Coordination: mechanisms and architectural patterns

Herman Bruyninckx
KU Leuven&TU Eindhoven
27 May 2020

Overview

- Why is Coordination needed?
- Three complementary mechanisms for Coordination:
  - flag arrays (“bitfields”)
  - Petri Nets arrays (“bitfields”)
  - Finite State Machines
- Architectures for:
  - data exchange
  - task queue processing

Why is Coordination needed?

Why is "Coordination" needed?

```
if (condA and conB) then {...}
```

In every algorithm, conditional statements like the one on the left occur.

There are two very different contexts at work behind the screens:

synchronous computing:

both `condA` and `condB` are computed in the same algorithm, using data that is **not shared** with any other algorithm.

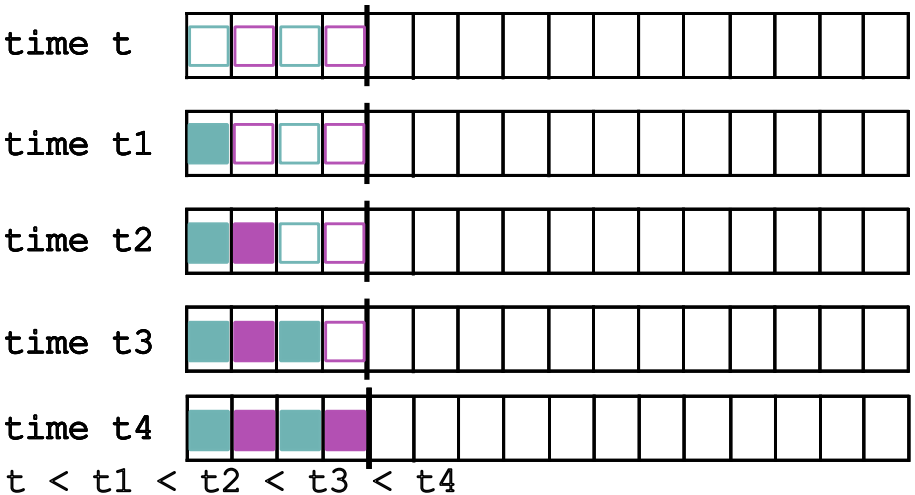
asynchronous computing:

one or both of `condA` and `condB` are computed with data that can also be **changed by another** algorithm.

Problem: the conditions on which one algorithm makes its decisions can change behind its back, while it is deciding
→ **inconsistent** decisions will occur, sooner or later!
Solution: let algorithms coordinate the **execution of their functions**!

Three mechanisms for Coordination

Mechanism 1: Flag arrays for protocols



There are two algorithms: Alg1 and Alg2
They share a **bitfield**, or **flag array**.

Each algorithm computes logical conditions, with data that is under its full control **only**...
...and fills in the **truth value** of such logical condition in the **agreed-upon** bit in the shared array.

Both algorithms also **share a protocol**.
That is, the **order** in which each algorithm fills in the next flag in the array.

Each algorithm only computes the logical conditions involved in the protocol **after** it has **observed** that the other algorithm has set the associated flag.

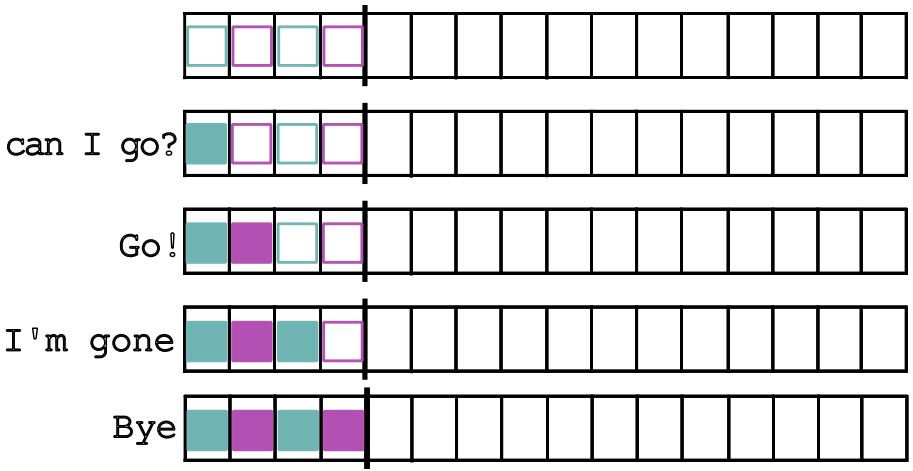
Mechanism 1: Flag arrays for protocols (2)

