Introduction to Cognitive Robotics

David Vernon

www.vernon.eu/cognitive_robotics

Lecture 42

www.vernon.eu/cognitive_robotics/CR_42.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 42

www.vernon.eu/cognitive_robotics/CR_42.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 TurtleSim
   - Creating process modules
   - Automatically choosing a process module for a motion
   - Using location designators with TurtleSim
   - Writing high-level plans for TurtleSim
   - Implementing failure handling for TurtleSim

The CRAM Beginner Tutorials

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

Implementing Failure Handling for Turtlesim

Based on Implementing failure handling for the TurtleSim
http://cram-system.org/tutorials/beginner/failure_handling

Failure Handling in CRAM

Overview

- Failure is normal in robotics so we need to handle it
- You can't avoid failure either
  - A robot's environment is typically non-deterministic
  - so you can't plan to avoid all failures
- In this part of the course we introduce the CRAM macro for handling failure: `with-failure-handling`

Failure Handling in CRAM

Overview

Just for reference, the following is the CRAM documentation on `with-failure-handling`

"Macro that replaces handler-case in cram-language. This is necessary because error handling does not work across multiple threads. When an error is signaled, it is put into an envelope to avoid invocation of the debugger multiple times. When handling errors, this envelope must also be taken into account.

We also need a mechanism to retry since errors can be caused by plan execution and the environment is highly non-deterministic. Therefore, it is possible to use the function 'retry' that is lexically bound within with-failure-handling and causes a re-execution of the body.

When an error is unhandled, it is passed up to the next failure handling form (exactly like handler-bind). Errors are handled by invoking the retry function or by doing a non-local exit. Note that with-failure-handling implicitly creates an unnamed block, i.e. 'return' can be used." ← This will be relevant when we write the failure handing code at the end

Failure Handling in CRAM

Overview

- To implement failure handling in a plan, we have to
  - Define a failure condition
  - Signal it somewhere sensible
  - Handle it
- Handling a failure often involved some recovery strategy and retrying
  - So that the robot can carry on with its task and pursuing its goal
  - For example, if a robot failed to grasp something, it might retract its arm, move to a new position that offers a better chance of successfully grasping the object, and retry

Failure Handling in CRAM

Overview

Here's an example of how `with-failure-handling` can be used

```
(with-failure-handling
    ((some-error-condition (e)
       (print e)
       (recover-from e)
       (retry)))
  (do-some-failure-prone-stuff))
```

When an error `e` is signalled ...

We print it, recover from it, and try again ...
which means re-executing `do-some-failure-prone-stuff`

This is the normal task-oriented code:
when something goes wrong, it signals an error `e`

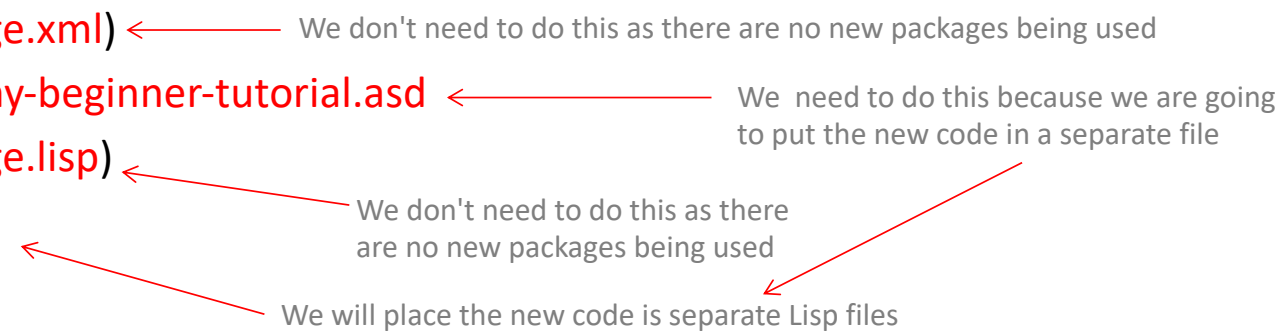
Start reading here ...

