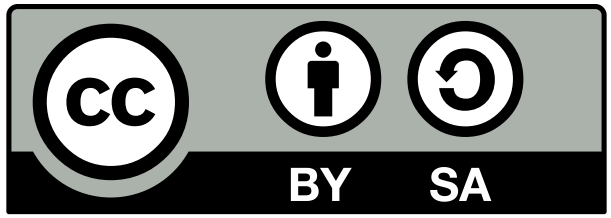
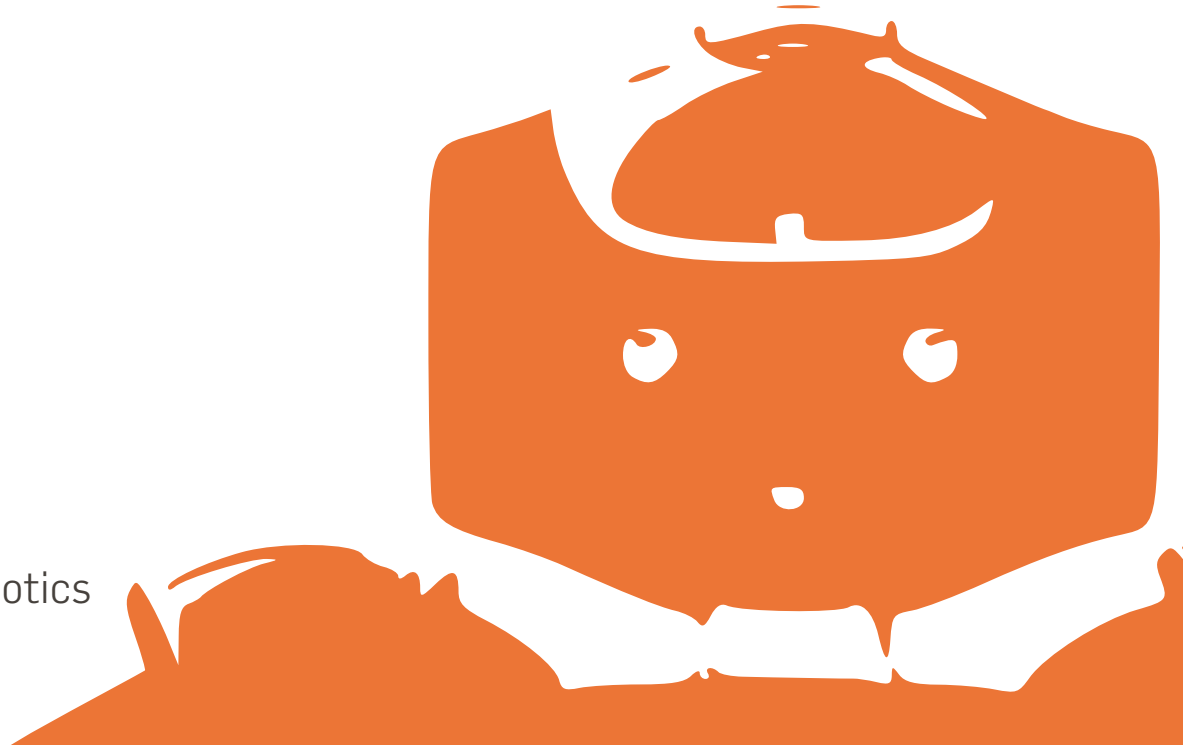


ROBOTICS WITH PLYMOUTH UNIVERSITY

ROCO318
Mobile and Humanoid Robots
Robot Control



Séverin Lemaignan

Centre for Neural Systems and Robotics
Plymouth University

How hard might it be?

LET'S THINKER A LITTLE...

Let's imagine you want to build a robot that **fetches beers from the fridge** and bring them back to you whenever you ask. It should **not kill the cat**, and it should **politely greet your mum** whenever it sees her.

You have:

- a map with the important landmarks like `fridge`
- the following modules:

`speech_synthesis` `open_fridge`

`pick_up_beer` `detect_cat`

`follow_path_to`

`plan_to` `detect_mum`

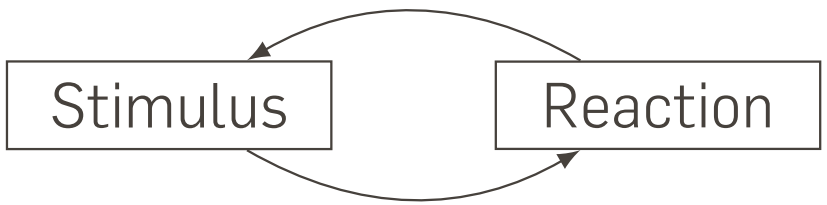
`process_verbal_commands`

Can you draw a control architecture that achieves just that?

CONTROL PARADIGMS

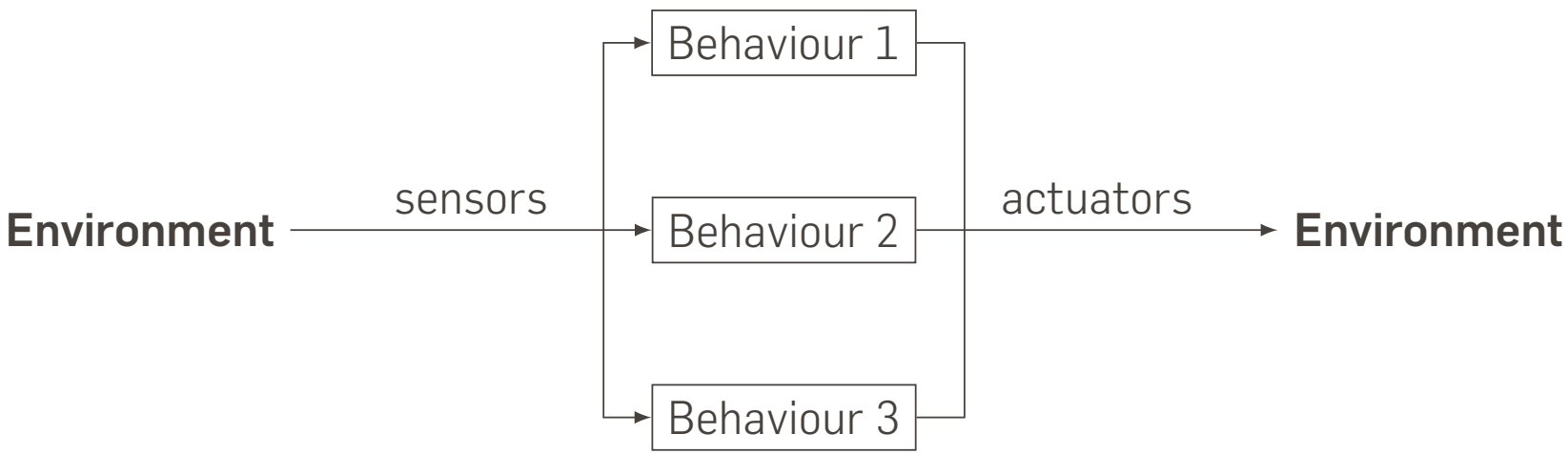
BEHAVIOURAL (OR REACTIVE) CONTROL

Behaviours are small programs that read sensors and control actuators. Each behaviour does **one simple thing**; it typically has access to all sensors/actuators.



BEHAVIOURAL (OR REACTIVE) CONTROL

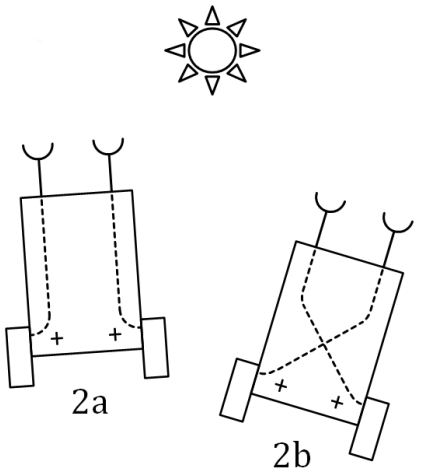
Behaviours are small programs that read sensors and control actuators. Each behaviour does **one simple thing**; it typically has access to all sensors/actuators.



We only want one behaviour at a time! Need to **prioritise**: behaviours can *override* or **subsume** less important ones.

EXTREME CASE: BRAITENBERG MACHINES

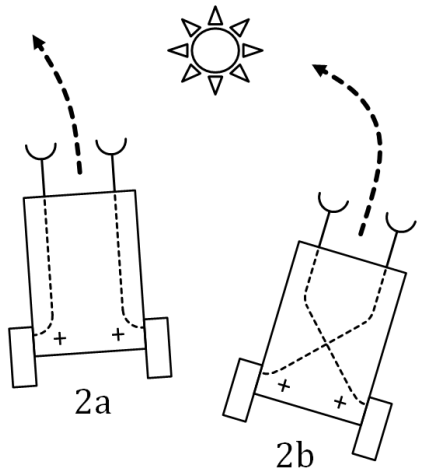
- Motion directly controlled by sensors (typically photocells)
- Yet the resulting behaviour may appear complex or even intelligent
- Can you guess the behaviours of vehicles 2a and 2b?



Source: Wikipedia

EXTREME CASE: BRAITENBERG MACHINES

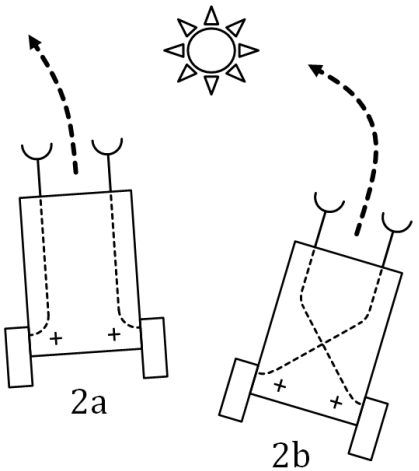
- Motion directly controlled by sensors (typically photocells)
- Yet the resulting behaviour may appear complex or even intelligent
- Can you guess the behaviours of vehicles 2a and 2b?

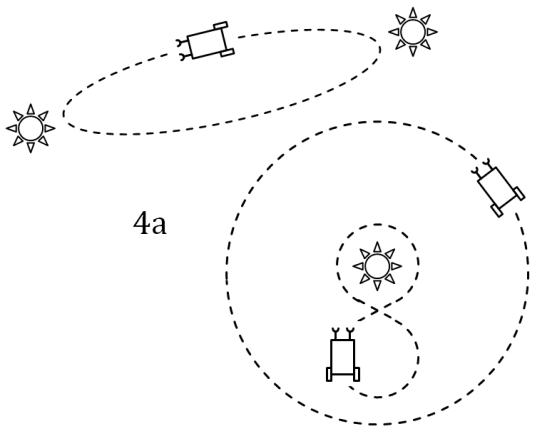


Source: Wikipedia

EXTREME CASE: BRAITENBERG MACHINES

- Motion directly controlled by sensors (typically photocells)
- Yet the resulting behaviour may appear complex or even intelligent





- Complex behaviours emerge (typically with non-linear control functions).

Source: Wikipedia

COMBINING BEHAVIOURS: EXAMPLE

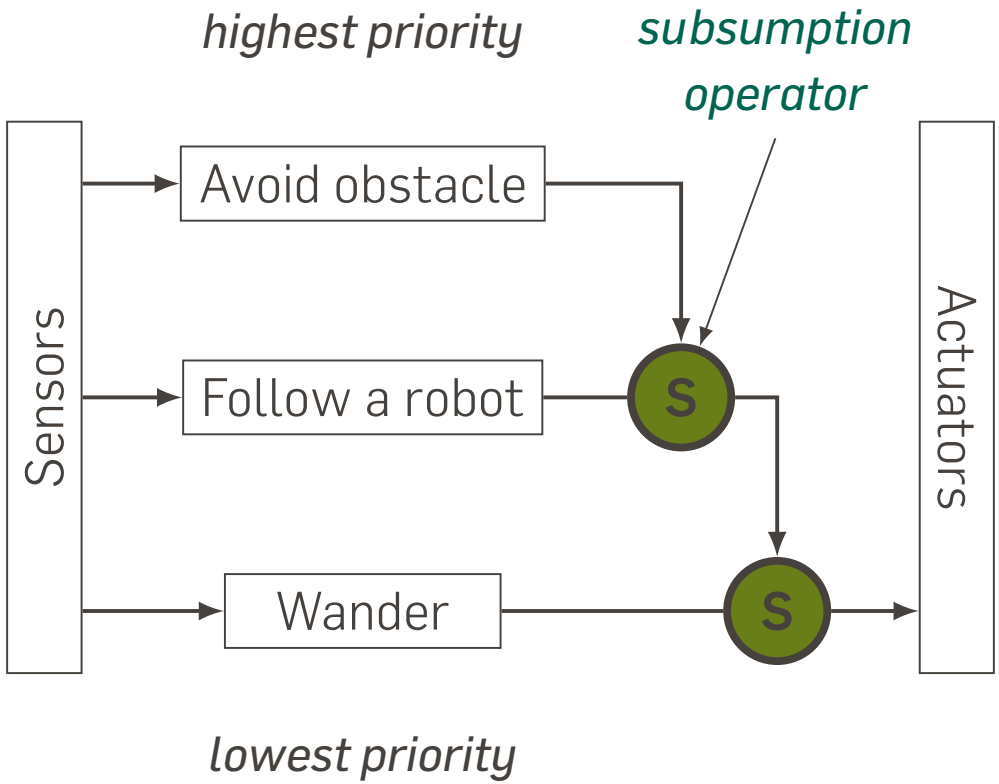
Three behaviours:

- **Follow a robot**
  - Sensor: IR communication
  - Behaviour: follow another robot
- **Avoid obstacle**
  - Sensors: bumpers
  - Behaviour: move away from the wall
- **Wander**
  - Sensors: encoders
  - Behaviour: move forward and turn

Which behaviour should have the highest priority? the lowest?

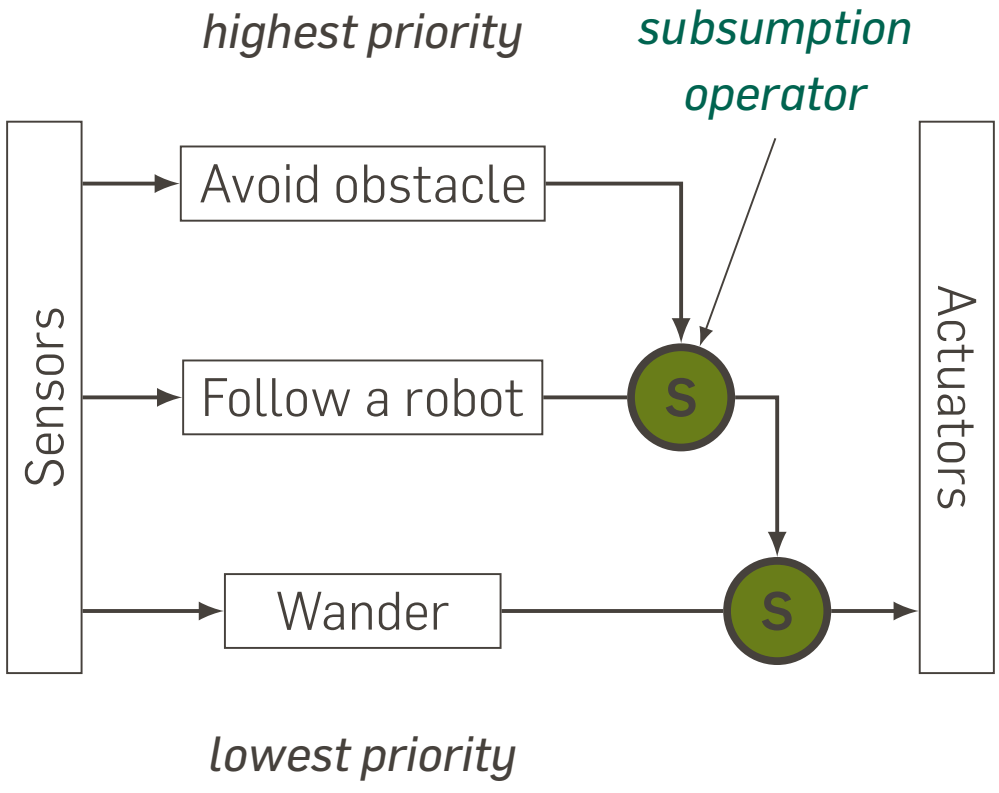
Source: example borrowed from Rice University ENGI128

COMBINING BEHAVIOURS: EXAMPLE



We combine behaviours by **subsuming** lower-priority behaviours whenever a higher-priority behaviour becomes active.

COMBINING BEHAVIOURS: EXAMPLE



We combine behaviours by **subsuming** lower-priority behaviours whenever a higher-priority behaviour becomes active.

→ **subsumption architecture**

BEHAVIOURAL CONTROL: STRENGTHS/WEAKNESSES

Strengths

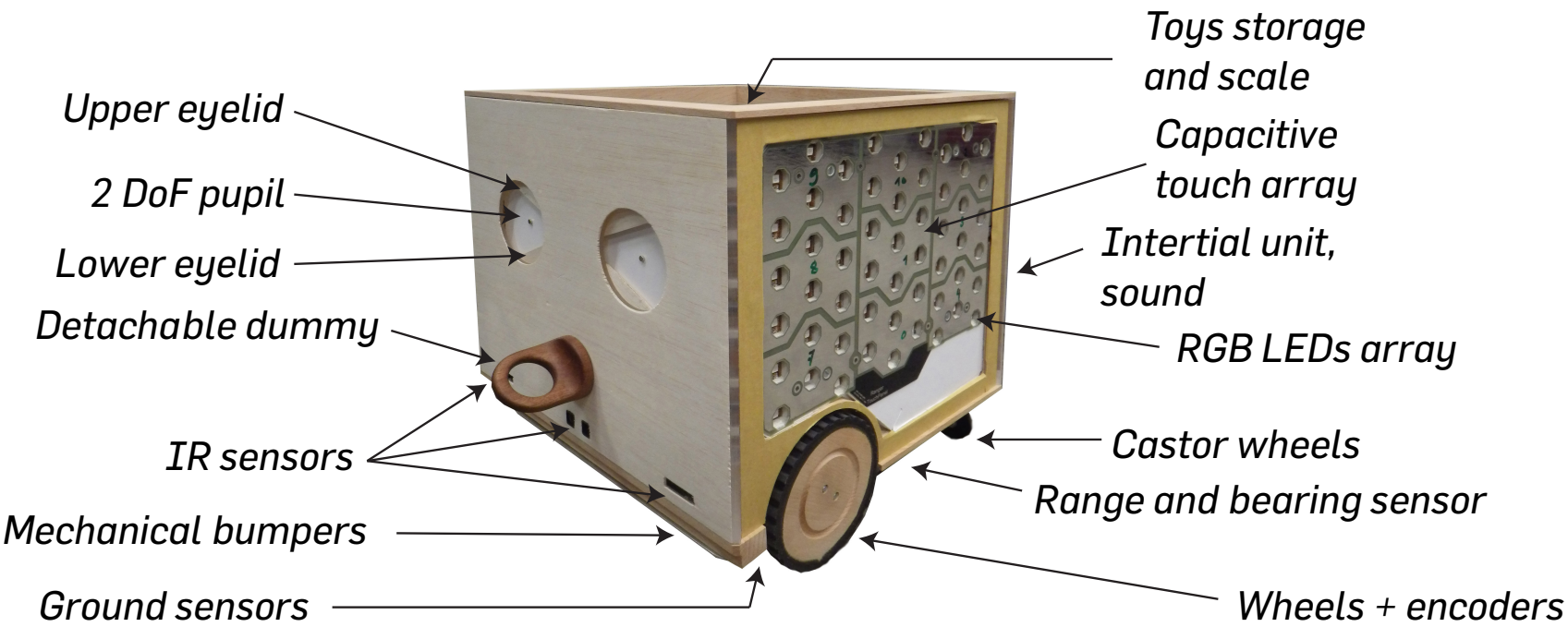
- **Incremental development**
- By definition **modular**
- Effective to react to events → well suited to **dynamic environments**

Weaknesses

- **goal-oriented behaviours hard to implement** ("what will my robot do?")
- **importance of the arbiter**: who inhibits (i.e. ***subsumes***) who might be context-dependent
- debugging difficult (need to trace which behaviours are active)

EVENT-ORIENTED PROGRAMMING: EXAMPLE OF RANGER

Ranger is a ‘box on wheels’ developed at EPFL



EVENT-ORIENTED PROGRAMMING

Event-oriented programming is a possible way of implementing a behavioural control paradigm:

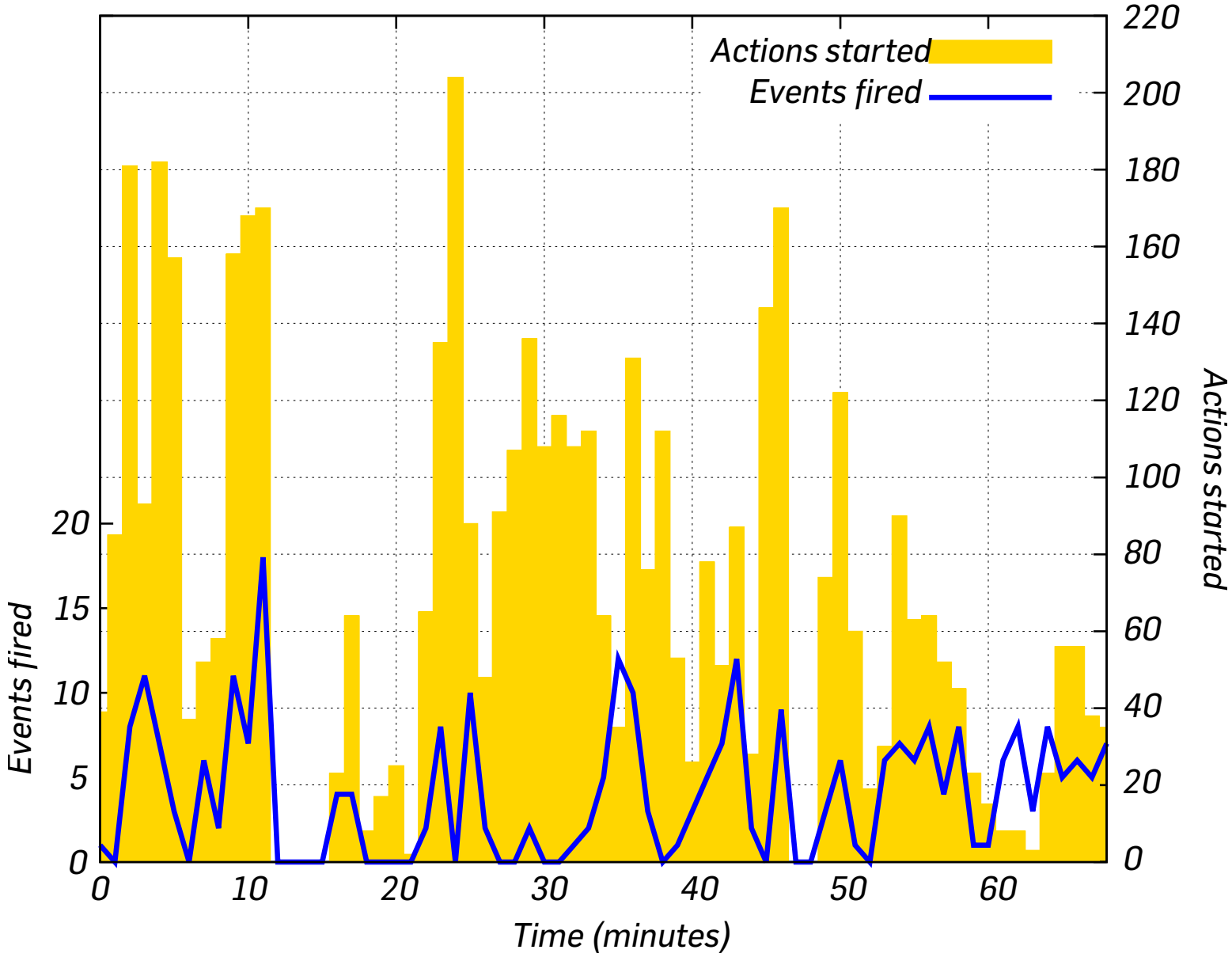
```
with Ranger() as robot:

    robot.background_blink()
    robot.look_at_touches()

    robot.whenever("dummy", becomes = True)
                                .do(on_dummy)
    robot.whenever("dummy", becomes = False)
                                .do(on_dummy_removed)
    robot.whenever("scale", increase = 0.3).do(on_toy)
    robot.whenever("bumper", becomes = True).do(on_bumper)

    while True:
        time.sleep(0.1)
```

EVENT-ORIENTED PROGRAMMING

Event-oriented programming is a possible way of implementing a behavioural control paradigm:

```
def on_dummy(robot):
    robot.look_at_dummy()
    robot.blink()
    sleep = robot.fall_asleep()
    robot.lightbar(RAINBOW).wait()
    sleep.wait()

def on_dummy_removed(robot):
    robot.light_bar(colors.rand())
    robot.wakeup().wait()
    robot.move(0.4, v = 0.8).wait()
    robot.idle().wait()
```

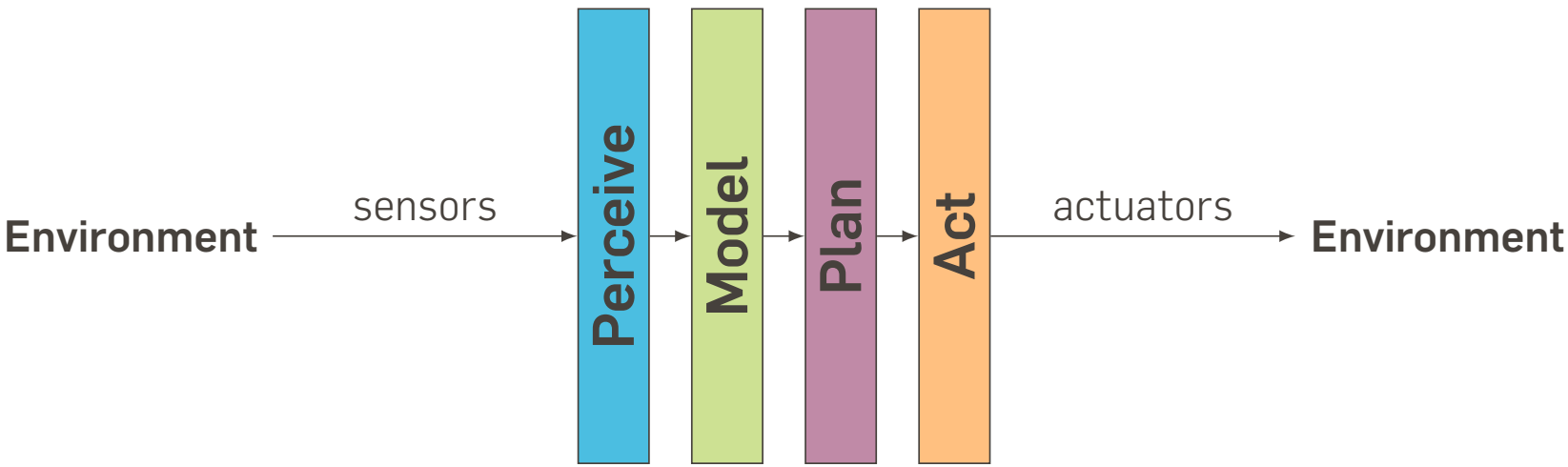
```
def on_bumper(robot):
    pulse = robot.pulse_row(0)
    while abs(robot.state.v) > 0.01:
        robot.sleep(0.2)
    pulse.cancel()

def on_toy(robot):
    robot.playsound(SOUNDS["toy_in"])
    robot.lightbar(RAINBOW).wait()
```


Actions started
Events fired
Events fired
Actions started
Time (minutes)
0
5
10
15
20
0
20
40
60
80
100
120
140
160
180
200
220
0
10
20
30
40
50
60

Note that this is a example of **aimless, purely reactive**, behaviour.

MODEL-PLAN-ACT

Basic paradigm for **deliberative architectures**.



TASK PLANNING

Turn a high-level goal (*bring me a beer!*) into 'simple' **primitive** actions.

A standard approach relies on **Hierarchical task networks** (**HTN**): uses partial-order contraints to *decompose actions* into *primitive operators*.

TASK PLANNING

Turn a high-level goal (*bring me a beer!*) into 'simple' **primitive** actions.

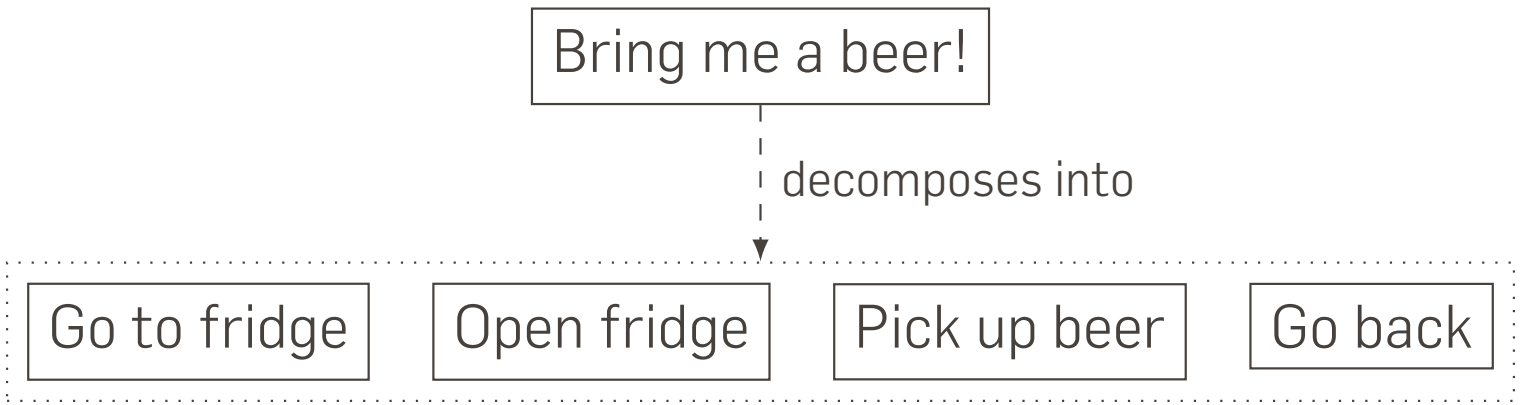
A standard approach relies on **Hierarchical task networks** (**HTN**): uses partial-order contraints to *decompose actions* into *primitive operators*.

Bring me a beer!

TASK PLANNING

Turn a high-level goal (*bring me a beer!*) into 'simple' **primitive** actions.

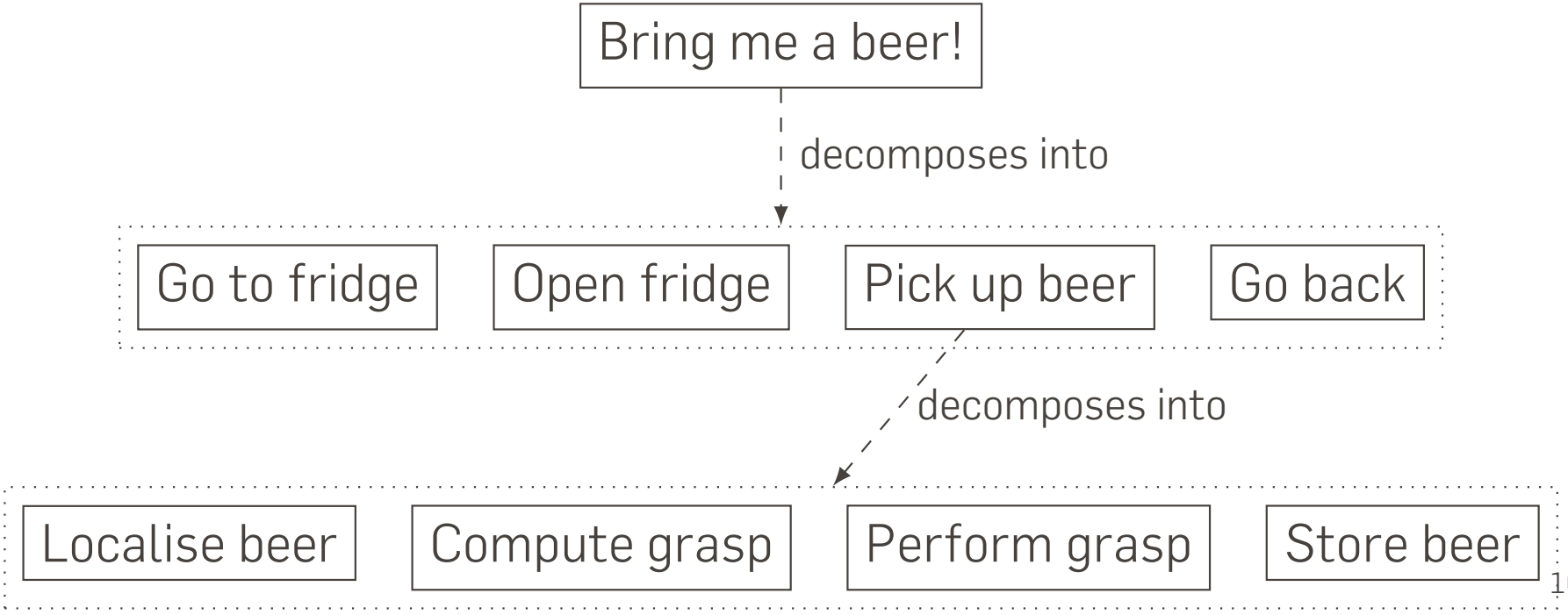
A standard approach relies on **Hierarchical task networks** (**HTN**): uses partial-order contraints to *decompose actions* into *primitive operators*.



TASK PLANNING

Turn a high-level goal (*bring me a beer!*) into 'simple' **primitive** actions.

A standard approach relies on **Hierarchical task networks** (**HTN**): uses partial-order contraints to *decompose actions* into *primitive operators*.



TASK PLANNING

What ***primitive action*** means is system-dependent: what an agent considers as primitive can be another agent's plans.

TASK PLANNING: ACTIONS

An action has **pre-conditions** and **post-conditions** (or *effects*).

```
Action <action name>
{
    preconditions {...};
    effects{...};
    cost{<cost_function_name>};
    duration{<duration_function_name>};
}
```

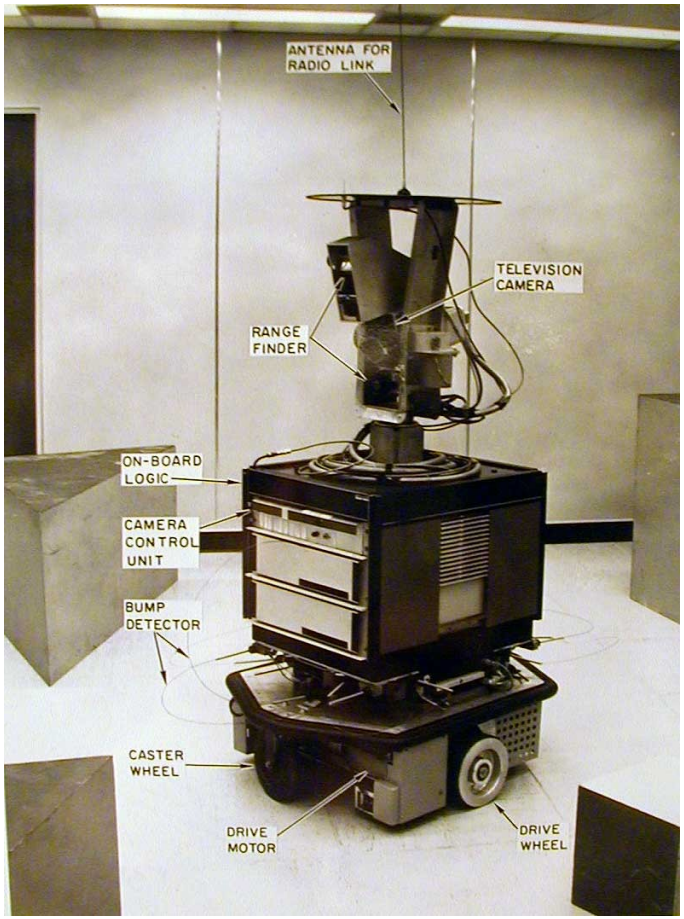
TASK PLANNING: ACTIONS

An action has **pre-conditions** and **post-conditions** (or *effects*).

```
Action <action name>
{
    preconditions {...};
    effects{...};
    cost{<cost_function_name>};
    duration{<duration_function_name>};
}

Action open_fridge
{
    preconditions {facing_fridge AND fridge_door.closed};
    effects {facing_fridge AND fridge_door.open};
}
```

EXAMPLE: SHAKEY THE ROBOT

Shakey the robot (Stanford, 1968), using the **STRIPS** planner



Go ...

Go to object bx

GOTOB(bx)

Preconditions: TYPE(bx,OBJECT),(∃rx)[INROOM(bx,rx) ∧ INROOM(ROBOT,rx)]
Deletions: AT(ROBOT,$1,$2), NEXTTO(ROBOT,$1)
Additions: *NEXTTO(ROBOT,bx)

Go to door dx.

GOTOD(dx)

Preconditions: TYPE(dx,DOOR),(∃rx)(∃ry)[INROOM(ROBOT,rx) ∧ CONNECTS(dx,rx,ry)]
Deletions: AT(ROBOT,$1,$2), NEXTTO(ROBOT,$1)
Additions: *NEXTTO(ROBOT,dx)

Go to coordinate location (x,y).

GOTOL(x,y)

Preconditions: (∃rx)[INROOM(ROBOT,rx) ∧ LOCINROOM(x,y,rx)]
Deletions: AT(ROBOT,$1,$2), NEXTTO(ROBOT,$1)
Additions: *AT(ROBOT,x,y)

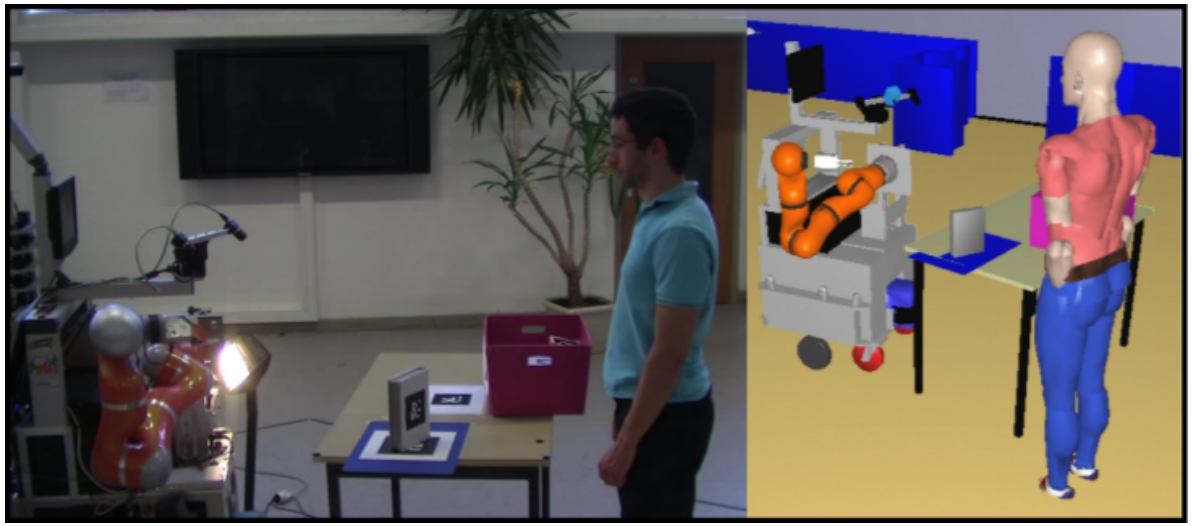
Go through door dx into room rx.

GOTHRUDR(dx,rx)

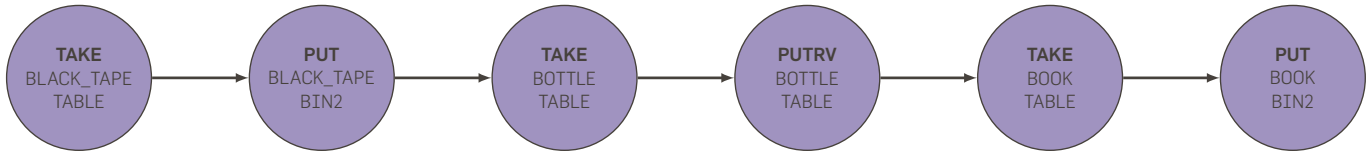
Preconditions: TYPE(dx,DOOR), STATUS(dx,OPEN), TYPE(rx,ROOM),
NEXTTO(ROBOT,dx) (∃rx)[INROOM(ROBOT,ry) ∧ CONNECTS(dx,ry,rx)]
Deletions: AT(ROBOT,$1,$2), NEXTTO(ROBOT$1), INROOM(ROBOT,$1)
Additions: *INROOM(ROBOT,rx)