Reading and writing bits in a bitfield can be done **atomically** on all CPUs
→ consistency of protocol flag array can be guaranteed!
→ consistency of evaluation of logical conditions can be guaranteed!

Caution:

- correct obedience of both algorithms to agreed-upon protocol can **not** be guaranteed, but depends on discipline of programmers.
- they must make sure that the truth values of conditions in each algorithm does not change as long as the coordination protocol is active.

Mechanism 1: Flag arrays for protocols (3)
Example: *Stop-and-go* coordination

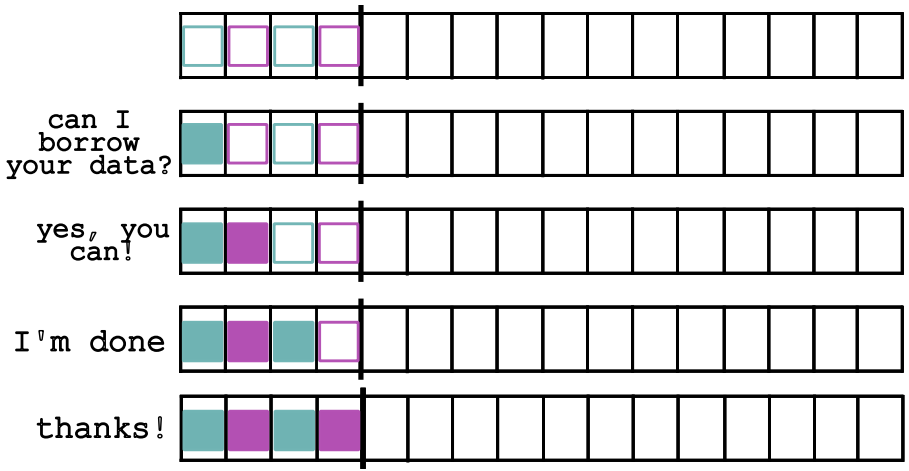


One algorithm waits before starting a particular computation…

…until another algorithm has finished its own particular computation.

They inform each other explicitly about the end of their mutual coordination.

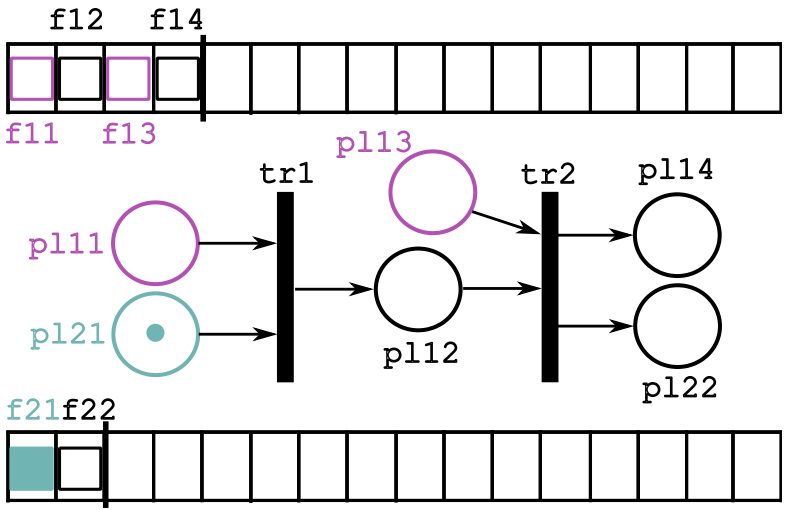
Mechanism 1: Flag arrays for protocols (4)
Example: *Data borrowing* coordination



Special case of *Stop-and-go* coordination:

- one algorithm **owns** data that other algorithms also need to work with.
- each of the other algorithms engages in a protocol with the “owner” to get its explicit agreement to use the data.
- the “borrower” informs the “owner” when it's done.