Failure Handling in CRAM

Failure detection and error signalling

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 conditions.lisp
- Test the code



Failure Handling in CRAM

Failure detection and error signalling

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

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

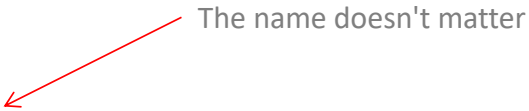
Failure Handling in CRAM

```
(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
                cram-executive
                std_srvs-srv)
   :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 "location-designators" :depends-on ("package"))
              (:file "action-designators" :depends-on ("package"))
              (:file "conditions" :depends-on ("package"))
              (:file "process-modules" :depends-on ("package"
                                                    "control-turtlesim"
                                                    "simple-plans"
                                                    "motion-designators"))
              (:file "selecting-process-modules" :depends-on ("package"
                                                              "motion-designators"
                                                              "process-modules"))
              (:file "high-level-plans" :depends-on ("package"
                                                     "motion-designators"
                                                     "location-designators"
                                                     "action-designators"
                                                     "process-modules"
                                                     "conditions"))))))
```

Add these lines

Failure Handling in CRAM

Start a ROS node




```
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
```

Failure Handling in CRAM

Failure detection and error signalling

Create a new Lisp file for the failure conditions 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$
```

Failure Handling in CRAM

Failure detection and error signalling

Create a new Lisp file for the failure conditions code:

Edit conditions.lisp

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

Failure Handling in CRAM

Failure detection and error signalling

Create a new Lisp file for the failure conditions code:

Edit conditions.lisp

Copy and paste the code from the following slide

This defines the condition `out-of-bounds-error` which inherits from `cpl:simple-plan-failure`

```
(in-package :tut)

(define-condition out-of-bounds-error (cpl:simple-plan-failure)
  ((description :initarg :description
                :initform "Turtle went out of bounds."
                :reader error-description))
  (:documentation "Turtle went out of bounds.")
  (:report (lambda (condition stream)
             (format stream (error-description condition)))))
```

We will signal this condition when the turtle move out of bounds

Failure Handling in CRAM

Failure detection and error signalling

We now need to extend the `navigate` plan in high-level-plans.lisp:

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$
```

Failure Handling in CRAM

Failure detection and error signalling

We now need to extend the `navigate` plan in high-level-plans.lisp:

Edit high-level-plans.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs high-level-plans.lisp
```

Failure Handling in CRAM

Failure detection and error signalling

We now need to extend the `navigate` plan in high-level-plans.lisp:

Edit high-level-plans.lisp

Copy and paste the code from the following slide,

- Adding the `defparameter` forms
- Overwriting the current implementation of the `navigate` function

```
(defparameter *min-bound* 0.5)
(defparameter *max-bound* 10.5)

(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (pursue
      (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
        (error 'out-of-bounds-error))
      (exe:perform (a motion (type moving) (goal ?v))))))
```

```
(defparameter *min-bound* 0.5)
(defparameter *max-bound* 10.5)

(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (pursue
      (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
        (error 'out-of-bounds-error))
      (exe:perform (a motion (type moving) (goal ?v))))))
```

Define the minimum and maximum valid values of turtle coordinates

We defined this function previously as

```
(defun navigate (?v)
  (exe:perform (a motion (type moving) (goal ?v))))
```

Define a local function `out-of-bounds` using `flet`
which returns T if the coordinates are outside the allowable minimum and maximum values

Create a fluent network with *turtle-pose* and the function `out-of-bounds`.

The function `out-of-bounds` takes `*turtle-pose*` as an argument, extracts the coordinates, and returns T if it is out of bounds.

The `whenever` macro executes its body (i.e. the error function) whenever the value of the passed fluent is non-NIL, i.e. T.

In parallel, using `pursue`, resolve the motion designator and execute the process module

We use `pursue` not `par` because we want the parallel form to return after the plan is executed.

`par` would never return, because not all of its children forms return.

`pursue` returns as soon as `perform` returns.

Failure Handling in CRAM

Launch the Lisp REPL

Open a new 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>
```

Failure Handling in CRAM

Test the error signalling

```
TUT> (top-level
   (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
     (navigate-without-pen '(4 8 0))
     (exe:perform (an action (type drawing) (shape house)))))
[(TURTLE-PROCESS-MODULES) INFO] 1503583551.243: TurtleSim pen control invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                 SETTING-PEN)
                                                (OFF
                                                 1)) {10047EB603}>'.

                                                 [ ... ]

[(TURTLE-PROCESS-MODULES) INFO] 1503583557.555: TurtleSim navigation invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                 MOVING)
                                                (GOAL
                                                 (9.08701467514038d0
                                                  12.503347396850586d0
                                                  0))) {1009740303}>'.
; Evaluation aborted on #<CRAM-BEGINNER-TUTORIAL::OUT-OF-BOUNDS-ERROR {10089BA023}>.
TUT>
```

