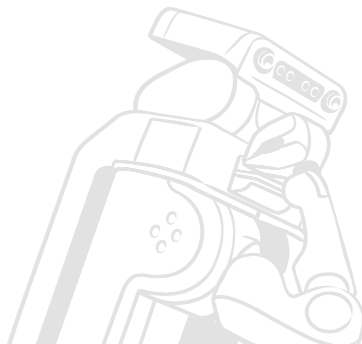
Introduction into ROS and Cram

Lorenz Mösenlechner (moesenle@in.tum.de)

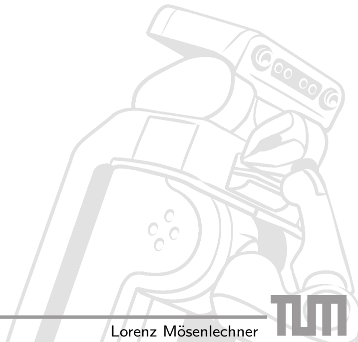
Intelligent Autonomous Systems Group
Technische Universität München

December 6, 2011



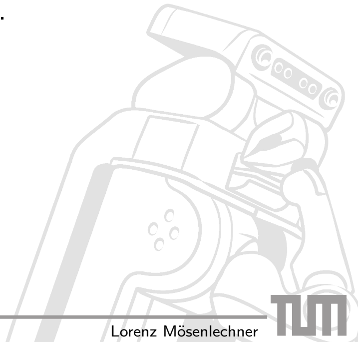
ROS

- ROS is a Meta-Operating System
- Powerful build system
- Middle-ware
  - Publisher-subscriber pattern
  - Synchronous services



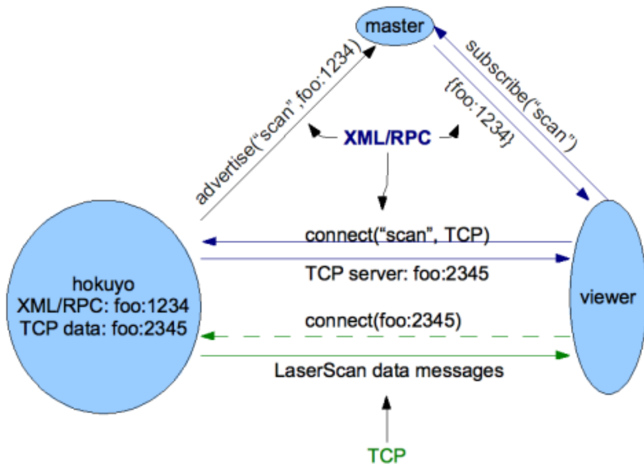
ROS Packages

- Package name = directory name
- Package path: .../stack-name/package-name
- manifest.xml for meta-information, e.g. dependencies, description, exported compile flags, ...
- roscd package-name



Publisher/Subscriber

- Typed topics (e.g /turtle1/pose, type turtlesim/pose)
- Publishers post stream of data on the topic
- Subscribers get a callback on every message on the topic
- Direct connection between all publishers and all subscribers


master
advertise("scan",foo:1234)
subscribe("scan")
{foo:1234}
XML/RPC
hokuyo
XML/RPC: foo:1234
TCP data: foo:2345
viewer
connect("scan", TCP)
TCP server: foo:2345
connect(foo:2345)
LaserScan data messages
TCP

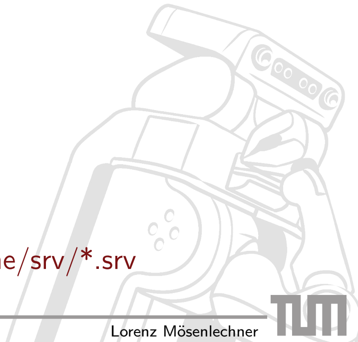
Messages

- Defined in package-name/msg/*.msg
- Basic data types:
  - int8,16,32,64
  - float32,64
  - string
  - time
  - duration
  - arrays: type[]
- Example:

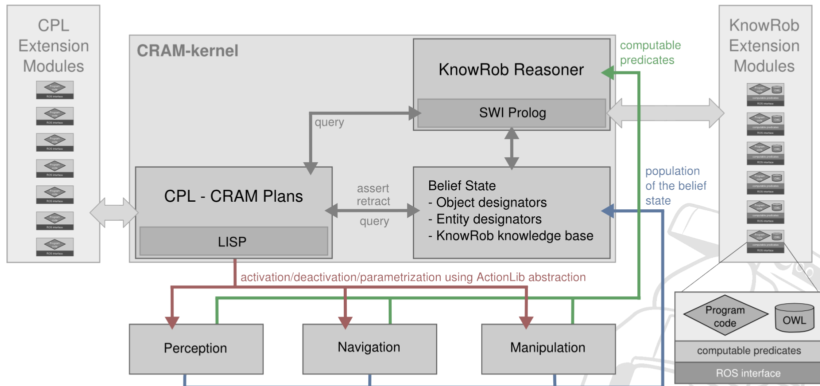
  Point.msg

  ```
  float64 x
  float64 y
  float64 z
  ```

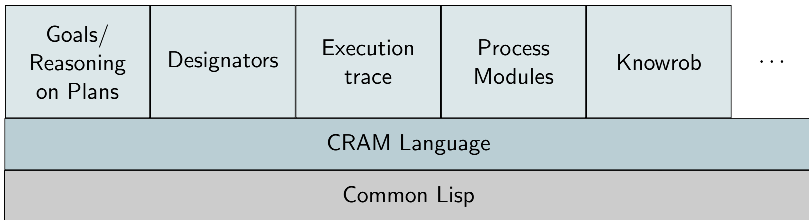
- ROS Services defined in package-name/srv/*.srv



Cognitive Robot Abstract Machine



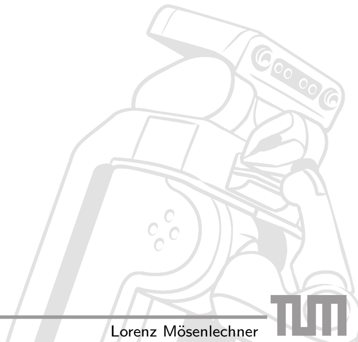
The CRAM Core



High-Level Robot Control

Task execution

- `Parallel`
- `Synchronization`
- `Robust and flexible`
- `Failure handling`



High-Level Robot Control

Task execution

- Parallel
- Synchronization
- Robust and flexible
- Failure handling

Requirements for the Language

- Expressive
- Easy to use

High-Level Robot Control

Task execution

- `Parallel`
- `Synchronization`
- `Robust and flexible`
- `Failure handling`

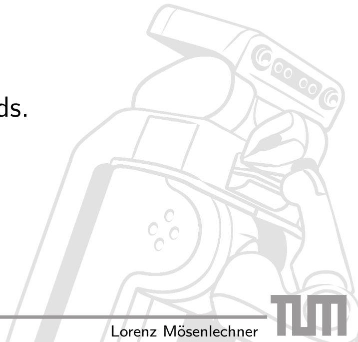
Requirements for the Language

- `Expressive`
- `Easy to use`

$\Rightarrow$ CPL is a Domain Specific Language fulfilling these requirements

Overview of the CRAM Language

- Implemented in Common Lisp.
- Compiles down to multithreaded programs.
- Programs are in native machine code.
- Provides control structures for parallel and sequential evaluation of expressions.
- Reactive control programs.
- Exception handling, also across threads.



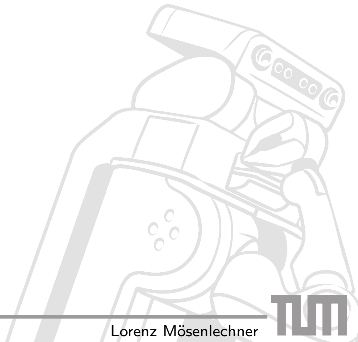
Example: Picking up an object

Example

```
(let* ((obj-pose (find-object obj))
       (pre-grasp-pos (calculate-pre-grasp obj-pose))
       (grasp-vector (cl-transforms:make-3d-vector 0 0 -0.1))
       (lift-vector (cl-transforms:make-3d-vector 0 0 0.1)))
  (open-gripper side)
  (take-collision-map)
  (with-failure-handling
      ((no-ik-solution (e)
         (move-to-different-place)
         (retry))
       (link-in-collision (e)
         (setf pre-grasp-pos (new-pre-grasp))
         (retry))
       (trajectory-controller-failed (e)
         (retry)))
     (move-arm-to-point side pre-grasp-pos)))
   ...)
```