Mechanism 2: Petri Nets for multi-algorithm coordination



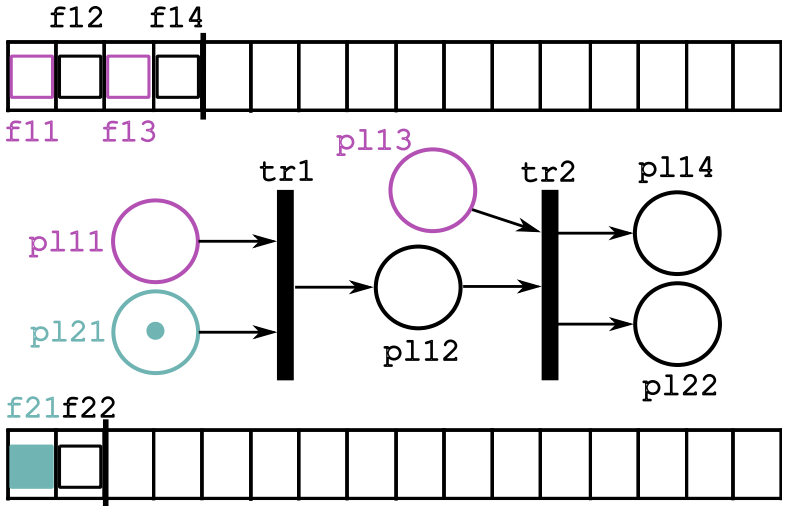
Flag arrays don't scale well for the coordination of many algorithms:

- the coordination order is often **not sequential** any more.
- flag arrays **imply** that implementers of **all** involved algorithms known about all the other algorithms involved in a coordination.

Solution:

- one algorithm (the "**mediator**") is responsible for the overall coordination.
- it engages in a **flag array protocol** with each of the coordinated algorithms
  → **decoupling** of having to know each other!
- it uses a Petri Net model to organise its own decision making
  → **non-sequential** ordering in decision making becomes possible!

Mechanism 2: Petri Nets for multi-algorithm coordination (2)



Primitives in a Petri Net:

- **token**: represents flag of one coordinated algorithm.
- **place**: holds zero or one token
- **transition**: *fires* when all its input places are full
  → makes them empty.
  → fills its output places.

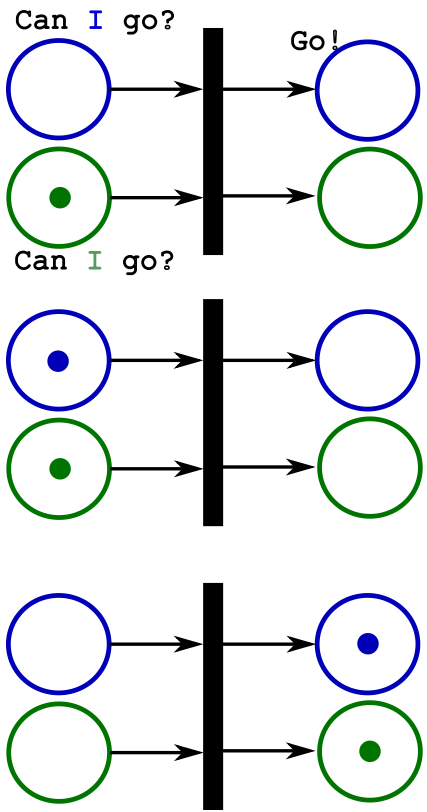
Marking reaction table of the Petri Net:

input places	transition	output places
pl11, pl21	tr1	pl12
pl12, pl13	tr2	pl14, pl22

Mediator is only "owner" of Petri Net:

- it can compute the Petri Net transitions without any interference of the other algorithms.
- it can engage in multiple flag array protocols, without interference.