Evaluation is aborted when the error is signalled

Failure Handling in CRAM

Recovering from failure

To recover from failure once it is detected and signaled

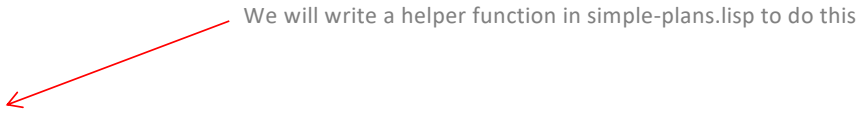
- Extend the navigate plan
- Add a function to implement a recovery strategy

Failure Handling in CRAM

Recovering from failure

The strategy we are going to implement is as follows:

When out of bounds:



1. Rotate towards the center of the world
2. Drive a bit forward
3. Calculate a new point inside the bounds to move to instead of the original target
4. Rotate towards this target
5. Move to the new target

Failure Handling in CRAM

Recovering from failure

Edit simple-plans.lisp

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$
```

Failure Handling in CRAM

Recovering from failure

Add a helper function to rotate the turtle so that it is facing a goal position

Edit simple-plans.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs simple-plans.lisp
```

Failure Handling in CRAM

Recovering from failure

Add a helper function to rotate the turtle so that it is facing a goal position

Edit simple-plans.lisp

Copy and paste the code from the following slide

This function is similar to move-to but just rotates the turtle

```
(defun rotate-to (goal &optional (threshold 0.05))
  (let ((reached-fl (< (fl-funcall #'abs
                                   (fl-funcall #'relative-angle-to goal *turtle-pose*))
                       threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
              0
              (calculate-angular-cmd goal))
             (wait-duration 0.01)))
      (send-vel-cmd 0 0))))
```

Forward velocity is zero

Failure Handling in CRAM

Recovering from failure

Now we implement the failure handling itself

Edit high-level-plans.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs high-level-plans.lisp
```

Failure Handling in CRAM

Recovering from failure

Extend the `navigate` plan in high-level-plans.lisp:

Edit high-level-plans.lisp

Copy and paste the code from the following slide:

- Overwriting the current implementation of the `navigate` function
- Adding the `out-of-bounds-error` function to recover from the failure

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

New

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

New

Rotate to towards the centre

Move a little towards towards the centre

If the optional goal has been provided then
rotate towards the goal

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

This form handles the `out-of-bounds` errors that are signalled in the body of `with-failure-handling`

Set the pen colour to red and the width to 2

New

We return to ensure the condition counts as having been handled (see the documentation at the beginning of this section of failure handling)

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

Calculate a new position inside the allowable bounds by:
extracting the x coordinate of the goal `(car ?v)`
the y coordinate of the goal `(cadr ?v)` ... note this is an abbreviation for `car(cdr ?v))`
choosing the minimum of the value and the maximum allowable value
choosing the maximum of the result and the minimum allowable value
making a list of the result and assigning it to `?corr-v`
For example, (6.5 12.5 0) would become (6.5 10.4 0)

New

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

New

Call `recover-from-oob` to move in bounds again

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

New

Move to the new position. inside the allowable bounds

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

New

Reset the pen

```
(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

New

We return to ensure the condition counts as having been handled
(see the documentation at the beginning of this section of failure handling)

