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

Lorenz Mösenlechner (moesenle@in.tum.de)
Intelligent Autonomous Systems Group
Technische Universität München

14 June 2012
Outline

1. Motivation

2. The Language

3. Reasoning about Plan Execution

4. Plan Parameterization and Inference of Locations

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

- CPL - CRAM Plans
- KnowRob Reasoner
- SWI Prolog
- CRAM-kernel
- Perception
- Navigation
- Manipulation

Motivation
CPL
Reasoning about Plan Execution
Plan Parameterization
Lab session

CPL Extension Modules

KnowRob Extension Modules

Computable predicates
Population of the belief state

Activation/deactivation/parametrization using ActionLib abstraction
The CRAM Core

Goals/Reasoning on Plans | Designators | Execution trace | Process Modules | Knowrob | …

CRAM Language

Common Lisp
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. Plan Parameterization and Inference of Locations

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:
  \[
  \text{foo(bar, 123)} \Rightarrow (\text{foo bar 123})
  \]

- Prefix notation for operators:
  \[
  1 + 2 + 3 + 4 + 5 \Rightarrow (+ 1 2 3 4 5)
  \]

- Expressions in “blocks”:
  \[
  \begin{align*}
  \text{with open("foo.txt", "w") as f:} \\
  &\text{f.write("bar\n")}
  \end{align*}
  \]
  \[
  \Rightarrow
  \begin{align*}
  \text{(with-open-file (f "foo.txt" :direction :output)} \\
  &\text{(format f "bar~%")})
  \end{align*}
  \]
Functions in Common Lisp and CRAM

- Define common lisp functions with

  \[
  \text{(defun <function-name> (<parameters*>)
  \text{(lisp-expressions*)})}
  \]

  **Example:**

  \[
  \text{(defun factorial (n)
  \text{(declare (type fixnum n))
  \text{(if (> n 1)
  \text{(* n (factorial (1- n)))
  1))}}}
  \]

- Define CRAM “functions” with `def-plan` and `def-top-level-plan` to make them visible in the task tree.

- Lisp functions can be called from CRAM.
Control structures in Common Lisp

Conditional evaluation

- One “if” and one “else” expression:
  
  ```lisp
  (if (> n 0)
      (foo)
      (bar))
  ```

- Only an “if” or only an “else” block:
  
  ```lisp
  (when (> n 0)
        (foo))
  (unless (> n 0)
           (bar))
  ```

- Complex conditionals:
  
  ```lisp
  (cond ((> n 0)
        (foo))
        (t
         (bar)))
  ```
Control structures in Common Lisp

Return values

- The return value of the last expression that is evaluated is used as return value.
- `prog1` returns the first return value, `prog2` the second, `progn` the last.
- Multiple return values with `values`.

```lisp
(defun foo () (values 1 2 3))
(multiple-value-bind (a b c) (foo)
  (format t "~a~a~a~%" a b c))
```
Variables

- Create local, lexically scoped variables with `let` or `let*`.
  
  ```
  (let ((a 1)
        (b 2))
...
  )
  ```

- Create global, dynamically scoped variables with `defvar` and `defparameter`.
  
  ```
  (defvar *foo* 1)
  (defparameter *bar* 2)
  ```
Data

Built-in data types

► Simple data types: symbols, strings, numbers:

'foo : bar "baz" 123 123.45 123.456 d0

► Built-in support for lists:

(list 1 2 3 4) '(1 2 3 4) '(1 2 3 , a , @b c)
(first (list 1 2 3 4)) (car (list 1 2 3 4))
(nth 5 some-list) (elt some-list 5)

► Built-in support for arrays:

(let ((matrix (make-array '(2 2) :element-type 'double-float)))
  (declare (type simple-array matrix))
  (aref matrix 0 1))

► Built-in support for hash-tables:

(let ((cache (make-hash-table)))
  (gethash 'foo cache))
Data

User-defined data types

- **Structs with defstruct:**

  ```lisp
  (defstruct 2d-vector
    (x 0.0d0 :type double-float)
    (y 0.0d0 :type double-float))
  
  (let ((vector (make-2d-vector :x 2.0d0)))
    (2d-vector-x vector))
  
  Classes (support for inheritance) with defclass:

