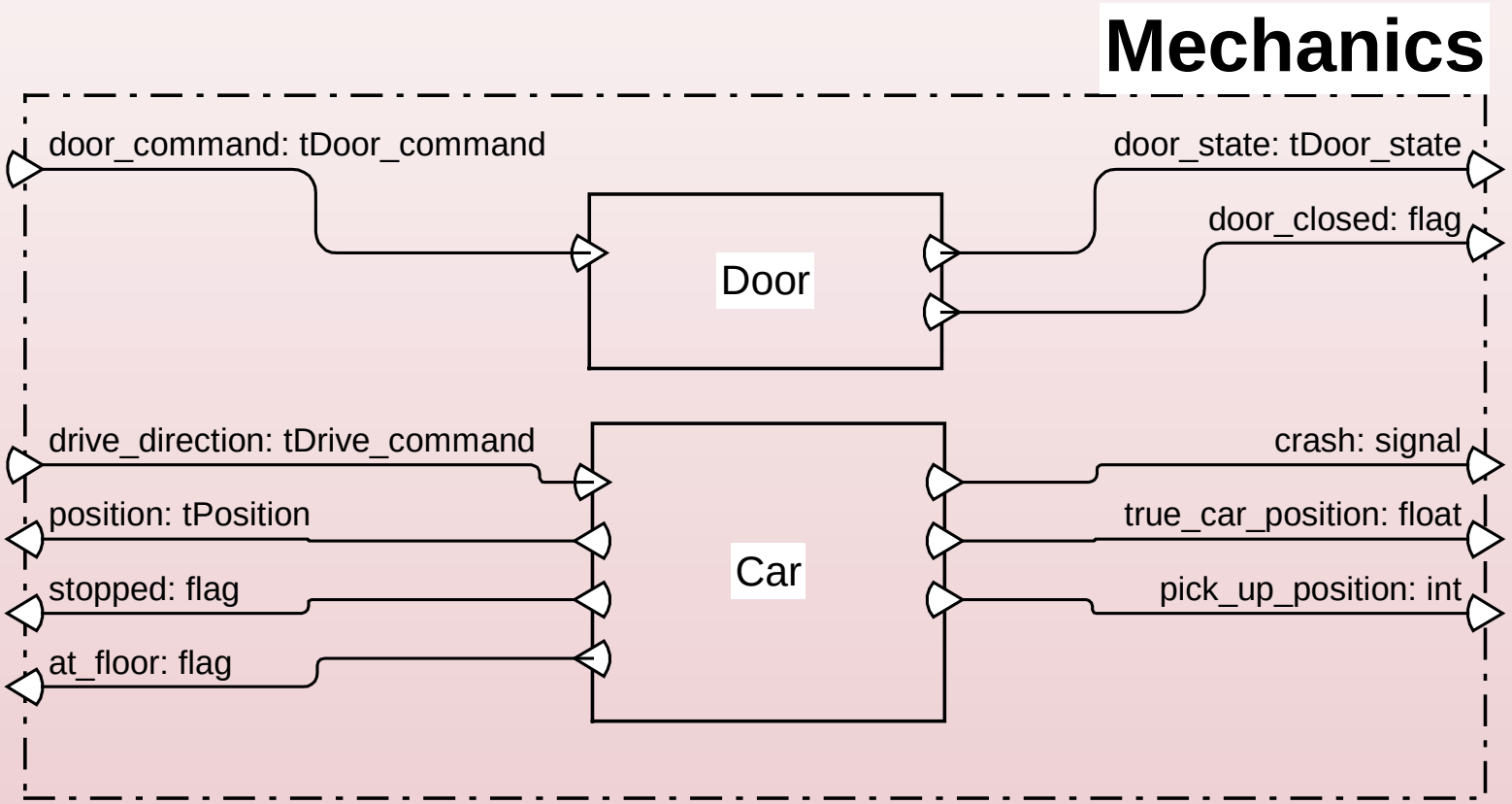
- Algorithms -> Programming language
- Management of related concepts -> ER/Object notation
- Reactive, event-response behaviour -> State-transition notation
- Interconnected subsystems -> Architecture notation

