Embedded Motion Control

——

Do's and Don'ts in the design of a robotic software architecture

Herman Bruyninckx
Eindhoven University of Technology
KU Leuven
http://people.mech.kuleuven.be/~bruyninc/

May 1, 2019

Architecture: hardware, software, information

Hardware:

- CPUs, memory
- communication lines (Ethernet, device IO,. . .)

Software:

- processes + threads in processes
- communication between processes
- shared data between threads

Information:

- data structures + functions that change them
- **activities**: own data + exchange data + schedule functions
- **tasks** that must be realised

Architecture: hardware, software, information

Hardware:

- CPUs, memory
- communication lines (Ethernet, device IO,...)

Software:

- processes + threads in processes
- communication between processes
- shared data between threads

Information:

- data structures + functions that change them
- **activities**: own data + exchange data + schedule functions
- **tasks** that must be realised

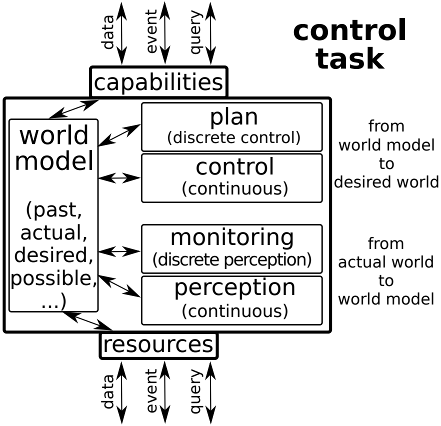
Do:

- information architecture first, software architecture later
- hardware architecture: is a given in this course

Don't assume information is:

- available all the time,
- available instantaneously,
- fresh,
- consistent,
- accurate

Design driver: what activities are needed for each task?



Design = **identify** and **integrate**:

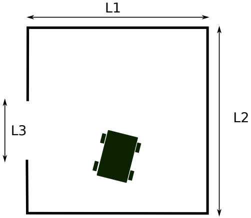
- *capabilities*: what does the application offer?
- *resources*: what does it rely on?
- *plan*: discrete set of "behavioural" states
- *control*: continuous-time feedback/feedforward
- *monitoring*: system dynamics trigger events
- *perception*: continuous-time sensor processing
- *world model*: **state** of "**everything**" that "**everyone**" must **know about**

The "world model" is a key activity in your architecture: it is the <u>memory</u> for all the other activities

Example task: escape from room

Plan:

1. initialize sensors and motors
2. move forward till wall is detected
3. follow wall on the right
4. turn right at first “hole”
5. stop



Resources:

- laser range finder: series of rays indicating free space, within minimal & maximum measurement range.
- encoders: actual velocity of robot.
- velocity control: instantaneously desired velocity of robot.
- effort value: percentage of “full available power” used for robot motion.
- keyboard button: events from keyboard

Design driver: abstraction & resolution of world model

Abstraction:

- is there a map?
- separate topology and geometry
- which primitives?
- polygonal?
- 1D? 2D? 3D?

Resolution:

- no more/less **spatial** detail than your task requires
- no more/less **temporal** detail than your task requires

Design driver: abstraction & resolution of world model

Abstraction:

- is there a map?
- separate topology and geometry
- which primitives?
- polygonal?
- 1D? 2D? 3D?

Resolution:

- no more/less **spatial** detail than your task requires
- no more/less **temporal** detail than your task requires

Do:

- throw **data** away when you know what **information** you're looking for
- remember the past
  $\rightarrow$ don't repeat yourself
- predict the future
  $\rightarrow$ only way to **monitor progress**
- use local references, all the time

Don't:

- use one global reference frame
- use grids (use polygons!)

Relation between sensing, planning, and control

- **you don't do planning**, but you select which plan to use at which time!
- you hardcode all plans that your robot could need $\rightarrow$ *explainability*!
- each phase in a plan:
  - corresponds to a particular expected situation in the task $\rightarrow$ explicit *intent*!
  - selects (i) map, (ii) sensor processing activity to update map, (iii) control activity to generate motion, (iv) monitor activity to generate phase-switching events.
  - needs a **measure of progress**

Relation between sensing, planning, and control

- **you don't do planning**, but you select which plan to use at which time!
- you hardcode all plans that your robot could need → *explainability*!
- each phase in a plan:
  - corresponds to a particular expected situation in the task → explicit *intent*!
  - selects (i) map, (ii) sensor processing activity to update map, (iii) control activity to generate motion, (iv) monitor activity to generate phase-switching events.
  - needs a **measure of progress**