  ```lisp
  (defclass parent ()
    ((parent-slot :reader parent-slot
      :initform nil
      :initarg :parent-slot)))

  (defclass child (parent) ()
    (make-instance 'child :parent-slot 5)
  ```
Side effects

- Lisp is not a purely functional language, it supports side effects.
- Set values with `setq` or `setf`. `setf` is more general.
  
  ```lisp
  (setf foo 123)
  (setf (aref matrix 0 1) 0.1d0)
  (setf (gethash :foo cache) 5)
  ```

- Various forms for in-place updates: `incf`, `decf`, `push`, `pushnew`, `pop`, ...
Function objects and mappers

- Function pointer with `'#': `'#factorial`
- Lambda functions:
  \[
  (\text{lambda } (x) (+ x 10))
  \]
- Calling a function pointer:
  \[
  (\text{funcall } '#factorial 5)
  \]
- Apply a function to each element of a list and get the list of return values:
  \[
  (\text{mapcar } '#factorial (1 2 3 4 5))
  (= \text{mapcar } (\text{lambda } (x) (+ x 10)) (1 2 3 4 5))
  \]
Common Lisp and SBCL provide much more

- Macros
- Object oriented programming (defmethod, defclass)
- File IO
- The loop macro
- Sockets
- Multithreading
- Foreign Function Interface
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)
Motivation CPL Reasoning about Plan Execution Plan Parameterization Lab session

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:
  
  ```cpl
  (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:
  
  \[
  (:\text{tag} \text{ var} \\
  (\text{move-to} \ x \ y))
  \]
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))

CPL
Lorenz Mösenlechner
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

1. Motivation

2. The Language

3. Reasoning about Plan Execution

4. Plan Parameterization and Inference of Locations

5. Lab session
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

Achieve(Loc(bottle, table))

Achieve(ObjectOpened(fridge))

Achieve(ObjInHand(bottle))

Achieve(ObjectClosed(fridge))

Achieve(ObjPlacedAt(bottle, table))
Recording of Execution Trace

Achieve(Loc(bottle, table))

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

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))

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))

Action:
- Grasp the bottle

Log:
- Achieve(ObjectOpended(fridge)) succeeded
- Achieve(ObjInHand(bottle)) running
- Perceived properties of bottle (object designator)
- . . .
Recording of Execution Trace

- \text{Achieve}(\text{Loc}(\text{bottle, table}))
- \text{Achieve}(\text{ObjectOpened}(\text{fridge}))
- \text{Achieve}(\text{ObjInHand}(\text{bottle}))
- \text{Achieve}(\text{ObjPlacedAt}(\text{bottle, table}))
- \text{Achieve}(\text{ObjectClosed}(\text{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

\[
\begin{align*}
\text{Achieve}(\text{Loc}(\text{bottle, table})) & \\
\text{Achieve}(\text{ObjectOpened}(\text{fridge})) & \\
\text{Achieve}(\text{ObjInHand}(\text{bottle})) & \\
\text{Achieve}(\text{ObjectClosed}(\text{fridge})) & \\
\text{Achieve}(\text{ObjPlacedAt}(\text{bottle, table})) & \\
\end{align*}
\]
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
Reasoning about Plan Execution

- The plate is on the cupboard at the beginning.
- The robot placed a plate on the table.
- The plate is supported by the table.
- The robot cannot see the plate on the table.

Time: 216.41 s
Reasoning about Plan Execution

- Holds(Loc(Plate, on(Cupboard)), At(0s))

- The robot placed a plate on the table
- The plate is supported by the table.
- The robot cannot see the plate on the table.

Time 216.41 s
Reasoning about Plan Execution

- Holds(Loc(Plate, on(Cupboard)), At(0s))

- Holds(Loc(Plate, on(Table)), At(TaskEnd(tsk)))
  - Task(tsk)
  - TaskGoal(tsk, Achieve(Loc(Plate, on(Table))))

- The plate is supported by the table.
- The robot cannot see the plate on the table.

