Lisp Tutorial

Jan Winkler

Institute for Artificial Intelligence
Universität Bremen

June 2, 2014
Summer Term

Today's Schedule

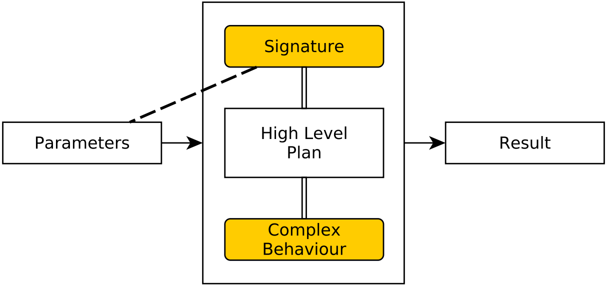
- High Level Plan Concepts
- The CRAM Plan Language
  - High Level Plans
  - Goals
  - Process Modules
  - Top Level Plans
- Language Constructs
  - `with-failure-handling` and `fail`
  - `with-policy`
  - `achieve`
  - `perform`
  - `pursue`, `par`, `seq`, `try-all`, `try-in-order`
- CRAM Usage Examples
- Insight Appendage from last lecture

High Level Plan Concepts

What is a High Level Plan?

Solves a **Task** that is handled as a **Black Box**.

High Level Plans describe **complex behaviour**, **encapsulating** the complexity and making the task available as a **module** (think building blocks) using a fixed **parameter signature** and returning a **result**.



High Level Plan Concepts

Examples of High Level Plans

Common tasks to encapsulate in robotics are:

- Picking up objects
- Placing objects
- Perceiving the environment
- Retrieve knowledge from the web and extract information from it
- Navigate to a given location
- Open a drawer

This is compliant with programming paradigms encapsulating functionality into **functions**.

CRAM Plan Language

What is CRAM?

CRAM is a high level **plan engine**

Its main purposes are:

- Management of a plan library
- Interfacing between abstract high level plans and robot specific control modules
- Supply developers with meaningful language constructs
- Encapsulate behaviour

CRAM Plan Language
High Level Plans

Syntactic description of high level plans in CRAM:

Example of a High Level Plan

```
(def-cram-function object-in-hand (?obj)
  ...)
```

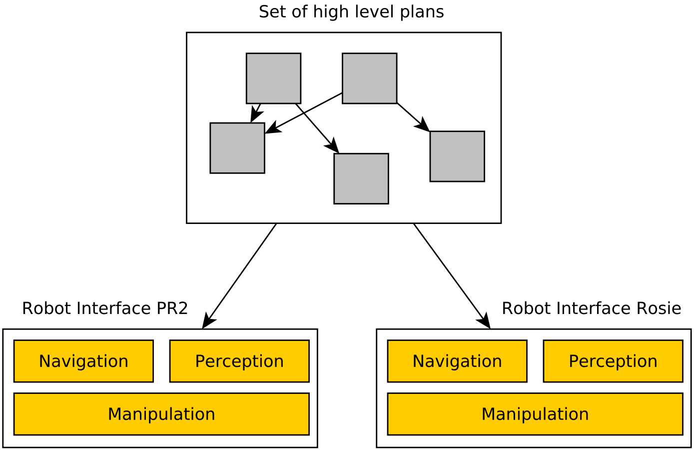
Every high level plan is accompanied by an abstract, achievable goal:

Example of a Goal Declaration

```
(declare-goal object-in-hand (object)
  "Orders the robot to pick up an object."
  (roslisp:ros-info (plan-library) "Pick up object ~a" object))
```

CRAM Plan Language

High Level Plans



CRAM Plan Language

Goals

Goals are **achievable states**.

A state is achieved, when the robot **believes it is true**.

Abstract examples (so-called *occasions*):

- *The object is in the hand.*
- *The object is on the table.*
- *I have perceived all objects in the table.*
- *I am in front of the table.*
- *The door is open.*

When **achieving goals** in CRAM, the command is formulated as

Make occasion ?O true

where ?O is a given goal.

