[17] M. Minsky, *The Society of Mind*, Touchstone, 1988.

[18] R. Norving, *Artificial Inteligence, a modern approach*, 1995, Prentice-Hall.

[19] W. S. Reilly, J. Bates, Natural Negotiation for Believable Agents, Tech. Report CMU-CS-95-164, Carnegie Mellon University, June 1995.

[20] A. Sloman, M. Croucher, "Why Robots Will Have Emotions", *Proc. IJCAI 1981*, Vancouver.

[21] M. Wooldridge, N. R. Jennings, "Intelligent Agents: Theory and Practice", January 1995.

Referências

[1] J. Bates, A. B. Loyall, W. S. Reilly, An Architecture for Action, Emotion, and Social Behaviour, Tech. Report CMU-CS-92-144, Carnegie Mellon University, May 1992.

[2] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Jour. of Robotics and Automation*, Vol. RA-2, No. 1, March 1989.

[3] António R. Damásio, *O Erro de Descartes*, Europa-América, 1994.

[4] W. H. Davies, P. Edwards, "Agent-K: An Integration of AOP and KQML," Dept. od Computing Science, King's College, Univ. od Aberdeen, 1995.

[5] J. E. Doran, *Experiments with a Pleasure-seeking Automaton*, Machine Intelligence, 1968.

[6] O. Etzioni, N. Lesh, R. Segal, Building Softbots for UNIX (Preliminary Report), Tech. Report 93-09-01, University of Washington, 1993.

[7] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, J. McGuire, S. Shapiro, C. Beck, Specification of the KQML Agent-Communication Language, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993.

[8] S. French, *Sequencing and Scheduling*, John Wiley & Sons, 1982.

[9] D. Gilbert, M. Aparicio, B. Atkinson, S. Brady, J. Ciccarino, B. Grosof, P. O'Connor, D. Osisek, S. Prikto, R. Spagna, L. Wilson, "Inteligent Agent Stategy," IBM, 1995.

[10] P. Graham, *ANSI Common Lisp*, Prentice-Hall, 1996.

[11] H. A. Kautz, B. Selman, M. Coen, S. Ketchpel, "An Experiment in the Design of Software Agents", AI Principles Research Dept, AT&T Bell Laboratories, 1993.

[12] J. R. Koza, *Genetic Programming*, MIT Press, 1992.

[13] A. B. Loyall, J. Bates, Hap — A Reactive, Adaptative Architecture for Agents, Tech. Report CMU-CS-91-147, Carnegie Mellon University, June 24, 1991.

[14] A. B. Loyall, J. Bates, Behaviour-based Language Generation for Believable Agents, Tech. Report CMU-CS-95-139, Carnegie Mellon University, March 1995.

[15] P. Maes, "Artificial Life meets Entertainment: Lifelike Autonomous Agents," MIT Media Lab, 1995.

