Introduction to Cognitive Robotics

David Vernon

www.vernon.eu/cognitive_robotics

Lecture 38

www.vernon.eu/cognitive_robotics/CR_38.pdf

The CRAM Cognitive Architecture: Cognitive Robot Abstract Machine

1. Overview of CRAM
2. The main tools: Lisp, Emacs, CRAM Language, ROS
3. CRAM Beginner Tutorials with Turtlesim
4. A pick-and-place CRAM plan with a simulation of the PR2 robot

Lecture 38

www.vernon.eu/cognitive_robotics/CR_38.pdf

The CRAM Cognitive Architecture: Cognitive Robot Abstract Machine

1. Overview of CRAM
2. The main tools: Lisp, Emacs, CRAM Language, ROS
3. CRAM Beginner Tutorials with Turtlesim
   - Creating a CRAM package
   - Controlling Turtlesim from CRAM
   - Implementing simple plans to move a turtle
   - Using Prolog for reasoning
   - Creating motion designators for the TurtleSim
   - Creating process modules
   - Automatically choosing a process module for a motion
   - Using location designators with the TurtleSim
   - Writing high-level plans for the TurtleSim
   - Implementing failure handling for the TurtleSim

The CRAM Beginner Tutorials

Based on CRAM tutorials
http://cram-system.org/tutorials

Creating Process Modules

Based on Using Prolog for reasoning
http://cram-system.org/tutorials/beginner/process_modules_2

Creating Process Modules

- Different robots require different controllers to cater for their different kinematics
- We want an abstract interface in the high level of task specification
  - We don't want to be concerned if a robot has two arms or one when specifying a task
- Process modules provides this abstraction
  - A well-defined robot-independent interface for robot control
  - Used for high level planning

Creating Process Modules

Example

This is a location designator

These are the symbolic parameters for the designator

```
(with-designators
    ((my-designator :location '((:close-to :fridge))))
  (pm-execute :navigation my-designator))
```

Run a process module associated with :navigation. and use `my-designator` as an input parameter

- The resolution of the designator (by some other module in the system) provides the location
- which the process module then uses to control the robot and navigate to that location
- The kinematics of the robot (i.e. differential drive or legged locomotion) are abstracted away

Creating Process Modules

Writing a process module for turtlesim

- Use a process module to execute a resolved motion designator
- To drive to some location

Creating Process Modules

As before, when developing new code, we need to

- Update the dependencies in package.xml
- Update the dependencies in cram-my-beginner-tutorial.asd
- Update the dependencies in package.lisp
- Add the new code to process-modules.lisp
- Test the code
  - Run the ROS master
  - Run the Lisp REPL, loading the new program, creating a ROS node
  - Run turtlesim
  - Run turtlesim_teleop
  - Call the new functions

The new code depends on the cram-process-modules package

We will place the new code is the Lisp file

Creating Process Modules

Update the ROS dependencies

Make sure you are in the cram_my_beginner_tutorial sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial
~/workspace/ros/src/cram_my_beginner_tutorial$
```

Creating Process Modules

Update the ROS dependencies

Edit package.xml

~/workspace/ros/src/cram_my_beginner_tutorial$ emacs package.xml

Creating Process Modules

Update the ROS dependencies

Edit package.xml

Add the following lines

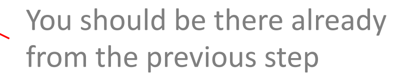
```
<exec_depend>cram_language</exec_depend>

<depend>turtlesim</depend>
<depend>roslisp</depend>
<depend>cl_transforms</depend>
<depend>geometry_msgs</depend>
<depend>cram_prolog</depend>
<depend>cram_designators</depend>
<depend>cram_process_modules</depend>
```

Creating Process Modules

Update the ASDF dependencies

Make sure you are in the cram_my_beginner_tutorial sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial
~/workspace/ros/src/cram_my_beginner_tutorial$
```



Creating Process Modules

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

```
~/workspace/ros/src/cram_my_beginner_tutorial$ emacs cram-my-beginner-tutorial.asd
```

Creating Process Modules

Update the ASDF dependencies

```
(defsystem cram-my-beginner-tutorial
  :depends-on (roslisp cram-language
               turtlesim-msg turtlesim-srv
               cl-transforms geometry_msgs-msg
               cram-designators cram-prolog
               cram-process-modules cram-language-designator-support)
  :components
  ((:module "src"
          :components
          ((:file "package")
           (:file "control-turtlesim" :depends-on ("package"))
           (:file "simple-plans" :depends-on ("package" "control-turtlesim"))
           (:file "motion-designators" :depends-on ("package"))
           (:file "process-modules" :depends-on ("package"
                                                 "control-turtlesim"
                                                 "simple-plans"
                                                 "motion-designators")))))
```