CRAM Plan Language

Process Modules

Process Modules supply the CRAM system with **low level functionality**.

This includes:

- Motor control
- Call of actual ROS services
- Triggering of perception routines
- Request of knowledge from an external knowledge base
- ...

They are **interchangeable** – All process modules have the **same interface**, allowing easy replacement (e.g. for a different robot).

CRAM Plan Language

Top Level Plans

Their **Nature**:

- **Similar in Structure** to High Level Plans, but **Conceptionally Different**
- No explicit `performs`: Does not trigger Primitive Actions
- Enduser-Programmable (i.e. *"Do what I mean, not what I say."*)

Their **Purpose**:

- Parameterizes the overall **Goal**
- Initializes the **Environment**
- Handles **Unanticipated Exceptions/Failures**

CRAM Plan Language

Top Level Plans

Example:

Example of a Top-Level Plan

```
(def-top-level-cram-function my-toplevel-plan ()
  (initialize-environment) ;; Initializes important variables
  (with-process-modules ;; Starts environment with
                        ;; necessary process modules active
    (with-designators ((cup (object `((type cup)
                                       (color red)))))
      (achieve `(object-in-hand ,cup)))))
```

CRAM Plan Language

Top Level Plans

Example with Failure Handling:

Example of a Top-Level Plan with Failure Handling

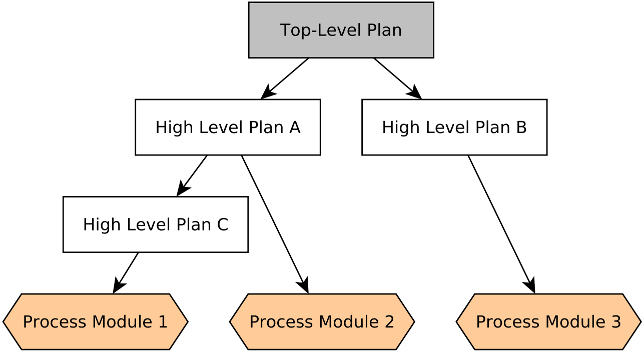
```
(def-top-level-cram-function my-toplevel-plan ()
  (initialize-environment) ;; Initializes important variables
  (with-process-modules ;; Starts environment with
                        ;; necessary process modules active
    (with-designators ((cup (object `((type cup)
                                      (color red)))))
      (with-retry-counters ((retry-count 3))
        (with-failure-handling
            (((or object-not-found manipulation-failure) (f)
               (do-retry retry-count ;; Only retry n times
                 (retry)))) ;; Retry on failure
          (achieve `(object-in-hand ,cup)))))))
```

Failure handling code can be up 80% of the code you write.

Much more than goes right can go wrong!

CRAM Plan Language
Top Level Plans

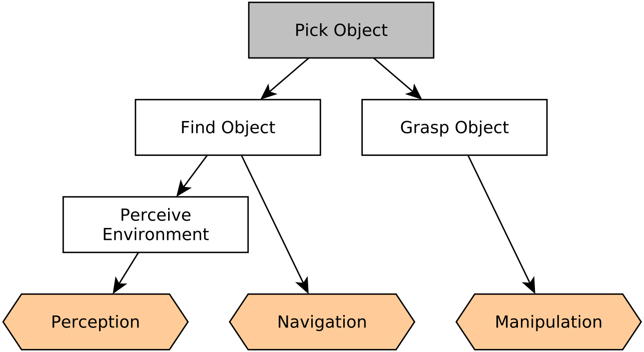
Top Level Plans control High Level Plans,
High Level Plans control other High Level Plans and Process Modules



CRAM Plan Language

Top Level Plans (with more life)