[16] D. McAllester, D. Rosenblitt, "Systematic Nonlinear Planning", *AAAI-91*, 1991.

```
;; Wakes every agent up, once for each, random order.
(defprim* 'run-every-agent (a-list)
   (let* ((random-seq-list '((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)))
          (agent-list (agent-eval agent a-list))
          (sequence (nth (random 6) random-seq-list)))
      ;         (format t "~&SEQUENCE : ~a~%" sequence)
      (mapcar  #'(lambda (token)
                   (give-token (nth (1- token) agent-list))) sequence)))

;; A kind of replacement for :gostate that, unlike it, doesn't force an immediate state change.
(defprim* 'set-state (new-state)
   (loop for a in (get-var agent '/sons)
      unless (equal (get-var a '/state) 'happy)
      do (set-var a '/state new-state)))

;; Puts each agent's already-read-list's on the screen.
(defprim 'dump-final-results ()
   (mapcar #'(lambda (an-agent) (format t "~&Reading list for agent #~a is ~a~%"
                                  (get-id an-agent) (get-var an-agent  '/already-read-list)))
      (get-var agent '/sons)))

;; Here's the big boss.
(defagent 'meta-agent
   :behaviour '((:onstate (init)
                 :rules (:then (do (let /sons (make-agents))
                                   (fill-blackboard)
                                   (post-time)
                                   (run-every-agent /sons)
                                   (rerun))
                           :gostate running))

                (:onstate (running)
                 :rules (:if (not (all-agents-happy))
                         :then (do (increase /elapsed-time)
                                   (remove-time)
                                   (post-time)
                                   (set-state drop-and-pick)
                                   (run-every-agent /sons)
                                   (set-state just-pick)
                                   (run-every-agent /sons)
                                   (check-for-happy-agents)
                                   (rerun))
                         :elsestate ending))
                (:onstate (ending)
                 :rules (:then (dump-final-results))))

   :knowledge '((/total-agents 3) (/happy-agents 0) (/elapsed-time 0))
   :initstate 'init)

;; One instance of the blackboard.
(defparameter bb (make-blackboard))

;; One instance of the meta-agent.
(defparameter agent (make-agent meta-agent
                      :id 0 :blackboards (list bb)))

(defun main ()
   (give-token agent))
```

```
      (:if (not (get-next-paper /paper))
       :then (post-happy /already-read-list)
       )
      )

     ;; **** Second state of each round.
     (:onstate (just-pick)
      :rules
      ;; **** If he didn't get a paper while in drop-and-pick state.
      (:if (and (not (am-I-reading))
                (get-time /time)
                (get-next-paper /paper))
       :then (do (pick-paper-from-bb /paper /new-slot)
                 ;; **** It is now that NIL is inserted if no suitable paper is found on bb.
                 (insert-new-paper /time /new-slot)))
      ))
   :initstate 'init)

;;   ********** Meta Agent **********
;; The guy who deals the cards.
;;
;;

;; Counts happy agents
(defprim 'check-for-happy-agents ()
   (when (find-post agent 0 :predicate 'happy :to-me t :remove t)
      (set-var agent '/happy-agents (1+ (get-var agent '/happy-agents)))))

;; Is everyone happy yet ?
(defprim 'all-agents-happy ()
   (equal (get-var agent '/happy-agents)
      (get-var agent '/total-agents)))

;; Creates a list with each agent having its wish-list properly set .
(defun make-agents ()
   (list (make-agent reader-agent :id 1 :blackboards (list bb)
            :knowledge '((/wish-list (FT DN S))))
      (make-agent reader-agent :id 2 :blackboards (list bb)
         :knowledge '((/wish-list (FT S DN))))
      (make-agent reader-agent :id 3 :blackboards (list bb)
         :knowledge '((/wish-list (S DN FT))))))

;; Just a silly little helper.
(defprim* 'increase (data)
   (set-var agent data (+ 1 (agent-eval agent data))))

;; Erases time from bb.
(defprim 'remove-time ()
   (find-post agent 0 :predicate 'time :from-me t :remove t))

;; Puts current time on bb.
(defprim 'post-time ()
   (format t "~&   **** TIME : ~a ****~%" (get-var agent '/elapsed-time))
   (put-post agent '* 0 'time (get-var agent '/elapsed-time)))

;; Puts newspapers on bb, ready to pick.
(defun fill-blackboard ()
   (mapcar #'(lambda (post)
                (put-post agent '* 0 'to-pick post))
      '(FT S DN)))
```

```
   (get-var agent current-time))

;; Picks next paper from internal list (wish-list)
(defprim* 'get-next-paper (paper)
   (set-var agent paper
      (nth (get-var agent '/already-read-number) (get-var agent '/wish-list)))
   (get-var agent paper))

;; Gets paper from bb
(defprim* 'pick-paper-from-bb (paper new-slot)
   (format t "~&Seeking ~a~%" (get-var agent paper))
   (set-var agent new-slot
      (pcontents
         (find-post agent 0 :predicate 'to-pick
            :test #'(lambda (post)
                      (equal (pcontents post) (get-var agent paper)))
            :remove t)))
   (format t "~&Got a new one : ~a~%" (get-var agent new-slot))
   (not (null (get-var agent new-slot))))

;; READS paper now, inserting it in already-read-list
(defprim 'insert-new-paper (elapsed-time new-slot)
   (format t "~&Inserting new paper ~a~%" new-slot)
   (set-var agent '/already-read-list (append (get-var agent '/already-read-list) (list new-slot)))
   ;; This is an under-cover 'if'
   (and new-slot
        (set-var agent '/already-read-number (1+ (get-var agent '/already-read-number)))
        (set-var agent '/now-reading t)))

;; The game is played in two rounds :
;;      1st - "drop-and-pick" : Each agent drops the paper he's just finished
;;                              reading, and tries to pick a new one.
;;      2nd - "just-pick"     : If the agent couldn't get the paper he wanted from the
;;                              bb, he still has another chance to try and do it now
;;                             that others have dropped theirs.
;;
;; NOTE: there are no :gostate's. State transition forced externally by Meta-agnet
;;
(defagent 'reader-agent
   :behaviour
   '((:onstate (init)
      :rules(:then (do (let /already-read-list NIL)
                       (let /already-read-number 0)
                      (let /current-time 0)
                      (let /now-reading NIL))
            ))
     ;; **** First state of each round
     (:onstate (drop-and-pick)
      :rules
      ;; **** Whenever he's sick of reading -> drop
      (:if (and (get-time /time) (am-I-reading) (sick-of-reading /time))
       :then (put-away-paper)
       )
      ;; **** Just in case there's something interesting on bb -> pick
      (:if (and (get-time /time)
                (get-next-paper /paper)
                (pick-paper-from-bb /paper /new-slot))
       :then (insert-new-paper /time /new-slot)
       )
      ;; **** If there are no more papers to read ...
```

```
(defparameter resources '(6 6 6 6))
(defparameter products '(1 1 1 2 2 2 3 3 3))

; Agents and Blackboard creation
(defparameter bb (make-blackboard))

(defparameter agents
  (loop for id from 1 to (length resources)
        for len in resources
        collect (make-agent resource
                            :id id
                            :knowledge (list (list '/total len)
                                             (list '/free  len)
                                             (list '/alloc nil))
                            :blackboards (list bb))))

; RUN ME!
(defun main ()
  (clear bb)
  (mapcar #'(lambda (p) (put-msg bb (list 0 '* 0 'product p))) products)
  (run agents))
```