Add this line

Add these lines

Creating Process Modules

Update the Lisp package to add `:cram-process-modules` and `cram-language-designator-support` to the namespace

Make sure you are in the cram_my_beginner_tutorial/src sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial/src
~/workspace/ros/src/cram_my_beginner_tutorial/src$
```

Creating Process Modules

Update the Lisp package to add `:cram-process-modules` and `cram-language-designator-support` to the namespace

Edit package.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs package.lisp
```

Creating Process Modules

Update the Lisp package to add `:cram-process-modules` and `cram-language-designator-support` to the namespace

Edit package.lisp

Add `:cram-designators` to the `(:use :cpl ...)` line and add the `(:import-from ...)` line

```
(defpackage :cram-my-beginner-tutorial
  (:nicknames :tut)
  (:use :cpl :roslisp :cl-transforms :cram-designators
   :cram-process-modules cram-language-designator-support)
  (:import-from :cram-prolog :def-fact-group :<- :lisp-fun))
```

Add these

Be careful with the closing bracket

Creating Process Modules

Create a new Lisp file for the process modules code

Make sure you are in the cram_my_beginner_tutorial/src sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial/src
~/workspace/ros/src/cram_my_beginner_tutorial/src$
```

Creating Process Modules

Create a new Lisp file for the process modules code

Edit process-modules.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs process-modules.lisp
```

Creating Process Modules

Create a new Lisp file for the process modules code

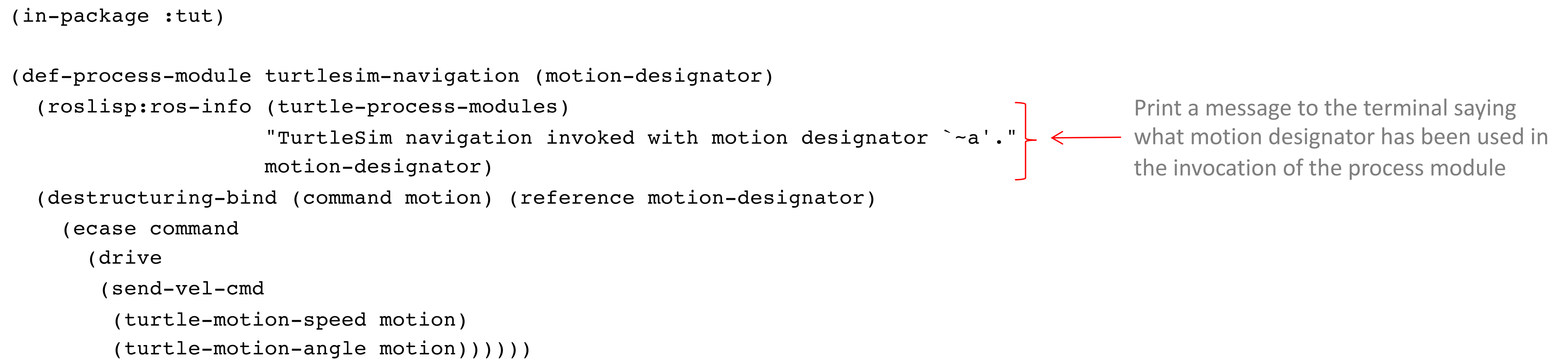
Edit process-modules.lisp

Copy and paste the code from the following slide

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))
```

The `cram-process-modules:def-process-module` macro is used to define the `turtlesim-navigation` process module with one parameter, `motion-designator`, a motion designator

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))
```



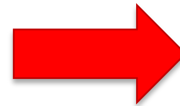

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))
```

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))
```

This binds the result of resolving (i.e. referencing) the designator passed as a parameter to the process module to the two variables `command` and `motion`.

Recall that the motion designators we wrote previously resolve with two values: the name of the motion and the motion itself.