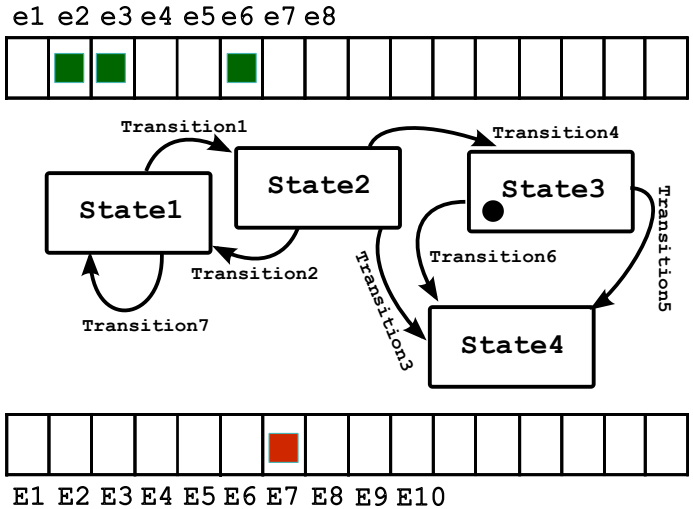
Mechanism 2: Petri Nets for multi-algorithm coordination (3)
Example: coordinated starting of multiple algorithms



Coordination behaviour:

- coordinated algorithms can become `ready-to-go` in any order.
- mediator waits till **both** coordinated algorithms are ready to start…
  …before making its transition.
- both coordinated algorithms can check their `Go!` flag at their own leisure…
  …and without having to known anything about each other's existence.

Mechanism 3: Finite State Machines

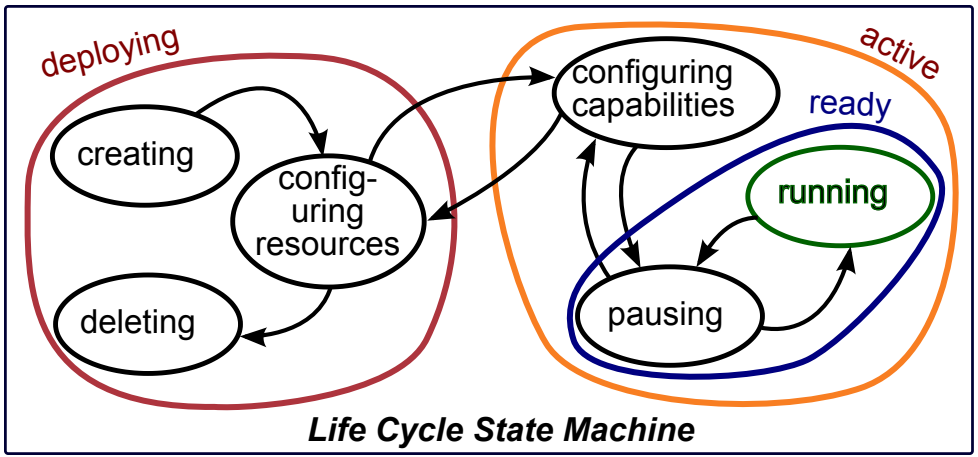


input event	transition	output event
e1	Transition1	
e2	Transition2	E2
e3	Transition3	E3
e2	Transition4	E4
e1	Transition5	
e6	Transition6	**E7**
e4	Transition7	E4

Differences with Petri Net:

- mediator algorithm can be in **one and only one** state at any given time.
- setting and cleaning of any flag in flag array can happen at any time, by any algorithm.
- mediator algorithm decides to take a transition away from its current state as soon as the associated event flags are true.
- possibly, a transition sets an event flag in the output array.
- mediator algorithm can decide to clean input and/or output event arrays at any time.