B.4 O problema do *jobshop*

Este exemplo inclui um meta-agente para controlar a execução dos agentes. Sendo assim, após uma chamada `(main)`, a execução acabará por chegar ao seu termo. Os resultados são impressos no ecrã quando o processo chega ao fim.

```
;;   ********** Reader Agent **********
;; Internal lists for each agent:
;;                       /wish-list
;;                       /already-read-list
;;

;; Self-explanatory.
(defprim 'am-I-reading ()
   (not (null (get-var agent '/now-reading))))

;; Future implementation shall take #slots>1 for each paper into account.
(defprim 'sick-of-reading (present-time)
   (equal present-time (1+ (length (get-var agent '/already-read-list)))))

;; Puts paper back on bb.
(defprim 'put-away-paper ()
   (let ((post (car (last (get-var agent '/already-read-list)))))
      (format t "~& ----->Putting back ... ~a~%" post)
      (put-post agent '* 0 'to-pick post))
   (set-var agent '/now-reading NIL))

;; Fetches current-time from bb.
(defprim* 'get-time (current-time)
   (set-var agent '/last-time (get-var agent current-time))
   (set-var agent current-time (pcontents (pick-post agent 0 :predicate 'time)))
```

```
  (let ((trade (pcontents post)))
    (cond ((null (second trade))
           ; Is a plain product move TO me
           (set-var agent '/alloc (cons (first trade) (get-var agent '/alloc))))
          ((null (first trade))
           ; Is a plain product move FROM me
           (setf (first (member (second trade) (get-var agent '/alloc))) 'kill-me)
           (set-var agent '/alloc (remove 'kill-me (get-var agent '/alloc))))
          (t
           ; Is a real trade
           (setf (first (member (second trade) (get-var agent '/alloc))) (first trade))))
    (set-var agent '/free (- (get-var agent '/total)
                             (apply #'+ (get-var agent '/alloc))))))

(defprim 'has-products ()
  (not (null (get-var agent '/alloc))))

(defprim 'has-free ()
  (not (zerop (get-var agent '/free))))

(defprim* 'make-complaint (complaint)
  (set-var agent complaint (cons (get-var agent '/free)
                                 (get-var agent '/alloc))))

(defprim 'post-proposal (trade complaint)
  (format t "~&Posting proposal trade=~a comp=~a~&" trade complaint)
  (put-post agent (psource complaint) 0 'proposal trade))

(defprim 'acceptable (proposal)
  (let ((he-gives (first (pcontents proposal)))
        (he-takes (second (pcontents proposal))))
    (and
     ; I have what he wants
     (or (null he-takes)
         (find he-takes (get-var agent '/alloc)))
     ; I can have what he gives
     (<= he-gives (+ (get-var agent '/free)
                     (if (null he-takes) 0 he-takes)))
     ; I improve
     (or (null he-takes)
         (< he-takes he-gives)))))

(defprim 'post-accept (proposal)
  (format t "~&Posting accept prop=~a~&" proposal)
  (put-post agent (psource proposal) 0  'accept (list (second (pcontents proposal))
                                                      (first (pcontents proposal)))))

(defprim 'post-reject (proposal)
  (format t "~&Posting reject prop=~a~&" proposal)
  (put-post agent (psource proposal) 0  'reject (list (second (pcontents proposal))
                                                      (first (pcontents proposal)))))


;;
;; Main Section
;; ~~~~~~~~~~~~

; Problem Statement
;(defparameter products '(3 3 3 2 2 2 2 2 8))
;(defparameter resources '(7 7 10 5))
```

```
                        (rem-my-complaint)
                        (do-trade /proposal))
                :gostate idle)

            (:if /proposal  ; Implicit: a proposal not accepted but recv
                :then (post-reject /proposal)))

    (:onstate (proposed)
     :rules (:if (accepted /trade :remove t)
                :then (do-trade /trade)
                :gostate idle)

            (:if (rejected /trade :remove t)
                :gostate idle)

            (:then (reject-proposals))))

  :initstate 'idle)


;;
;; Primitives
;; ~~~~~~~~~~

(defprim* 'find-product (product)
  (set-var agent product (pick-post agent 0 :predicate 'product)))

(defprim 'feasible (product)
  (<= (first (pcontents product))
      (get-var agent '/free)))

(defprim 'use-product (product)
  (set-var agent '/alloc (cons (first (pcontents product))
                               (get-var agent '/alloc)))
  (set-var agent '/free (- (get-var agent '/total)
                           (apply #'+ (get-var agent '/alloc)))))

(defprim* 'good (trade complaint)
  (let ((other-left (first (first (pcontents (agent-eval agent complaint)))))
        (other-have (rest  (first (pcontents (agent-eval agent complaint)))))
        (me-left (get-var agent '/free))
        (me-have (get-var agent '/alloc)))
    (set-var agent trade
             (when (<= other-left me-left)
                   (pick-random-atom
                    ; Try first plain product moves
                    (let ((hip-moves (loop for p_j in me-have
                                           when (<= p_j other-left)
                                           collect (list p_j nil))))
                      (if (null hip-moves)
                          ; If there is no plain moves, look for trades
                          (loop for p_i in other-have
                                append (loop for p_j in me-have
                                             when (and (<  0 (- p_j p_i))
                                                       (<= (- p_j p_i) other-left))
                                             collect (list p_j p_i)))
                        ; Otherwise use plain moves
                        hip-moves)))))))

(defprim 'do-trade (post)
```