Time: 216.41 s
Motivation
CPL Reasoning about Plan Execution
Plan Parameterization Lab session

Reasoning about Plan Execution

- Holds(Loc(Plate, on(Cupboard)), At(0s))

- Holds(Loc(Plate, on(Table)), At(TaskEnd(tsk)))
  \( \land \) Task(tsk)
  \( \land \) TaskGoal(tsk, Achieve(Loc(Plate, on(Table))))

- Holds(Supporting(Table, Plate), At(t))
- \( \neg \)Holds(ObjectVisible(Plate), At(t))

Time 216.41 s
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

Achieve(Loc(Obj, Loc))

Perceive(Loc(Obj, Loc))

Achieve(ObjPlacedAt(Obj, Loc))

Achieve(ObjInHand(Obj))
Predicates for reasoning on execution traces

(task ?tsk)  

(task-goal ?tsk ?goal)  
(task-start ?tsk ?t)  
(task-end ?tsk ?t)  
(subtask ?tsk ?subtsk)  
(subtask+ ?tsk ?subtsk)  
(task-outcome ?tsk ?status)  
(task-result ?tsk ?result)  

?tsk is a task on the interpretation stack.  
Unifies the goal of the task.  
Unifies the start time of the task.  
Unifies the end time of the task.  
Asserts that subtask is a direct subtask of task.  
Assets that subtask is a subtask of task.  
Unifies the final status of a task (Failed, Done or Evaporated).  
Unifies the result of a task.
Outline

1. Motivation

2. The Language

3. Reasoning about Plan Execution

4. Plan Parameterization and Inference of Locations

5. Lab session
Plan parameterizations: Designators

- Symbolic descriptions of Objects, Locations and Actions
- Examples:
  - (location (on table) (reachable-by Rosie))
  - (object (type cup))
  - (action (to reach) (the object (name Cup1)))
- Example plan:

  (with-designators
   ((cup-loc (location ‘(on counter-top)))
    (cup (object ‘(type cup) (at ,cup-loc)))
    (pick-up-loc ‘(location (to reach) (obj cup))))
   (at-location pick-up-loc
    (achieve ‘(obj-grasped ,cup))))

CPL

Lorenz Mösenlechner
Example: Location to put-down pancake mix

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making)))))

Concepts

- Generative model
- Stability reasoning
- Reachability reasoning
- Visibility reasoning and perspective taking
- Light-weight temporal projection
Example: Location to put-down pancake mix

(a location
   (for pancake-mix)
   (on counter)
   (reachable-for Rosie)
   (visible-for James)
   (not-hindering (the activity (type pancake-making))))

Concepts

- Generative model
- Stability reasoning
- Reachability reasoning
- Visibility reasoning and perspective taking
- Light-weight temporal projection
Example: Location to put-down pancake mix

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making)))))

Concepts

- Generative model
- Stability reasoning
- **Reachability reasoning**
- Visibility reasoning and perspective taking
- Light-weight temporal projection
Example: Location to put-down pancake mix

(a location
   (for pancake-mix)
   (on counter)
   (reachable-for Rosie)
   (visible-for James)
   (not-hindering (the activity (type pancake-making)))))

Concepts

- Generative model
- Stability reasoning
- Reachability reasoning
- Visibility reasoning and perspective taking
- Light-weight temporal projection
**Example: Location to put-down pancake mix**

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making))))

### Concepts

- Generative model
- Stability reasoning
- Reachability reasoning
- Visibility reasoning and perspective taking
- Light-weight temporal projection
First-order representation of Designators