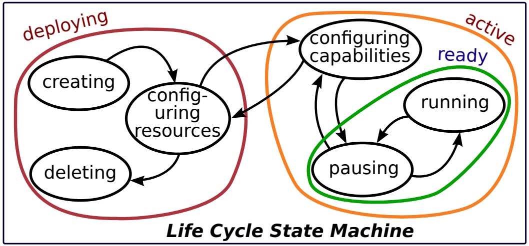
Do introduce a *Finite State Machine*:

- there are always different *behavioural states* in the execution of a task.
- **monitor** the assumptions in each state
- **monitor** the progress of the task
- separate monitoring from **decision making**

Don't rely on *instantaneous* behaviour:

- *Sense-Plan-Act* architecture
- *potential fields*
- feedback/feedforward with time window of one sample

Pattern: Life Cycle State Machine



- every **activity** needs one
- every **task** needs one
- every **resource** needs one
- *m*-to-*n* relation monitor $\leftrightarrow$ transition + dependency on *context*

$\rightarrow$ nesting is needed. . . !

$\rightarrow$ separate activity is needed for FSM(s). . . ! (called "Coordinator")

Sensing, monitoring, control, plan execution, world model updating: are all activities

Do:

- assign ownership of each data structure to one, and only one, of those activities
- allow an activity to read data owned elsewhere
- allow an activity to advice other activities to update data
- allow an activity to transfer ownership to another activity

Don't:

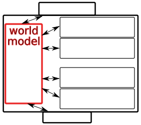
- expect all activities to execute instantaneously
- send around all data all the time, to all activities
- connect sensing and control without world model in between (Only there is the **context** needed to interpret information and to configure activities)

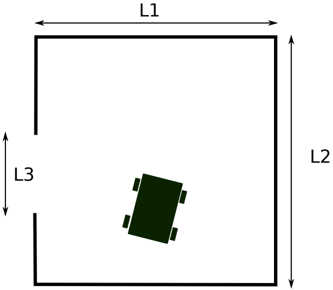
Event Loop: software pattern for an Activity

```
when triggered  // by operating system
 do {
  communicate() // get data from other activities
  coordinate()  // decide what phase of plan to switch to
  configure()   // set all parameters and select functions
  compute()     // execute control, perception, monitoring, plan
                // functions synchronously, one after the other
  coordinate()  //
  communicate() // send data to other activities
  sleep()       // the loop deactivates itself, until next deadline
 }
```

Next slides: relevant snippets of a design

Initial world model: parameterized room with a hole



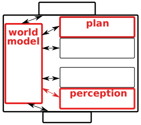


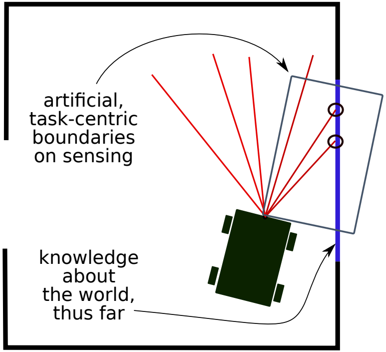
At **initialization**, this is **assumed**:

- the robot is *inside* a room
- the room has a *rectangular* shape as in the figure
- the room has *one door*, with a *width enough* to let the robot pass through
- the position and orientation of the robot in the room are *not known*
- the size of the room is *not known*

→ enough to encode world model + relevant plans!

Perception (sensor processing) during "Follow wall"



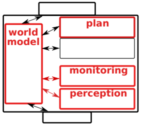


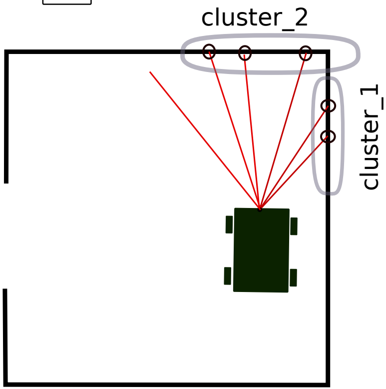
The sensor provides more data than strictly necessary to do the job:

- select a *region of interest* (grey box) that fits to the *plan* (= only interested in right-hand side)
- fit a *line* through a *large enough* cluster of measurements
- do this over a *time window* of measurements
- monitor whether nothing is closer than the line

So, *perception* ⇔ least-squares fitting of a line of limited length through a clever, plan-directed selection of current and previous "hits"

Monitoring to decide to add next wall to control scope

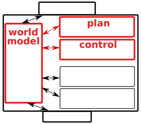


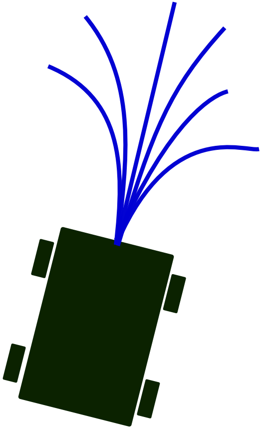


Monitoring has four **hypotheses** to follow:

1. *local* horizon to fit *wall*, on the *right*, as expected by the *task* context, to configure the *control*.
2. further horizon in *forward* direction:
   - 2.1 to monitor whether there is "something", to react to in *plan*;
   - 2.2 to find another *line cluster*, orthogonal to first one, to update the *world model* with a new *corner*.
3. the leftmost rays can be discarded
   → reduces the computational load.
   → improves interpretation of the data
4. *all* measurements *could* be *neglected* until needed again, based on *planned* speed of motion.