Failure Handling in CRAM

Recovering from failure

For the sake of completeness, all the code for high-level-plans.lisp is on the next following slide:

```
(in-package :tut)

(defun draw-house ()
  (with-fields (x y)
      (value *turtle-pose*)
    (exe:perform (an action (type drawing) (shape rectangle) (width 5) (height 4.5)))
    (navigate-without-pen (list (+ x 3) y 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 2.5)))
    (navigate-without-pen (list (+ x 0.5) (+ y 2) 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 1)))
    (navigate-without-pen (list x (+ y 4.5) 0))
    (exe:perform (an action (type drawing) (shape triangle) (base-width 5) (height 4)))))

(defun draw-simple-shape (vertices)
  (mapcar
   (lambda (?v)
     (exe:perform (an action (type navigating) (target ?v))))
   vertices))

(defun navigate-without-pen (?target)
  (exe:perform (a motion (type setting-pen) (off 1)))
  (exe:perform (an action (type navigating) (target ?target)))
  (exe:perform (a motion (type setting-pen) (off 0))))

(defparameter *min-bound* 0.5)
(defparameter *max-bound* 10.5)

(defun navigate (?v)
  (flet ((out-of-bounds (pose)
           (with-fields (x y)
               (value pose)
             (not (and (< *min-bound* x *max-bound*)
                       (< *min-bound* y *max-bound*))))))
    (with-failure-handling
        ((out-of-bounds-error (e)
           (ros-warn (draw-simple-simple) "Moving went-wrong: ~a" e)
           (exe:perform (a motion (type setting-pen) (r 204) (g 0) (b 0) (width 2)))
           (let ((?corr-v (list
                           (max 0.6 (min 10.4 (car ?v)))
                           (max 0.6 (min 10.4 (cadr ?v)))
                           0)))
             (recover-from-oob ?corr-v)
             (exe:perform (a motion (type moving) (goal ?corr-v))))
           (exe:perform (a motion (type setting-pen) (off 0)))
           (return)))
      (pursue
        (whenever ((fl-funcall #'out-of-bounds *turtle-pose*))
          (error 'out-of-bounds-error))
        (exe:perform (a motion (type moving) (goal ?v)))))))

(defun recover-from-oob (&optional goal)
  (rotate-to (make-3d-vector 5.5 5.5 0))
  (send-vel-cmd 1 0)
  (wait-duration 0.2)
  (when goal
    (rotate-to (apply #'make-3d-vector goal))))
```

Failure Handling in CRAM

Test the plan with failure handling

Load the system

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

Switch to the package

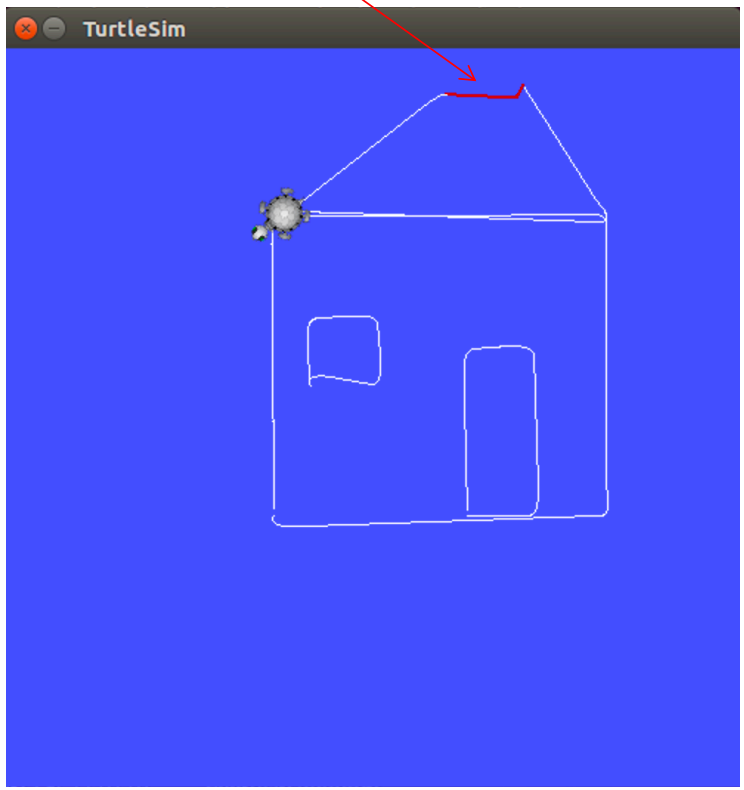
```
CL-USER> (in-package :tut)
TUT>
```

Failure Handling in CRAM

Test the plan with failure handling

```
TUT> (top-level
       (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
         (navigate-without-pen '(4 4 0))
         (exe:perform (an action (type drawing) (shape house)))))
```

The red lines are drawn when recovering from failure.
First, move a small distance towards the center
Then, calculate new achievable goal coordinates (6.5 10.4)
and move there