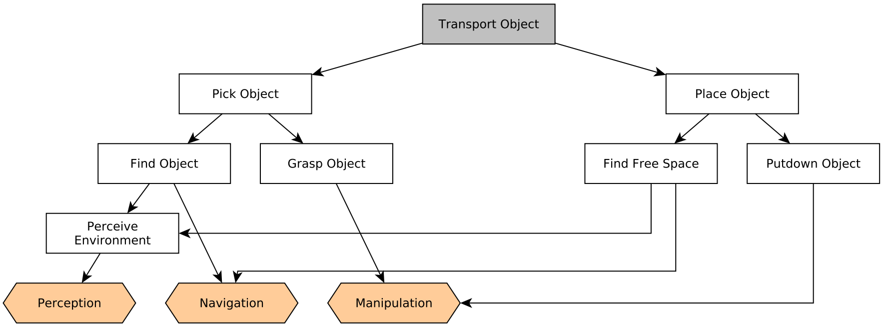
Top Level Plans control High Level Plans,
High Level Plans control other High Level Plans and Process Modules



CRAM Plan Language

Simplified Transport Object Top Level Plan (Nested Plans)

Top Level Plans can be High Level Plans $\Longrightarrow$ Nesting of Plans



Language Constructs

Purpose of Specialized Language Constructs

Specialized Language Constructs allow:

- Easily readable code (constructs have **semantic meaning**)
- Faster programming (complex functionality available transparently)
- **Encapsulation** of proven-to-work-code (central mechanisms)
- Reasoning about plans during runtime
  - Plus for Lisp: Analysis of plan code while executing it (e.g. **look-ahead**, **action projection**)
  - Dynamic reparameterization

Language Constructs
Failure Handling

During plan performance, all kinds of things can go wrong. Modules must signal failures (**type**, detailed information about **what** went wrong, **where** it went wrong, and possibly **why**).

Failure signals can happen in an **anticipated** or **unanticipated** manner.

Anticipated failures can be caught and (mostly) handled well.

Unanticipated failures must be detected and handled using heuristics.

Language Constructs

In CRAM: `with-failure-handling`, `fail`

Wrapping code in failure handling blocks:

Example of Failure Handling code

```
(with-failure-handling
    (((or object-not-found manipulation-failure) (f)
       (retry))) ;; Retry on failure
  (achieve `(object-in-hand ,cup)))
```

Conceptually similar to `try..catch..finally` language constructs in Java, C++, ...

Has an implicit `rethrow`-mechanism when failure did not succeed in `retry` (send failure one level up).

Language Constructs
Policies

While a plan (or any kind of code) is executed, a set of **contextual constraints** must be considered.

Policies describe this kind of constraints:

- Hold the coffee mug upright while moving it
- Keep away at least half a meter from the cliff
- Whatever you do, don't place objects on the hot stove
- Make sure that you always have the baby in view

Policies are usually not coded for a **specific situation**, but are **general** and should be **applyable to arbitrary situations/plans**.

Language Constructs

In CRAM: `with-policy`

Defining a policy: define-policy

```
(define-policy timeout-policy (time)
  "Policy that executes the given ‘body’ code until either that
  code finishes or a given time ‘time’ in seconds has passed. If
  the timeout is reached before the end of ‘body’, the
  ‘policy-check-condition-met’ condition is signalled."
  (:check (sleep* time) t))
```

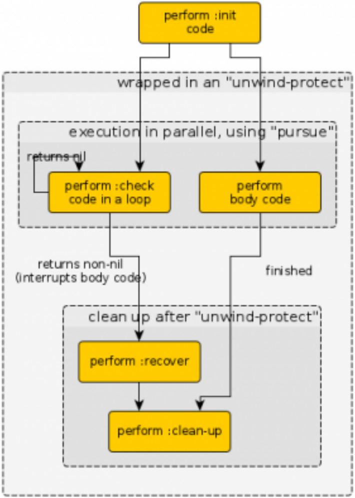
Using a policy: with-policy

```
(with-policy timeout-policy (5)
  (code-that-must-be-timeout-interrupted))
```

See also: `http://cram-system.org/doc/plans`

Language Constructs

In CRAM: `with-policy`



Language Constructs

Making Occasions True – Changing the World according to Belief

Again: **Goals**

Achieving a goal means changing the robot's belief about the world.

