Introduction to Cognitive Robotics

David Vernon

www.cognitiverobotics.net

Lecture 31

www.cognitiverobotics.net/CR31.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

The CRAM Language

Based on CRAM documentation
http://cram-system.org/doc

The CRAM Language

The CRAM language is an extension of Lisp

- Recall Lisp's natural extensibility and suitability for task-specific programming languages … the CRAM language is an extension of Lisp

- Exploiting macros and functions

- Making heavy use of the operating system's multi-threading, especially for one of the key extensions: fluents

The CRAM Language

The CRAM language is an extension of Lisp

- The following provides an overview of a small subset of these extensions

- mainly to allow you to understand

  – The Beginner Tutorials

  – The detailed explanation of the pick-and-place CRAM plan example (intermediate tutorial: "How to write a simple mobile manipulation plan")

- For more details, see the CRAM language resources at the end

Fluents

- A fluent is a proxy object for some Common Lisp object

- It is used as a variable that allows a thread to effectively monitor and act on a change in value of a Lisp object

```
(wait-for fluent)
```

  – The current thread blocks (i.e. waits and does nothing) if `(value fluent)` is `nil`

```
(wait-for (pulsed fluent))
```

More on pulsed fluents below

  – The current thread blocks until the value of the fluent changes

Fluents

`(whenever fluent)`

- Iteratively repeats the body except when `(value fluent)` is `nil` before a new iteration
- If it is nil, the thread blocks until the fluent becomes true
- Unless `return` is called explicitly, the **whenever** form never terminates:
  it either repeatedly evaluates the body or blocks

Fluents

- A fluent is created with the function `make-fluent`
- The fluent is accessed through the reader function `value`
- A fluent can be set as follows
  `(setf (value fluent) <value>)`
- A fluent can be pulsed with `(pulse fluent)`

Fluents

Example

Create the fluent and initialize its value

```
(let ((fl (make-fluent :test-fluent :value nil)))
  (spawn-perception fl)
  (wait-for fl)
  (format t "Received value: ~a~%" (value fl)))
```

Pass as an argument to a hypothetical function to spawn a perception thread

Block until the value is not `nil`

Fluents

Example

```
(let ((fl (make-fluent :test-fluent :value nil)))
  (spawn-perception fl)
  (whenever (fl)
    (when (eq (value fl :done))
      (return-from whenever))
    (format t "Received value: ~a~%" (value fl))))
```

return when the fluent value is `:done`

Block until the value is not `nil`
otherwise, execute body repeatedly

Repeatedly print the value of the fluent

If you only want to print the value when it changes use

```
(whenever ((pulsed fl))
   ...)
```

Fluent Networks

- Fluents can be combined to create fluent networks
- A fluent network updates is value whenever one of the constituent fluents changes its value
- Combination is effected using (overloaded) relational, arithmetic, and logical operators

`<`, `>`, `=`, `eq`, `eql`, `+`, `-`, `*`, `/`

`fl-and`, `fl-or`, `fl-not`

Recall the native `and` returns the first non-nil value
`fl-and` returns a fluent containing `T` or `nil`

Fluent Pulses

- Whenever the value of a fluent is set to a different value, the fluent is pulsed automatically
- You can also pulse the fluent (without changing the value) with the `pulse` method
- To construct a fluent network that reacts if its constituent fluents are pulsed, use the `pulsed` combinator

  e.g. `(wait-for (pulsed fluent))`

Fluent Pulses

- `whenever` in combination with a pulse works like an infinite loop waiting for a pulse and then executing the body when the pulse occurs

- What happens if a fluent is pulsed when the body is being executed, i.e. what happens if there is a "missed" pulse?

- The `(pulsed fl)` form has three optional keywords to specify how pulses that occur during the execution of the body of a `whenever` are to be handled

Fluent Pulses

`(pulsed fl :handle-missed-pulses :once)`

Any number of missed pulses cause cause exactly one additional execution of the `whenever` body

`(pulsed fl :handle-missed-pulses :never)`

Missed pulses don't cause cause any additional execution of the `whenever` body

`(pulsed fl :handle-missed-pulses :always)`

The number of iterations of the `whenever` body exactly matches the number of missed pulses: the `whenever` body gets executed for every value change

Definition of Functions

- When defining functions to implement CRAM plans, use `def-cram-function` instead of `defun`