Overview CRAM Language

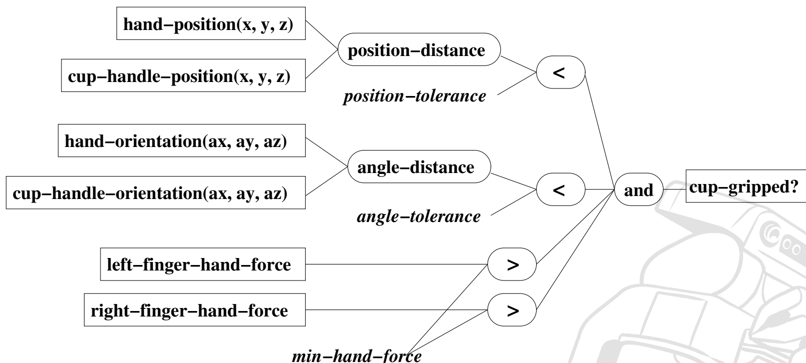
- Fluents
- Sequential evaluation
- Parallel evaluation
- Exceptions and failure handling
- Task suspension



Fluents

- Fluents are objects that contain a value and provide synchronized access.
- Create with `(make-fluent :name 'fl :value 1)`
- Wait (block thread) until a fluent becomes true: (wait-for fl)
- Execute whenever a fluent becomes true: (whenever fl)
- Can be combined to fluent networks that update their value when one fluent changes its value. (wait-for ($>$ x 20))

Fluent networks



CRAM Control Flow

Sequential Evaluation

- Execute expressions sequentially:

```
(seq
  (do a)
  (do b)
```

CRAM Control Flow

Sequential Evaluation

- `Execute expressions sequentially:`

```
(seq
  (do a)
  (do b)
```

- `Execute expressions sequentially until one succeeds:`

```
(try-in-order
  (do a)
  (do b)
```

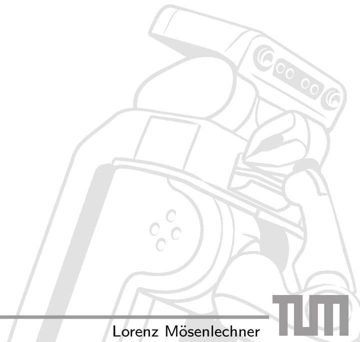
CRAM Control Flow

Parallel Evaluation

- `Execute in parallel, succeed when` **all** `succeed, fail if` **one** `fails:` `(par ...)`

Examples:

```
(par
  (open-right-gripper)
  (open-left-gripper)
```



CRAM Control Flow

Parallel Evaluation

- `Execute in parallel, succeed when` **all** `succeed, fail if` **one** `fails:` `(par ...)`
- `Execute in parallel, succeed when` **one** `succeeds, fail if` **one** `fails:` `(pursue ...)`

Examples:

```
(par
  (open-right-gripper)
  (open-left-gripper)
```

```
(pursue
  (wait-for (< (distance robot p) 5))
  (update-nav-cmd x)
```

CRAM Control Flow

Parallel Evaluation

- `Execute in parallel, succeed when` **all** `succeed, fail if` **one** `fails:` `(par ...)`
- `Execute in parallel, succeed when` **one** `succeeds, fail if` **one** `fails:` `(pursue ...)`
- `Try in parallel, succeed when` **one** `succeeds, fail if` **all** `fail:` `(try-all ...)`

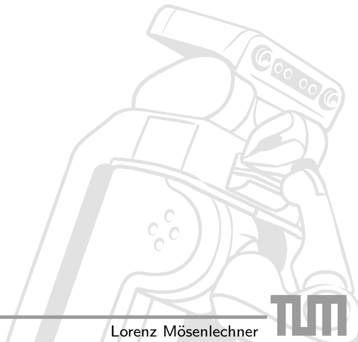
Examples:

```
(par
  (open-right-gripper)
  (open-left-gripper)
```

```
(pursue
  (wait-for (< (distance robot p) 5))
  (update-nav-cmd x)
```