B.3 O problema da alocação de recursos

Tal como nos exemplos anteriores, a chamada `(main)` dá início à execução do sistema. A ausência de um meta-agente obriga a que seja necessário interromper o processo manualmente. Para observar as variáveis de todos os agentes, para aferir da solução chegada, pode-se utilizar a chamada `(dump agents)`.

```
;;;
;;; Unidimensional Resource Allocation Problem
;;; ------------------------------------------
;;;

(unless (boundp '*RUBA*) (load "ruba.lsp"))


;;
;; Agent definition
;; ~~~~~~~~~~~~~~~~
;; Data representation:
;;   /total = total resource size
;;   /free  = free space left
;;   /alloc = list of products
;;
;; Complaint format:
;;   (<left> . <have>)
;;             ^^^^^^\____ list of products
;;
;; General Trade format (in posts/accepts/rejects):
;;   (<give> <take>)
;;

(defagent 'resource
  :behaviour
  '(
    (:onstate (idle complained)
     :rules (:if (and (find-product /product)
                      (feasible /product))
                 :then (do (rem-post /product)
                           (use-product /product)
                           (rerun))))

    (:onstate (idle)
     :rules (:if (and (has-products) (has-free))
                 :then (do (make-complaint /complaint)
                           (post-complaint /complaint))
                 :gostate complained))

    (:onstate (complained)
     :rules (:if (and (get-complaint /complaint)
                      (good /trade /complaint))
                 :then (do (post-proposal /trade /complaint)
                           (rem-my-complaint))
                 :gostate proposed)

            (:if (and (get-proposal /proposal :remove t)
                      (acceptable /proposal))
                 :then (do (post-accept /proposal)
                           (reject-proposals)
```

```
  (and (>= r 0) (< r bsize)
       (>= c 0) (< r bsize)))

(defun gen-zone (row col bsize)
  (loop with lst
        for n upfrom 1 below bsize
        collect (list row (mod (+ col n) bsize)) into lst
        collect (list (mod (+ row n) bsize) col) into lst
        append  (let* ((c (mod (+ col n) bsize))
                       (r1 (- (+ row col) c))
                       (r2 (+ (- row col) c)))
                  (append (when (is-on r1 c bsize)
                                (list (list r1 c)))
                          (when (is-on r2 c bsize)
                                (list (list r2 c))))) into lst
        finally (return lst)))

(defun get-hits (board row col)
  (loop for p in (gen-zone row col (board-size board))
        count (aref board (first p) (second p)) into n
        finally (return n)))

(defun heuristic (board)
  (loop for r upfrom 0 below (board-size board)
        sum (loop for c upfrom 0 below (board-size board)
                  when (aref board r c) sum (get-hits board r c) into n
                  finally (return n)) into total
        finally (return total)))


;;
;; Main section
;; ~~~~~~~~~~~~

(defparameter bb (make-blackboard))

(defparameter agents
  (loop for id from 1 to *nqueens*
        collect (make-agent queen-vb :id id :blackboards (list bb))))
;                           ^^^^^^^^---> Change this to either queen-vb or queen-wb

(defparameter theboard (make-board *bsize*))


; RUN ME!
(defun main ()
  (clear bb)
  (put-msg bb (list nil '* 0 'board theboard))
  (run agents))

; To inspect the board do ....... (print-board theboard)
; To dump the agents do ......... (dump agents)
; To dump the blackboard do ..... (dump bb)
; To dump all at once do ........ (dump (list agents bb))
; hey, ain't this beautiful? 8-)
```

```
                                collect (list row col)))))
    (unless (null pos) (put-on-board pos (get-id agent) board)))))

(defprim 'pick-place ()
  (let ((board (first (pcontents (find-post agent 0 :predicate 'board)))))
    (labels ((get-random-place ()
                               (let ((row (random *bsize*))
                                     (col (random *bsize*)))
                                 (if (aref board row col) (get-random-place)
                                   (list row col)))))
            (get-random-place))))


;;
;; Low-level functions
;; ~~~~~~~~~~~~~~~~~~~

(defun remove-from-board (pos id board)
  (setf (aref board (first pos) (second pos)) nil))

(defun put-on-board (pos id board)
  (assert (not (aref board (first pos) (second pos))))
  (setf (aref board (first pos) (second pos)) id))


;
; These ones are recycled
;

(defun make-board (bsize)
  (make-array (list bsize bsize)))

(defun board-size (board)
  (array-dimension board 0))

(defun print-board (board &optional (stream *standard-output*))
  (loop for r upfrom 0 below (board-size board)
        do (loop for c upfrom 0 below (board-size board)
                 do (princ (if (aref board r c) *queen-char* *board-char*) stream))
           (fresh-line stream)))

(defun print-board2 (board &optional (stream *standard-output*))
  (loop for r upfrom 0 below (board-size board)
        do (loop for c upfrom 0 below (board-size board)
                 do (princ (if (aref board r c) (aref board r c) *board-char*) stream))
           (fresh-line stream)))

(defun random-place (board nqueens)
  (loop repeat nqueens
        do (setf board (labels ((place-one (b)
                                           (let ((r (random (board-size board)))
                                                 (c (random (board-size board))))
                                             (if (aref b r c)
                                                 (place-one b)
                                               (setf (aref b r c) t)))
                                           b))
                               (place-one board)))
        finally (return board)))

(defun is-on (r c bsize)
```

```
                          :then (send-warn /someone)))

               (:onstate (out-board)
                :rules (:then (move-to (pick-place))
                        :gostate in-board)))

  :initstate 'out-board)


;;
;; Primitives
;; ~~~~~~~~~~

(defprim 'recv-warn ()
  (find-post agent 0 :predicate 'warning :to-me t :remove t))

(defprim 'recv-shake ()
  (find-post agent 0 :predicate 'shake :to-me t :remove t))

(defprim 'send-warn (target)
  (put-post agent target 0 'warning nil))

(defprim 'send-shake ()
  (put-post agent '* 0 'shake nil))

(defprim 'move-to (new-place)
  (let ((board (first (pcontents (find-post agent 0 :predicate 'board)))))
    (assert board)
    (when (get-var agent '/position)
          (remove-from-board (get-var agent '/position) (get-id agent) board))
    (put-on-board new-place (get-id agent) board)
    (set-var agent '/position new-place)))

(defprim 'get-out ()
  (let ((board (first (pcontents (find-post agent 0 :predicate 'board)))))
    (assert board)
    (when (get-var agent '/position)
          (remove-from-board (get-var agent '/position) (get-id agent) board)
          (set-var agent '/position nil))))

(defprim* 'attacking (victim)
  (let* ((mypos (get-var agent '/position))
         (board (first (pcontents (find-post agent 0 :predicate 'board))))
         (pos (pick-random-atom
               (loop for p in (gen-zone (first mypos) (second mypos) *bsize*)
                     when (aref board (first p) (second p)) collect p))))
    (set-var agent victim (unless (null pos)
                                  (aref board (first pos) (second pos))))))

(defprim* 'find-vacancy (old place)
  (let ((board (first (pcontents (find-post agent 0 :predicate 'board))))
        (pos (agent-eval agent old)))
    (prog2
     (unless (null pos) (remove-from-board pos (get-id agent) board))
     (set-var agent place
              (pick-random-atom
               (loop for row from 0 below *bsize*
                     append (loop for col from 0 below *bsize*
                                  when (and (zerop (get-hits board row col))
                                            (not (aref board row col)))
```

```
; RUN ME!
(defun main ()
  (give-token agent))
```