Understanding the Problem

- Mostly, you can't solve a problem, if you don't understand it.
- Build simulators of the environment
  - Easier to communicate and re-use
  - Facilitates **automated validation of designs**
  - Add details as the development requires
  - Validate with problem domain experts

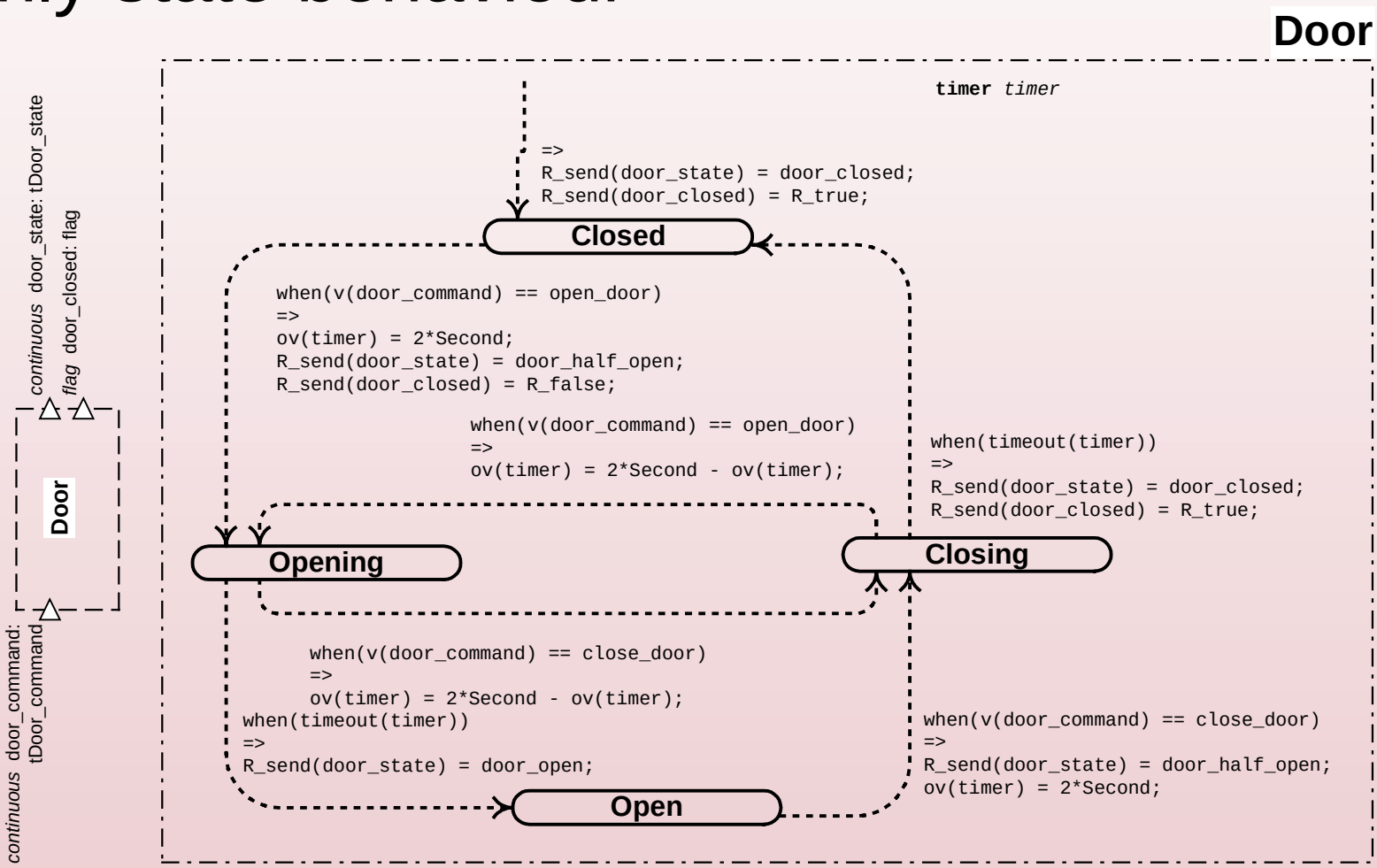
A Problem Model - Lift Mechanics

- Two independent subsystems



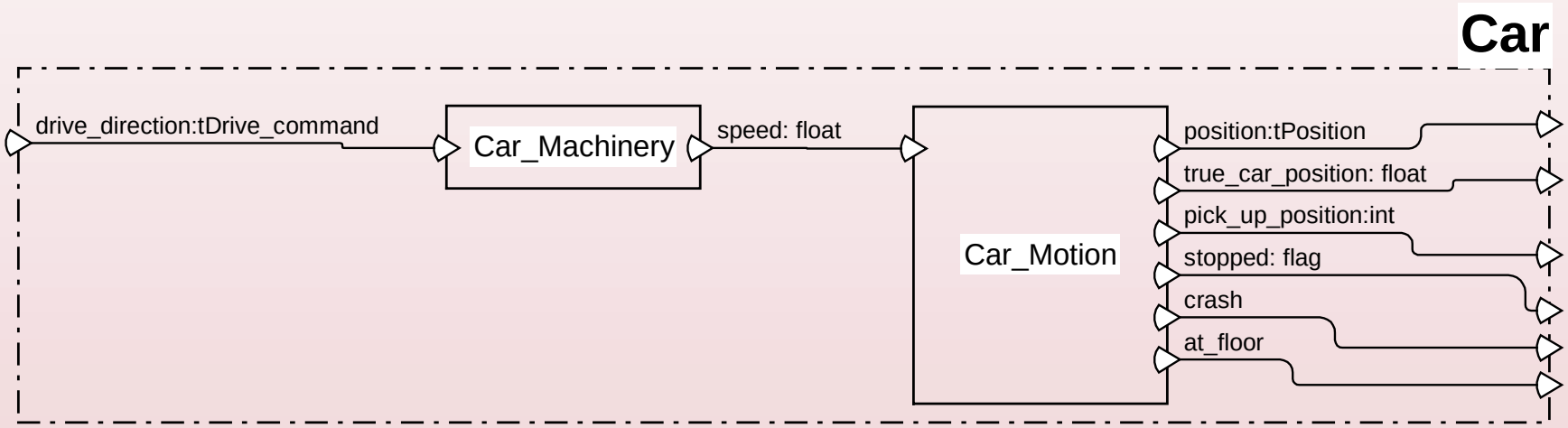
A Problem Model - Door Mechanics

- Only state behaviour