Mechanism 3: Finite State Machines (2)
Example: Life Cycle State Machine (LCSM)



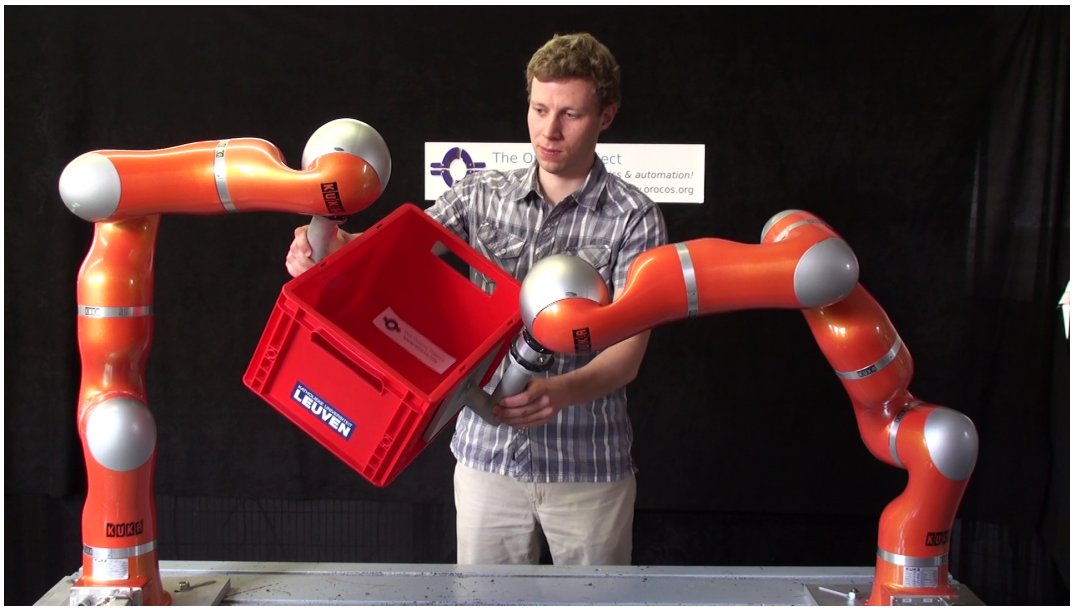
Life Cycle State Machine

The **hierarchy** of states is only in the **model**:
→only **leaf states** matter for the software.
→**other** state: **view** on set of leaf states.

Purpose: to coordinate the behaviour of an **activity**:

- activity = **set** of many algorithms running together.
- state of the activity: particular **configuration** of these algorithms.
- before being "active", the activity must make sure its resources have been correctly configured.
- while being "active", the activity can decide to pause its behaviour, temporarily.

Mechanism 3: Finite State Machines (3)
Example: Task execution State Machine



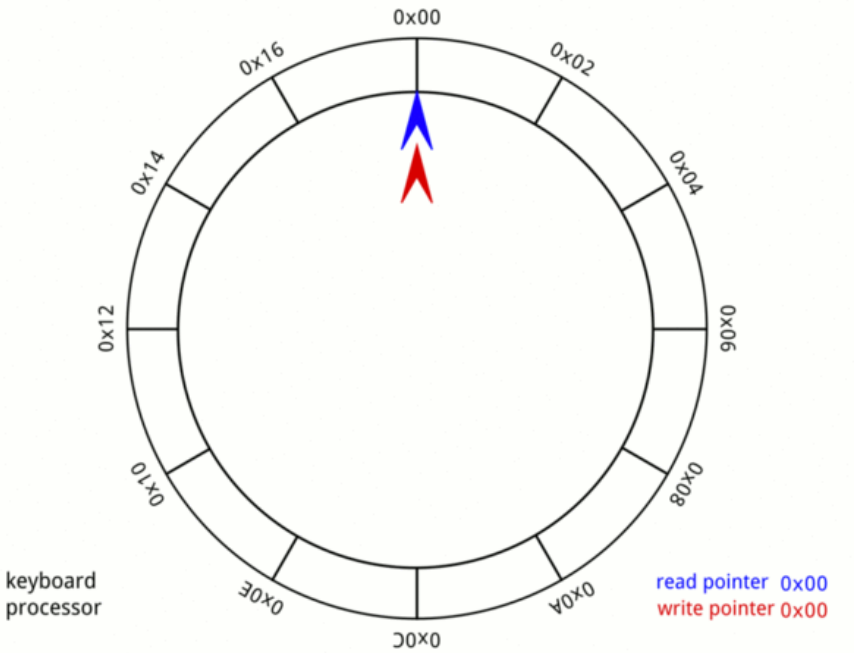
- each robot has its own FSM:
  - each state has different control settings.
  - each state reacts to different events.