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making))))

\[\downarrow\]

\begin{align*}
\text{posesOn(Counter, PancakeMix, ?Poses) } & \land \text{ member(?P, ?Poses)} \\
& \land \text{ assertPose(PancakeMix, ?P) } \land \text{ stable(PancakeMix)} \\
& \land \text{ reachable(Rosie, PancakeMix) } \land \text{ blockingObjects(Rosie, PancakeMix, \emptyset)} \\
& \land \text{ visible(James, PancakeMix)} \\
& \land \text{ projectPlan(MakePancakes, ?Tl) } \land \text{ bagof(?F, flawsInTimeline(?Tl, ?F), \emptyset)}
\end{align*}
First-order representation of Designators

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making))))

⇓

posesOn(Counter, PancakeMix, ?Poses) ∧ member(?f, ?Poses)
  ∧ assertPose(PancakeMix, ?P) ∧ stable(PancakeMix)
  ∧ reachable(Rosie, PancakeMix) ∧ blockingObjects(Rosie, PancakeMix, {})
  ∧ visible(James, PancakeMix)
  ∧ projectPlan(MakePancakes, ?Tl) ∧ bagof(?F, flawsInTimeline(?Tl, ?F), {})

CPL
Lorenz Mösenlechner
First-order representation of Designators

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making))))

↓

\[
\text{posesOn(Counter, PancakeMix, ?Poses) } \land \text{ member(?P, ?Poses)}
\land \text{ assertPose(PancakeMix, ?P) } \land \text{ stable(PancakeMix)}
\land \text{ reachable(Rosie, PancakeMix) } \land \text{ blockingObjects(Rosie, PancakeMix, \emptyset)}
\land \text{ visible(James, PancakeMix)}
\land \text{ projectPlan(MakePancakes, ?Tl) } \land \text{ bagof(?F, flawsInTimeline(?Tl, ?F), \emptyset)}
\]
First-order representation of Designators

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making))))

⇓

posesOn(Counter, PancakeMix, ?Poses) ∧ member(?P, ?Poses)
  ∧ assertPose(PancakeMix, ?P) ∧ stable(PancakeMix)
  ∧ reachable(Rosie, PancakeMix) ∧ blockingObjects(Rosie, PancakeMix, ∅)
  ∧ visible(James, PancakeMix)
  ∧ projectPlan(MakePancakes, ?Tl) ∧ bagof(?F, flawsInTimeline(?Tl, ?F), ∅)
First-order representation of Designators

(a location
  (for pancake-mix)
  (on counter)
  (reachable-for Rosie)
  (visible-for James)
  (not-hindering (the activity (type pancake-making))))

⇓

posesOn(Counter, PancakeMix, ?Poses) ∧ member(?P, ?Poses)
  ∧ assertPose(PancakeMix, ?P) ∧ stable(PancakeMix)
  ∧ reachable(Rosie, PancakeMix) ∧ blockingObjects(Rosie, PancakeMix, ∅)
  ∧ visible(James, PancakeMix)
  ∧ projectPlan(MakePancakes, ?Tl) ∧ bagof(?F, flawsInTimeline(?Tl, ?F), ∅)
Inference algorithm

\[ \text{posesOn(Counter, Cup, \textit{?Poses})} \land \text{member(?P, \textit{?Poses})} \land \text{assertPose(Cup, ?P)} \land \text{stable(Cup)} \]
Inference algorithm

posesOn(Counter, Cup, ?Poses) ∧ member(?P, ?Poses)
∧ assertPose(Cup, ?P) ∧ stable(Cup)

1. posesOn(Counter, Cup, ?Poses)
2. member(?P, ?Poses)
3. assertPose(Cup, ?P)
4. stable(Cup)
Inference algorithm

\[ \text{posesOn(Counter, Cup, ?Poses)} \land \text{member(?P, ?Poses)} \land \text{assertPose(Cup, ?P)} \land \text{stable(Cup)} \]

1. \text{posesOn(Counter, Cup, ?Poses)}
2. \text{member(?P, ?Poses)}
3. \text{assertPose(Cup, ?P)}
4. \text{stable(Cup)}

Draw a pose sample
Inference algorithm

posesOn(Counter, Cup, ?Poses) \land member(?P, ?Poses) \\
\land assertPose(Cup, ?P) \land stable(Cup)