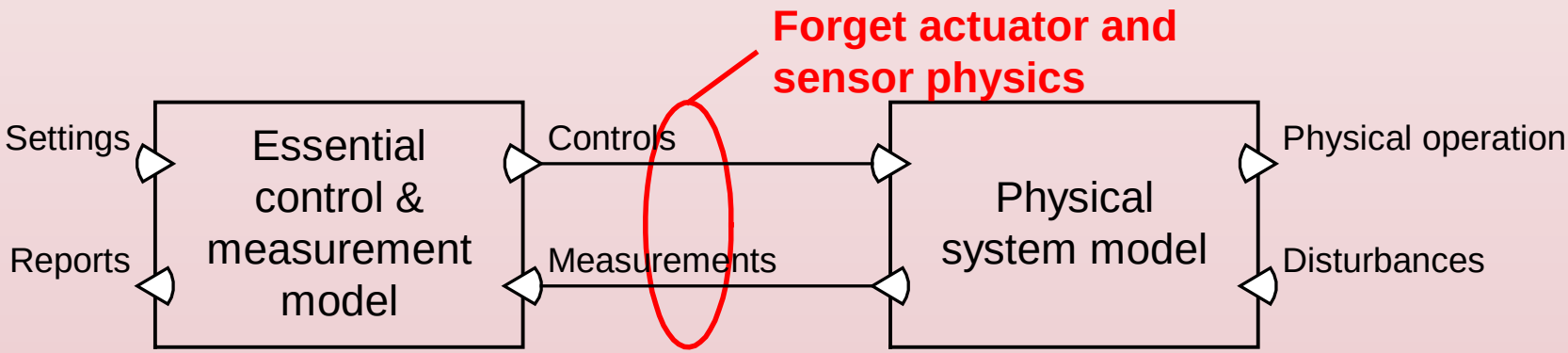
A Problem Model - Car Mechanics

- Two interconnected subsystems



Start with the essential

- Why
  - to keep the effort focused to the essential
  - to avoid unnecessary redesign
  - to have a rigid basis to remaining effort
  - to address controllability of the physical process early enough!
- How
  - Ask: “What is the reason the system will exist?”
  - Solve the main problem in detail
  - Interface and implementation concerns come later
  - simulate against the problem model