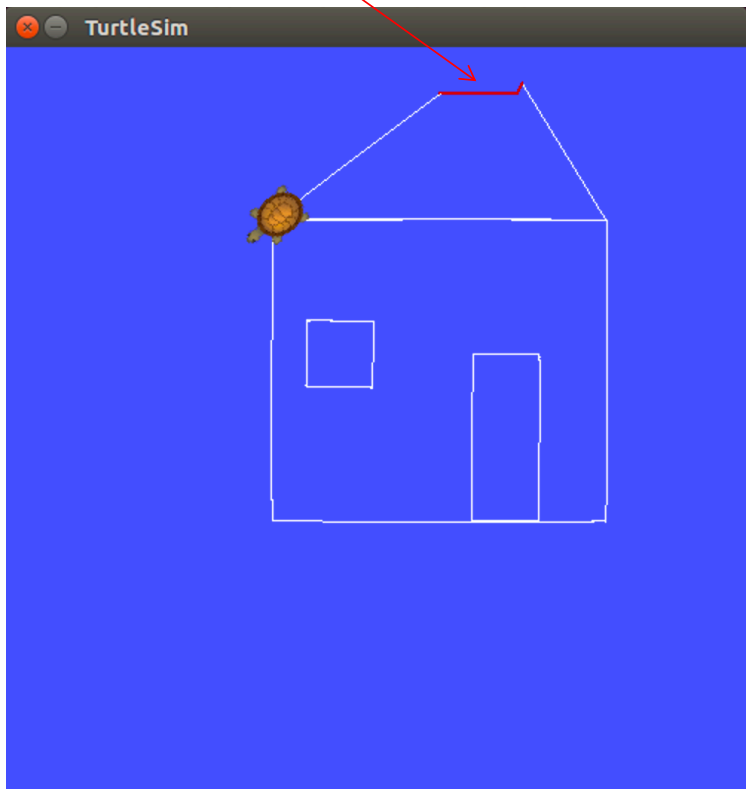


Failure Handling in CRAM

Test the plan with failure handling

```
TUT> (top-level
       (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
         (navigate-without-pen '(4 8 0))
         (exe:perform (an action (type drawing) (shape house)))))
```

The red lines are drawn when recovering from failure.
First, move a small distance towards the center
Then, calculate new achievable goal coordinates (6.5 10.4)
and move there



This is the result if we replace the move-to function with one based on the divide-and-conquer algorithm we covered earlier in the course, using a threshold of 0.01 on distance

Failure Handling in CRAM

Retry counters

Sometimes, it makes sense to try performing an action or a motion a number of times

- Test different positions
- The world is non-deterministic and something might have changes

Failure Handling in CRAM

Retry counters

We can do this using the `with-retry-counters` macro

- We can define an arbitrary number of retry counters
- We then can use `do-retry` with one of these counters
  - This decreases the number of remaining retries for that counter
- `do-retry` only executes its body when there are retries left for the given retry-counter

Failure Handling in CRAM

From the documentation

Just for reference, the following is the CRAM documentation on `with-retry-counters`

"Lexically binds all counters in 'counter-definitions' to the initial values specified in 'counter-definitions'. 'counter-definitions' is similar to 'let' forms with the difference that the counters will not be available under the specified names in the lexical environment established by this macro. In addition, the macro defines the local macro (DO-RETRY <counter> <body-form>*) to decrement the counter and execute code when the maximal retry count hasn't been reached yet and the function '(RESET-COUNTER <counter>)."

Failure Handling in CRAM

Retry counters

Here is an example of how to use it

```
(with-retry-counters ((some-error-counter 3))
  (with-failure-handling
      ((some-error-condition (e)
         (print e)
         (do-retry some-error-counter
           (recover-from e)
           (retry))))
    (do-some-failure-prone-stuff)))
```

Three retries for this counter: we will try executing `do-some-failure-prone-stuff` three times after. which the `do-retry` body won't be executed and the `some-error-condition` won't be handled

The `with-failure-handling` is now the body of `with-retry-counter`

Retry using this. counter

The recovery part is now the body of `do-retry`

Now retry the `do-some-failure-prone-stuff`

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
Automatically choosing a process module for a motion	http://cram-system.org/tutorials/beginner/assigning_actions_2
Using location designators with the TurtleSim	http://cram-system.org/tutorials/beginner/location_designators_2
Writing plans for the TurtleSim	http://cram-system.org/tutorials/beginner/high_level_plans
Implementing failure handling for the TurtleSim	http://cram-system.org/tutorials/beginner/failure_handling

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`