LAAS-CNRS

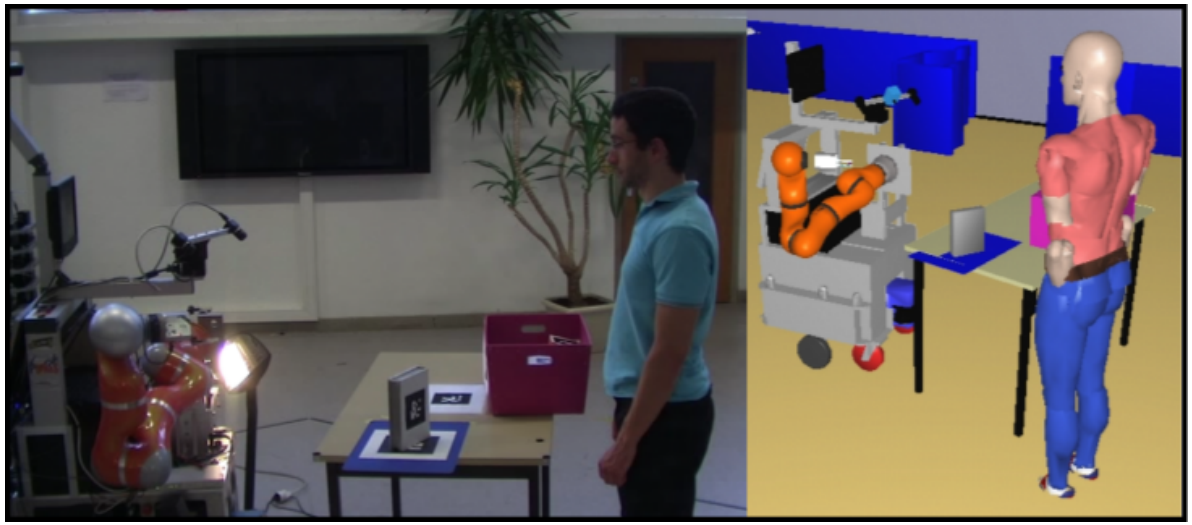
TASK PLANNING

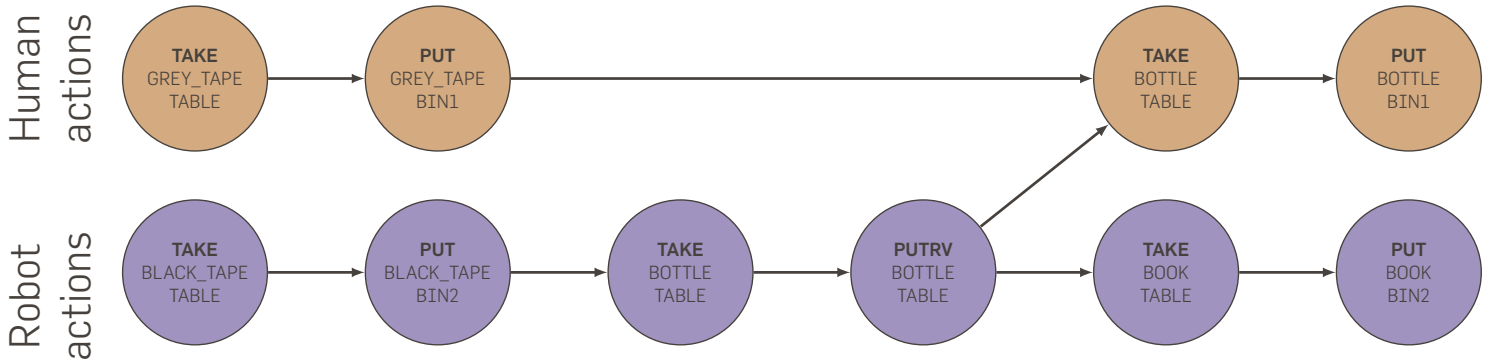


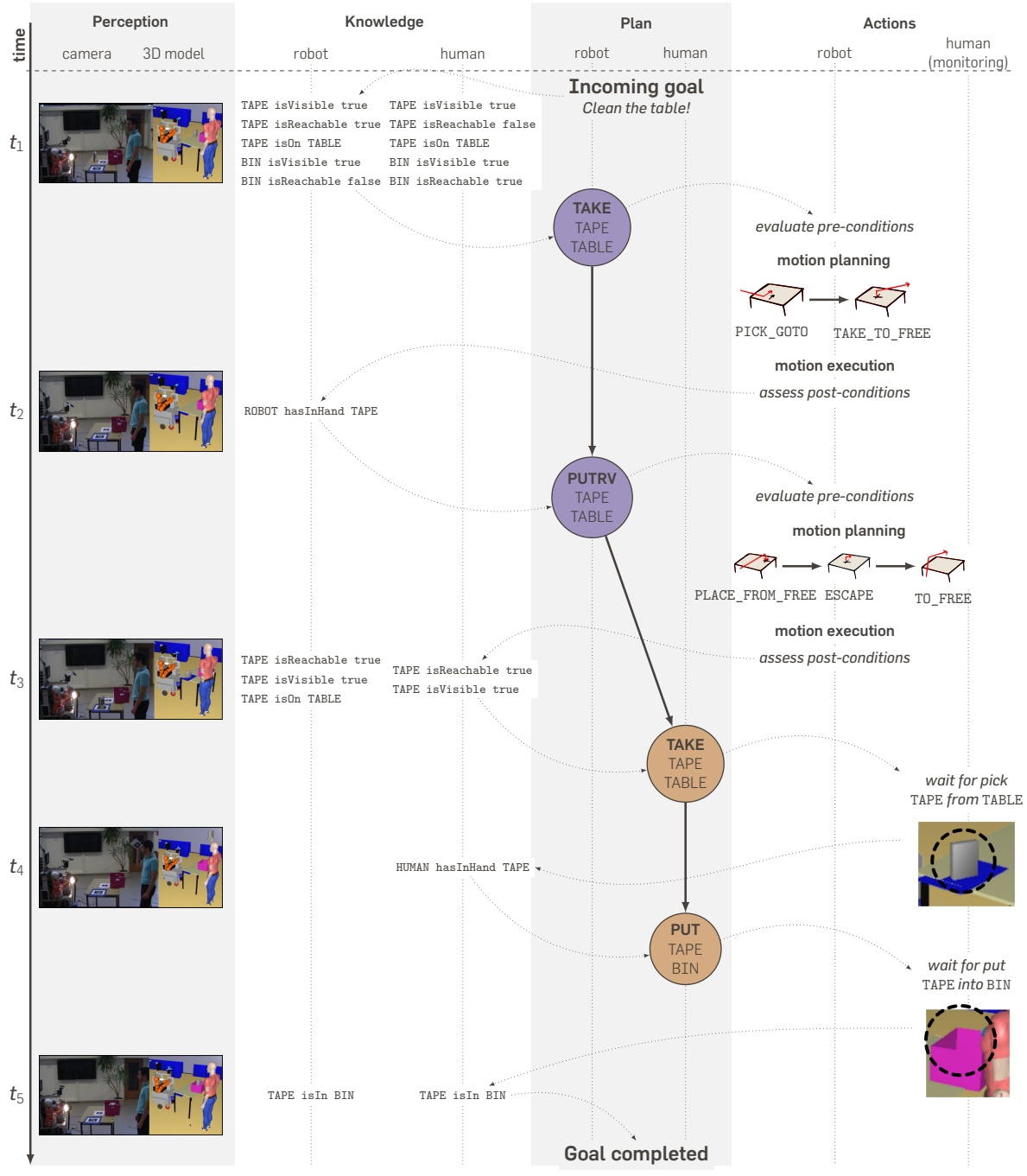
Robot actions

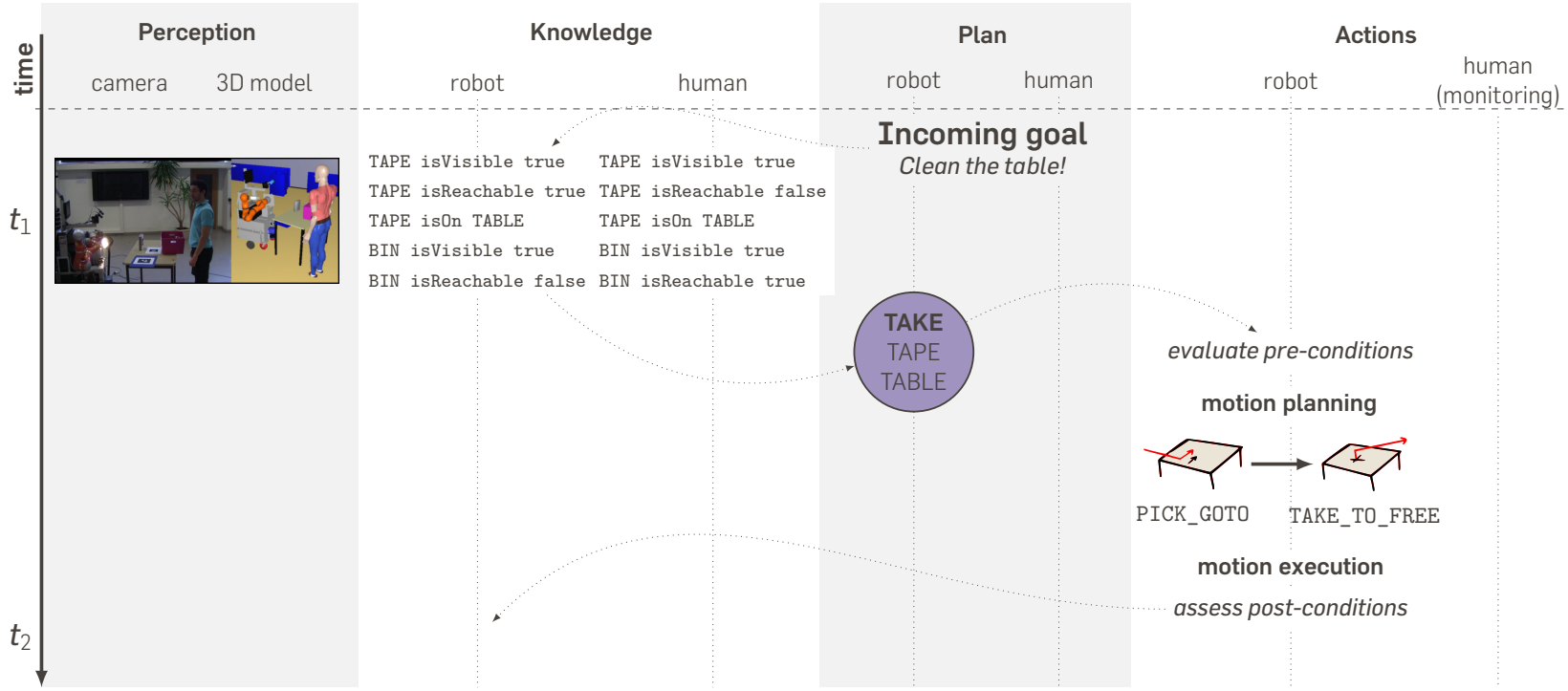


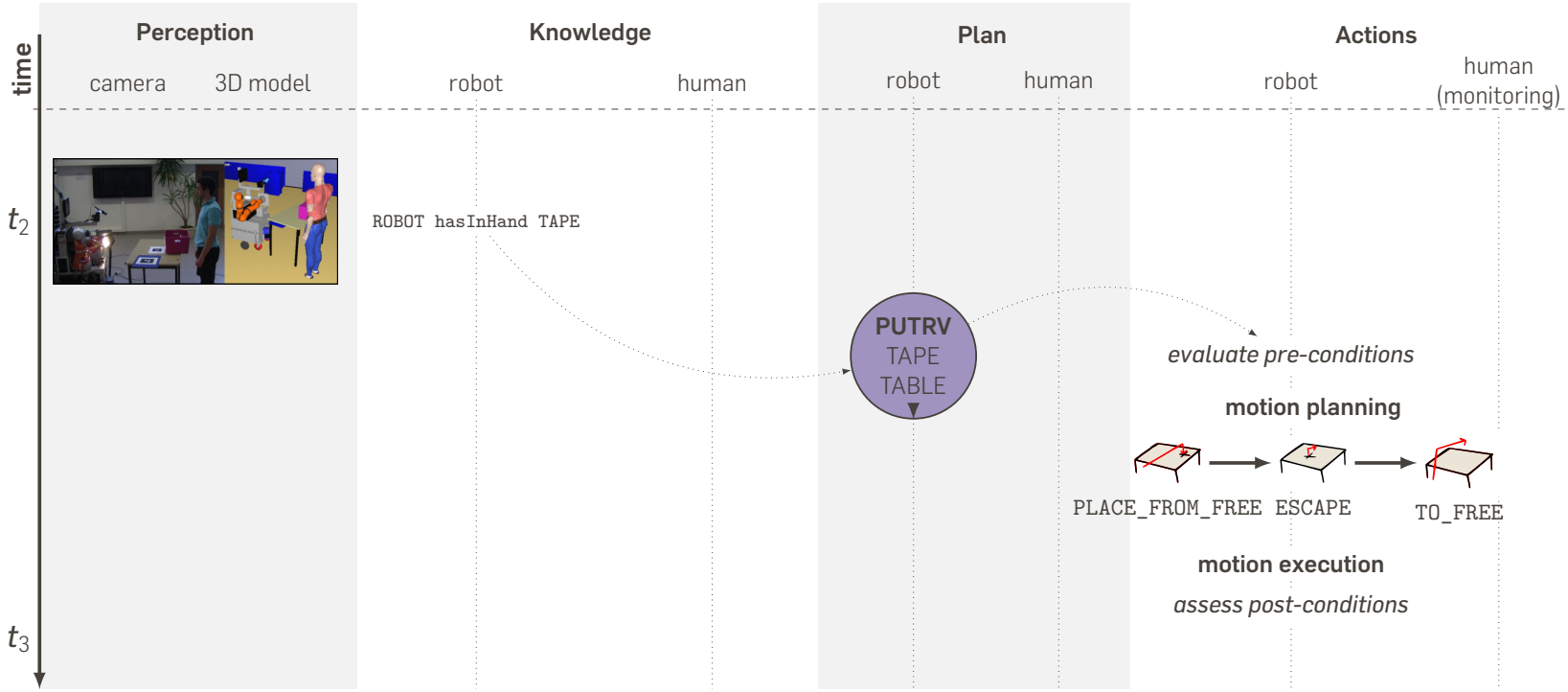
TASK PLANNING

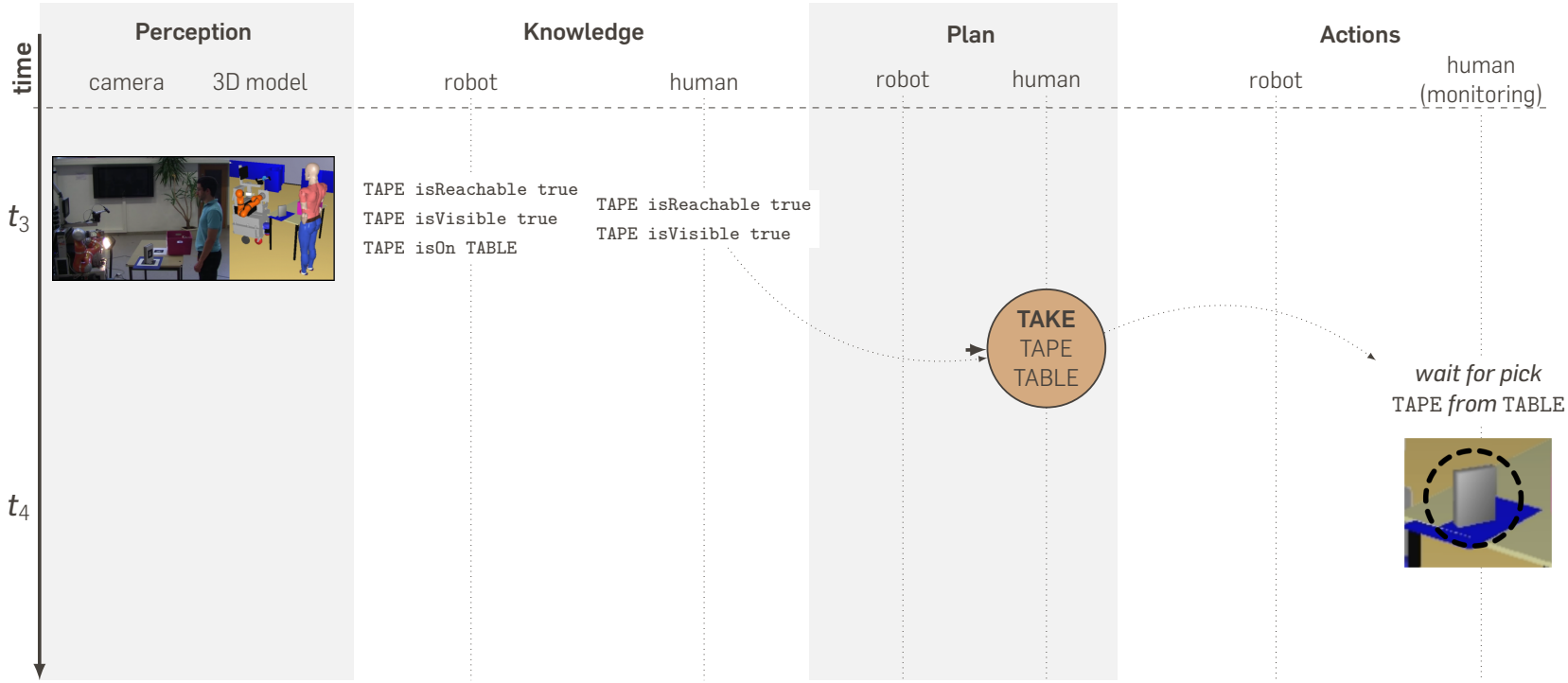


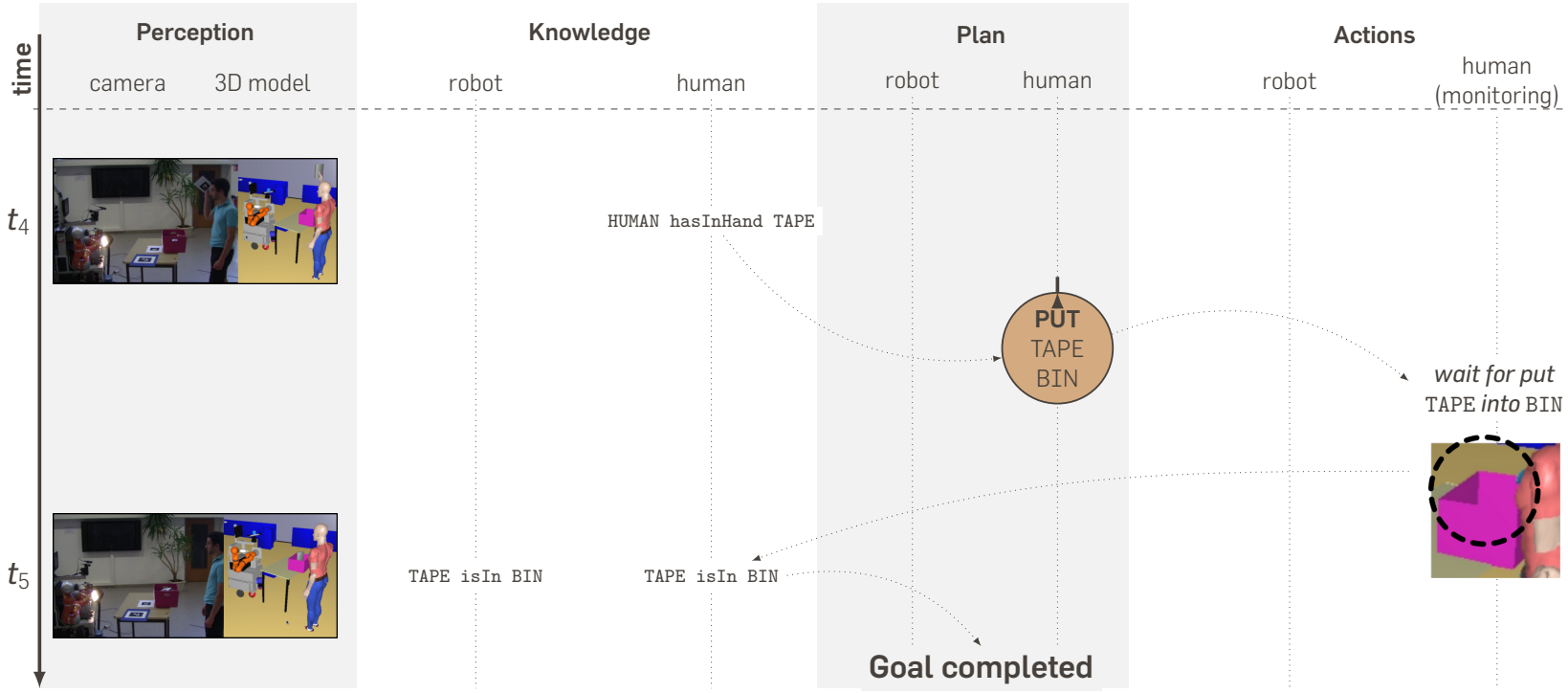



time
Perception
camera
3D model
Knowledge
robot
human
Plan
robot
human
Actions
robot
human (monitoring)
Incoming goal
Clean the table!
t1
TAPE isVisible true
TAPE isReachable true
TAPE isOn TABLE
BIN isVisible true
BIN isReachable false
TAPE isVisible true
TAPE isReachable false
TAPE isOn TABLE
BIN isVisible true
BIN isReachable true
TAKE TAPE TABLE
evaluate pre-conditions
motion planning
PICK_GOTO
TAKE_TO_FREE
motion execution
assess post-conditions
t2
ROBOT hasInHand TAPE
PUTRV TAPE TABLE
evaluate pre-conditions
motion planning
PLACE_FROM_FREE
ESCAPE
TO_FREE
motion execution
assess post-conditions
t3
TAPE isReachable true
TAPE isVisible true
TAPE isOn TABLE
TAPE isReachable true
TAPE isVisible true
TAKE TAPE TABLE
wait for pick TAPE from TABLE
t4
HUMAN hasInHand TAPE
PUT TAPE BIN
wait for put TAPE into BIN
t5
TAPE isIn BIN
TAPE isIn BIN
Goal completed


Perception
Knowledge
Plan
Actions
time
camera
3D model
robot
human
robot
human
robot
human
(monitoring)
Incoming goal
Clean the table!
TAPE isVisible true
TAPE isReachable true
TAPE isOn TABLE
BIN isVisible true
BIN isReachable false
TAPE isVisible true
TAPE isReachable false
TAPE isOn TABLE
BIN isVisible true
BIN isReachable true
TAKE
TAPE
TABLE
evaluate pre-conditions
motion planning
PICK_GOTO
TAKE_TO_FREE
motion execution
assess post-conditions
t1
t2


Perception
Knowledge
Plan
Actions
time
camera
3D model
robot
human
robot
human
robot
human
(monitoring)
t_2
t_3
ROBOT hasInHand TAPE
PUTRV
TAPE
TABLE
evaluate pre-conditions
motion planning
PLACE_FROM_FREE
ESCAPE
TO_FREE
motion execution
assess post-conditions


Perception
Knowledge
Plan
Actions
time
camera
3D model
robot
human
robot
human
robot
human
(monitoring)
t_3
t_4
TAPE isReachable true
TAPE isVisible true
TAPE isOn TABLE
TAPE isReachable true
TAPE isVisible true
TAKE
TAPE
TABLE
wait for pick
TAPE from TABLE


Perception
Knowledge
Plan
Actions
time
camera
3D model
robot
human
robot
human
robot
human
(monitoring)
t_4
t_5
HUMAN hasInHand TAPE
PUT
TAPE
BIN
wait for put
TAPE into BIN
TAPE isIn BIN
TAPE isIn BIN
Goal completed

MODEL/PLAN/ACT: STRENGTHS/WEAKNESSES

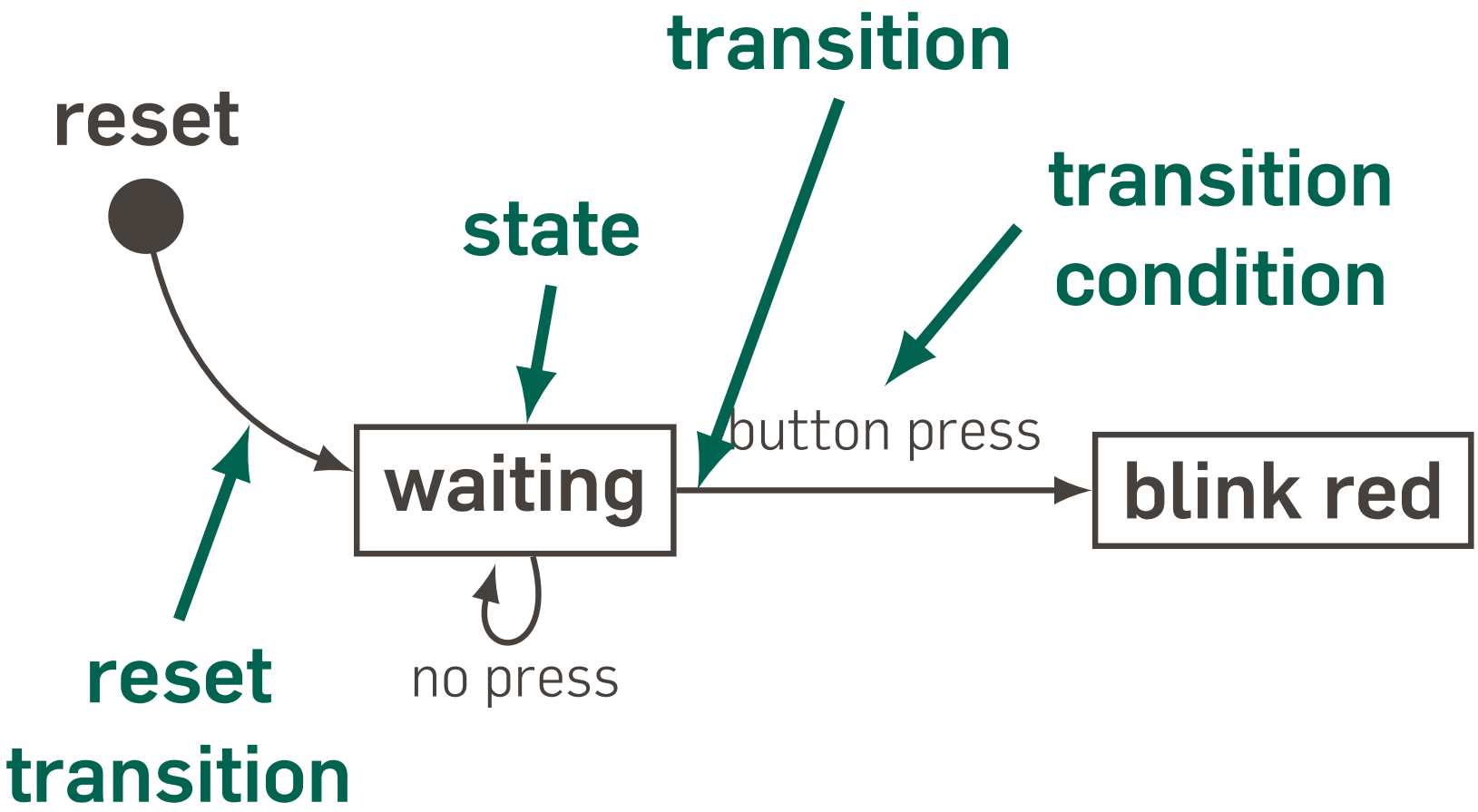
Strengths

- High-level goals are explicit
- Scale well with task complexity
- Easy to switch from one task to the other (planning domain is symbolic and explicit)