B.2 O problema das 8 rainhas

A chamada (`main`) põe o programa em execução. Como não há um meta-agente, a execução terá que ser interrompida manualmente. Esta interrupção leva os ambientes LISP para um nível de *debugging*, a partir do qual se poderá inspeccionar o estado do sistema. No fim do código estão algumas sugestões de como isto poderá ser feito.

```
;;;
;;; N-Queen on a MxM board problem solver
;;; -------------------------------------
;;;
;;; Data representation: (<row> <col>)
;;;


(unless (boundp '*RUBA*) (load "ruba.lsp"))

(defparameter *queen-char* #\*)
(defparameter *board-char* #\O)

(defparameter *bsize* 8)
(defparameter *nqueens* 8)


;;
;; Agents
;; ~~~~~~

; A conservative vacancy-based Queen
(defagent 'queen-vb
  :behaviour '((:onstate (in-board)
                :rules (:if (and (recv-shake) (find-vacancy /position /vacancy))
                            :then (move-to /vacancy)))

               (:onstate (out-board)
                :rules (:if (find-vacancy nil /vacancy)
                            :then (move-to /vacancy)
                            :gostate in-board
                            :else (send-shake))))

  :initstate 'out-board)

; A lazy warning-based Queen
(defagent 'queen-wb
  :behaviour '((:onstate (in-board)
                :rules (:if (recv-warn)
                            :then (move-to (pick-place)))

                       (:if (attacking /someone)
```

```
                :then (post-complaint /have)
                :gostate complained)
           (:if (like /have)
                :then (post-happy /have)
                :gostate done))

    (:onstate (complained)
     :rules (:if (and (get-complaint /comp) (like (object /comp)))
                :then (do (post-proposal /comp)
                          (rem-my-complaint))
                :gostate proposed)
           (:if (and (get-proposal /prop :remove t) (not (dislike (object /prop))))
                :then (do (post-accept /prop)
                          (reject-proposals)
                          (rem-my-complaint)
                          (do-trade /prop))
                :gostate idle)
           (:if (and /prop (dislike (object /prop)))
                :then (post-reject /prop)))

    (:onstate (proposed)
     :rules (:if (accepted /trade :remove t)
                :then (do-trade /trade)
                :gostate idle)
           (:if (rejected /trade :remove t)
                :gostate idle)
           (:if t :then (reject-proposals))))
  :initstate 'idle)

(defagent 'meta-agent
  :behaviour '((:onstate (init)
                :rules (:then (do (let /sons (make-agents))
                                  (rerun))
                        :gostate running))

               (:onstate (running)
                :rules (:if (not (all-agents-happy))
                        :then (do (run-agents /sons :times 1)
                                  (check-for-happy-agents)
                                  (rerun))))))

  :knowledge '((/total-agents 3) (/happy-agents 0))
  :initstate 'init)


;;
;; Main section
;; ------------

(defun make-agents ()
  (list (make-agent balls-agent :id 1 :blackboards (list bb)
                    :knowledge '((/have RED)    (/want GREEN)))
        (make-agent balls-agent :id 2 :blackboards (list bb)
                    :knowledge '((/have BLUE)   (/want RED)))
        (make-agent balls-agent :id 3 :blackboards (list bb)
                    :knowledge '((/have GREEN)  (/want BLUE)))))

(defparameter bb (make-blackboard))

(defparameter agent (make-agent meta-agent :id 0 :blackboards (list bb)))
```

```
;;
;; Domain-dependent primitives
;; ---------------------------

(defprim 'like (object)
  (let ((want (get-var agent '/want)))
    (equal object want)))

(defprim 'dislike (object)
  (let ((want (get-var agent '/want))
        (have (get-var agent '/have)))
    (if (equal object want) nil
      (if (equal have want) t nil))))

(defprim 'object (post)
  (first (pcontents post)))

(defprim 'do-trade (post)
  (set-var agent '/have (first (pcontents post))))

(defprim 'post-proposal (complaint)
  (format t "~&Posting proposal comp=~a~&" complaint)
  (put-post agent (psource complaint) 0 'proposal
            (list (get-var agent '/have)
                  (first (pcontents complaint)))))

(defprim 'post-accept (proposal)
  (format t "~&Posting accept prop=~a~&" proposal)
  (put-post agent (psource proposal) 0  'accept (list (second (pcontents proposal))
                                                      (first (pcontents proposal)))))

(defprim 'post-reject (proposal)
  (format t "~&Posting reject prop=~a~&" proposal)
  (put-post agent (psource proposal) 0  'reject (list (second (pcontents proposal))
                                                      (first (pcontents proposal)))))


;
; Master agent primitives
;

(defprim 'check-for-happy-agents ()
  (when (find-post agent 0 :predicate 'happy :to-me t :remove t)
        (set-var agent '/happy-agents (1+ (get-var agent '/happy-agents)))))

(defprim 'all-agents-happy ()
  (equal (get-var agent '/happy-agents)
         (get-var agent '/total-agents)))


;;
;; Agent definition
;; ----------------

(defagent 'balls-agent
  :behaviour
  '(
    (:onstate (idle)
     :rules (:if (not (like /have))
```

```
    (set-var agent proposal post)
    post))

(defprim* 'accepted (trade &key remove)
  (let ((post (find-post agent 0 :predicate 'accept :to-me t :remove remove)))
    (unless (null post)
            (set-var agent trade post)
            post)))

(defprim* 'rejected (trade &key remove)
  (let ((post (find-post agent 0 :predicate 'reject :to-me t :remove remove)))
    (unless (null post)
            (set-var agent trade post)
            post)))

(defprim 'post-happy (object)
  (format t "~&Posting HAPPY~&")
  (put-post agent '* 0  'happy (list object)))

(defprim 'reject-proposals ()
  (format t "~&Rejecting proposals.~&")
  (loop (let ((prop (find-post agent 0 :predicate 'proposal :to-me t :remove t)))
          (if (null prop) (return)
            (apply-prim agent 'post-reject (list prop))))))

(defprim 'rem-my-complaint ()
  (format t "~&Removing my complaint.~&")
  (loop (let ((comp (find-post agent 0 :predicate 'complaint :from-me t :remove t)))
          (if (null comp) (return)))))

(defprim 'rem-post (post)
  (format t "~&Removing post=~a.~&" post)
  (rem-post agent 0 post))
```