We really should another one for `move` but we'll add that in a moment



For the inference rules we defined (see next slide for a reminder) there are two names, `drive` and `move`, and two motions, a `turtle-motion` structure and a `3d-vector` for `drive` and `move` motions, respectively

```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```

Recall that the motion designators we wrote previously resolve with two values: the name of the motion and the motion itself.

For the inference rules we defined there are two names, `drive` and `move`, and two motions, a `turtle-motion` structure and a `3d-vector` for `drive` and `move` motions, respectively

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))
```

The `ecase` Lisp expression is similar to a `case` expression except it signals an error if there is no match between the value of argument and the sequence of keys. Here we have just one key, `drive`, but we really should another one for `move`

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))
```

The `send-vel-cmd` function is the one we wrote previously to control the turtle.
We invoke it here with two arguments extracted from the resolved motion designator

Recall that these are the functions for accessing the slots of a structure

Creating Process Modules

Create a new Lisp file for the process modules code

Edit process-modules.lisp

Add the code from the following slide

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

Define a new function `drive` that has two parameters `?speed` and `?angle`

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

This is a CPL (CRAM Plan Language) macro so it needs to run inside a `top-level` form

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

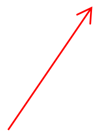
Activate the turtle process modules
(there's just one at the moment and that's `turtlesim-navigation`)

We can add more later and they will all run concurrently

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

Create a motion designator `trajectory` (of type `driving`) and pass the two parameter values to it

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```



Call the `cram-process-modules:pm-execute` macro to use the `turtlesim-navigation` process module to follow the trajectory specified by the `trajectory` designator

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

Don't be confused: the function `drive` has nothing to do with the `drive` command (label) in the ecase

Creating Process Modules

Now, let's experiment with this code

First, we need to make sure a ROS master is running

If you have not already done it, open a terminal and enter

```
~$ roscore
```

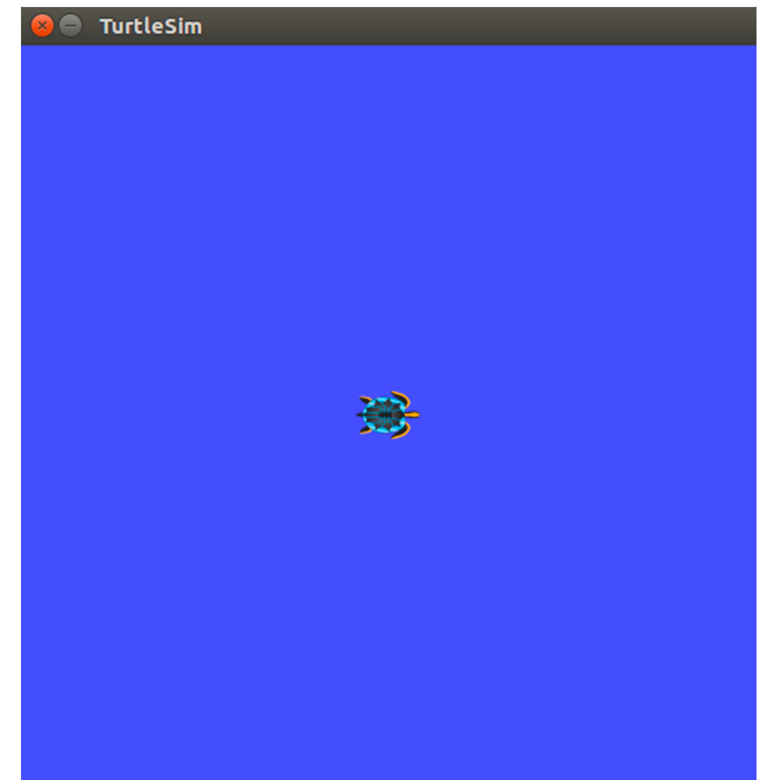
Creating Process Modules

Now, start turtlesim

Open a new terminal and enter

~$ rosrun turtlesim turtlesim_node

This is what you should see



Creating Process Modules

Launch the Lisp REPL

If you have not already done it, open a terminal and enter

```
~/workspace/ros$ roslisp_repl
```

Load the system

```
CL-USER> (ros-load:load-system "cram_my_beginner_tutorial" :cram-my-beginner-tutorial)
```

Switch to the package

```
CL-USER> (in-package :tut)
TUT>
```

Creating Process Modules

Start a ROS node

The name doesn't matter

