Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 / MoSIG M1 Second Semester 2014/2015

Lesson 6 27 February 2015

Control of Reasoning and Decision Trees

Using Context to Structure Rules

It is possible to organize an expert system as a finite state machine, where each state corresponds to a rule "context" (also called a phase). Contexts can be organized into cycles, networks or trees. The transition between contexts can be coded reactively (as rules) or declaratively (as facts). Declarative control is slightly slower but provides the advantages of being easy to inspect, easy to change, and easier to debug. Examples include:

Diagnostic Systems (e.g. MYCIN) - tree of contexts
Systems for self-monitoring and self repair – cycles

Within each context, a set of rules encode the systems knowledge.

For larger systems, it is good practice to group rules into contexts. Each context concerns some sort of calculation coded as a body of rules that may fire in any order.

We can use a Salience hierarchy to assure that Context changes when all rules are fired.

Level	Salience	
Constraints	30	;; Rules that eliminate hypotheses
Expertise	20	;; Domain knowledge
Query	10	;; Rules that interrogate the user
Control	0	;; Context transitions

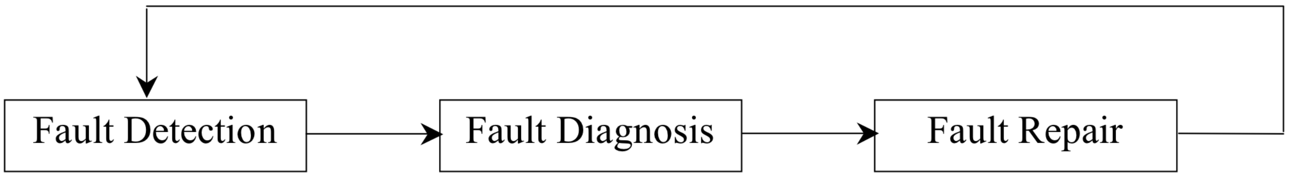
Contexts and Control Elements

Contexts are indicated by the existence of a token: an element in the facts list that indicates the current context.

```
(context <Name of the context>)
```

A classic example is a self-monitoring and self-repair system used for satellites and space applications. Such systems typically operate in cycle with 3 contexts:

Fault-Detection: A set of rules that test the integrity of subsystems
Fault-Diagnosis: A set of rules that determine the origin of an error.
Fault-Repair: A set of rules that reconfigure the system to repair a fault.



One can imagine many ways to code control.
For example,

1) It is possible to code the fault condition in each rule. However this is expensive and can allow multiple interacting faults to interfere with each other.

2) Use a "context" element to mark each context and then use a rule to manage the transitions. This is the preferred solution.

The transition rules encode the control of the system. These can be reactive or declarative. For example:

```
(defrule detection-to-diagnosis
   ?context <- (context detection)
   (fault ?f detected)
=>
   (retract ?context)
   (assert (context diagnosis))
   (printout t "Fault " ?f " detected!" crlf)
)

(defrule diagnosis-to-repair
   ?context <- (context diagnosis)
   (fault ?f detected)
   (fault ?f diagnosis ?d)
=>
   (retract ?context)
   (assert (context repair))
   (printout t " Fault " ?f " diagnosis " ?d crlf)
)

(defrule repair-to-detection
   ?c <- (context repair)
   (fault ?f diagnosis ?d repair ?r)
=>
   (retract ?c)
   (assert (context detection))
   (printout t "Fault repaired" crlf)
)
```

Each context contains a collection of rules for domain knowledge that diagnosis and suggest a repair for the fault. We can use a salience hierarchy to assure execution of domain knowledge

Salience Hierarchy:

Level	Salience	
Constraints	30	;; Rules that eliminate hypotheses
Expertise	20	;; Domain knowledge
Query	10	;; Rules that interrogate the user
Control	0	;; Context transitions

```
(defrule find-fault
   (declare (salience 20))
   (context identification)
=>
   (printout t "So, what broke this time? ")
   (assert (fault id (read)))
)
```

;; function to set a component on or off

```
(deffunction set (?a ?b) (printout t ?a " is set to " ?b
crlf))

(set motor off)


;;;
;;; repair knowledge in facts
;;;;

(defacts repair-knowledge
   (replace  motor1 motor2)
   (replace  sensor-1 sensor2)
)


(defrule repair-fault
   (declare (salience 20))
   (context repair)
   (fault identified  ?p)
   (replace  ?p ?replacement)
=>
   (set ?p off)    ;; function defined by user.
   (set ?replacement on)
   (assert (fault repaired))
)
```

Declarative Control Structures

An alternative to coding the context transitions in explicit rules, is to encode the context transitions in a declarative data structure and use a single generic transition rule.

```
(deffacts control-list
    (context detection)
    (next-context detection diagnosis)
    (next-context diagnosis repair)
    (next-context repair detection)
)

(defrule context transition rule.
    (declare (salience -10))
    ?P <- (context ?context)
    (next-context ?context ?next)
=>
    (retract ?P)
    (assert (context ?next))
)
```

This is an example of a DECLARATIVE representation of control knowledge.
Declarative structures make it possible to treat knowledge representations as data for calculation. A declarative representation can be used as data to reason about knowledge.