B Código fonte dos exemplos de aplicação do RUBA

B.1 O problema das bolas

Para correr este exemplo basta emitir o comando `(main)`. O processo é controlado por um meta-agente que se encarregará de terminar a execução quando a solução for encontrada.

```
(unless (boundp '*RUBA*) (load "ruba.lsp"))

;;
;; Post format
;; -----------
;;
;; (<source-id> <destination-id> <blackboard-id> <predicate> . <contents>)
;;
;;   <pred>: trade      (<give> <take>)
;;           proposal   (<give> <take>)
;;           complaint  (<have>)
;;
```

```
        thereis (agent-eval agent arg)))

(defprim* 'do (&rest args)
  (loop for sexp in args
        do (agent-eval agent sexp))
  t)


;
; General primitives
;

(defprim* 'run-agents (a-list &key while until times)
  (let* ((agent-list (agent-eval agent a-list))
         (nagents (length agent-list))
         (token (random nagents)))
    (loop while (if (null while) t (agent-eval agent while))
          until (if (null until) nil (agent-eval agent until))
          do (unless (null times)
                     (if (zerop times) (return) (decf times)))
             (setf token (new-token token nagents))
             (give-token (nth token agent-list)))))

(defprim* 'let (var value)
  (set-var agent var (agent-eval agent value)))

(defprim 'rerun ()
  (set-var agent '/rerun t))


;
; Primitives derived from "fruitcake"
;

(defprim 'on-bb (obj)
  (not (null (find obj (get-board (nth-bb 0 agent))))))

(defprim 'put-bb (obj)
  (put-msg (nth-bb 0 agent) obj))

(defprim 'bb-empty ()
  (null (get-board (nth-bb 0 agent))))


;
; Primitives derived from "balls"
;

(defprim 'post-complaint (object)
  (format t "~&Posting complaint ~a~&" object)
  (put-post agent '* 0  'complaint (list object)))

(defprim* 'get-complaint (complaint &key remove)
  (let ((post (pick-post agent 0 :predicate 'complaint :remove remove)))
    (unless (null post)
            (set-var agent complaint post)
            post)))

(defprim* 'get-proposal (proposal &key remove)
  (let ((post (find-post agent 0 :predicate 'proposal :to-me t :remove remove)))
```

mentos *keyword* do LISP. Usando a função `pick-random-atom` é possível obter um elemento de uma lista, escolhido aleatoriamente.

```
;;
;; Low Level Functions
;; -------------------

(defun new-token (old total)
  (let ((token (random total)))
    (if (= old token) (new-token old total) token)))

(defun is-var (sym)
  (and (symbolp sym) (char= (char (symbol-name sym) 0) *binding-prefix*)))

(defun is-sexp (sexp)
  (and (listp sexp) (not (null sexp))))

(defun is-special (prim)
  (and (listp prim) (eql :special (first prim))))

(defun get-tag (tag lst &key multi)
  "Returns the argument(s) that follow a given keyword (tag);
   If multi=t, returns a list of arguments until another keyword is found"
  (let ((items (rest (member tag lst))))
    (cond ((null items) nil)
          (multi (labels ((get-items (l)
                            (cond ((null l) nil)
                                  ((keywordp (first l)) nil)
                                  (t (cons (first l) (get-items (rest l)))))))
                   (get-items items)))
          (t (first items)))))

(defun pick-random-atom (list) "Picks a random atom from the given list"
  "Picks a random element of a given list"
  (unless (null list)
          (nth (random (length list)) list)))
```

As primitivas de base do RUBA são definidas abaixo. Estas incluem também algumas primitivas criadas especificamente para as aplicações desenvolvidas da linguagem, mas que se revelaram de utilidade mais genérica.

```
;;
;; Generic Primitives
;; ------------------

;
; LISP special forms replacement
;

(defprim* 'and (&rest args)
  (loop for arg in args
        always (agent-eval agent arg)))

(defprim* 'or (&rest args)
  (loop for arg in args
```

```
;; --------------------

; Defines an agent class
(defun defagent (name &key behaviour knowledge initstate)
  "Defines as agents, ie creates a agent-class"
  (set name (make-instance 'agent-class :behaviour behaviour))
  (read-list-of-bindings (eval name) knowledge)
  (unless (null initstate) (set-var (eval name) '/state initstate)))

; Creates an agent instance
(defun make-agent (class &key blackboards knowledge id initstate)
  "Creates a working agents, given it's class"
  (let ((a (make-instance 'agent :blackboards blackboards)))
    (clone-from a class)
    (if (numberp id) (setf (get-id a) id))
    (unless (null knowledge) (read-list-of-bindings a knowledge))
    (unless (null initstate) (set-var a '/state initstate))
    (set-var a '/myself a)
    a))

; Creates a blackboard instance
(defun make-blackboard ()
  "Creates a blackboard"
  (make-instance 'blackboard))

; Runs the agents [MAIN PROGRAM]
(defun run (agent-list)
  "Randomly gives tokens to the given agents; endless"
  (let* ((nagents (length agent-list))
         (token (random nagents)))
    (loop (setf token (new-token token nagents))
          (give-token (nth token agent-list)))))

; Defines a primitive [global scope]
(defmacro defprim (name args &rest body)
  "Defines a primitive"
  `(setf (gethash ,name *primitives*) #'(lambda (agent ,@args) ,@body)))

; Like defprim but *doesn't* eval-agent its arguments
(defmacro defprim* (name args &rest body)
  "Defines a special primitive (primitive*)"
  `(setf (gethash ,name *primitives*)
         (list :special #'(lambda (agent ,@args) ,@body))))

(defun dump-prims (&optional (str t))
  "Dump all primitives"
  (format str "~&RUBA Primitives (~a):~&" (hash-table-count *primitives*))
  (maphash #'(lambda (key val)
               (format str "  ~a ~[~;(is special)~] args: ~a~&"
                       key
                       (if (and (listp val) (find :special val)) 1 0)
                       (rest (second (function-lambda-expression
                                (if (listp val) (first (last val))
                                  val))))))
           *primitives*))
```