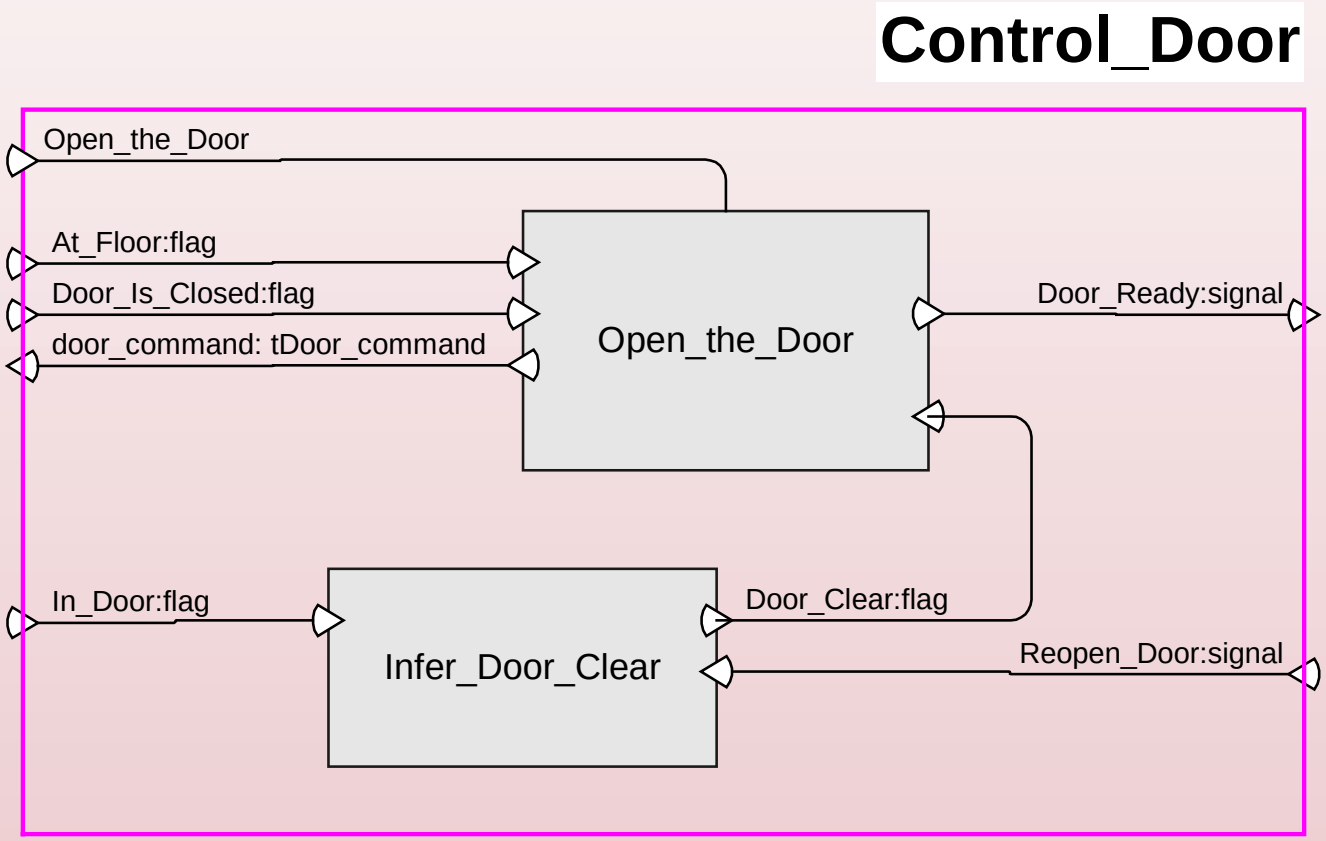


Test, test, test ...

- Test immediately, when you get something
- It is MUCH cheaper to catch an error in early phase than later on!