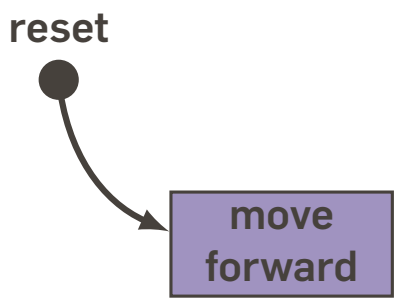
Weaknesses

- Modeling and planning might be (very) computationally expensive
- Difficult to deal with unexpected events: not well suited for dynamic environments (often requires replanning)

FINITE STATE MACHINES

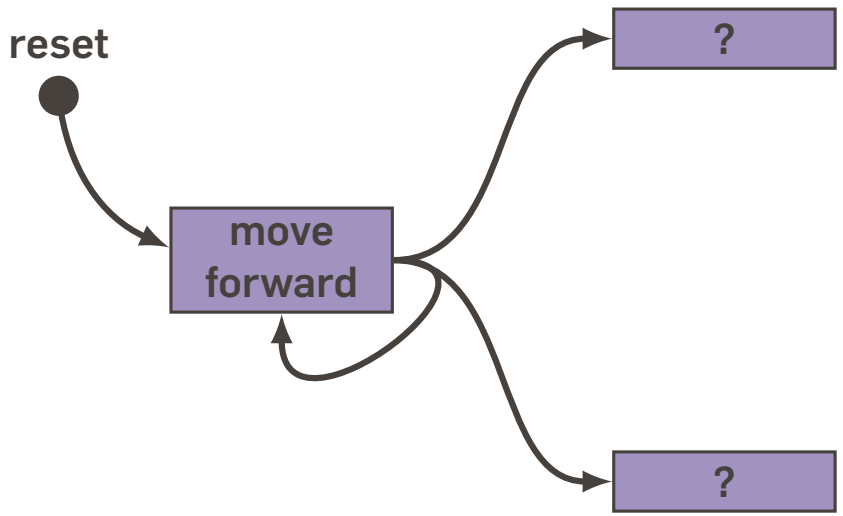


FINITE STATE MACHINE: WALL AVOIDANCE EXAMPLE



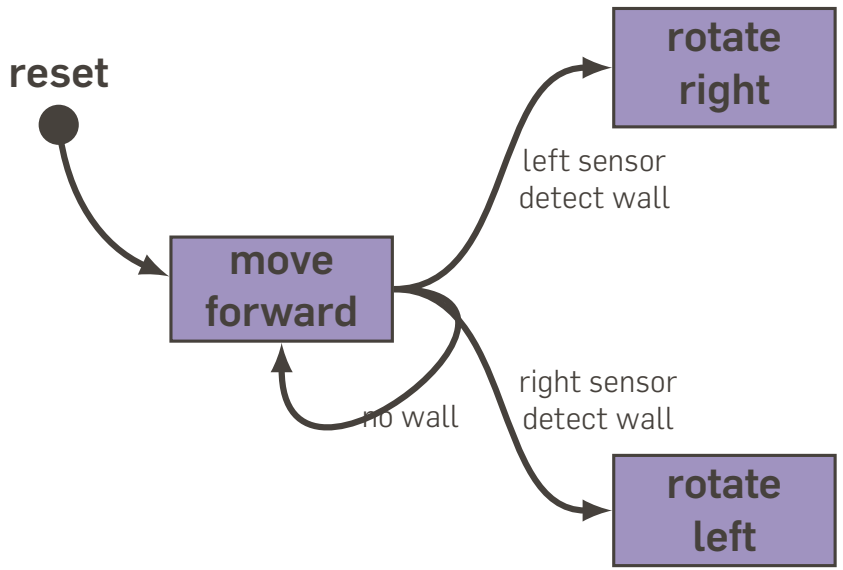
Source: example borrowed from Rice University ENGI128

FINITE STATE MACHINE: WALL AVOIDANCE EXAMPLE



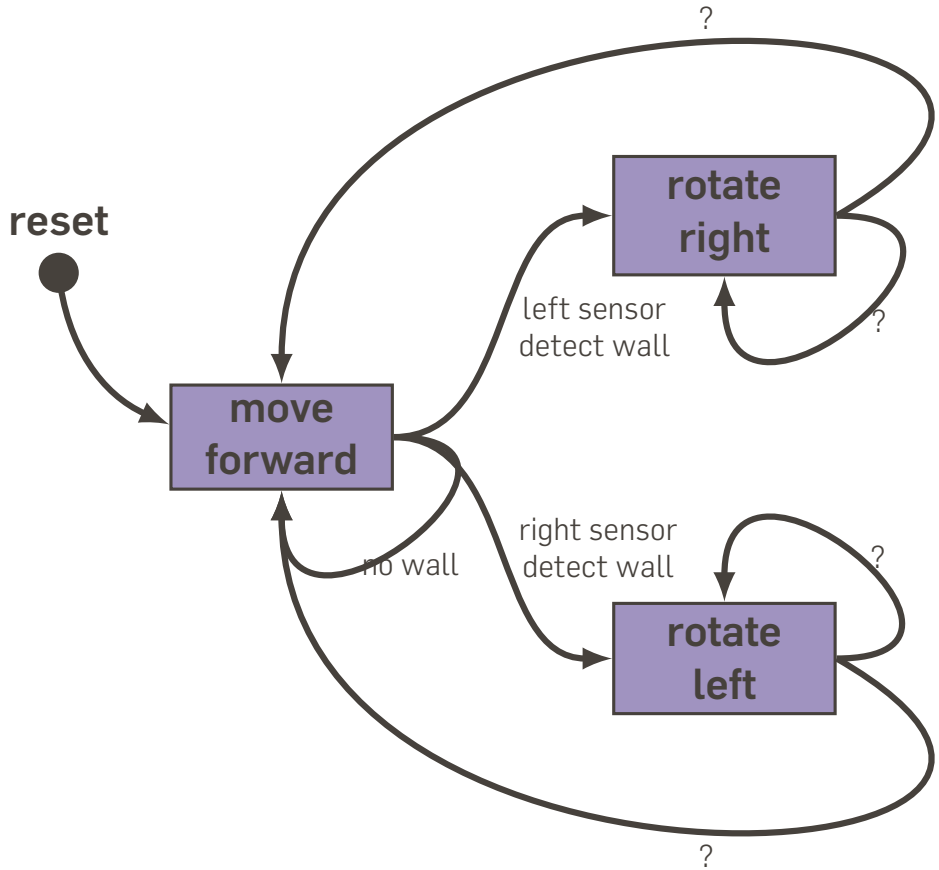
Source: example borrowed from Rice University ENGI128

FINITE STATE MACHINE: WALL AVOIDANCE EXAMPLE



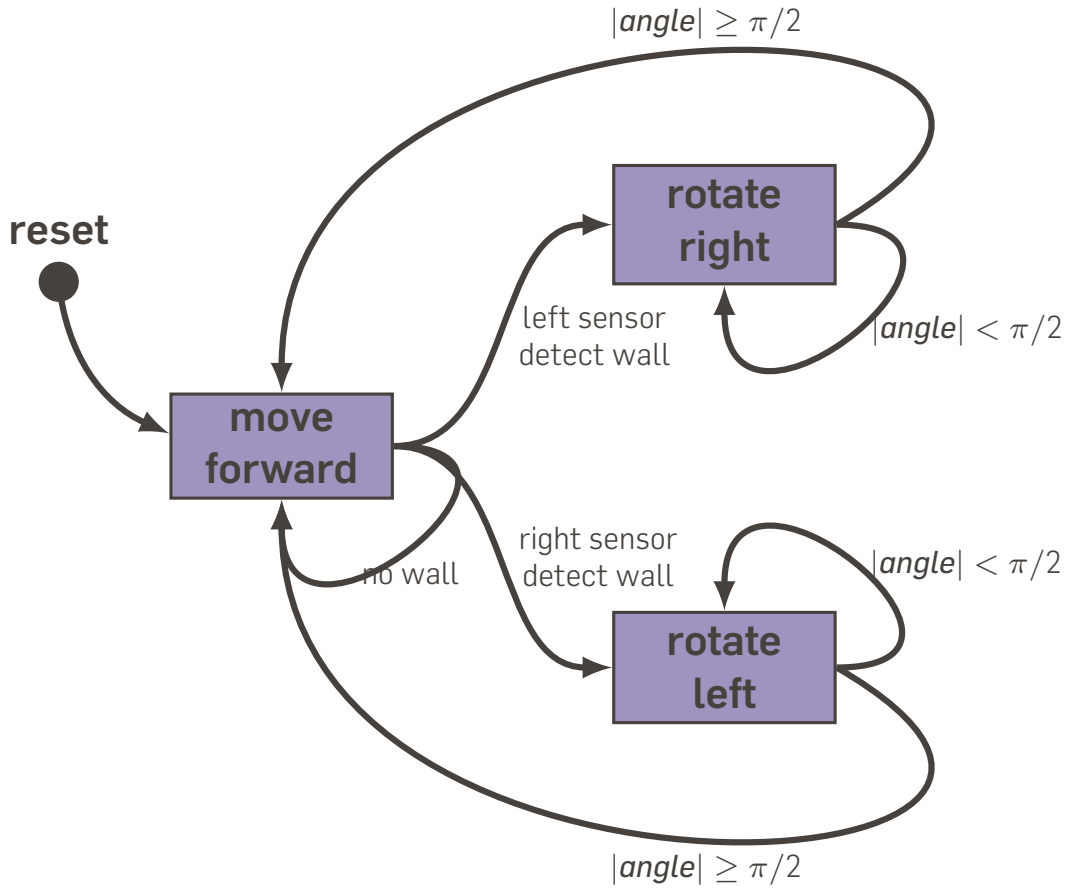
Source: example borrowed from Rice University ENGI128

FINITE STATE MACHINE: WALL AVOIDANCE EXAMPLE

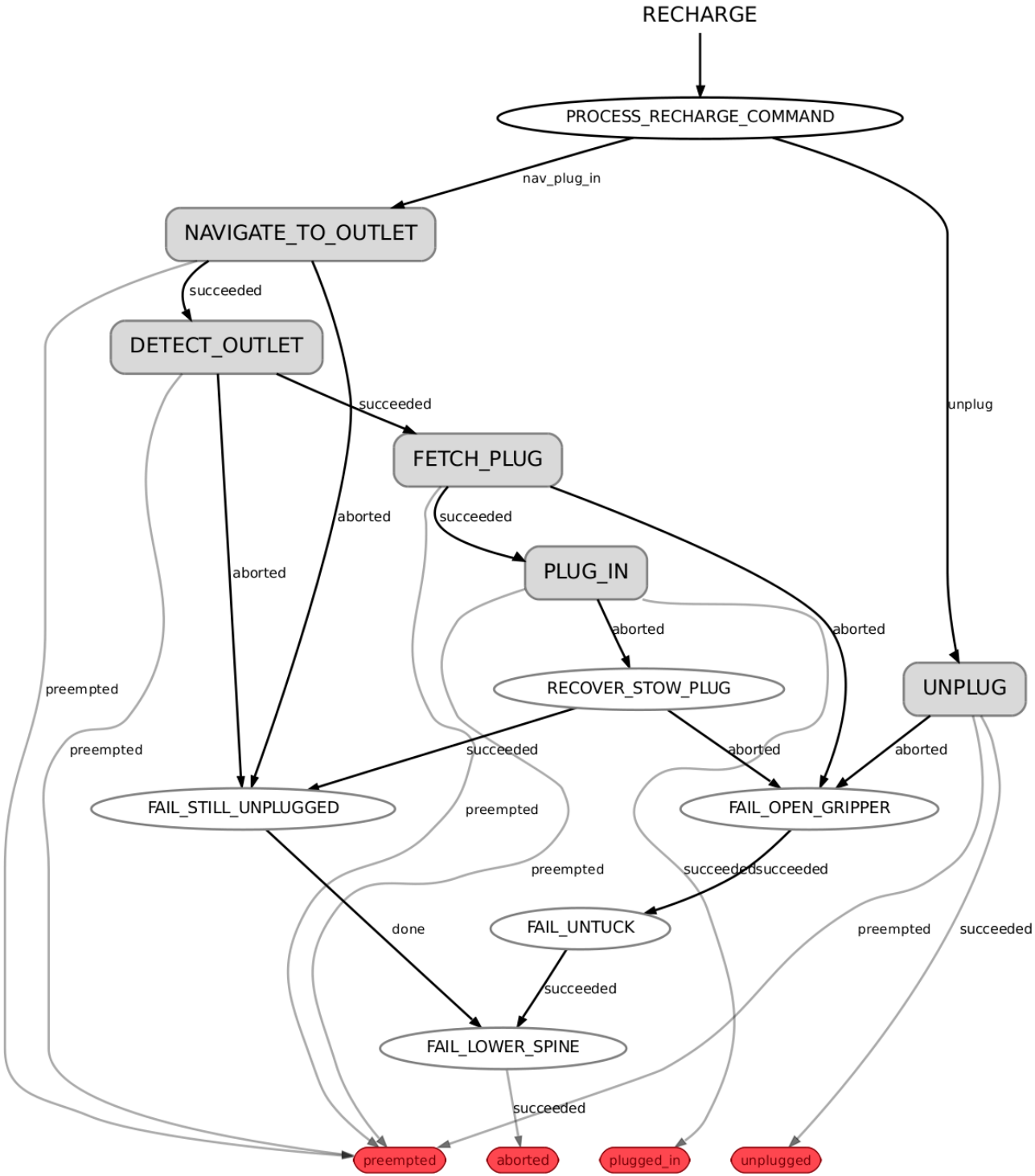


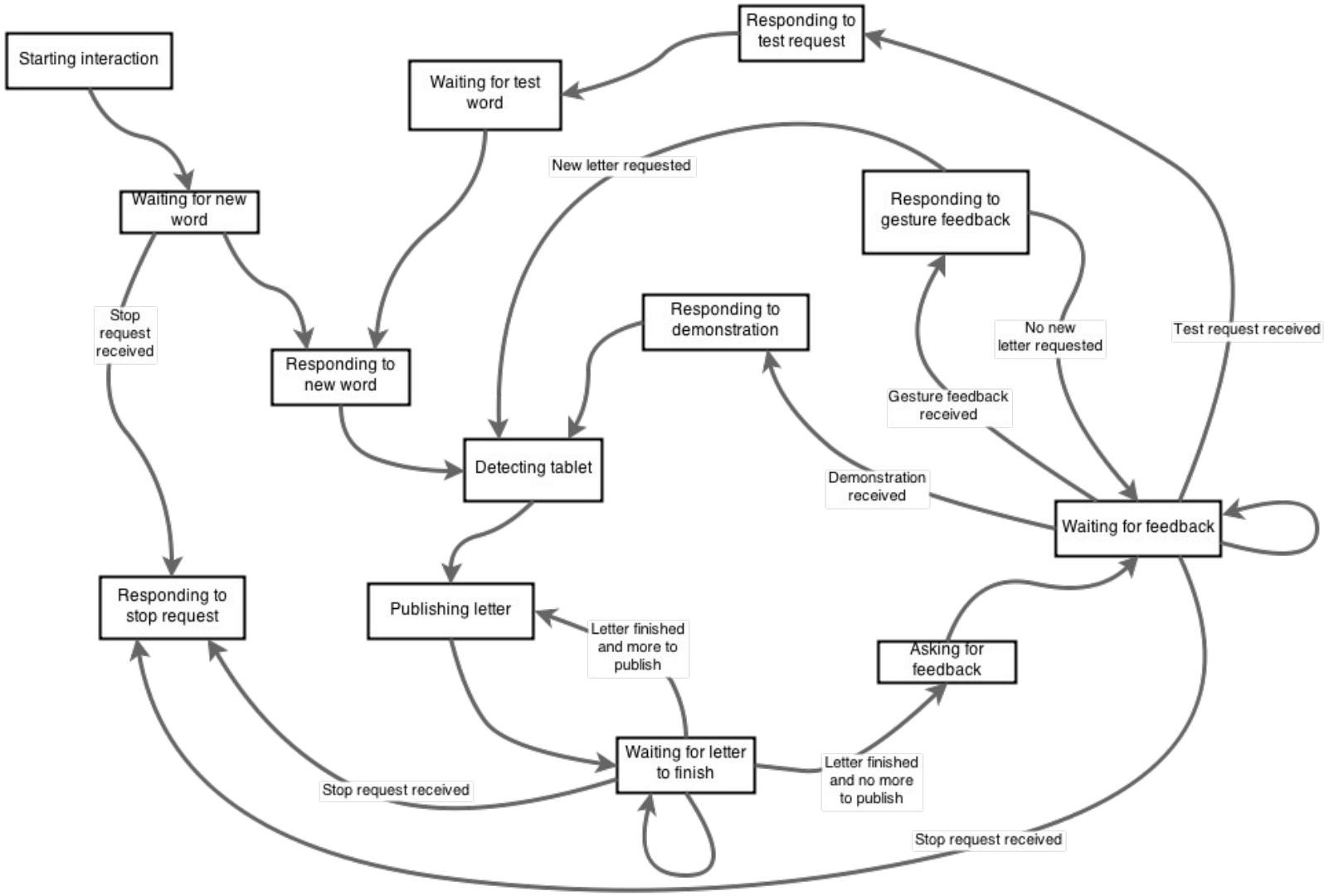
Source: example borrowed from Rice University ENGI128

FINITE STATE MACHINE: WALL AVOIDANCE EXAMPLE



Source: example borrowed from Rice University ENGI128

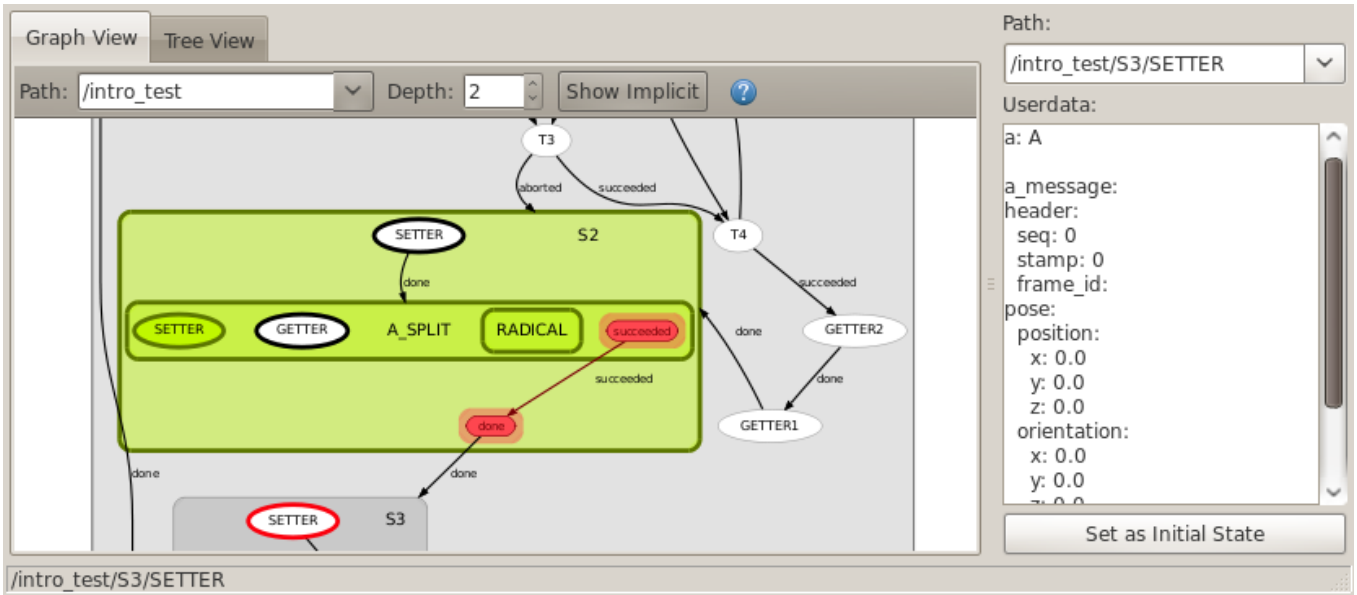

RECHARGE
PROCESS_RECHARGE_COMMAND
nav_plug_in
unplug
NAVIGATE_TO_OUTLET
succeeded
DETECT_OUTLET
succeeded
FETCH_PLUG
aborted
succeeded
aborted
PLUG_IN
aborted
aborted
preempted
RECOVER_STOW_PLUG
UNPLUG
preempted
succeeded
aborted
aborted
FAIL_STILL_UNPLUGGED
preempted
FAIL_OPEN_GRIPPER
preempted
succeeded
succeeded
done
FAIL_UNTUCK
preempted
succeeded
succeeded
FAIL_LOWER_SPINE
succeeded
preempted
aborted
plugged_in
unplugged


Starting interaction
Waiting for new word
Stop request received
Responding to stop request
Responding to new word
Waiting for test word
Responding to test request
New letter requested
Detecting tablet
Responding to demonstration
Publishing letter
Letter finished and more to publish
Waiting for letter to finish
Stop request received
Letter finished and no more to publish
Asking for feedback
Responding to gesture feedback
Gesture feedback received
No new letter requested
Test request received
Demonstration received
Waiting for feedback
Stop request received

FSM WITH ROS: THE SMACH PACKAGE

SMACH is a ROS-independent (but very weill integrated with ROS!) Python library to build state machines

- support for hierarchical (i.e. nested) state machines
- provides support for concurrency as well



RELATION TO BEHAVIOURAL/DELIBERATIVE PARADIGMS

Finite State Machines are **not a control paradigm** per-se.