Seguem-se algumas funções de baixo nível usadas ao longo do RUBA. A função `get-tag` é útil para fazer o *parsing* de uma lista de pares *keyword* argumento(s), à semelhança dos argu-

```
  "Finds a post given the specifications it must obey"
  (let ((post (find predicate (get-board (nth-bb bb-id a))
                    :test #'(lambda (pred msg)
                              (and (if to-me (is-for-me a msg) t)
                                   (if from-me (is-from-me a msg) t)
                                   (or (null predicate)
                                       (equal pred (ppred msg)))
                                   (or (null test)
                                       (apply test (list msg))))))))
    (if remove (rem-post a bb-id post))
    post))

(defmethod pick-post ((a agent) bb-id &key predicate remove)
  "Picks a random post from a set that satisfies the requirements"
  (let ((post (pick-random-atom (mapcan #'(lambda (post)
                                            (if (and (is-for-me a post)
                                                     (not (is-from-me a post))
                                                     (if (null predicate) t
                                                       (equal (ppred post) predicate)))
                                                (list post)))
                                        (get-board (nth-bb bb-id a))))))
    (if remove (rem-post a bb-id post))
    post))

(defmethod is-for-me ((a agent) post)
  "Is this post for me?"
  (and (not (equal (psource post) (get-id a) ))
       (member (pdest post) (list '* (get-id a)))))

(defmethod is-from-me ((a agent) post)
  "Is this post from me?"
  (equal (psource post) (get-id a)))


;
; Agent dump methods
; ~~~~~~~~~~~~~~~~~~

(defmethod dump ((a agent) &optional (str t))
  "Dumps the agent contents"
  (format str "~&Agent ID #~a (~a) Profile~&  Blackboards: ~a~&  Knowledge:~&"
          (get-id a) a (get-blackboards a))
  (maphash #'(lambda (key val) (format str "    ~a = ~a~&" key val))
           (get-knowledge a)))

(defmethod dump ((lst list) &optional (str t))
  "Dumps a list of 'things' (either blackboards or agents)"
  (mapcar #'dump lst))
```

A interface do RUBA com o exterior é efectuada primariamente pelas funções e *macros* que se seguem. Estas permitem definir e criar agentes, tal como criar *blackboards*, etc. Na parte III do relatório pode ser encontrada a explicação de cada um deles, excepto a última (`dump-prims`) que imprime informações sobre cada uma das primitivas definidas no RUBA.

```
;;
;; High Level Functions
```

```
        while (check-state a statement)
        do (eval-rule a rule)))

(defmethod check-state ((a agent) statement)
  "Check if statement corresponds to current state"
  (let* ((onstate (get-tag :onstate statement :multi t))
         (state (if (null (second onstate))
                    (get-var a '/state)
                  (get-var a (second onstate)))))
    (if (null onstate) t (member state (first onstate)))))

; [IMPORTANT]
(defmethod eval-rule ((a agent) rule)
  "Evaluates a single rule"
  (let ((if-cond (get-tag :if rule))
        (then-act (get-tag :then rule))
        (else-act (get-tag :else rule))
        (dest-state (get-tag :gostate rule :multi t))
        (else-state (get-tag :elsestate rule :multi t)))
    (if (or (null if-cond) (agent-eval a if-cond))
        (prog1 (unless (null then-act)
                       (agent-eval a then-act)
                       (format t "~&THEN-Activated rule ~a~&" rule))
               (unless (null dest-state)
                       (set-var a (if (null (second dest-state))
                                      '/state (second dest-state))
                                  (first dest-state))
                       (format t "~&Going to state ~a~&" (first dest-state))))
      (prog1 (unless (null else-act)
                     (agent-eval a else-act)
                     (format t "~&ELSE-Activated rule ~a~&" rule))
             (unless (null else-state)
                     (set-var a (if (null (second else-state))
                                    '/state (second else-state))
                              (first else-state))
                     (format t "~&Going to state ~a (ELSE)~&" (first else-state)))))))
```

Os métodos do agente que se seguem manipulam mensagens segundo o formato anteriormente referido. Algumas destas funções já foram descritas na parte III do relatório. O método `dump` pode aplicar-se a um `blackboard`, a um `agent` ou a uma lista destes objectos, imprimindo no ecrã detalhes sobre o estado interno do(s) objecto(s) em causa.

```
;
; Message handling methods
; ~~~~~~~~~~~~~~~~~~~~~~~~

(defmethod put-post ((a agent) dest bb-id pred contents)
  "Puts a post on a blackboard indexed as bb-id"
  (put-msg (nth-bb bb-id a)
           (append (list (get-id a) dest bb-id pred) contents)))

(defmethod rem-post ((a agent) bb-id post)
  "Removes a post"
  (rem-msg (nth-bb bb-id a) post))

(defmethod find-post ((a agent) bb-id &key predicate from-me to-me test remove)
```

```
;;
;; Class agent
;; -----------

; Class definition
(defclass agent (agent-class)
  ((id :accessor get-id :initarg :id :initform -1)
   (blackboards :accessor get-blackboards :initarg :blackboards)))

(defmethod nth-bb (n (a agent))
  "Returns the n-th indexed blackboard"
  (nth n (get-blackboards a)))

(defmethod clone-from ((a agent) (ac agent-class))
  "Clone rules and bindings from a given agent class"
  (setf (get-behaviour a) (get-behaviour ac))
  (maphash #'(lambda (k v) (set-var a k v)) (get-knowledge ac)))

(defmethod give-token ((a agent))
  "Gives a token to an agent"
  (format t "~&--- Agent #~a has token! ---~&" (get-id a))
  (run-agent a))

(defmethod agent-eval-args ((a agent) args)
  "Evaluates each argument of list args"
  (mapcar #'(lambda (s) (agent-eval a s)) args))

; [IMPORTANT]
(defmethod agent-eval ((a agent) sexp)
  "Agent expression evaluator"
  (cond ((is-sexp sexp)
         (let ((prim (gethash (first sexp) *primitives*)))
           (if (null prim)
               (apply (first sexp) (agent-eval-args a (rest sexp)))
             (cond ((is-special prim)
                    (apply (second prim) (cons a (rest sexp))))
                   (t (apply prim (cons a (agent-eval-args a (rest sexp)))))))))
        ((is-var sexp) (get-var a sexp))
        (t sexp)))

(defmethod apply-prim ((a agent) prim args)
  "Applies a primitive to a set of arguments [the agent instance arg is hidden]"
  (apply (gethash prim *primitives*) (cons a args)))


;
; RUBA II additions start here
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

; [IMPORTANT]
(defmethod run-agent ((a agent))
  "Run the agent's behaviour"
  (loop (set-var a '/rerun nil)
        (mapcar #'(lambda (statement) (eval-statement a statement))
                (get-behaviour a))
        (unless (get-var a '/rerun) (return))))

(defmethod eval-statement ((a agent) statement)
  "Evaluate a language statement"
  (loop for rule in (get-tag :rules statement :multi t)
```

```
  (mapcar #'(lambda (msg) (format str "  ~a~&" msg))
          (get-board bb)))

(defmethod dump-posts ((bb blackboard) &optional (str t))
  "Dumps a blackboard contents, pretty-printing as a post message"
  (format str "~&Blackboard (~a) Profile (post format):~&" bb)
  (mapcar #'(lambda (msg)
              (format str "  > src=~a dest=~a bb=~a pred=~a~&    contents=~a~&"
                      (psource msg) (pdest msg) (pbb msg) (ppred msg)
                      (pcontents msg)))
          (get-board bb)))
```