Lazy control: “power + steer” motions





One easy **possibility** for “control”:

- apply a set of constant speeds to each wheel $\rightarrow$ set of known trajectories of the robot in the near future, to choose from
- sparsity/density of trajectories can be chosen, in a plan-directed way
- time/space horizon of trajectories can be chosen, in a plan-directed way
- **control** can be as simple as **selecting** the “best” trajectory and apply the corresponding wheel velocities during a **long** period
- **Do**: separate control of (i) path (“steer”), and (ii) velocity along path (“power”)

Motion specification paradigm of "Guarded motion"

Combines open loop motion with monitoring

Compares to mainstream paradigm of:

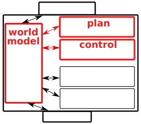
- motion *specification* via *trajectories*
- motion *control* via *trajectory tracking*

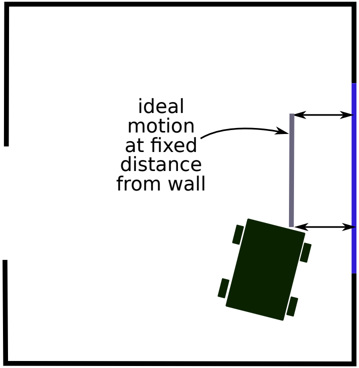
Advantages of *guarded motion* approach:

- motion need not be *specified* in full configuration space of *control*
  $\rightarrow$ avoids bringing in artificial constraints for task
  $\rightarrow$ allows to comply to physical constraints of hardware
- sensing and control are *decoupled* via world model + plan
  $\rightarrow$ decision about *behaviour* is *owned* by plan activity
  $\rightarrow$ present/past/future can be taken into account differently
  $\rightarrow$ dependence on **context** of task becomes easier!

Motion specification for control

First simple example



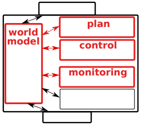


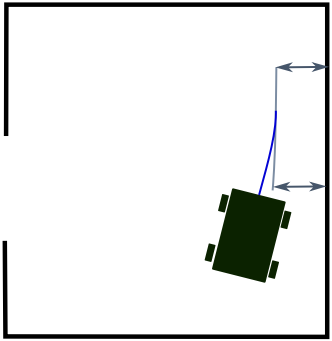
- *plan* places **reference trajectory** (grey line) in world model
- at fixed offset with respect to the best fitting wall line
- and with **goodness of fit** function for the actual robot motion

→ the controller need not change when representation and/or location of reference trajectory change

Control

Simple solution: best fitting open loop trajectory

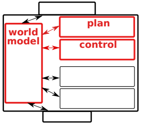


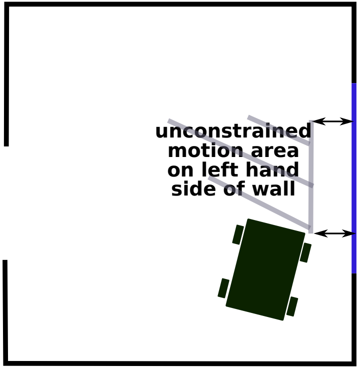


Control is simple:

- generate the "spray" of feedforward trajectories
- select a "good enough" fit
- apply corresponding open loop motor values
- until **monitoring** tells us that deviation becomes "too large"

Motion specification for control
Second simple example

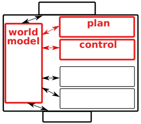


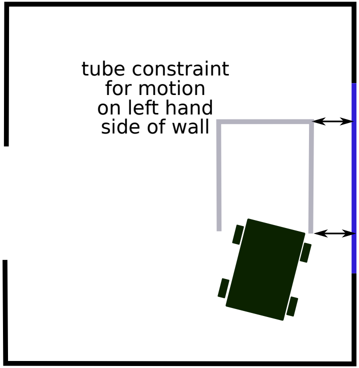


Another easy possibility:

- the robot is allowed to move "anywhere" left of the wall.