Instead, they are a mathematical formalism to describe states and transitions.

States themselves might be behavioural components, deliberative components, or **nested state machines**.

FINITE STATE MACHINE: STRENGTHS/WEAKNESSES

Strengths

- Behaviour formally provable
- Easier to debug: control flow is explicit

Weaknesses

- Less modular than behavioural approaches
- Difficult to deal with unexpected events: not well suited for dynamic environments

CONTROL ARCHITECTURES

CONTROL STRATEGIES SO FAR

Behavioural or reactive

- *bottom-up* approach
- lots of independent modules executing concurrently, monitoring sensor values, triggering actions and possibly inhibiting each-others
- hard to organize into complex behaviours; gets messy quickly

CONTROL STRATEGIES SO FAR

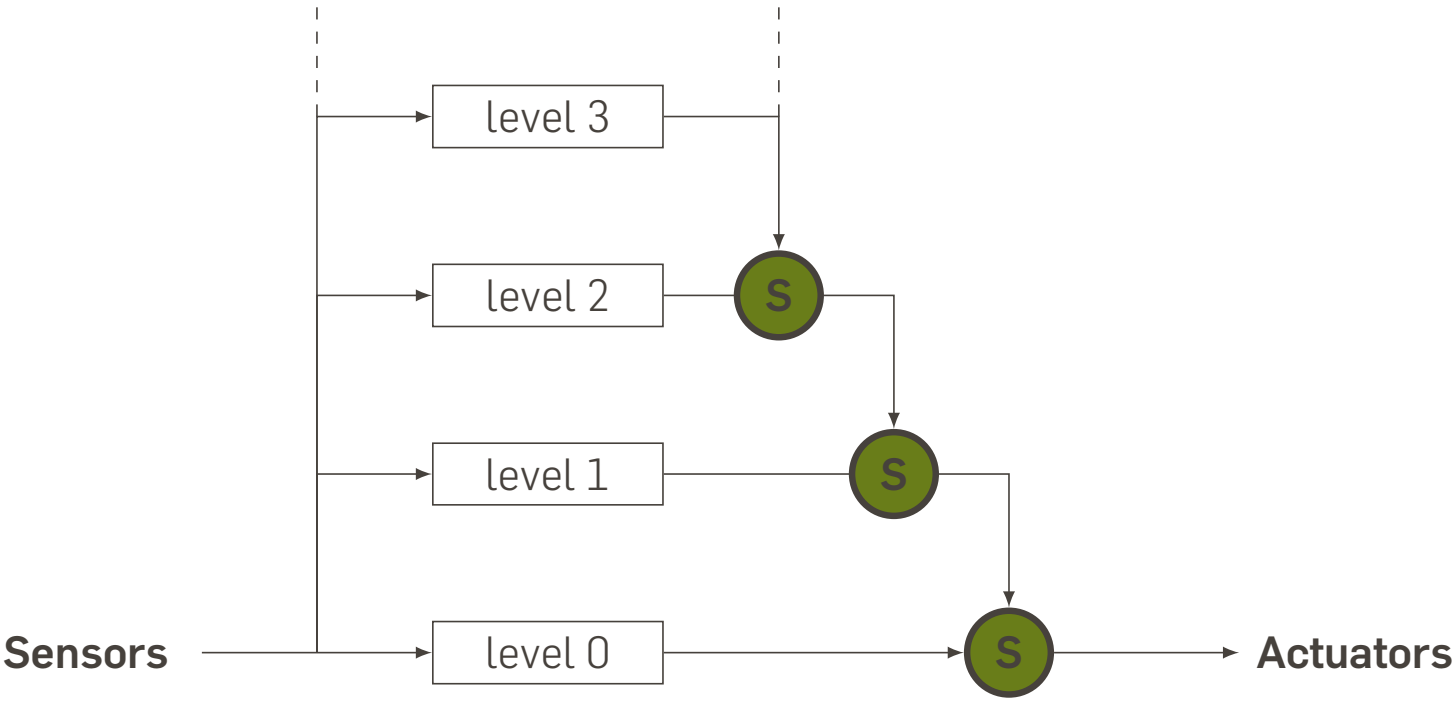
Behavioural or reactive

- *bottom-up* approach
- lots of independent modules executing concurrently, monitoring sensor values, triggering actions and possibly inhibiting each-others
- hard to organize into complex behaviours; gets messy quickly

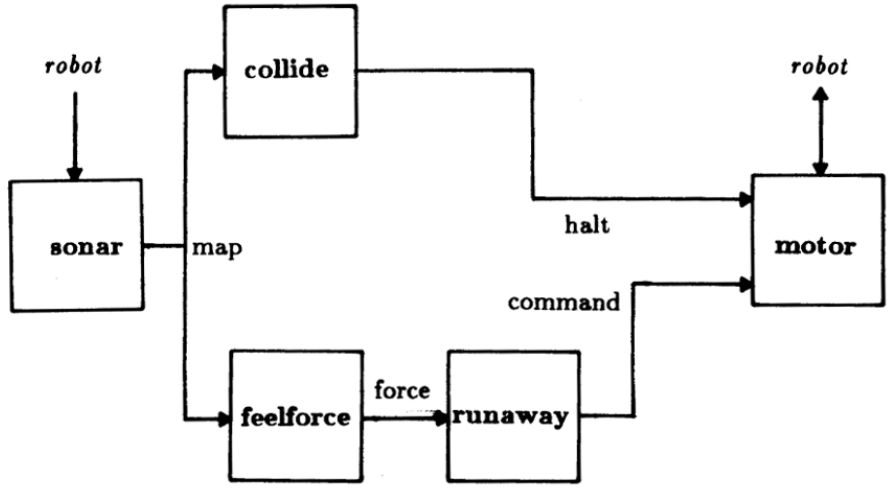
Hierarchical: classic model/plan/act

- *top-down* approach
- starts with high-level goals, decompose into sub-tasks
- not very agile

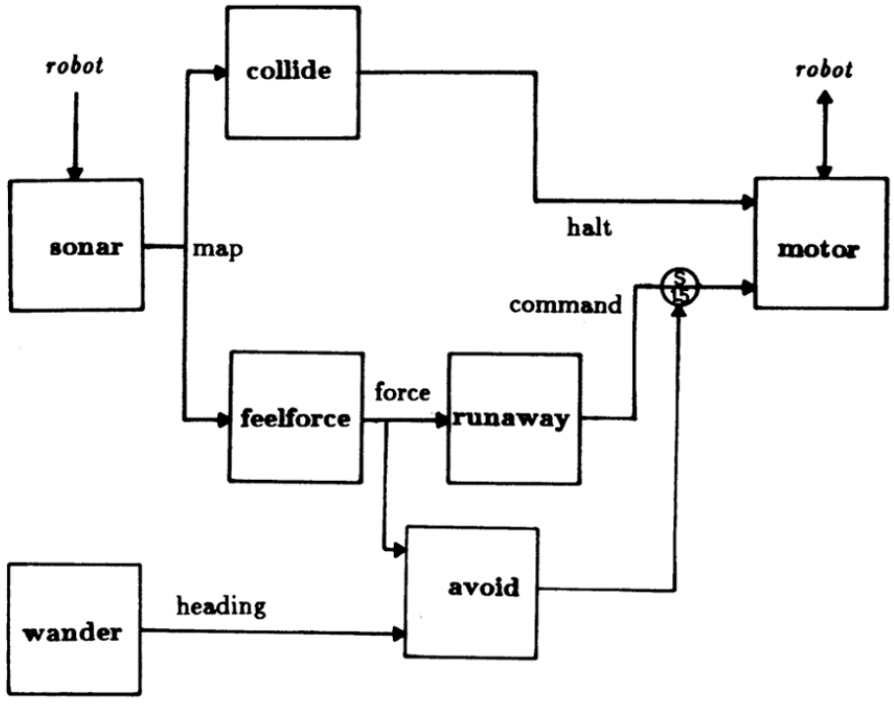
SUBSUMPTION ARCHITECTURE



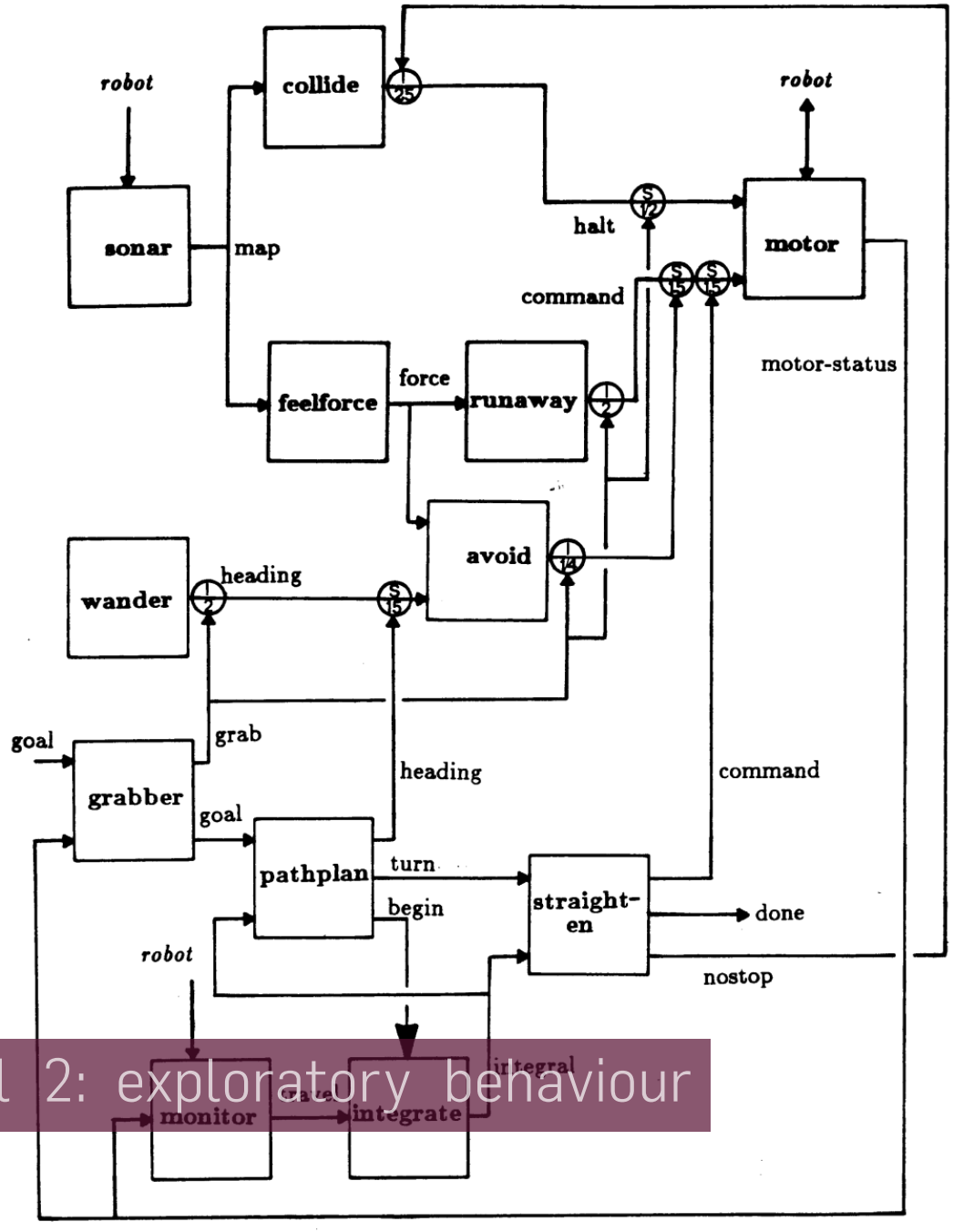
Source: Brooks, A robust layered control system for a mobile robot, 1986



Level 0: obstacle avoidance



Level 1: wander around aimlessly



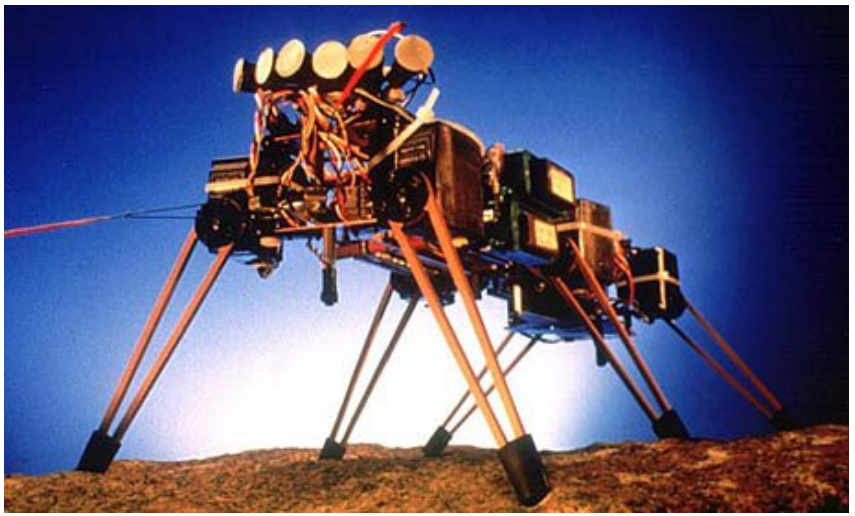
Level 2: exploratory behaviour

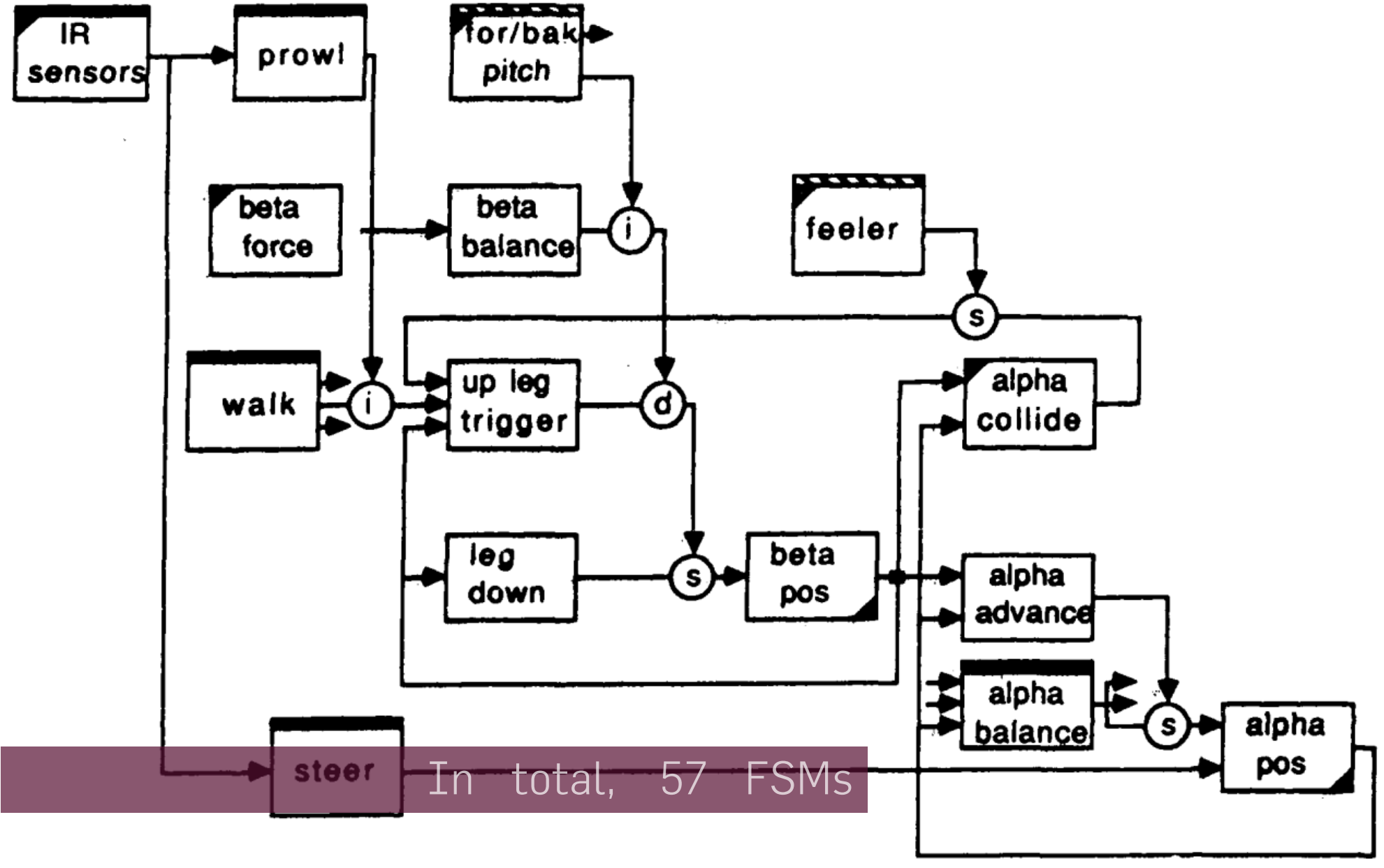
EXAMPLE: THE MIT GENGHIS ROBOT

MIT Genghis: Rodney Brooks in 1989

Really simple hardware

- 6 legs, 2 motors per leg
  - motor for forward/back,
  - motor for up/down
- 2 bump sensors (feelers)
- 2 ground detection sensors (switches)
- 6 heat sensors (but they weren't used for walking)




IR sensors
prowl
for/bak pitch
beta force
beta balance
i
feeler
s
walk
i
up leg trigger
d
alpha collide
leg down
s
beta pos
alpha advance
alpha balance
s
alpha pos
steer
In total, 57 FSMs

CONTROL STRATEGIES SO FAR

Behavioural or reactive

- *bottom-up* approach
- lots of independent modules executing concurrently, monitoring sensor values, triggering actions and possibly inhibiting each-others
- hard to organize into complex behaviours; gets messy quickly

Hierarchical: classic model/plan/act

- *top-down* approach
- starts with high-level goals, decompose into sub-tasks
- not very agile

Hybrid approaches?

- Deliberative at high level, reactive at low level

LEVELS OF CONTROL

Control problems are usually split into three levels:

- **Low-level control**

  Example: where to place a leg as robot takes its next step
  Generally, continuous-valued problems
  Short time scale (under a second); high frequency loop

Source: CMU lecture on Cognitive Robotics

LEVELS OF CONTROL

Control problems are usually split into three levels:

- **Low-level control**

  Example: where to place a leg as robot takes its next step
  Generally, continuous-valued problems
  Short time scale (under a second); high frequency loop

- **Intermediate level control**

  Navigating to a destination, or picking up an object → *capabilities*
  Continuous or discrete valued problems
  Time scale of a few seconds

Source: CMU lecture on Cognitive Robotics

LEVELS OF CONTROL

Control problems are usually split into three levels:

- **Low-level control**

  Example: where to place a leg as robot takes its next step
  Generally, continuous-valued problems
  Short time scale (under a second); high frequency loop

- **Intermediate level control**

  Navigating to a destination, or picking up an object → *capabilities*
  Continuous or discrete valued problems
  Time scale of a few seconds

- **High level control**

  What is the plan for moving these boxes out of the room?
  Discrete problems, long time scale (minutes)

Source: CMU lecture on Cognitive Robotics

LAYERED ARCHITECTURES

High-level layer
Deliberative/decisional layer

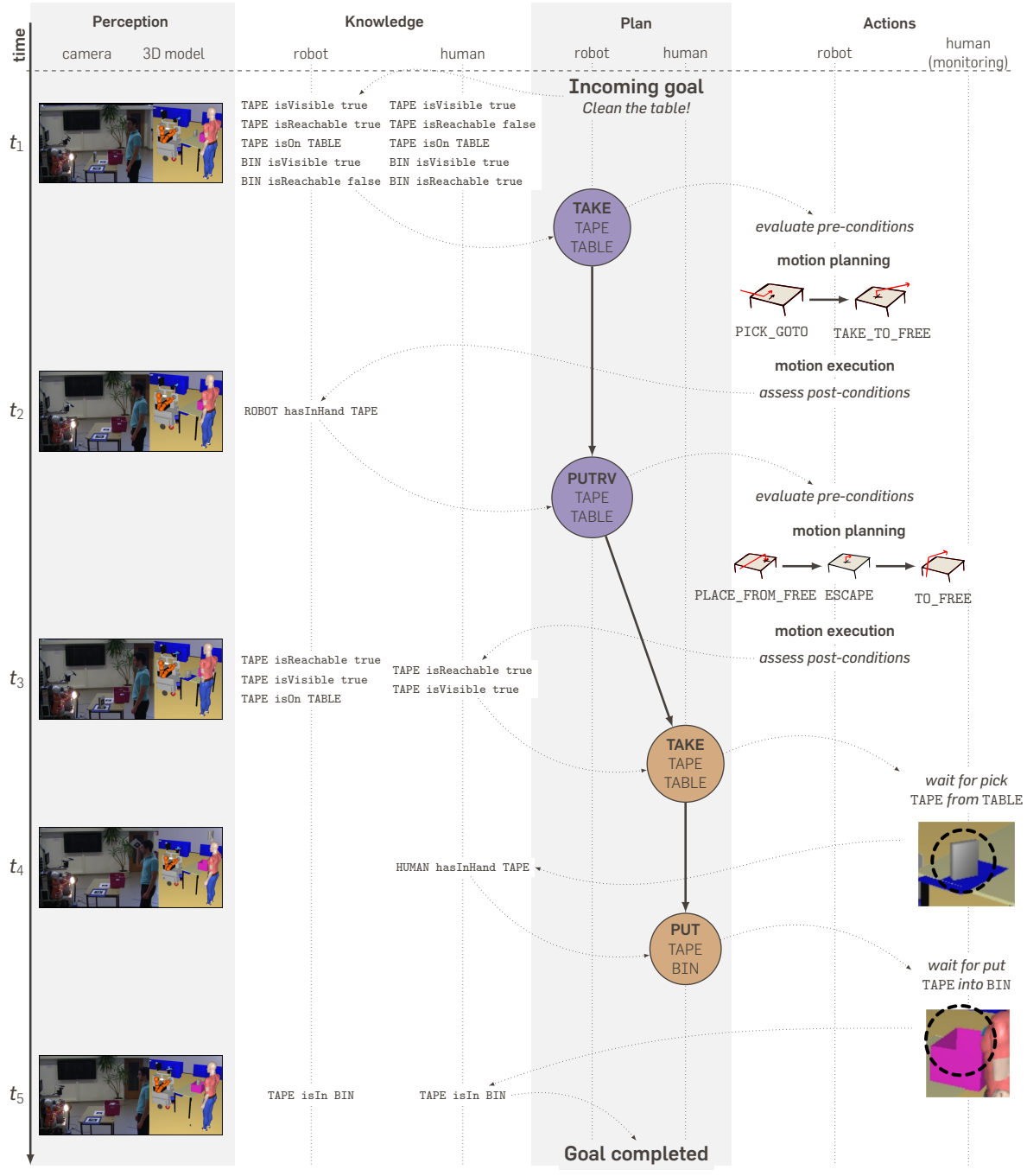
Intermediate layer
Executive layer

Low-level layer
Functional layer

Each layer has different time constraints (from real-time to possibly 'slow'); they typically use different control paradigms