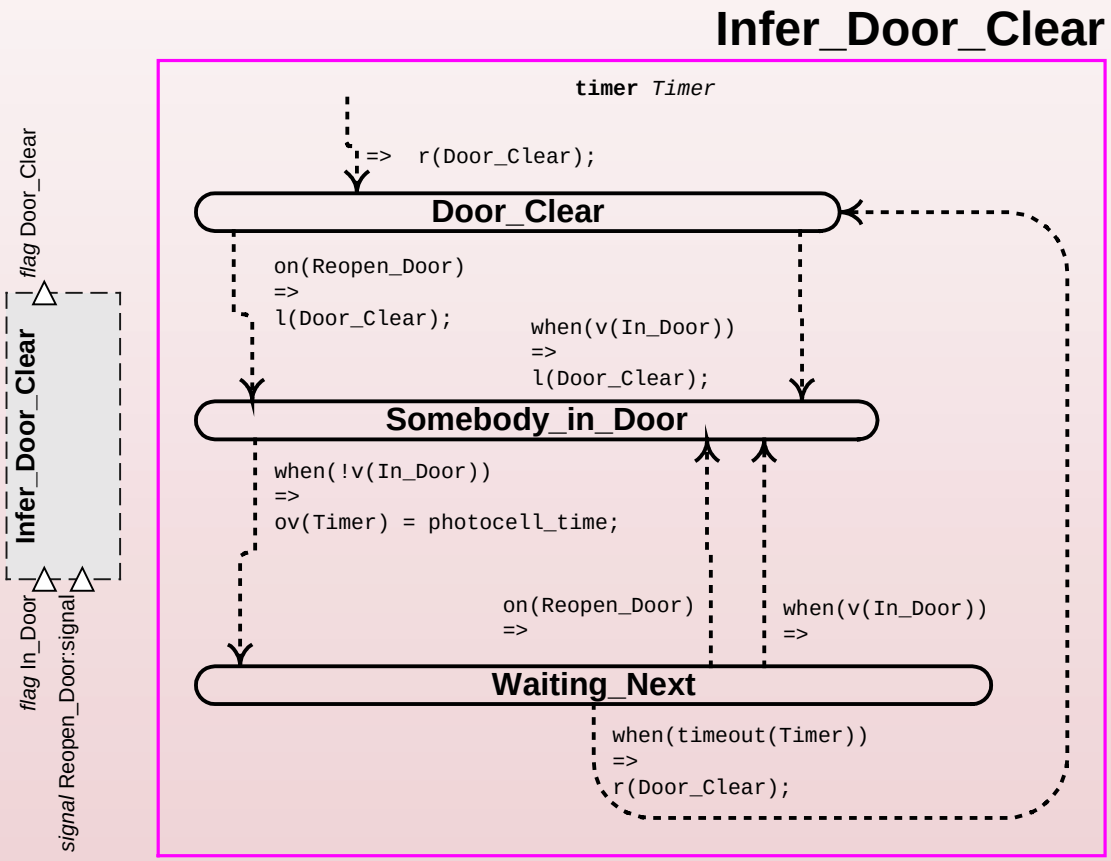
Essential Control - Architecture

- Subsystems with own responsibilities



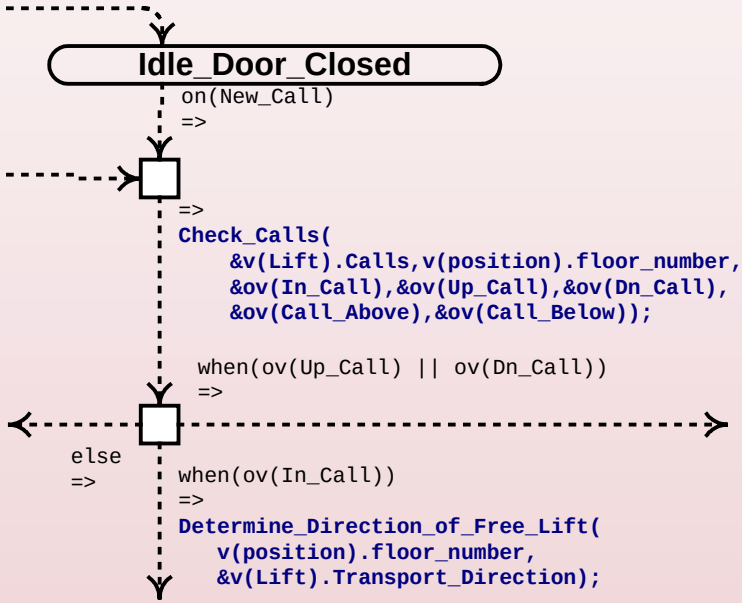
Essential Control - Behaviour

- Example of simple state control



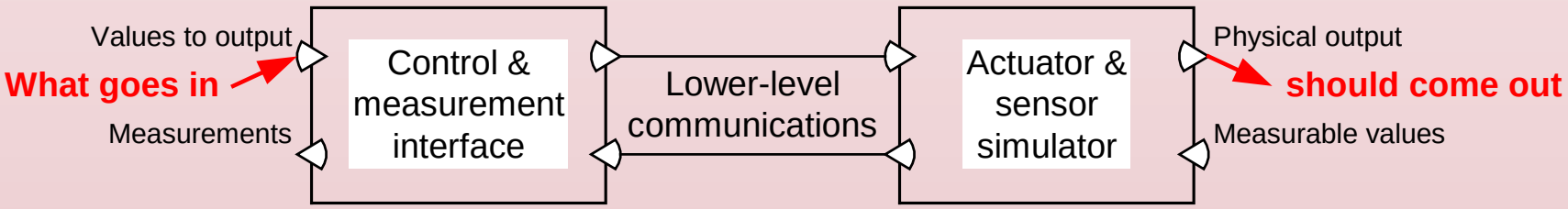
Essential Control - Using Algorithms

- Algorithms written *(manually or otherwise)* in programming language are called from state machines



Interfacing - High Level

- Develop parts with environment simulators
  - Why
    - to find errors and validate the designs as early as possible
  - How
    - test the combined behaviour of a system component and the corresponding environment simulator



Interfacing - Low Level

- Test separately to see that all signals go through with acceptable quality
- Direct interfacing with higher level modules helps integration phase

Physical Challenges

- Reaction speed and timing accuracy
  - Find reaction paths from the architecture diagrams
  - Calculate required CPU time
  - Allocate operations along reaction paths to processors, interrupt handlers and tasks according to required response times and CPU load

Implementation

- Systematic coding rules or Reagenix generator may be used for architecture diagrams and state machines
- The diagrams and C-code are verified and validated specifications, if some functions must be written in assembly or designed in HW

Implementation 2

- Some microcontroller applications can be completely generated in Reagenix
- New Reagenix interfacing wizard exists now for AT-Mega128

Summary

- Avoid complexity by keeping things simple and separate
- Develop by modelling
- Test unit and subsystem models with simulator counterparts
- Use generator to produce code for rapid development cycle