```
TUT> (start-ros-node "turtle1")
[(ROSLISP TOP) INFO] 1292688669.674: Node name is turtle1
[(ROSLISP TOP) INFO] 1292688669.687: Namespace is /
[(ROSLISP TOP) INFO] 1292688669.688: Params are NIL
[(ROSLISP TOP) INFO] 1292688669.689: Remappings are:
[(ROSLISP TOP) INFO] 1292688669.691: master URI is 127.0.0.1:11311
[(ROSLISP TOP) INFO] 1292688670.875: Node startup complete
```

Creating Process Modules

Call the function we wrote to perform the initialization

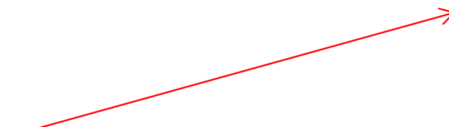
```
TUT> (init-ros-turtle "turtle1")
```

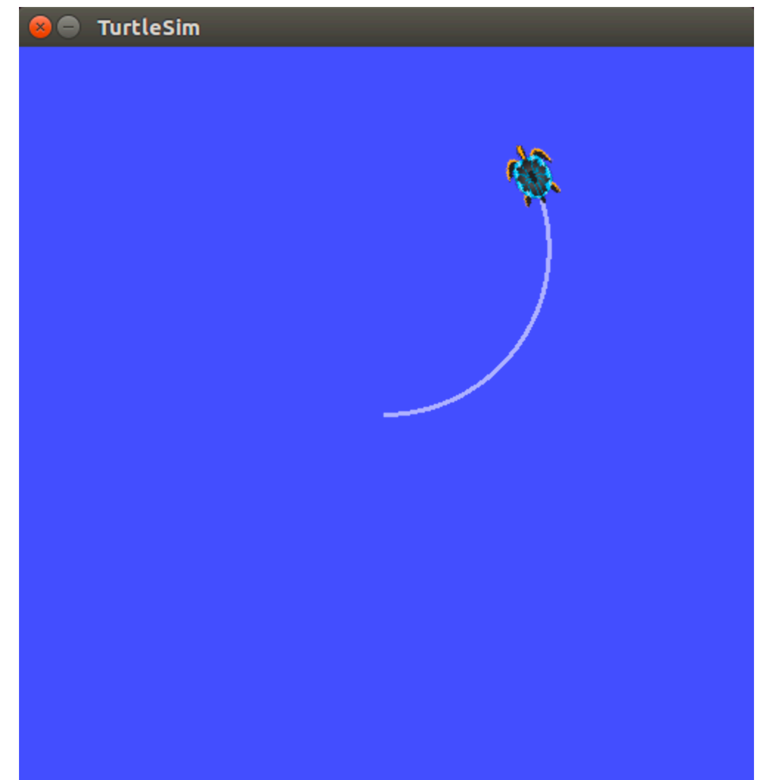
Creating Process Modules

Now, let's call `drive`

TUT> (drive 5 2)

This is what you should see





Creating Process Modules

Now, let's call `drive`

```
TUT> (drive 5 2)
[(TURTLE-PROCESS-MODULES) INFO] 1562698751.679: TurtleSim navigation invoked with motion designator `#<A MOTION
   (TYPE DRIVING)
   (SPEED 5)
   (ANGLE 2)>'.
2
```

Creating Process Modules

When adding a process module we have a choice:

1. Add it to an existing process module

2. Create a new process module

Creating Process Modules

To prevent unwanted behaviour when executing multiple designators in parallel or in succession, a process module only resolves one designator at a time

- Therefore, if the process module to be added uses the same resources as an existing one, e.g. two or more process modules for robot locomotion,

  we should add it the existing process module

- Otherwise, e.g adding a process module for robot manipulation to process modules for locomotion, we should add it as a new process module

Note: if a process module is called while still executing, the new call is queued as executed when the current call is finished ... we'll see that in just a moment

Creating Process Modules

Let's add process modules, one for locomotion and one for setting the pen:

1. The locomotion one (in this case, for a move motion) should be added to the existing process module

2. The set-pen module can be a new process module

Creating Process Modules

Extend the Lisp file for the process modules code

Edit process-modules.lisp

Add the code in black from the following slide

(or, alternatively, copy copy-paste it all, overwriting the existing code)

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion)))
      (move
        (move-to motion)))))

(def-process-module turtlesim-pen-control (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim pen control invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (set-pen
       (call-set-pen
        (pen-motion-r motion)
        (pen-motion-g motion)
        (pen-motion-b motion)
        (pen-motion-width motion)
        (pen-motion-off motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion)))
      (move
        (move-to motion)))))
```