DELIBERATIVE ARCHITECTURE FOR INTERACTION

One example: the LAAS deliberative architecture for interaction


time
Perception
camera
3D model
Knowledge
robot
human
Plan
robot
human
Actions
robot
human (monitoring)
Incoming goal
Clean the table!
t1
TAPE isVisible true
TAPE isReachable true
TAPE isOn TABLE
BIN isVisible true
BIN isReachable false
TAPE isVisible true
TAPE isReachable false
TAPE isOn TABLE
BIN isVisible true
BIN isReachable true
TAKE TAPE TABLE
evaluate pre-conditions
motion planning
PICK_GOTO
TAKE_TO_FREE
motion execution
assess post-conditions
t2
ROBOT hasInHand TAPE
PUTRV TAPE TABLE
evaluate pre-conditions
motion planning
PLACE_FROM_FREE
ESCAPE
TO_FREE
motion execution
assess post-conditions
t3
TAPE isReachable true
TAPE isVisible true
TAPE isOn TABLE
TAPE isReachable true
TAPE isVisible true
TAKE TAPE TABLE
wait for pick TAPE from TABLE
t4
HUMAN hasInHand TAPE
PUT TAPE BIN
wait for put TAPE into BIN
t5
TAPE isIn BIN
TAPE isIn BIN
Goal completed

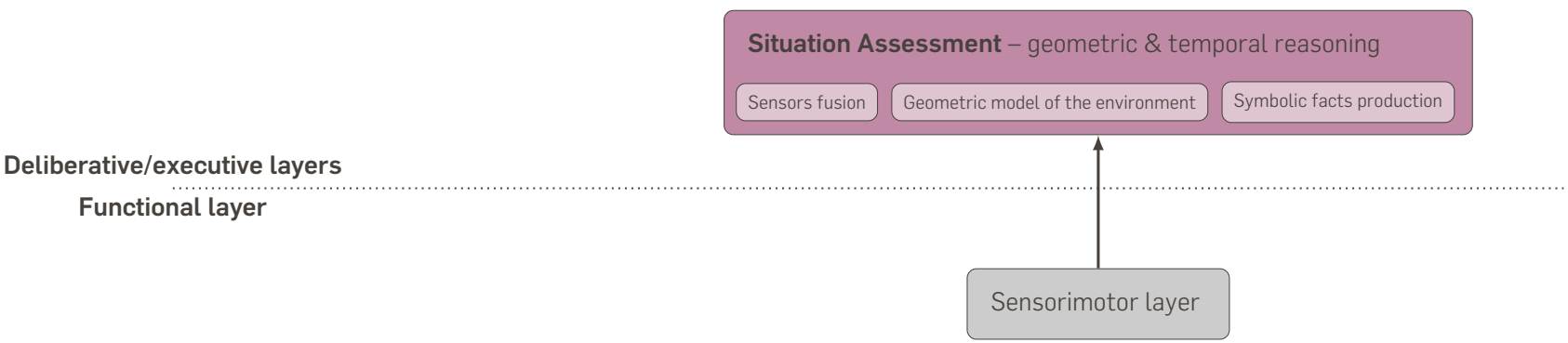
DELIBERATIVE ARCHITECTURE FOR INTERACTION

Deliberative/executive layers

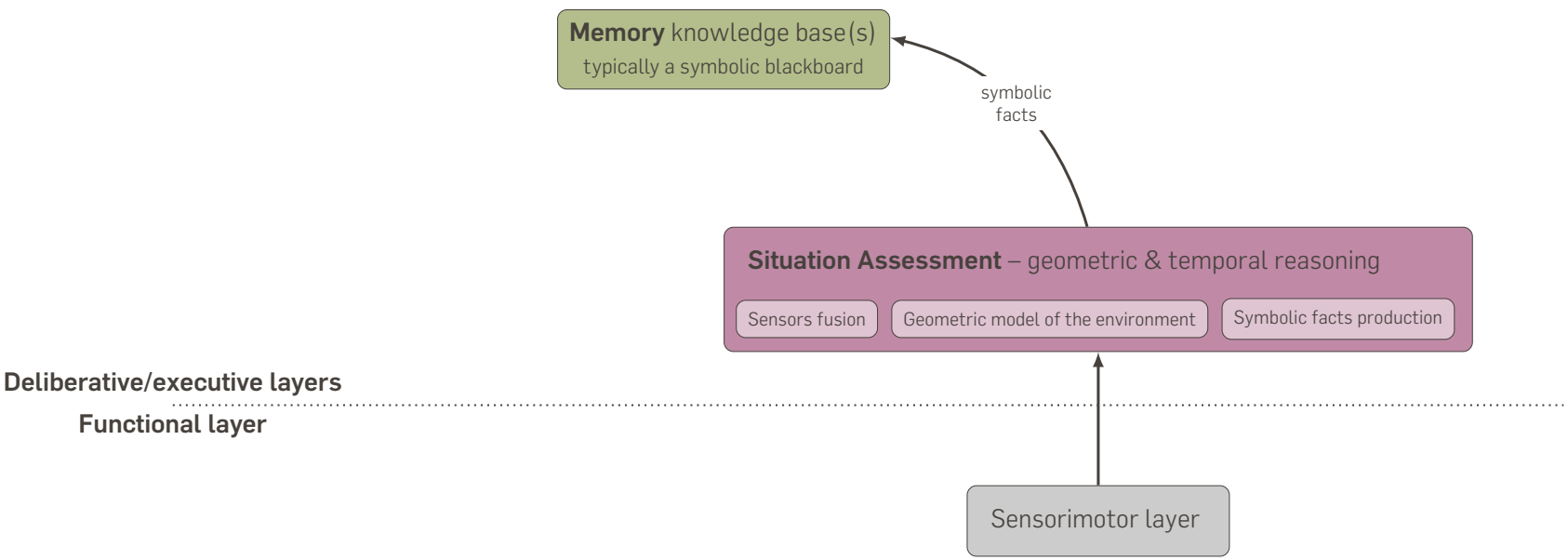
Functional layer

Sensorimotor layer

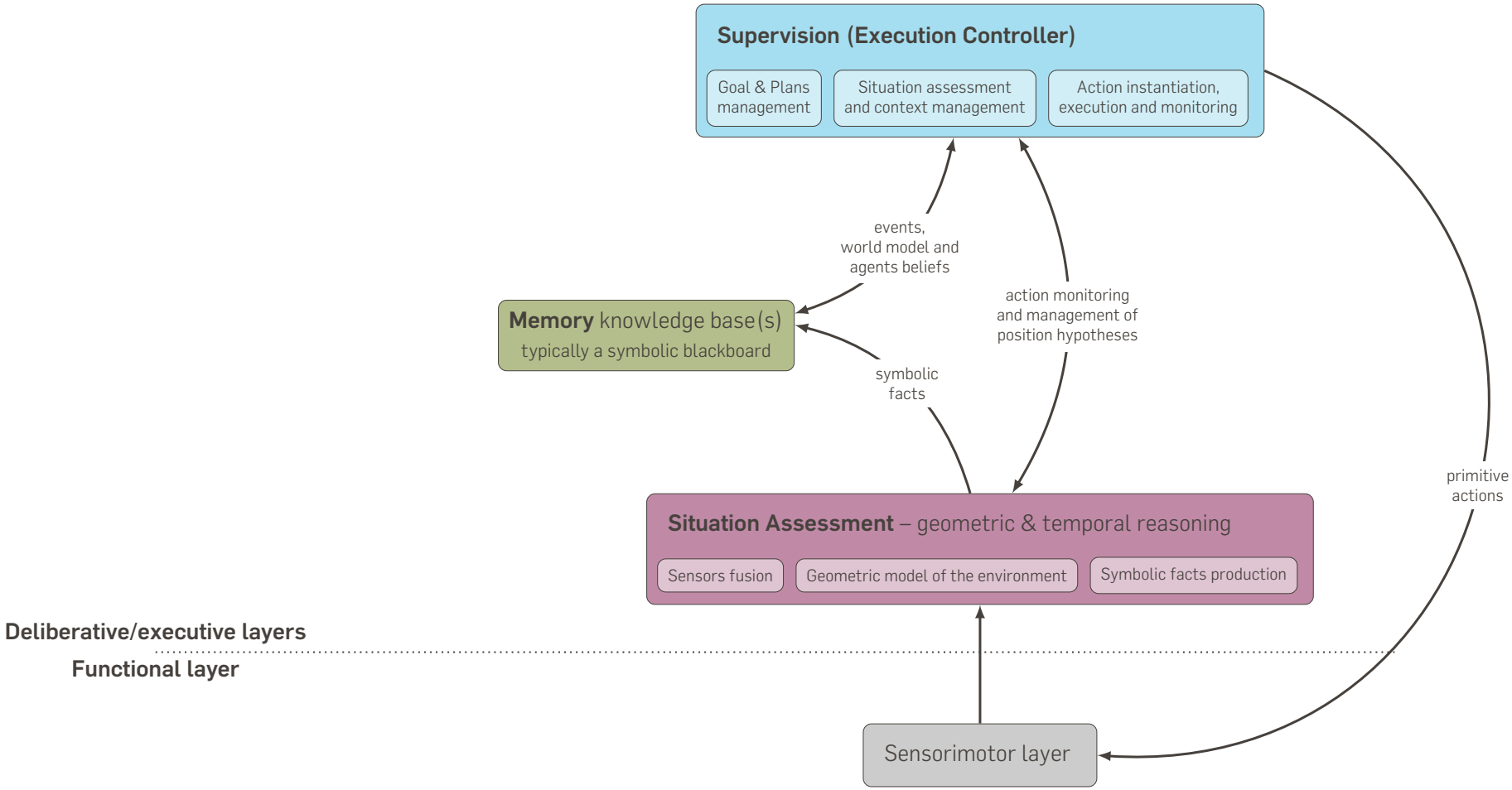
DELIBERATIVE ARCHITECTURE FOR INTERACTION



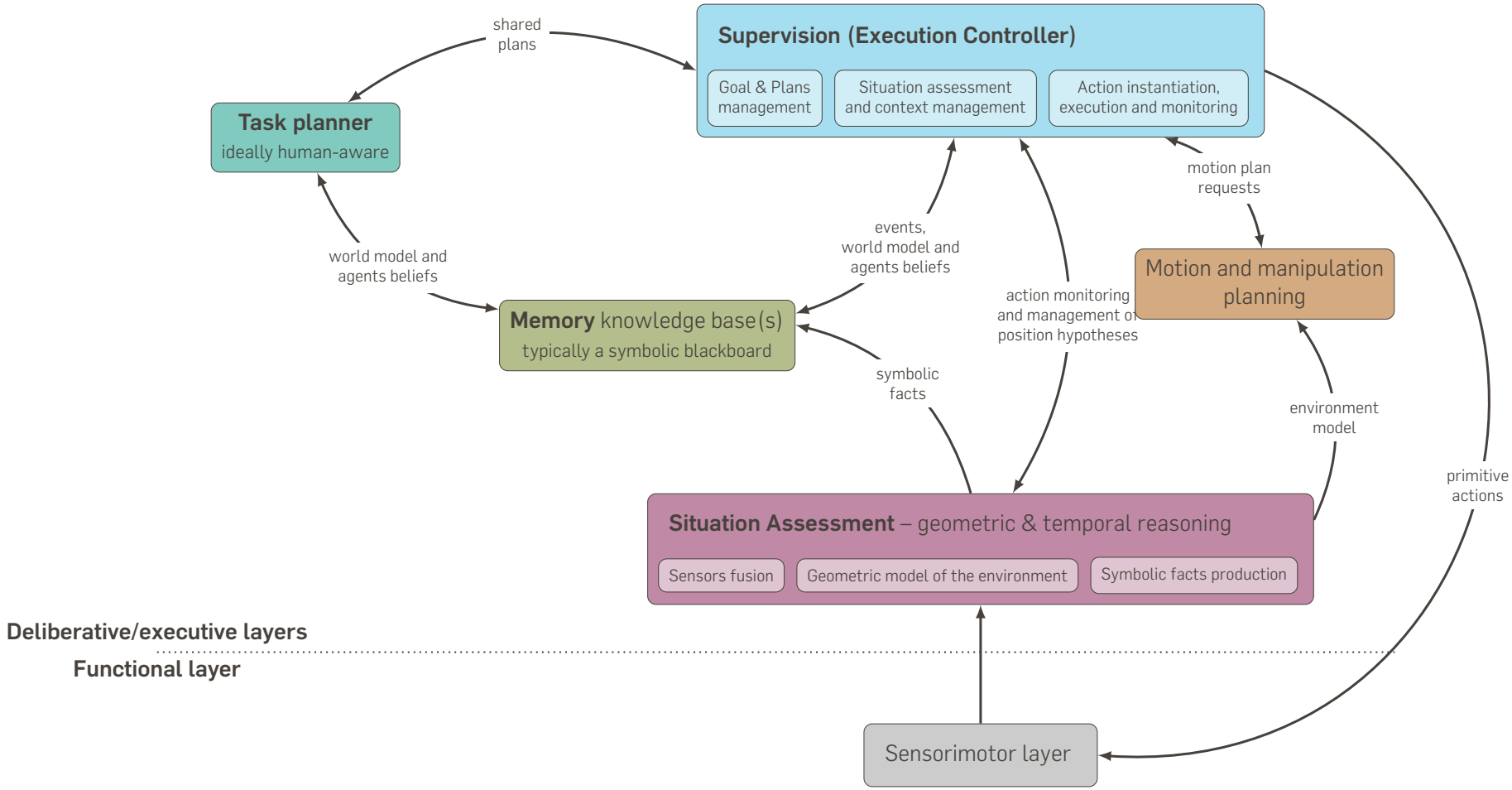
DELIBERATIVE ARCHITECTURE FOR INTERACTION



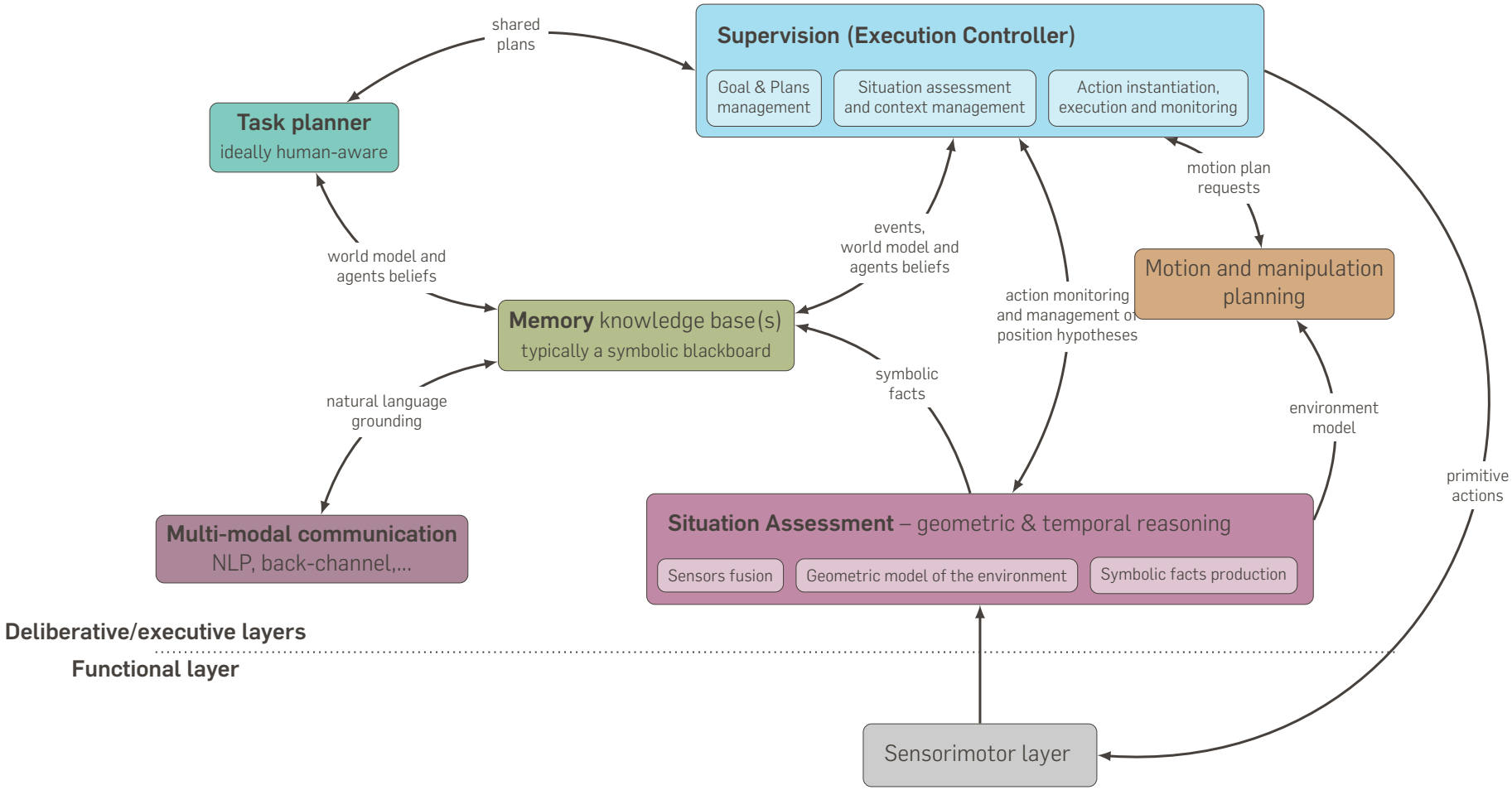
DELIBERATIVE ARCHITECTURE FOR INTERACTION



DELIBERATIVE ARCHITECTURE FOR INTERACTION

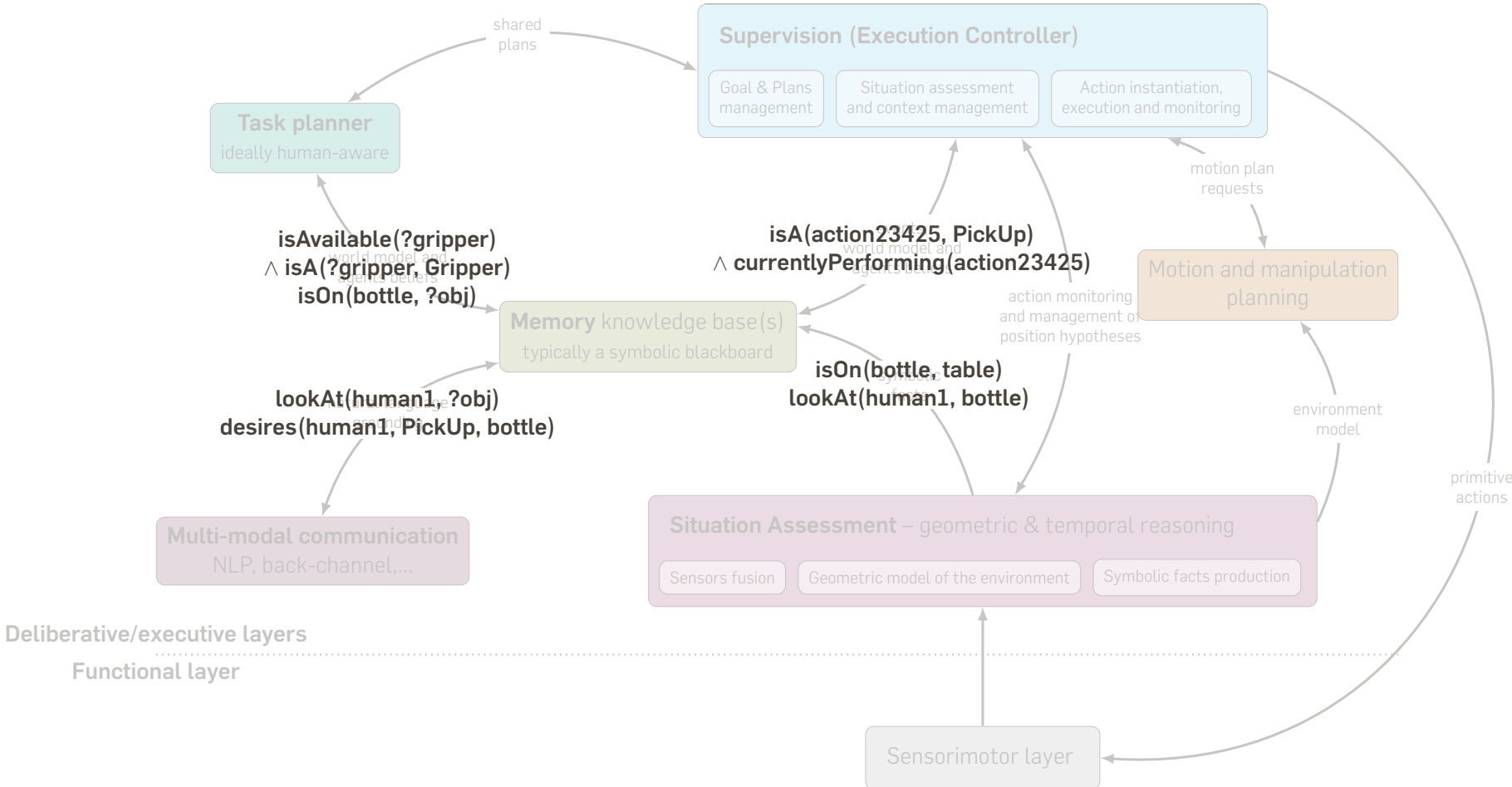


DELIBERATIVE ARCHITECTURE FOR INTERACTION



How do modules actually communicate with each other?