Motion specification for control
Third simple example



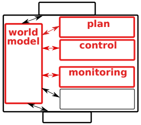


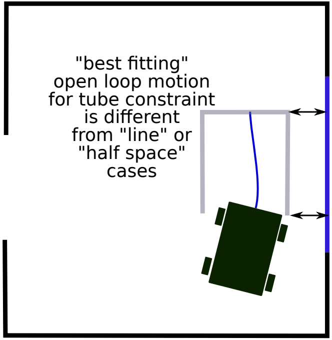
Another easy possibility:

- the robot is allowed to move "anywhere" inside a "tube" at some distance from the wall.

Control

Other simple solution: best fitting open loop trajectory





The *tube* approach has

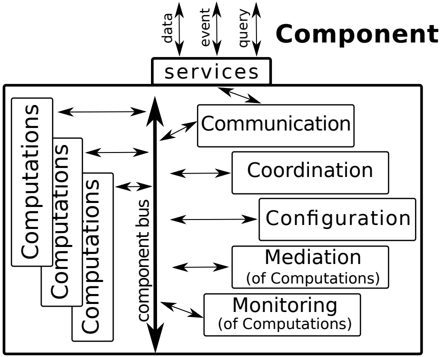
- other **optimal** trajectories
- other **tolerances**
- but same **monitoring**

Slides before: modelling
Now we need to go to software

- *models* must be turned into *data structures* in a programming language.
- *functions* on these data structures must be written, everywhere where the previous slides used *verbs* like “fit”, “place”, “select”,...
- the *order* of the computations (“*schedule*”) must be determined
- each *state* in the *plan* corresponds to one set of all of the above
- the *timing* of the computations (“*sampling*”) must be determined
- the *execution* of the computations (“*dispatching*”) must be done
- the *communication* with sensors, motors, keyboard,... must be realised

The **resource usage** *of all of the above must be* **mediated**!
(*mediation* = *monitoring* for saturation + reaction in *plan*)

“Component”: formalisation of the software
And a guideline to implement robot applications

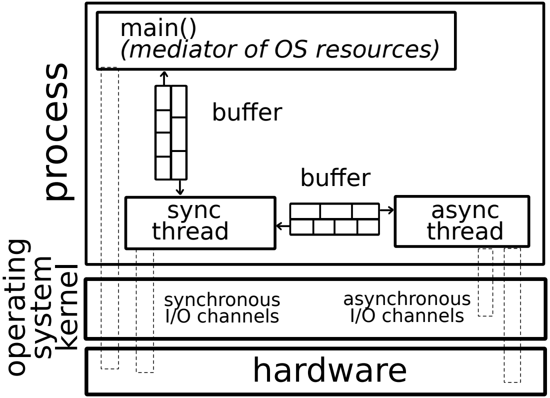


Component model **integrates** the following:

- *Computations*: all data + functions to execute
- *Communication*: read/write I/O data
- *Coordination*: decide to switch plan **state**
- *Configuration*: set right parameters
- *Mediation*: make trade-offs for scarce resources
- *Monitoring*: of CPU, BUS, RAM, IO *resources*

“Component”: can be a **process**, but also a **thread** inside a process.

Software pattern of threads in a process
Typical activity has three types of threads



- one **main**: configures threads, memory and communication
- one **synchronous thread**: event loop for "realtime" control, *never blocking*
- one or more **asynchronous threads**: each communicating with a resource or other activity, *possibly blocking*

Do's and Don'ts

in mapping activities to processes/threads

Do:

- separate synchronous and asynchronous parts in each activity
- couple them via buffers

- separate OS configuration of threads from implementation (priorities, memory reservation, IO reservation, timing,...)

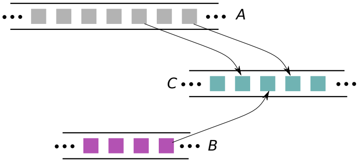
Don't:

- run just one activity in each thread, without knowing exactly why you don't run more
- use priorities on threads to influence scheduling order of activities

Exchanging data between threads and processes

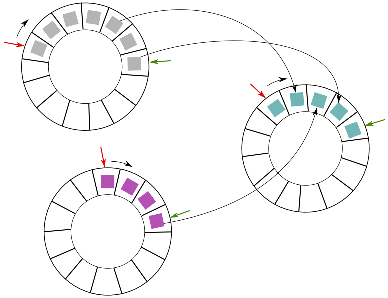
Inter-Processes Communication:

- *Publisher-Subscribe*, via individual data topics spread over different processes (**Don't**!)
- **Producer-Consumer stream**, via ringbuffer between one Producer activity and one Consumer activity (**Do**!)



Threads:

- shared data, protected via mutexes (**Don't!**)
- Producer-Consumer streams, via ringbuffers (**Do!**)



Next lecture . . . (?)

C reference implementations for:

- process/thread architecture: with mutex and without
- Producer/Consumer ringbuffer
- sensor fusion (same sensor over multiple times)

Bring your information architecture:

- and walk away with your software architecture. . .