1. posesOn(Counter, Cup, ?Poses)
2. member(?P, ?Poses)
3. assertPose(Cup, ?P)
4. stable(Cup)
Inference algorithm

\[ \text{posesOn}(\text{Counter, Cup, ?Poses}) \land \text{member(?P, ?Poses)} \]
\[ \land \text{assertPose(Cup, ?P)} \land \text{stable(Cup)} \]

1. \( \text{posesOn}(\text{Counter, Cup, ?Poses}) \)
2. \( \text{member(?P, ?Poses)} \)
3. \( \text{assertPose(Cup, ?P)} \)
4. \( \text{stable(Cup)} \)

Simulate for 50ms, *fail!*
Inference algorithm

posesOn(Counter, Cup, ?Poses) \land member(?P, ?Poses) \\
\land assertPose(Cup, ?P) \land stable(Cup)

1. posesOn(Counter, Cup, ?Poses)
2. member(?P, ?Poses)
3. assertPose(Cup, ?P)
4. stable(Cup)

Backtrack, draw another pose sample
Inference algorithm

\[ \text{posesOn(Counter, Cup, ?Poses)} \land \text{member(?P, ?Poses)} \land \text{assertPose(Cup, ?P)} \land \text{stable(Cup)} \]

1. posesOn(Counter, Cup, ?Poses)
2. member(?P, ?Poses)
3. assertPose(Cup, ?P)
4. stable(Cup)
Inference algorithm

\[
\text{posesOn(Counter, Cup, ?Poses)} \land \text{member(?P, ?Poses)} \\
\land \text{assertPose(Cup, ?P)} \land \text{stable(Cup)}
\]

1. \text{posesOn(Counter, Cup, ?Poses)}
2. \text{member(?P, ?Poses)}
3. \text{assertPose(Cup, ?P)}
4. \text{stable(Cup)}

Simulate for 50ms, \textit{succeed!}
## Built-in Predicates of CRAM Reasoning

<table>
<thead>
<tr>
<th>Stability</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contact(O_1, O_2)</code></td>
<td>Contact between objects</td>
</tr>
<tr>
<td><code>stable(O)</code></td>
<td>Stability of object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visibility</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>visible(P, O)</code></td>
<td>Object visible from pose P</td>
</tr>
<tr>
<td><code>occluding(P, O_1, O_2)</code></td>
<td>Object $O_2$ occludes object $O_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reachability</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reachable(R, O)</code></td>
<td>Object $O$ is reachable by robot $R$</td>
</tr>
<tr>
<td><code>blockingObjects(R, O, B)</code></td>
<td>B is the list of objects that “block” $O$</td>
</tr>
</tbody>
</table>
Stability reasoning

- Bullet physics engine
- Infer collisions and stability of scenes
- Simulate for a short period of time (5s)
- Example: mugs can be placed on plates but not the other way around
Built-in Predicates of CRAM Reasoning

Visibility reasoning

- OpenGL to infer visibility of objects
- Render scene from camera frame, each object in a different color and count pixels
- Infer visibility of objects and occluding objects

See also “Geometric Tools for Perspective Taking for Human-Robot Interaction” [Marin-Urias 2008]
Built-in Predicates of CRAM Reasoning

**Reachability**

- Use inverse kinematics to infer reachability
- No grasp planning, reach for object center
- Blocking objects = objects in collision with IK solution
Temporal projection

(a location ... (not-hindering (the activity (type (pick-up Cup1))))))
Temporal projection

(a location ...
  (not-hindering (the activity (type (pick-up Cup1))))))

⇓

projectPlan(PickUp(Cup1), ?Ti) ∧ bagof(?F, flawsInTimeline(?F), ∅)

