The Cram Plan Language — Plan-based Control of Autonomous Robots

Lorenz Mösenlechner (moesenle@in.tum.de)

Intelligent Autonomous Systems Group
Technische Universität München

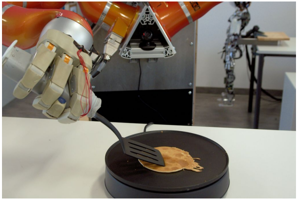
November 4, 2010

Outline

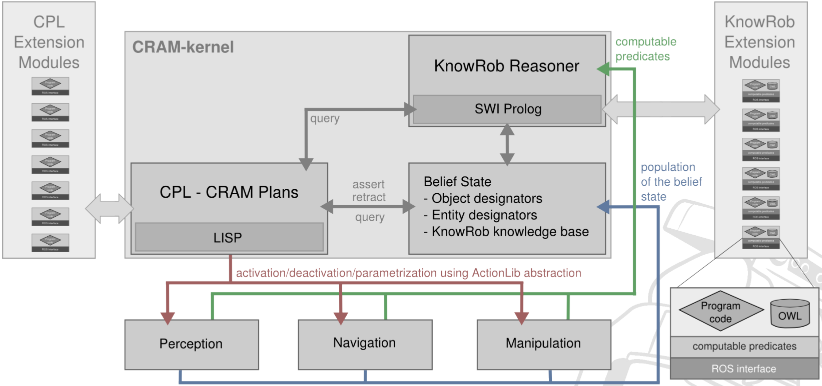
1. Motivation
2. The Language
3. Reasoning about Plan Execution
4. Outlook
5. Lab session

Motivation

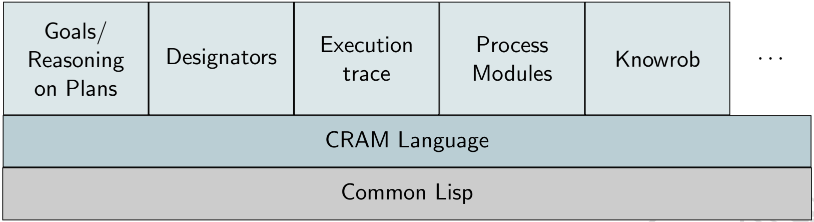
- Goal: perform complex activity in a human household
- Implementing reliable robot control programs is hard
- Complex failure handling is required
- Tasks synchronization, parallel execution, resource management, ...



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`

⇒ CPL is a Domain Specific Language fulfilling these requirements

Outline

1. Motivation
2. The Language
3. Reasoning about Plan Execution
4. Outlook
5. Lab session

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)))
   ...)
```

Basic Lisp Syntax

- Parenthesis around complete expression:
  foo(bar, 123) $\Rightarrow$ (foo bar 123)
- Prefix notation for operators:
  $1+2+3+4+5 \;\Rightarrow$ (+ 1 2 3 4 5)
- Expressions in "blocks":