Finally, we add the case to handle the motion designator for the `move` command
Since `move-to` takes a 3d-vector as a parameter, we can pass the second resolved argument `motion` to it directly

```
(def-process-module turtlesim-pen-control (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim pen control invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (set-pen
       (call-set-pen
        (pen-motion-r motion)
        (pen-motion-g motion)
        (pen-motion-b motion)
        (pen-motion-width motion)
        (pen-motion-off motion))))))
```

Add this code for the new process module for setting the pen

It uses the call-set-pen function we wrote previously

```
(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))
```

Creating Process Modules

Executing process modules in parallel

First, let's call the same process module twice to see how it behaves

- We'll use the `par` macro
- One process module will use a motion designator for driving
- The other process module will use a motion designator for moving

Creating Process Modules

Remember:

- If the turtlesim environment gets a bit messy,
  you can clear the background by entering the following from a terminal

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ rosservice call /clear
```

- Or you can reset it completely by entering the following from a terminal
  (this creates a new turtle in the default pose)

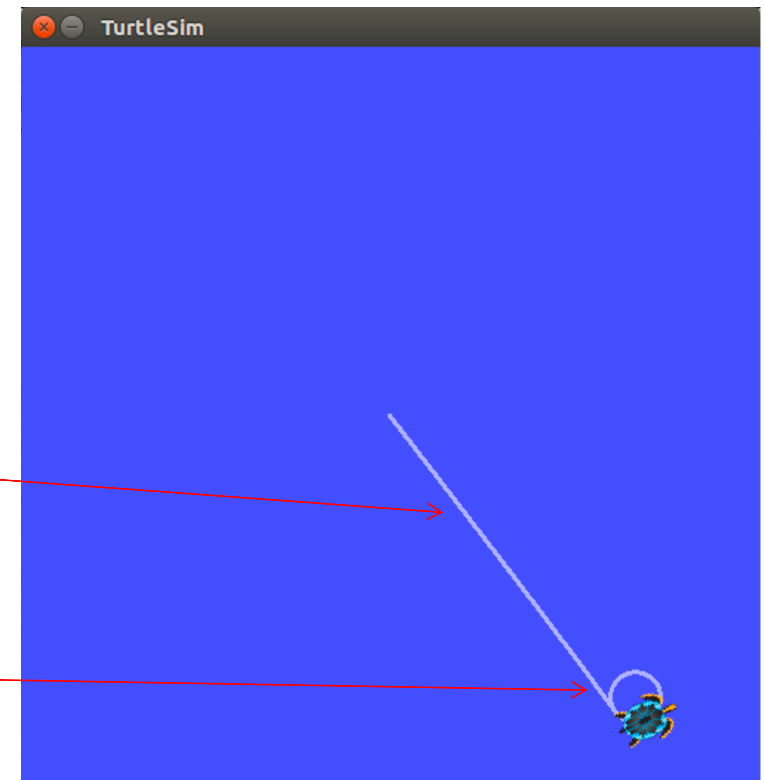
```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ rosservice call /reset
```

Creating Process Modules

```
TUT> (top-level
  (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
   (let ((goal (desig:a motion (type moving) (goal (9 1 0))))
         (trajectory (desig:a motion (type driving) (speed 3) (angle 8))))
     (cpl:par
       (pm-execute 'turtlesim-navigation goal)
       (pm-execute 'turtlesim-navigation trajectory)))))
```

First the turtle moves to the goal

Then. it drives in a circle



Creating Process Modules