1. Execute plan in projection mode
2. Projection generates a timeline
3. Match pre-defined flaws on the timeline
Plan projection and Timeline generation

\[ \text{Achieve} (\text{ObjInHand}(\text{Cup1})) \]

\[ \text{Perceive}(\text{Cup1}) \]

\[ \text{AtLocation}(\text{ReachLoc}) \]

\[ \vdots \]

\[ \text{ObjectGrasped}(\text{Cup1}) \]

\[ \text{ObjectLifted}(\text{Cup1}) \]
Plan projection and Timeline generation

\[
\text{Achieve(ObjInHand(Cup1))}
\]

\[
\text{Perceive(Cup1)} \quad \text{AtLocation(ReachLoc)}
\]

\[
\text{ObjectGrasped(Cup1)} \quad \text{ObjectLifted(Cup1)}
\]
Plan projection and Timeline generation

\[
\text{Achieve}(\text{ObjInHand}(\text{Cup1}))
\]

\[
\text{Perceive}(\text{Cup1})
\]

\[
\text{AtLocation}(\text{ReachLoc})
\]

\[
\text{ObjectGrasped}(\text{Cup1}) \quad \text{ObjectLifted}(\text{Cup1})
\]
Plan projection and Timeline generation

\[
\text{Achieve}(\text{ObjInHand}(\text{Cup1}))
\]

\[
\text{Perceive}(\text{Cup1})
\]

\[
\text{AtLocation}(\text{ReachLoc})
\]

\[
\text{ObjectGrasped}(\text{Cup1}) \quad \text{ObjectLifted}(\text{Cup1})
\]
Plan projection and Timeline generation

\[
\text{Achieve}(\text{ObjInHand}(\text{Cup1}))
\]

\[
\text{Perceive}(\text{Cup1})
\]

\[
\text{AtLocation}(\text{ReachLoc})
\]

\[
\text{ObjectGrasped}(\text{Cup1}) \quad \text{ObjectLifted}(\text{Cup1})
\]
Inferring behavior flaws

- Projection creates a timeline
- Discrete transitions in world
- Execution trace of plan execution
- Predicates:
  - \( \text{holds}(\text{occ}, \text{at}(t)) \), \( \text{holds}(\text{occ}, \text{during}(t_1, t_2)) \), \( \text{holds}(\text{occ}, \text{throughout}(t_1, t_2)) \): Assert occasions (states)
  - \( \text{occurs}(\text{ev}, t) \): Assert occurrence of an event
- Example: Robot collides with an object
  \[
  \begin{align*}
  \text{TaskGoal}(\text{?tsk}, \text{Achieve}(\text{ObjectInHand(\text{Cup1})))) \\
  &\land \text{TaskStartedAt}(\text{?tsk}, \text{?t}_1) \land \text{TaskEndedAt}(\text{?tsk}, \text{?t}_2) \\
  &\land \text{holds}(\text{Contact(\text{Robot}, \text{?o}), during(\text{?t}_1, \text{?t}_2))} \land (\text{?o} \neq \text{Cup1})
  \end{align*}
  \]
Other applications of the system

Integration into executive to:

- keep a consistent representation of the world
- generate (artificial) sensor data
Other applications of the system

Integration into executive to:
- keep a consistent representation of the world
- generate (artificial) sensor data
Outline

1. Motivation
2. The Language
3. Reasoning about Plan Execution
4. Plan Parameterization and Inference of Locations
5. Lab session
Tutorial setup

- Make sure that Emacs is installed
- Make sure that `ros-electric-roslisp-common` and the `cram_pl` stack are installed.
- Execute
  
  ```
  cp `rospack find roslisp_repl`/swank.lisp ~/.swank.lisp
  ```
- You can run a LISP REPL with: `rosrun roslisp\_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
Go through tutorials 1 to 4 at http://ros.org/wiki/cram_pl/Tutorials.
Then solve the following task: we want to simulate two turtles in Turtlesim. One is controlled by the keyboard teleoperation node and one by CRAM. The CRAM controlled turtle should behave as follows:

1. It should drive along a “rectangular” path with the corners being close to the window corners. The exact coordinates do not matter too much, the turtle should just turn towards the next corner when it gets close to a corner.

2. If the turtle controlled by the keyboard comes too close, the CRAM turtle should stop. If the distance increases again, the CRAM turtle should continue driving on its rectangular path.