- `def-cram-function` uses the same syntax as `defun`

Top-level

- CRAM language forms must be executed in a top-level environment:

```
(top-level
   ...
)
```

- Alternatively, you can define a plan using

```
(def-top-level-plan
   ...
)
```

This contains an implicit top-level form

Concurrency

- Functions can be called sequentially or concurrently

- Sequential execution

```
(seq
   ...
)
```

  - Execute forms sequentially
  - Equivalent to `progn`
  - Fails if one of the component sub-forms fails
  - Succeeds when all of the component sub-forms succeed

Concurrency

- Functions can be called sequentially or concurrently

- Sequential execution

```
(try-in-order
   ...
)
```

  - Execute forms sequentially
  - Fails if all of the component sub-forms fail
  - Succeeds if one of the component sub-forms succeeds

Concurrency

- Functions can be called sequentially or concurrently

- Parallel execution

```
(par
   ...
)
```

  - Execute forms in parallel
  - Fails if one of the component forms fails
  - Succeeds when all the component sub-forms succeed

Concurrency

- Functions can be called sequentially or concurrently

- Parallel execution

```
(pursue
   ...
)
```

  – Execute forms in parallel
  – Fails if one of the component forms fails
  – Succeeds when one the component sub-forms succeeds
  – All other forms are evaporated (abandoned) when one form succeeds

Concurrency

- Functions can be called sequentially or concurrently

- Parallel execution

```
(pursue
  (wait-for (robot-at waypoint))
  (loop do
    (update-navigation-cmd waypoint)
    (sleep 0.1))
)
```

  - Terminates successfully when the robot is at the waypoint
  - Navigation commands to approach the waypoint are sent to the robot repeatedly

Concurrency

- Functions can be called sequentially or concurrently

- Parallel execution

```
(try-all
   ...
)
```

  - Execute forms in parallel
  - Fails if all of the component forms fail
  - Succeeds when one the component sub-forms succeeds

Exception Handling

Plan failures are generated and thrown using `(fail ...)`

`(with-failure-handling ...)`

Wraps function calls so that, if a failure occurs, failure handling is executed

Exception Handling

```
(with-failure-handling
               ((trajectory-controller-failed (e)
                   (declare (ignore e))
                   (retry)))
             (move-arm-to-point side pre-grasp-pos grasp))
```

body

- Monitor for failures of type `trajectory-controller-failed`
- In the event of a failure, call `(retry)` to run the body again on each failure
- Keep retrying until we succeed
- (Not a great strategy; the fetch-and-place example has a more sophisticated strategy using these constructs)

CRAM Reasoning

- The `cram_reasoning` package contains a full-featured Prolog interpreter (written in Lisp)

- The interpreter return a lazy list of solutions

  Solutions are generated on demand by accessing the element of the list

CRAM Reasoning

`(prolog ...)` executes the interpreter and proves a goal

```
CRS> (prolog '(member ?x (a b c)))
(((?X . 1)) . #S(CRAM-UTILITIES::LAZY-CONS-ELEM :GENERATOR ...)
```

First solution of the lazy list result; this solution is a tuple (i.e. a cons) with variable name in the `car` and the value in the `cdr`

force expansion of the lazy list

```
CRS> (force-ll (prolog '(member ?x (a b c))))
(((?X . 1)) ((?X . 2)) ((?X . 3)))
```

List of all three conses

CRAM Reasoning

`(var-value ...)` accesses the values

```
CRS> (var-value '?x (lazy-car (prolog `(member ?x (a b c)))))
A
```

Result

CRAM Reasoning

- Two ways to define a predicate
  - use a fact-group
  - Implement a Lisp function to get the current bindings and the predicate's parameters, and return a list of binding sets

CRAM Reasoning

```
(def-fact-group name (public-predicate*) fact-definition)
```

- Defines a fact group
- The `public-predicate` field declares which predicates can be defined in other fact-groups
- Predicates are defined using `<-`

```
(def-fact-group member-group ()
  (<- (member ?x (?x . ?_))
  (<- (member ?x (?y . ?z)
    (member ?x ?z))
```

CRAM Language Resources

CRAM Language	http://cram-system.org/doc/package/cram_language
CRAM Reasoning	http://cram-system.org/doc/package/cram_reasoning

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`

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`