DELIBERATIVE ARCHITECTURE FOR INTERACTION



MIDDLEWARES

ROBOTIC MIDDLEWARES

The core role of a **middleware** is to **provide a set of abstractions to ease the development of robotic software components**.

- It **abstracts away the hardware platform** (uniform driver interfaces)

ROBOTIC MIDDLEWARES

The core role of a **middleware** is to **provide a set of abstractions to ease the development of robotic software components**.

- It **abstracts away the hardware platform** (uniform driver interfaces)
- It abstracts away where computations are physically performed (by providing **distributed computation/network transparency**)

ROBOTIC MIDDLEWARES

The core role of a **middleware** is to **provide a set of abstractions to ease the development of robotic software components**.

- It **abstracts away the hardware platform** (uniform driver interfaces)
- It abstracts away where computations are physically performed (by providing **distributed computation/network transparency**)
- It supports **modularity/reusability** (**standard interfaces**: modules which implement these interfaces can be easily swapped)

ROBOTIC MIDDLEWARES

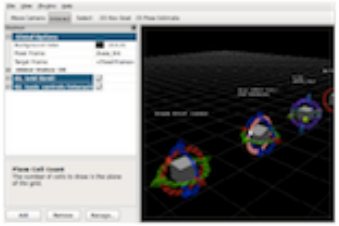
The core role of a **middleware** is to **provide a set of abstractions to ease the development of robotic software components**.

- It **abstracts away the hardware platform** (uniform driver interfaces)
- It abstracts away where computations are physically performed (by providing **distributed computation/network transparency**)
- It supports **modularity/reusability** (**standard interfaces**: modules which implement these interfaces can be easily swapped)
- It help with debugging (by providing **introspection** mechanisms)

MIDDLEWARES IN A BROADER SENSE

Plumbing

+



Tools

+

Capabilities

+

Ecosystem

MANY EXISTING MIDDLEWARES

Major ones:

- OROCOS (Europe)
- YARP (Europe)
- OpenRTM (Korea)
- ROS – Robot Operating System (originally US, now very much international)
- OpenRAVE (US)

MANY EXISTING MIDDLEWARES

Major ones:

- OROCOS (Europe)
- YARP (Europe)
- OpenRTM (Korea)
- **ROS** – Robot Operating System (originally US, now very much international)
- OpenRAVE (US)

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules
- $\rightarrow$ **a middleware**

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

- A set of conventions to write and package robotic softwares

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)

ROS

- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

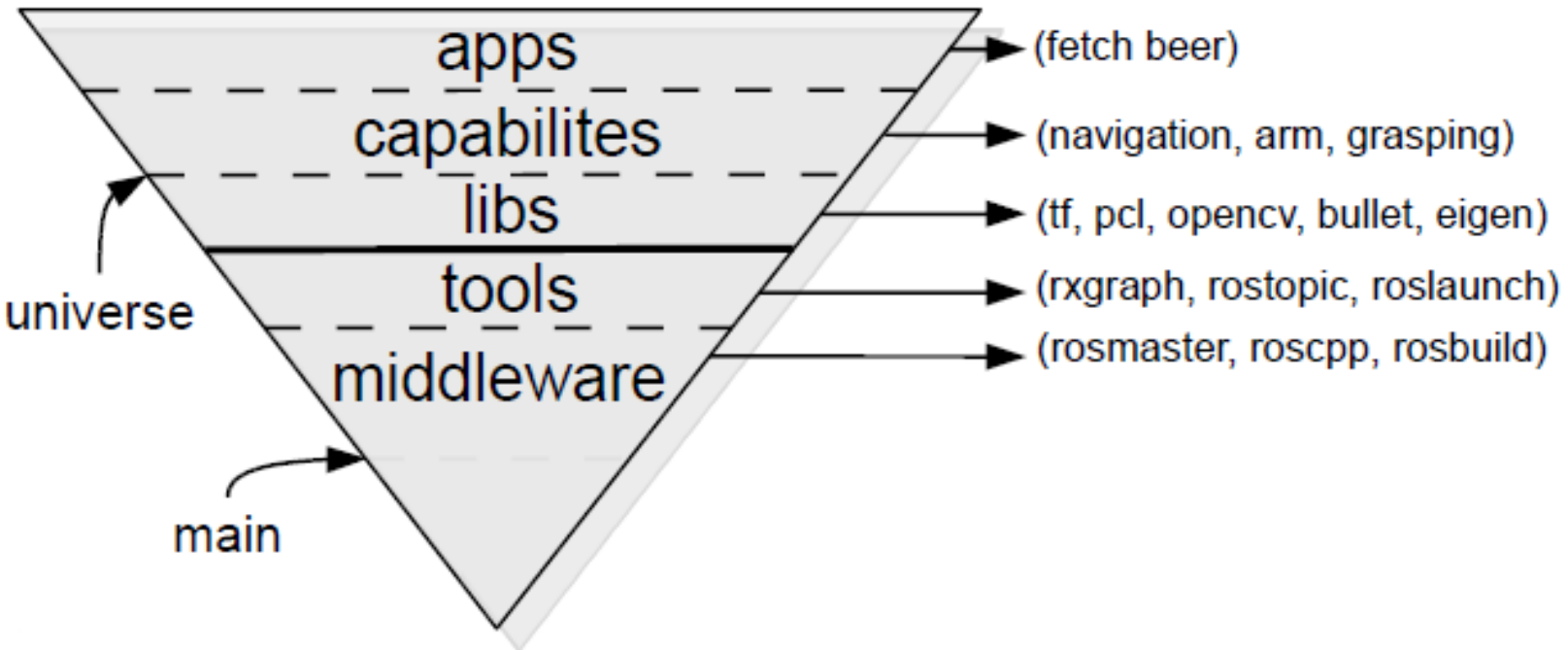
- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)
- A set of tools to run and monitor the nodes

ROS

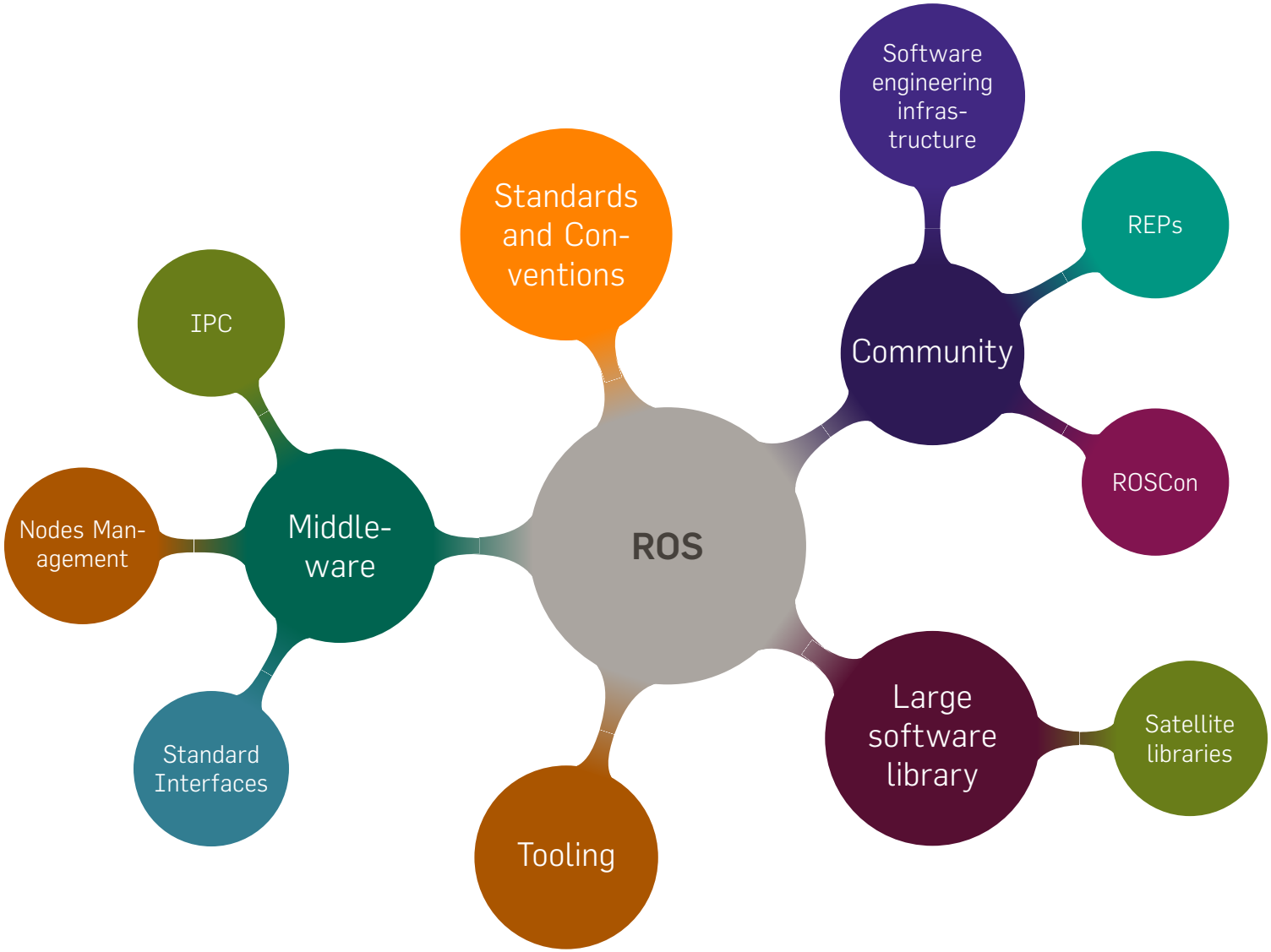
- A fairly simple peer-to-peer message passing system designed with robotics in mind
- An API to this system (in several languages – C++ and Python are 1st tier)
- A set of standard message types that facilitate interoperability between modules

- A set of conventions to write and package robotic softwares
- Deep integration of a few key open-source libraries (OpenCV, PCL, tf)
- A set of tools to run and monitor the nodes
- Engagement of a large academic community, leading to a library of thousands of nodes

ROS STRUCTURE

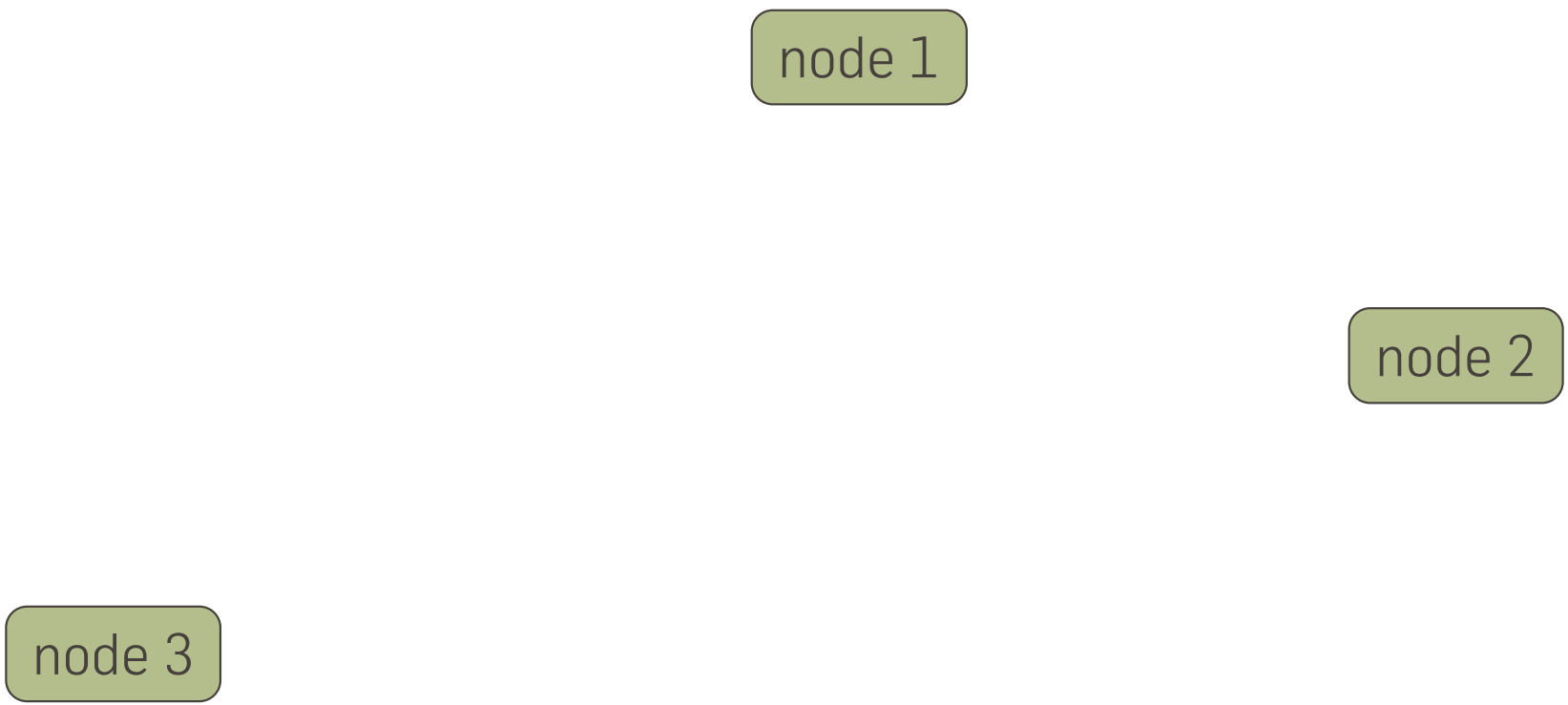


ROS ECOSYSTEM

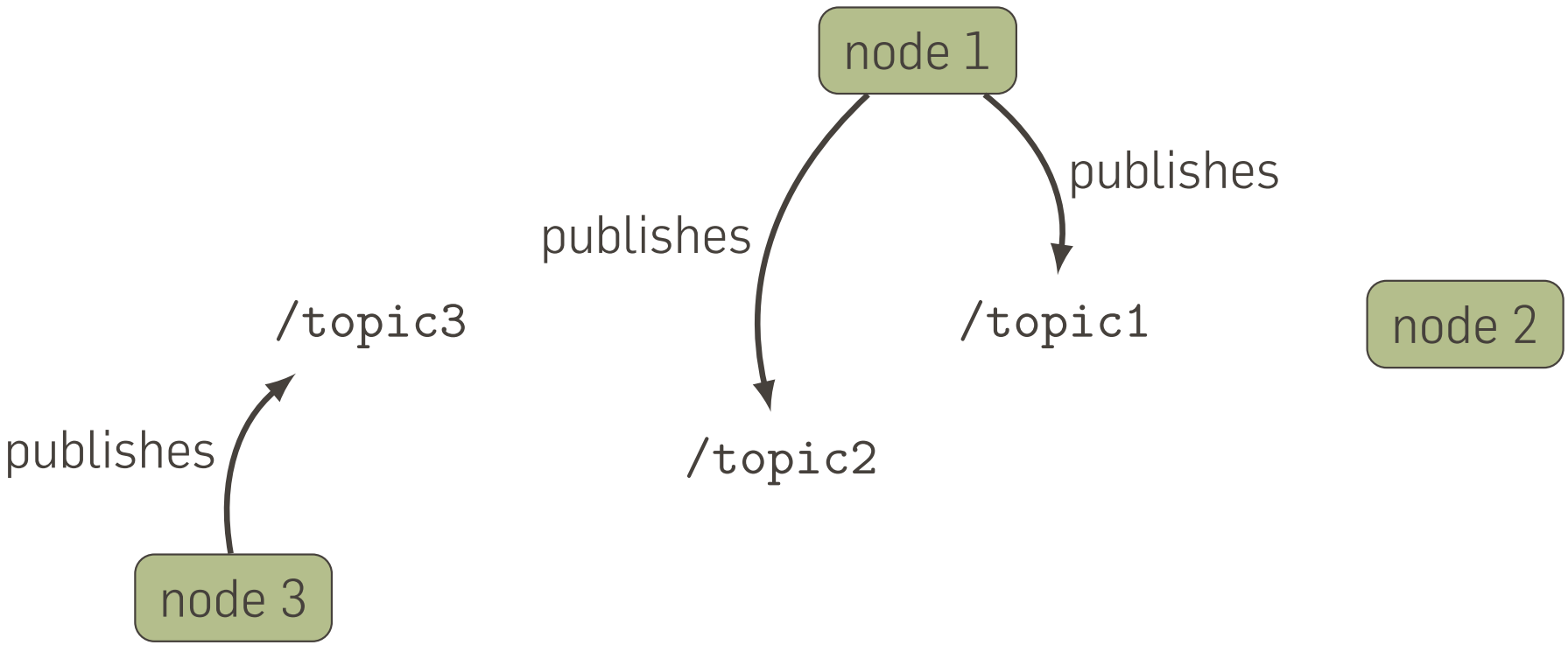


MIDDLEWARES' CORE PRINCIPLE: TALKING NODES

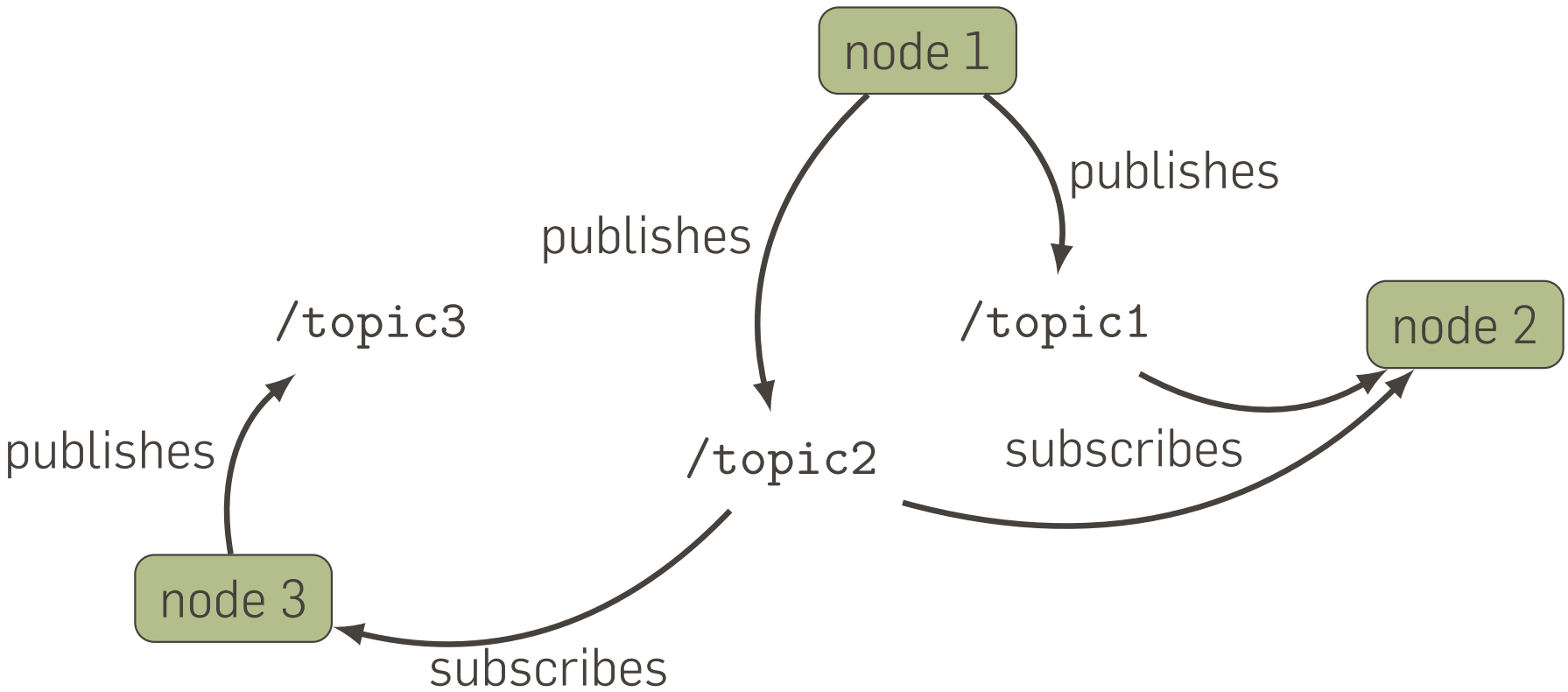
Nodes are software components which realise a function (can be a simple behaviour, a data filter, a complex FSM, or anything else really)