- the task's FSM coordinates these two robot FSMs:
  → by sending events.
  → LCSM events are essential!

Architectures

Data exchange architectures

Data “communication” pattern: ring buffer



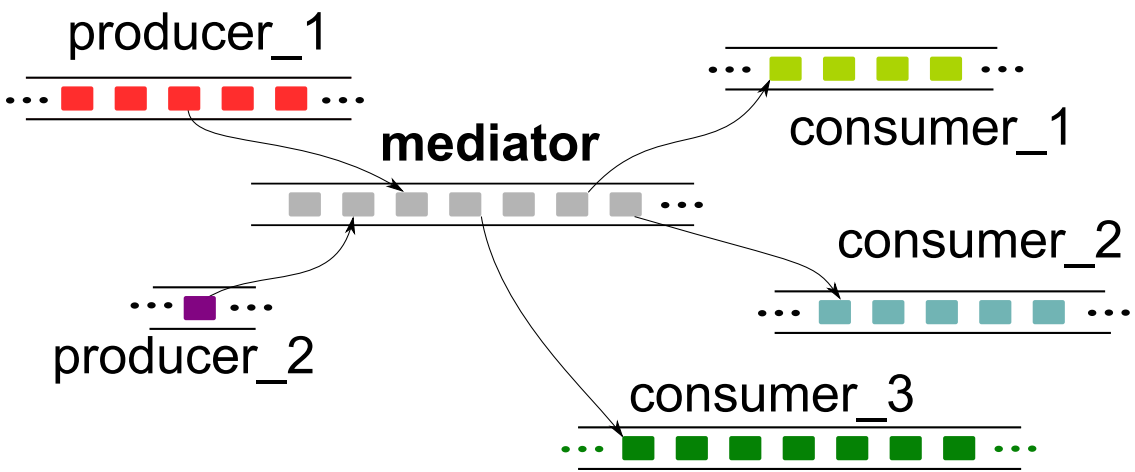
(See animation on Wikipedia.)

Principle:

- `producer` can fill the part of the buffer it **owns**.
- same holds for `consumer`.
- `producer` can **transfer** ownership to `consumer`, by advancing one pointer.
- same holds for `consumer`.
- ownership transfer can **always** be done without disrupting the `consumer`.

Result: “communication” of data with zero overhead!

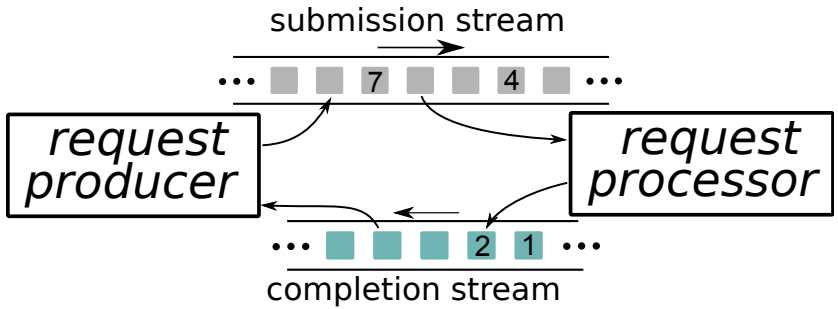
Data "communication" pattern: ring buffer (2)
Multiple producers – multiple consumers



Principle:

- `mediator` algorithm has one `ring buffer` "*stream*" with each `producer`.
- same holds for each `consumer`.
- `mediator` decides on the **transfer policy** between `producers` and `consumers`.