```
with open("foo.txt", "w") as f:
  f.write("bar\n")
```

$$\Downarrow$$

```
(with-open-file (f "foo.txt" :direction :output)
  (format f "bar~%"))
```

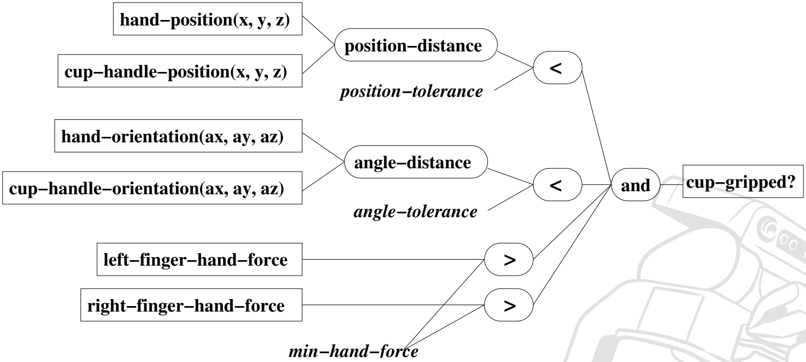
Overview CRAM Language

- 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:

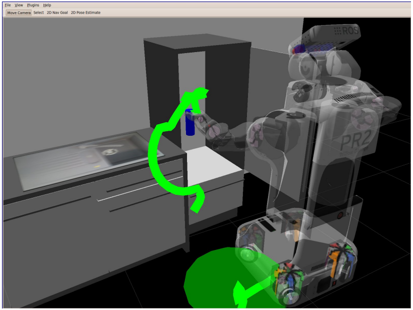
```
(pursue
  (whenever c
    (with-task-suspended nav
      ...))
  (:tag nav
    (move-to x y)
```

- Execute code just before a task is suspended:

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

Outline

Reasoning based on Execution Traces



- Why did you leave the cup on the table while clearing it?
- Where did you stand while performing a task?
- What did you see?
- How did you move?
- How did you move the arm while grasping the bottle?
- ...

Our approach

1. Record execution trace
2. Provide an interface to the execution trace through a first-order representation

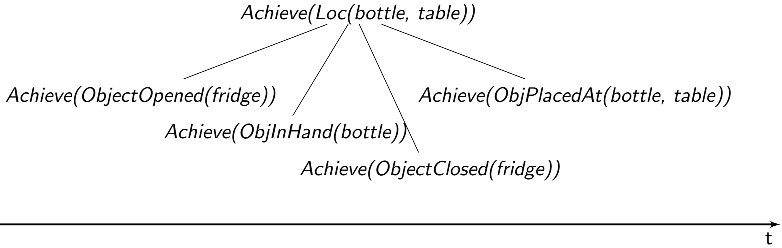
Our approach

1. Record execution trace
   - Belief state
   - State of plan execution, tasks, activation, deactivation, results
2. Provide an interface to the execution trace through a first-order representation

Our approach

1. Record execution trace
   - Belief state
   - State of plan execution, tasks, activation, deactivation, results
2. Provide an interface to the execution trace through a first-order representation
   - Symbolic annotations of plans
   - Causal relations through plan hierarchy
   - Symbolic representation of objects in plans

Recording of Execution Trace



Recording of Execution Trace

Achieve(Loc(bottle, table))

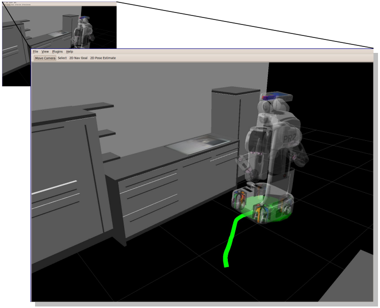
Achieve(ObjectOpened(fridge))

Achieve(ObjInHand(bottle))

Achieve(ObjectClosed(fridge))

Achieve(ObjPlacedAt(bottle, table))

t



Action:

- Move to fridge

Log:

- Achieve(Loc(bottle,table)) running
- Achieve(Loc(Robot, l)) running
- Trajectory of robot
- ...