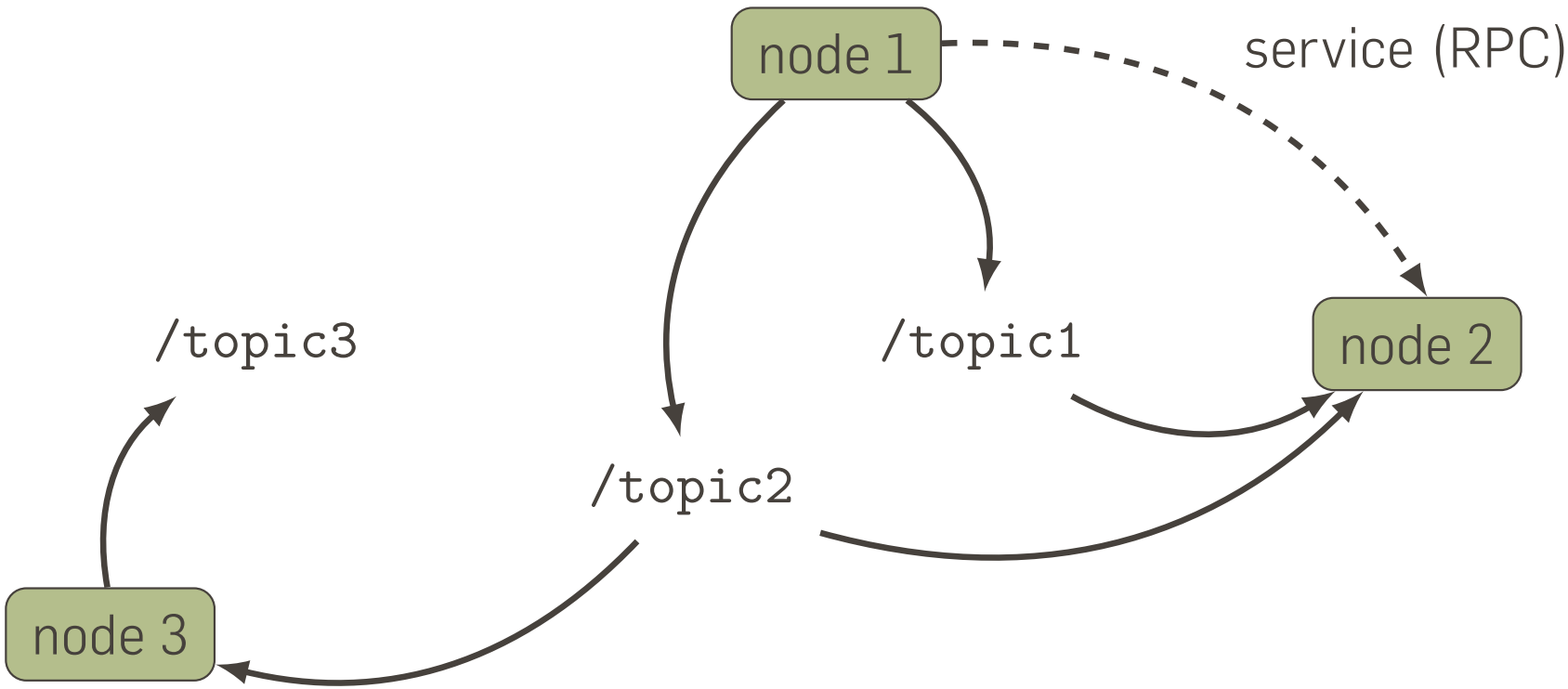
MIDDLEWARES' CORE PRINCIPLE: TALKING NODES



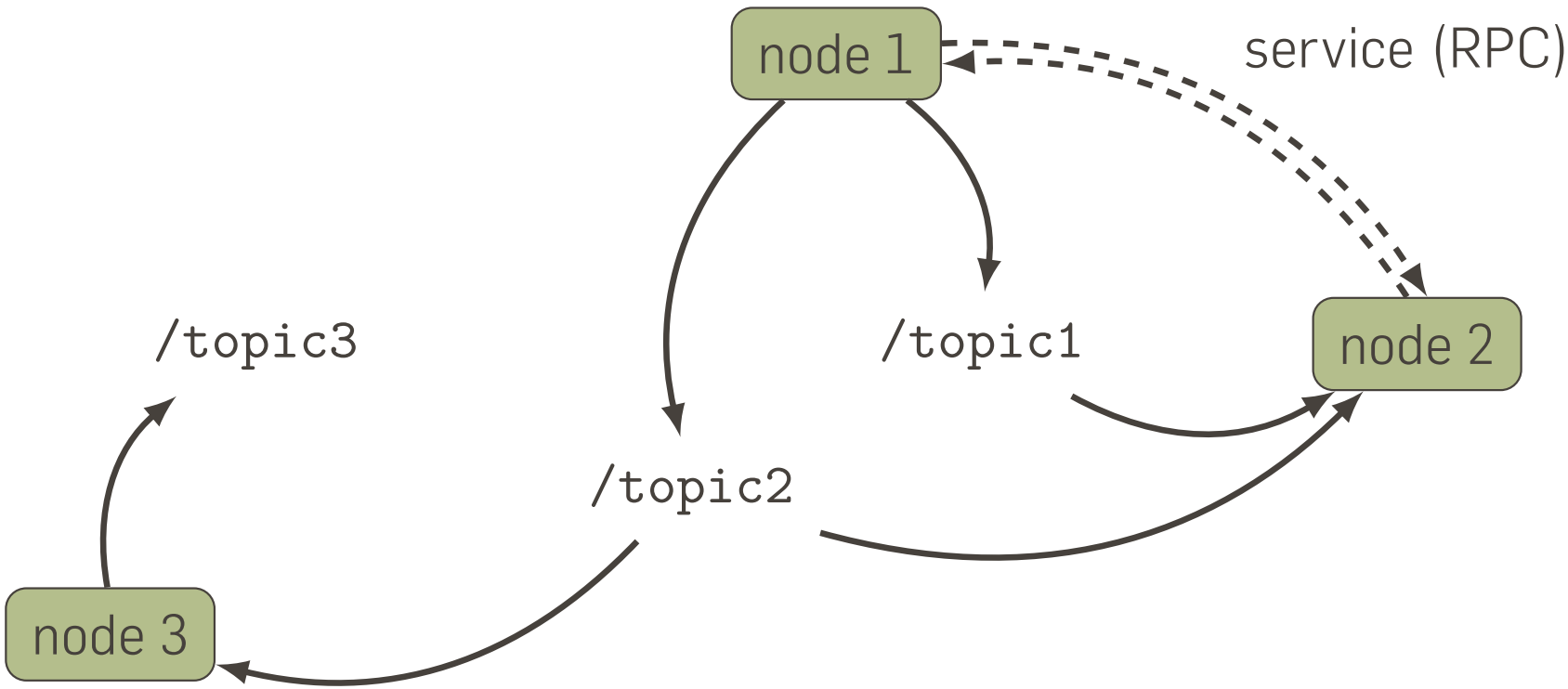
MIDDLEWARES' CORE PRINCIPLE: TALKING NODES



MIDDLEWARES' CORE PRINCIPLE: TALKING NODES

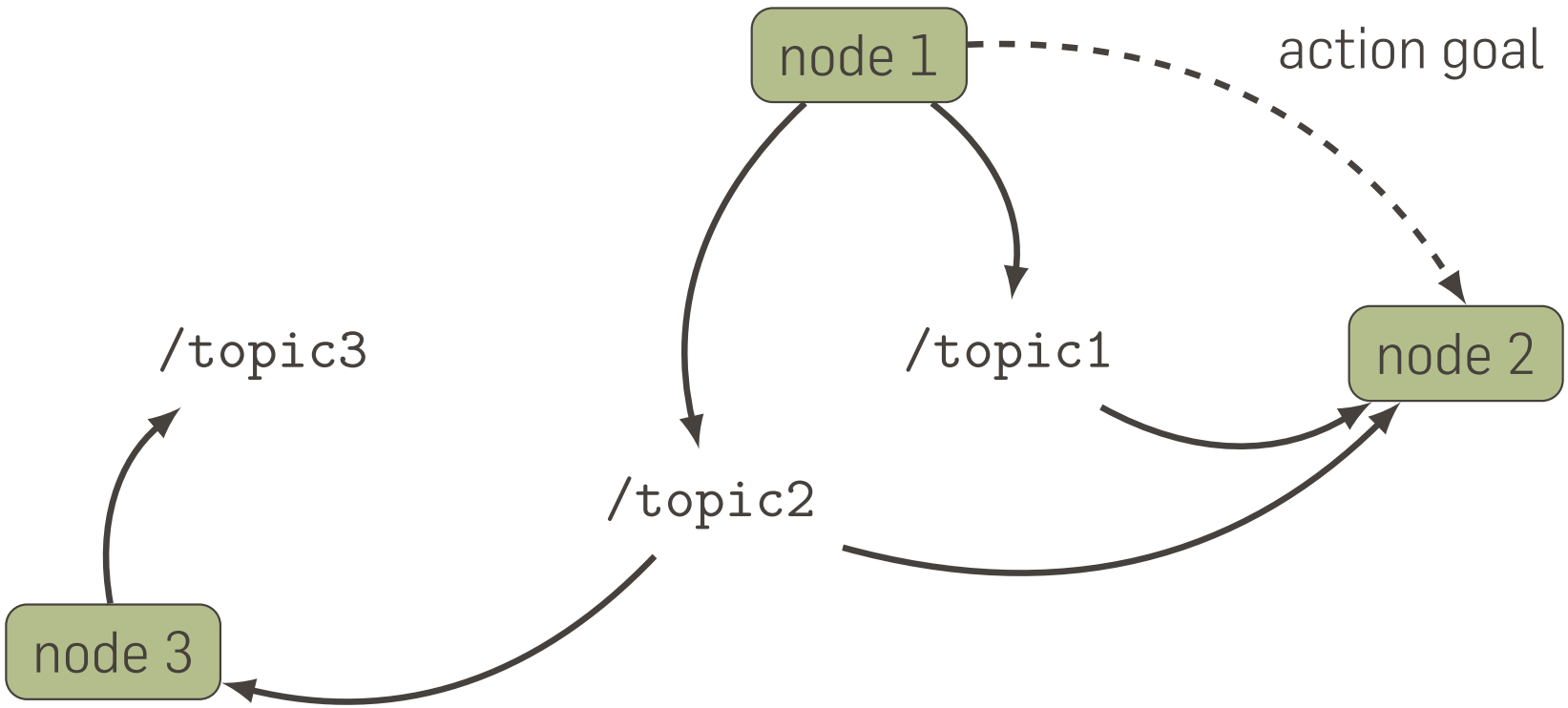


MIDDLEWARES' CORE PRINCIPLE: TALKING NODES

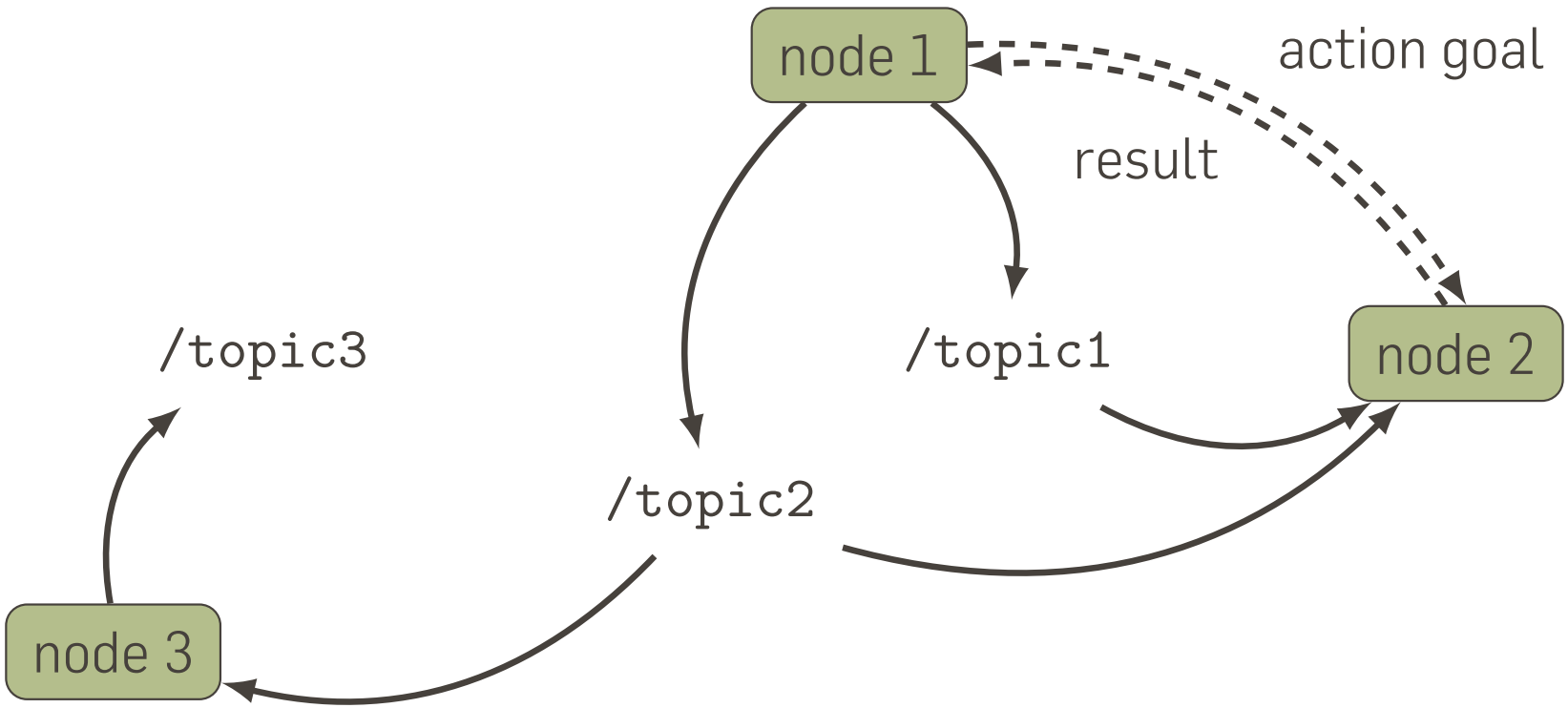


Services: **synchronous**

MIDDLEWARES' CORE PRINCIPLE: TALKING NODES

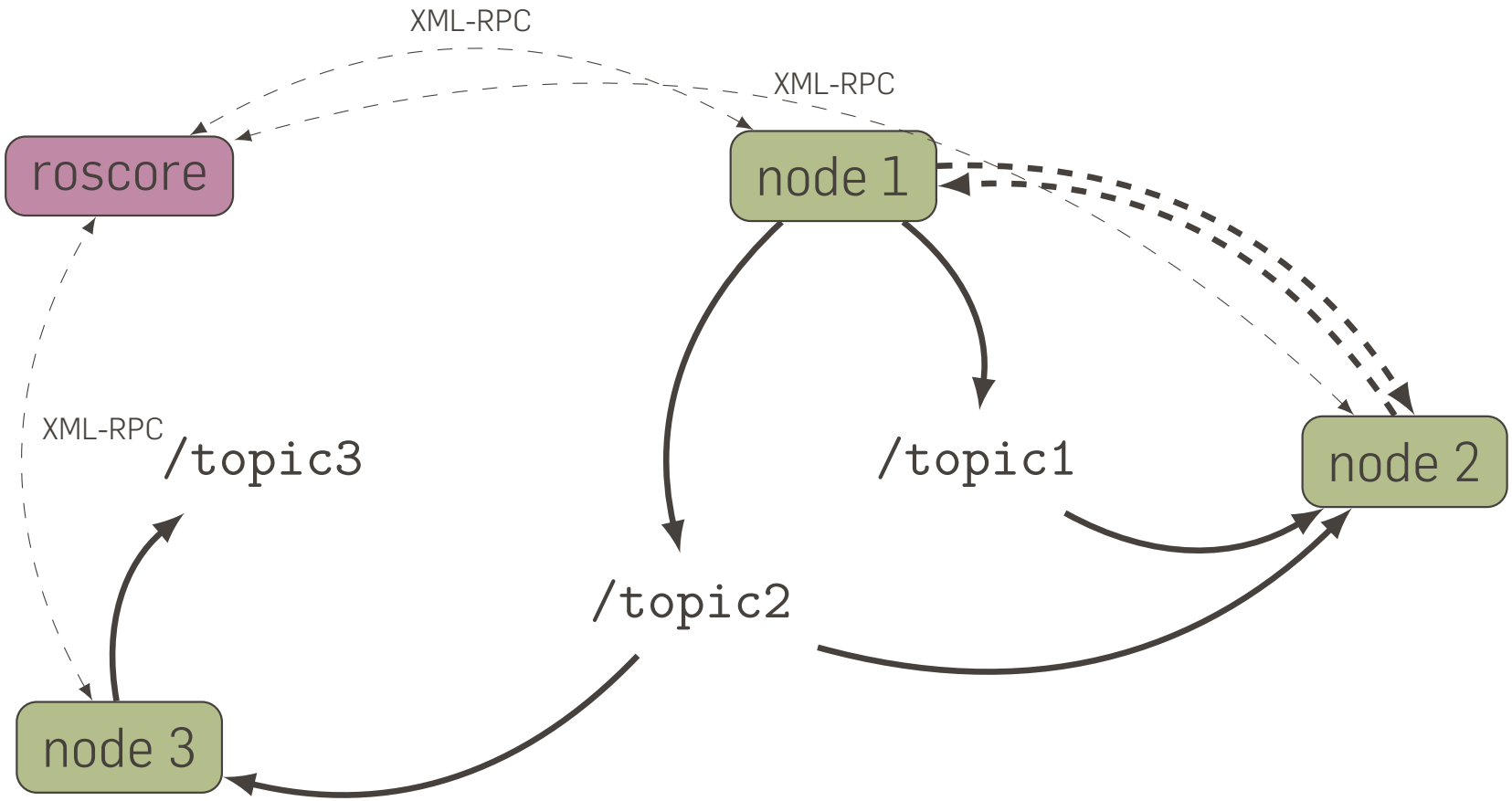


MIDDLEWARES' CORE PRINCIPLE: TALKING NODES



Actions: **asynchronous**

MIDDLEWARES' CORE PRINCIPLE: TALKING NODES



`ROS_MASTER_URI=http://<host>:<port>`

MIDDLEWARES: INTERFACE DEFINITIONS

Interfaces are called **messages** in ROS.

Message definitions are the interface definitions in ROS.

For example, a 6D pose:

```
$ rosmsg show geometry_msgs/Pose
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

MIDDLEWARES: INTERFACE DEFINITIONS

An image:

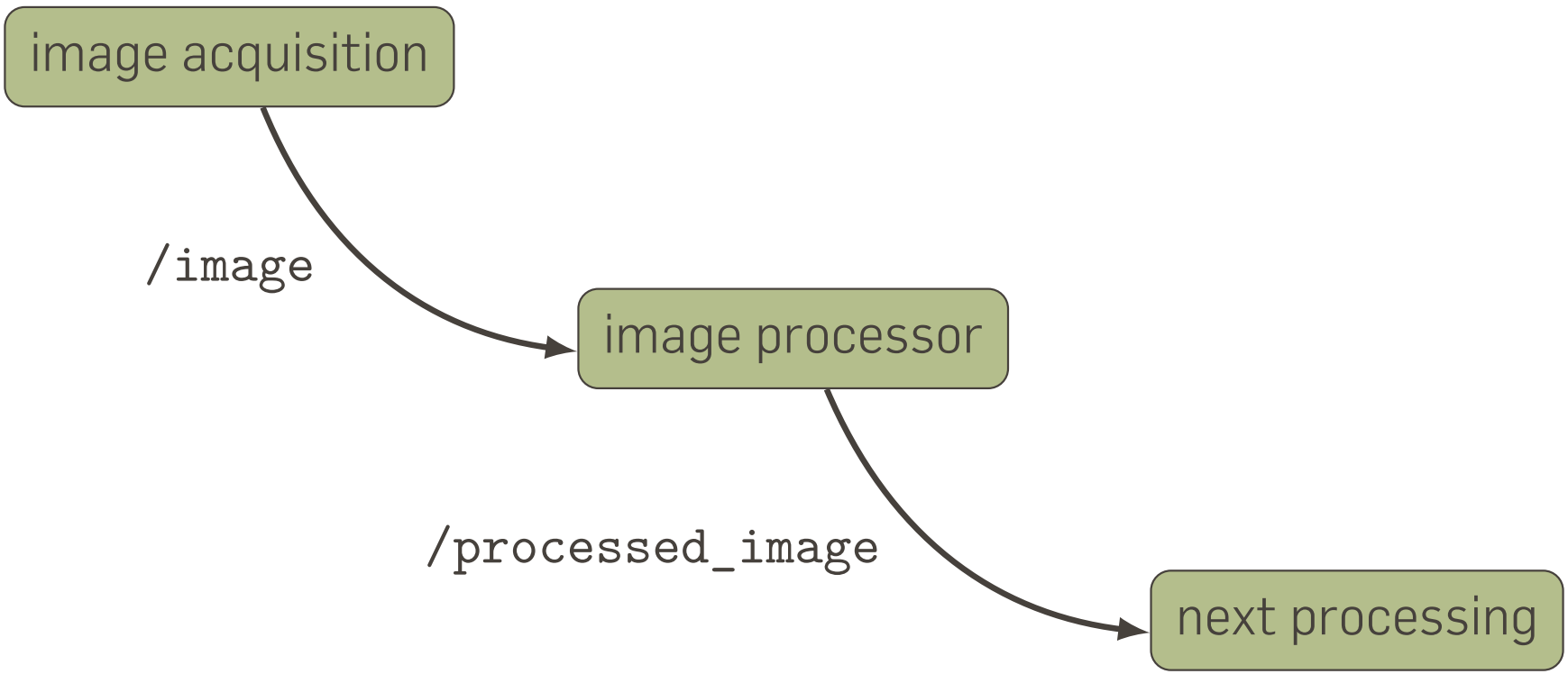
```
$ rosmsg show sensor_msgs/Image
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

MIDDLEWARES: INTERFACE INSTANTIATION

At run-time, messages are **instantiated** and **published over topics**:

```
$ rostopic echo /camera/image_raw
header:
    seq: 56
    stamp:
      secs: 1449243166
      nsecs: 415330019
    frame_id: /camera_frame
height: 720
width: 1280
encoding: rgb8
is_bigendian: 0
step: 3840
data: [32, 57, 51, 36, 61, 55, 41, 63, 60,...
```

EXAMPLE: A SIMPLE IMAGE PROCESSING PIPELINE



```
import sys, cv2, rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

def on_image(image):
    cv_image = bridge.imgmsg_to_cv2(image, "bgr8")
    (rows,cols,channels) = cv_image.shape
    cv2.circle(cv_image, (cols/2,rows/2), 50,(0,0,255), -1)
    cv2.imshow("Image window", cv_image)
    cv2.waitKey(3)
    image_pub.publish(bridge.cv2_to_imgmsg(cv_image, "bgr8"))

rospy.init_node('image_processor')
bridge = CvBridge()
image_sub = rospy.Subscriber("image",Image, on_image)
image_pub = rospy.Publisher("processed_image",Image)

while not rospy.is_shutdown():
    rospy.spin()
```

```
$ roslaunch gscam v4l.launch
$ python image_processor.py image:=/v4l/camera/image_raw
$ rosrun image_view image_view image:=/processed_image
```

That's all, folks!

Questions:
Portland Square B316 or **severin.lemaignan@plymouth.ac.uk**

Slides:
github.com/severin-lemaignan/module-mobile-and-humanoid-robots