```
TUT> (top-level
  (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
   (let ((goal (desig:a motion (type moving) (goal (9 1 0))))
        (trajectory (desig:a motion (type driving) (speed 3) (angle 8))))
    (cpl:par
      (pm-execute 'turtlesim-navigation goal)
      (pm-execute 'turtlesim-navigation trajectory)))))
[(TURTLE-PROCESS-MODULES) INFO] 1500997686.711: TurtleSim navigation invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                        MOVING)
                                       (GOAL
                                        (9 1
                                         0))) {1006DBC893}>'.
WARNING:
  Process module #<TURTLESIM-NAVIGATION
            {10074C0293}> already processing input. Waiting for it to become free.
[(TURTLE-PROCESS-MODULES) INFO] 1500997690.065: TurtleSim navigation invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                        DRIVING)
                                       (SPEED
                                        3)
                                       (ANGLE
                                        8)) {1006DBCC73}>'.
```

Activate both process modules

Create a motion designator of type moving and set the goal

Create a motion designator of type driving and set the speed and angle

Run the following process modules in parallel

The two are not executed in parallel
and the second one is queued for execution

Creating Process Modules

Executing process modules in parallel

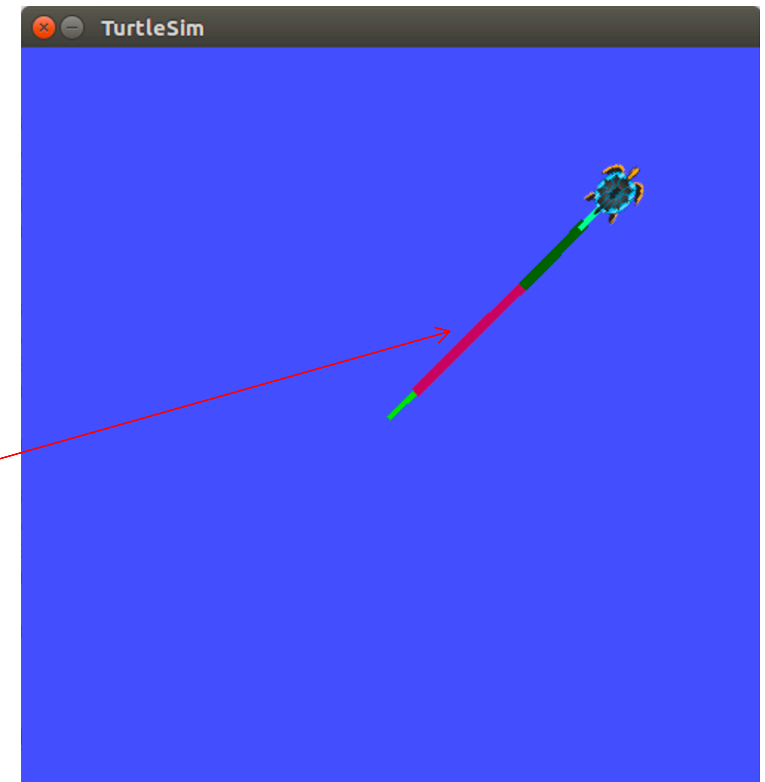
Next, let's call the two different process modules to see how it behaves

- We'll use the `par` macro
- One process module will use a motion designator for setting-pen
  - We will change the pen colour and width twice a second
- The other process module will use a motion designator for moving
- Now, the turtle moves and changes the pen in parallel

Creating Process Modules

```
TUT> (top-level
  (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
   (let ((goal (desig:a motion (type moving) (goal (9 9 0)))))
    (cpl:par
     (pm-execute 'turtlesim-navigation goal)
     (dotimes (i 10)
      (pm-execute 'turtlesim-pen-control
          (let ((?r (random 255))
              (?g (random 255))
              (?b (random 255))
              (?width (+ 3 (random 5))))
          (desig:a motion (type setting-pen) (r ?r) (g ?g) (b ?b) (width ?width))))
      (sleep 0.5))))))
```

The pen changes as the turtle moves



Creating Process Modules

Some notes:

- This approach is used
  - to ensure that a single resource on a robot is not used by several functions simultaneously
  - while allowing parallel execution functions using independent resources
- This is fine for low level motions but won't always be appropriate
  - For example, in high level task specification when grasping an object while the robot is moving
  - We need more sophisticated ways of handling possible interference between the sub-tasks

CRAM Beginner Tutorials

Create a CRAM Package	http://cram-system.org/tutorials/beginner/package_for_turtlesim
Controlling turtlesim from CRAM	http://cram-system.org/tutorials/beginner/controlling_turtlesim_2
Implementing simple plans to move a turtle	http://cram-system.org/tutorials/beginner/simple_plans
Using Prolog for reasoning	http://cram-system.org/tutorials/beginner/cram_prolog
Creating motion designators for the TurtleSim	http://cram-system.org/tutorials/beginner/motion_designators
Creating process modules	http://cram-system.org/tutorials/beginner/process_modules_2

Background Reading

G. Kazhoyan, Lecture notes: Robot Programming with Lisp 7. Coordinate Transformations, TF, ActionLib, slides 5-8.
`https://ai.uni-bremen.de/_media/teaching/7_more_ros.pdf`

`http://wiki.ros.org/tf/Overview/Transformations`

T. Rittweiler, CRAM – Design and Implementation of a Reactive Plan Language, Bachelor Thesis, Technical University of Munich, 2010.
`https://common-lisp.net/~trittweiler/bachelor-thesis.pdf`