Decision Trees

A decision trees can be used for diagnosis and classification problems.
They require that the space of problems can be reduced to a series of yes/no tests

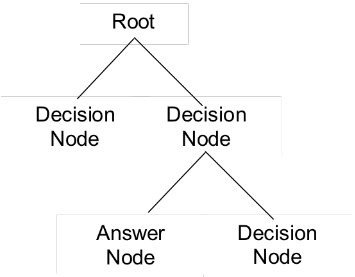
A decision tree is composed of decision nodes and leaves.
The answers are found at the leaves.

We can represent a node as a fact of the form:

(node <name> decision ?question ?yes ?no) ;; ?yes and ?no are names of nodes
and
(node <name> answer ?answer) ;; ?answer is a string

Example: Learn to guess the animal

The following example is a system that learns to guess an animal by asking questions.



```
(deffacts tree
(node  root decision "Is it warm blooded?" n1 n2)
(node  n1 decision "Does it purr ?" n3 n4)
(node  n2 answer snake)
(node  n3 answer cat)
(node  n4 answer dog)
)

;; initialization rule

(defrule init
   (initial-fact)
=>
 (assert (current-node root))
)
```

this could also be done with:

```
(defrule init
=>
 (assert (current-node root))
)
;; rule to request a decision node

(defrule make-decision
   ?N <- (current-node   ?name)
   (node  ?name decision ?q ?yes ?no)
=>
   (retract ?N)
   (format t "%s (yes or no) " ?q)
   (bind ?answer (read))
   (if (eq ?answer yes)
      then (assert (current-node ?yes) )
      else (assert (current-node ?no))
   )
)

;;  rule to give answers

(defrule give-answer
   ?N <- (current-node   ?name)
   (node  ?name answer ?r)
=>
   (printout t "I guess that it is a " ?r crlf)
   (printout t "Am I right? (yes or no) ")
   (bind ?rep (read))
   (if (eq ?rep yes)
      then (assert (context play-again))
         (retract ?N)
      else (assert (context correct-answer))
   )
)

;; rule to play again


(defrule play-again
   ?context <- (context play-again)
=>
   (retract ?context)
   (printout t "play again? (yes or no) ")
   (bind ?rep (read))
```

```
    (if (eq ?rep yes)
       then (assert (current-node   root))
       else (save-facts "animal.dat")
              (halt)
    )
)

;; rule to learn the correct answer

(defrule learn-correct-answer
    ?P <- (context correct-answer)
    ?N <- (current-node   ?name)
    ?D <- (node  ?name answer ?r)
=>
    (retract ?P ?N ?D)  ;;    Ask the correct answer
    (printout t "What animal was it? ")
    (bind ?new (read))
    (printout t  "What question should I ask to tell a "
       ?new " from a " ?r "? ")
    (bind ?question (readline))
    (bind ?newnode1 (gensym*))
    (bind ?newnode2 (gensym*))
    (assert (node  ?newnode1 answer ?new))
    (assert (node  ?newnode2 answer ?r))
    (assert
    (node ?name decision ?question ?newnode1 ?newnode2))
    (assert (context play-again))
)


;; Rule to open the file animal.dat

(defrule init
    (initial-fact)
    (play-again)
=>
 (assert (file (open "animal.dat" data "r")))
)

;; rule to close the file animal.dat

(defrule no-file
   ?f <- (file FALSE)
=>
   (retract ?f)
   (assert (current-node root))
```

```
)

;; rule to read the file.

(defrule init-file
   ?f <- (file TRUE)
=>
 (bind ?in (readline data))
 (printout t ?in crlf)
 (if (eq ?in EOF) then (assert (eof))
  else
    (assert-string ?in)
    (retract ?f)
    (assert (file TRUE))
   )
)

(defrule eof
   (declare (salience 30))
   ?f <- (file TRUE)
   ?eof <- (eof)
=>
   (retract ?f ?eof)
   (close data)
   (assert (current-node root))
)
```