Recording of Execution Trace

Achieve(Loc(bottle, table))

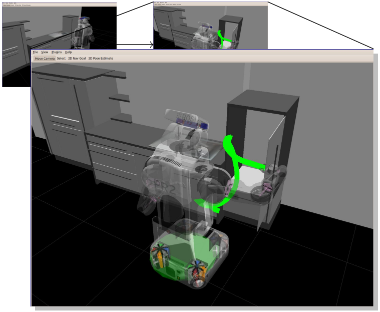
Achieve(ObjectOpened(fridge))

Achieve(ObjInHand(bottle))

Achieve(ObjectClosed(fridge))

Achieve(ObjPlacedAt(bottle, table))

t



Action:

- Open fridge

Log:

- Achieve(Loc(Robot, l)) succeeded
- Achieve(ObjectOpened(fridge)) running
- Trajectory of arm
- ...

Recording of Execution Trace

Achieve(Loc(bottle, table))

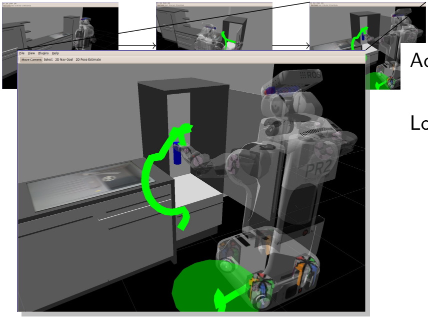
Achieve(ObjectOpened(fridge))

Achieve(ObjInHand(bottle))

Achieve(ObjectClosed(fridge))

Achieve(ObjPlacedAt(bottle, table))

t



Action:

- Grasp the bottle

Log:

- Achieve(ObjectOpended(fridge)) succeeded
- Achieve(ObjInHand(bottle)) running
- Perceived properties of bottle (object designator)
- ...

Recording of Execution Trace

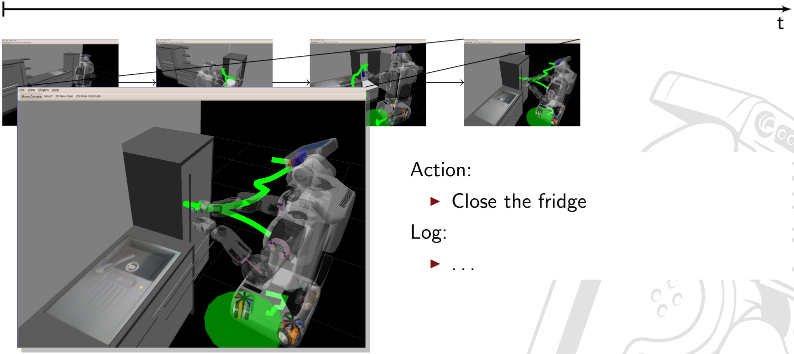
Achieve(Loc(bottle, table))

Achieve(ObjectOpened(fridge))

Achieve(ObjInHand(bottle))

Achieve(ObjPlacedAt(bottle, table))

Achieve(ObjectClosed(fridge))



Action:

- Close the fridge

Log:

- ...

Recording of Execution Trace

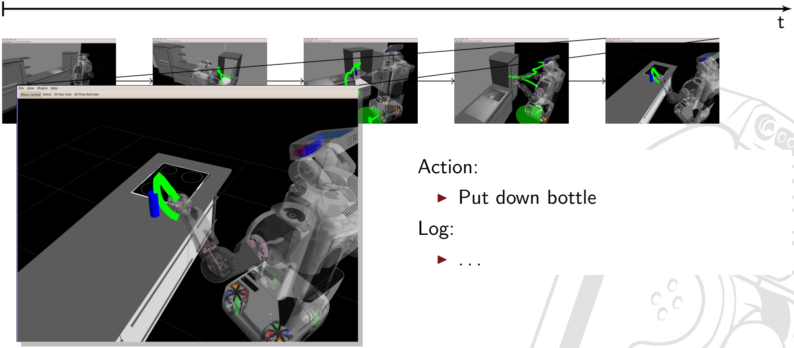
Achieve(Loc(bottle, table))

Achieve(ObjectOpened(fridge))

Achieve(ObjInHand(bottle))

Achieve(ObjectClosed(fridge))

Achieve(ObjPlacedAt(bottle, table))



Action:

- Put down bottle

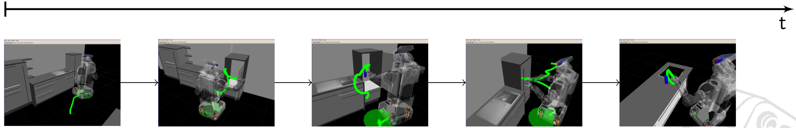
Log:

- ...

Recording of Execution Trace

Achieve(Loc(bottle, table))

- *Achieve(ObjectOpened(fridge))*
- *Achieve(ObjInHand(bottle))*
- *Achieve(ObjectClosed(fridge))*
- *Achieve(ObjPlacedAt(bottle, table))*



Goals and Reasoning

- Reasoning about programs is complex.
- We annotate only the interesting parts to infer the semantics of a plan.
  - `achieve`: Make true if not already true
  - `perceive`: Try to find object and return a information about it
  - `at-location`: Execute code at a specific location

Annotating Plans

- Achieve goals:

```
(def-goal (achieve (loc ?obj ?loc))
  (achieve `(object-in-hand ,?obj :right))
  (achieve `(object-placed-at ,?obj ,?loc)))
```

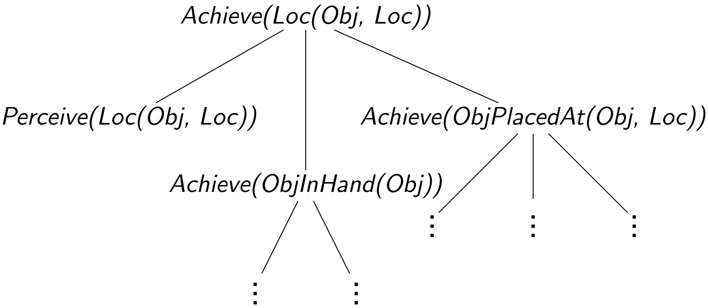
- Perceive:

```
(def-goal (perceive ?obj-name)
  (find-obj ?obj-name))
```

- At-location:

```
(at-location (?loc)
  (achieve `(object-in-hand ,?obj :right)))
```

Plan Representation



Predicates for reasoning on execution traces

`(task ?tsk)`	*?tsk* is a task on the interpretation stack.
`(task-goal ?tsk ?goal)`	Unifies the goal of the task.
`(task-start ?tsk ?t)`	Unifies the start time of the task.
`(task-end ?tsk ?t)`	Unifies the end time of the task.
`(subtask ?tsk ?subtsk)`	Asserts that *subtask* is a direct subtask of *task*.
`(subtask+ ?tsk ?subtsk)`	Assets that *subtask* is a subtask of *task*.
`(task-outcome ?tsk ?status)`	Unifies the final status of a task (Failed, Done or Evaporated).
`(task-result ?tsk ?result)`	Unifies the result of a task.

Outline

1. Motivation
2. The Language
3. Reasoning about Plan Execution
4. Outlook
5. Lab session

More CRAM Modules

- Designators (symbolic descriptions of objects)
- Process modules
- Reasoning about locations and inference of locations
- On-line reasoning in the execution trace
- Knowrob (tomorrow)
- ...

Outline

1. Motivation
2. The Language
3. Reasoning about Plan Execution
4. Outlook
5. Lab session

Tutorial setup

- Make sure that Emacs is installed
- Make sure that `ros-cturtle-roslisp-common` and the `cram_pl` stack is installed.
- All tutorial-related files are in the cram_tutorials package:

  `roscd cram_tutorials`
- You can run a LISP REPL with:

  `rosrun cram_emacs_repl repl`

The Lisp REPL

- REPL = Read-Eval-Print-Loop
- Interactive development environment
- Inspection of variables

Important commands

- `Ctrl-up and Ctrl-down for moving in history`
- `Change package with (in-package :roslisp)`
- `When in debugger, press number of restart Abort to abort debugging`
- `Enter in debugger opens stack frames or calls the inspector`
- `''l'' to go back in inspector`
- `''q'' to exit inspector`

Interactive roslisp (Move base)

Roslisp

```
;; Load actionlib_lisp
CL-USER> (ros-load:load-system "actionlib_lisp" :actionlib)

;; Create ros node
CL-USER> (roslisp:start-ros-node "move_base_lisp_client")

;; Instantiate action client
CL-USER> (defvar *move-base-client*
  (actionlib:make-action-client
    "/move_base" "move_base_msgs/MoveBaseAction"))
```

Interactive roslisp (Move base)

Roslisp

```
;; Send action goal and wait
CL-USER> (actionlib:call-goal *move-base-client*
  (make-msg "move_base_msgs/MoveBaseGoal"
    (frame_id header) "base_link"
    (x position pose) 1.0
    (w orientation pose) 1.0))
```