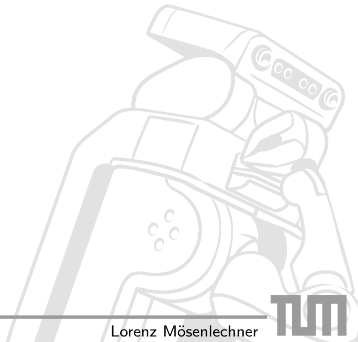
Failure Handling

- Create exception class:
  (define-condition nav-failed (plan-error) ())



Failure Handling

- Create exception class:
  (define-condition nav-failed (plan-error) ())
- Throw exception: (fail 'nav-failed)



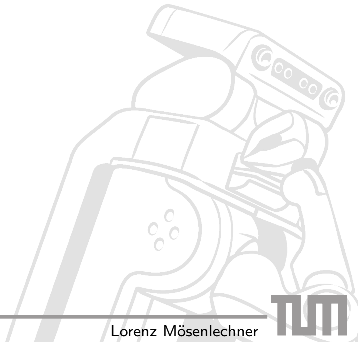
Failure Handling

- Create exception class:

```
(define-condition nav-failed (plan-error) ())
```

- Throw exception: (fail 'nav-failed)
- Handle exceptions:

```
(with-failure-handling
    ((obj-not-reachable (e)
       (move-to-better-location)
       (retry)))
  (pursue
    (seq
     (sleep timeout)
     (fail timeout))
    (grasp-obj obj)))
```



Failure Handling

- Create exception class:

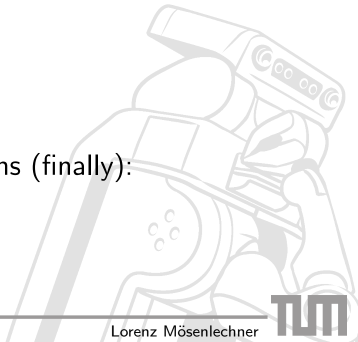
```
(define-condition nav-failed (plan-error) ())
```

- Throw exception: (fail 'nav-failed)
- Handle exceptions:

```
(with-failure-handling
    ((obj-not-reachable (e)
        (move-to-better-location)
        (retry)))
  (pursue
    (seq
     (sleep timeout)
     (fail timeout))
    (grasp-obj obj))
```

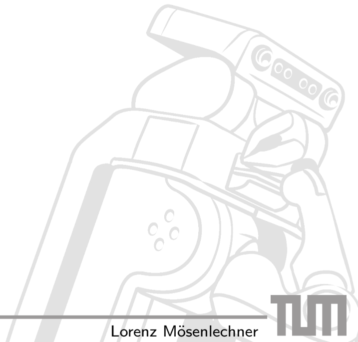
- Execute expressions even on exceptions (finally):

```
(unwind-protect
      (grasp-object)
   (move-arms-to-save-position))
```



Tagging, Suspension, Protection forms

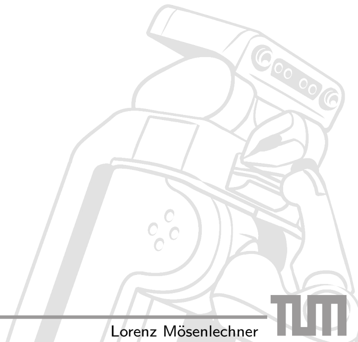
- Name sub-expressions and bind them to variables in the current lexical scope:
  (:**tag** var
    (move-to x y)



Tagging, Suspension, Protection forms

- Name sub-expressions and bind them to variables in the current lexical scope: (:tag var ...)
- Execute expressions with a parallel task suspended:

```
(pursue
  (whenever c
    (with-task-suspended nav
      ...))
  (:tag nav
    (move-to x y)
```



Tagging, Suspension, Protection forms

- Name sub-expressions and bind them to variables in the current lexical scope: (:tag var ...)
- Execute expressions with a parallel task suspended:

```
(pursue
  (whenever c
    (with-task-suspended nav
      ...))
  (:tag nav
    (move-to x y)
```

- Execute code just before a task is suspended:

```
(suspend-protect
    (move-to x y)
  (stop-motors)
```