Submission-Completion architecture
For “dialogues” between algorithms



Use cases:

- database-like “queries”
- pointers to data structures, to “borrow” access.

- one algorithm *hands over* “**stuff to do**” to another algorithm.
- that algorithm returns processed results, *at its own pace*.
- each hand-over has a **unique ID**, and includes the ID of submitter & executor,
  → execution of “stuff” can be **traced**…
  …and **reacted** upon.

Event loop architecture

Event loop architecture: “execution engine”

Decouples the Communicate, Coordinate, Configure, Compute parts in algorithms

```
when triggered  // by operating system, which deals with all
                // asynchronous side effects.
 do {           // the serial control flow structure of the event loop.
  communicate() // get all "messages" with events, data & queries,
                // provided by other asynchronous algorithms.
  coordinate()  // handle the events in these messages, and
                // decide which ones to react to.
  configure()   // some events imply reconfiguration of event loop.

  compute()     // execute your (serialized set of) synchronous algorithms,
                // which in themselves are side effect-free computations.

  coordinate()  // the computations above can generate events that imply
                // reconfiguration, of this event loop or other algorithms.
  communicate() // the computations above can generate events, data & queries
                // that other asynchronous algorithms must know about.

  sleep()       // the loop deactivates itself, until the earliest deadline
                // (default, or requested in the steps above).
 }
```

Further reading

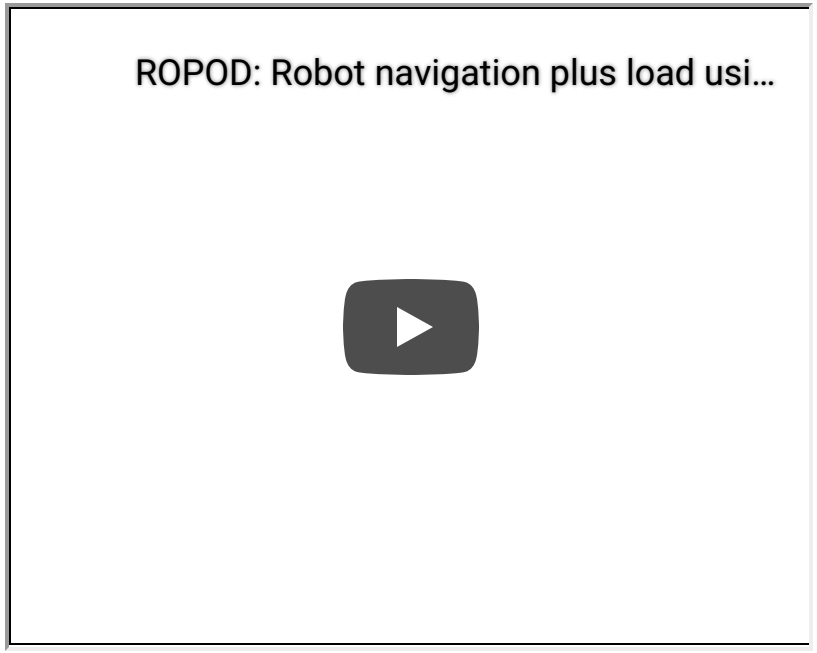
https://robmosys.pages.mech.kuleuven.be/.
Especially Chapters 2 and 5.

Wiki pages of H2020 project RobMoSys

Questions or remarks: contact me at:

herman . bruyninckx at kuleuven . be

Lots of algorithms…



(Direct link to video above.)

- **sensor processing**: encoders, accelerometers, laser distance sensors,…
- **motion control**: velocity or torque control for every wheel, task control of the whole platform,…
- **task execution**: decide when to switch to which part of the robot's task plan.
- **monitoring**: check all constraints and assumptions that should not be violated.
- **resources**: configure and interface hardware, communication, CPU,…

Any robot controller runs *a lot* of algorithms *"at the same time"*!

n