Potential pitfalls:

- Faulty belief (wrong deduction of events in the worls resulting in wrong conclusions)
- Faulty perception (misinterpretation on a sensor level leads to wrong information)

Language Constructs

In CRAM: `achieve`

Trying to achieve a goal `object-in-hand` with object of `type cup`:

Achieving object-in-hand

```
(with-designators ((obj-cup (object `((type cup)))))
  (achieve `(object-in-hand ,obj-cup)))
```

If the robot *believes* that it has the object `obj-cup` already in one of its hands, nothing happens (*implicit success*).

Otherwise, perform the high level plan `object-in-hand`.

Language Constructs

Action Primitives – Acting in the Real World

Primitive Actions can include:

- Moving the robot's base
- Triggering perception
- Moving an arm to a specific pose
- Moving the robot's head
- Switching on a flashlight
- Requesting information from a knowledge base
- . . .

A request must be formulated such that the **responsible entity** recognizes it from the action queue.

Language Constructs

In CRAM: perform

Action primitive performance using **action designators**:

Performing an action

```
(def-cram-function object-in-hand (?obj)
  (with-designators
      ((action-grasp (action `((type trajectory)
                               (to grasp)
                               (obj ,?obj)))))
    (perform action-grasp)))
```

All process modules are requested to check whether they can perform this action according to the **vague description**. If one fails, the next is tried.

Language Constructs

Order of Execution, Implicit Success and Failure

Tasks can be performed in different orderings, and dependent on the success/failure of other tasks.

Common **ordering manners**:

- Sequential execution (perform one by one)
- Parallel execution (perform *n* tasks at once, waiting for all to finish)

Common **result dependencies**:

- Try all until one succeeds (parallel and sequential order)
- Perform all until one finishes (only makes sense in parallel order)

Combining them:

- Try all in parallel until one succeeds (vaporizing the rest)
- Try all in order until one succeeds (then stopping execution)

Language Constructs

In CRAM: `pursue`, `par`, `seq`, `try-all`, `try-in-order`

Available constructs in CRAM:

- `seq`: Perform forms in sequence
  (same as `progn`, but has semantic meaning!)
- `par`: Perform forms in parallel, return when all have returned
- `pursue`: Perform forms in parallel until one finishes, vaporizing the rest (return the returned form's value)
- `try-in-order`: Try forms in order until one succeeds, then return
- `try-all`: Try forms in parallel, return when one succeeds

For `try-in-order` and `try-all`:

When one succeeds, result is success; when all fail, result is failure.

CRAM Usage Examples
Past Projects and Ongoing Research

Currently actively used in **at least four European Research Projects**

Central plan component in the **SUTURO student project** (`http://www.suturo.de/`)

Actively developed **open source** by six full-time researchers in Bremen, and more in other universitites around the world.

Find past research papers here:
`http://cram-system.org/publications`

Overall information about CRAM:
`http://cram-system.org/`

Insight Appendage

Corrections to the NXT Lisp Template Code

When anyone encounters problems with the NXT Lisp Template (stop at any point if it starts working),

1. Check out the repository again (`roscd nxt_lisp && git pull`)
2. `rospack profile && source  /.bashrc`
3. Ask Jan

There were problems that are not always reproducable, don't hesitate to ask.

Project Description

Preparational Document for your NXT Lisp Project

Prepare a one-page document about what your project is, containing:

- Your group's members names and e-mail addresses
- The task you will solve
- A short description of your approach
- Possibly a catchy name for your project/robot

This has two purposes:

- Make clear to yourself (and your group) **what your goal is**
- Let the supervisor understand better **what to look for** in your work

Assembly And Programming

The rest of today's course will be dedicated for assembly and programming of your projects.

Be sure to check the `project-sheet.tex` file in StudIP. You'll need to fill in a few details and **hand it in until next week** via e-mail.

Important: Get a group number today!