A classe `agent-class` é definida no excerto que se segue. Relembre-se que as instâncias desta classe representam classes de agentes, a partir das quais os agentes propriamente ditos são criados. Os *slots* desta classe contêm as regras da linguagem (`behaviour`) e as variáveis do agente (`knowledge`) implementadas com uma *hash table*. Os métodos `set-var` e `get-var` são usados para aceder a estas variáveis.

```
;;
;; Class agent-class
;; -----------------

; Class definition
(defclass agent-class ()
  ((behaviour :accessor get-behaviour :initarg :behaviour)
   (knowledge :accessor get-knowledge :initform (make-hash-table))))

; Sets a binding [IMPORTANT]
(defmethod set-var ((ac agent-class) var val)
  "Sets an agent's variable value"
  (setf (gethash var (get-knowledge ac)) val))

; Retrieves a binding [IMPORTANT]
(defmethod get-var ((ac agent-class) var)
  "Retrieves an agent's variable value"
  (gethash var (get-knowledge ac)))

; Reads a list of bindings values
(defmethod read-list-of-bindings ((ac agent-class) bindings)
  (mapcar #'(lambda (b) (set-var ac (first b) (second b))) bindings))
```

Na parte que se segue do código fonte, define-se a classe `agent`. As instâncias desta classe é que vão ser verdadeiramente os agentes. Esta classe é derivada da `agent-class`, herdando portanto as regras e as variáveis. Os *slots* que esta classe adiciona correspondem a uma identificação numérica com que cada agente é identificado, e a lista dos *blackboards* com que vai entrar em comunicação. Os métodos mais importantes neste bloco são o `give-token` que atribui um testemunho a um agente, o `agent-eval` que avalia uma expressão segundo as regras do RUBA, o `apply-prim` que aplica uma primitiva a uma lista de argumentos, o `run-agent` que executa o programa do agente, o `eval-statement` e o `eval-rule` que avaliam declarações e regras respectivamente.

```
;;
;; Global variables
;; ----------------

(defconstant *binding-prefix* #\/)

(defvar *primitives* (make-hash-table))

;;
;; Some macros [settable]
;; -----------

(defmacro psource (msg)    `(first ,msg))
(defmacro pdest (msg)      `(second ,msg))
(defmacro pbb (msg)        `(third ,msg))
(defmacro ppred (msg)      `(fourth ,msg))
(defmacro pcontents (msg)  `(nthcdr 4 ,msg))
```

De seguida define-se a classe `blackboard`, tal como os seus métodos. A classe contém apenas o *slot* `board` que representa a lista das mensagens actualmente no *blackboard*. Os três métodos têm a utilidade de, respectivamente colocar e retirar uma mensagem do *blackboard*, e limpar completamente o mesmo.

```
;;
;; Class blackboard
;; ----------------

(defclass blackboard ()
  ((board :accessor get-board :initform nil)))

(defmethod put-msg ((bb blackboard) msg)
  "Puts a message on a blackboard"
  (setf (get-board bb) (append (get-board bb) (list msg))))

(defmethod rem-msg ((bb blackboard) msg)
  "Removes a message from a blackboard"
  (setf (get-board bb) (remove msg (get-board bb))))

(defmethod clear ((bb blackboard))
  "Clears a blackboard contents"
  (setf (get-board bb) nil))
```

Os métodos que se seguem, mostram no ecrã o conteúdo do *blackboard* a que são aplicados. A diferença entre estas duas funções é que a segunda usa o formato de mensagens atrás referido, explicitando o emissor, o destinatário da mensagem, etc.

```
;
; Blackboard dump methods
; ~~~~~~~~~~~~~~~~~~~~~~

(defmethod dump ((bb blackboard) &optional (str t))
  "Dumps a blackboard contents"
  (format str "~&Blackboard (~a) Profile:~&" bb)
```

A Código fonte do RUBA anotado

Neste apêndice apresenta-se o código fonte do RUBA devidamente anotado. O sistema RUBA está contido no ficheiro `ruba.lsp`. A utilização do RUBA num exemplo concreto requer que este ficheiro seja previamente carregado antes do ficheiro da aplicação. Pode-se fazer isto de duas formas: carregar o `ruba.lsp` manualmente, ou adicionar a linha

```
(unless (boundp '*RUBA*) (load "ruba.lsp"))
```

no princípio do código fonte da aplicação. Desta última forma, o `ruba.lsp` é apenas carregado uma vez. A função `(unruba)` força o carregamento do `ruba.lsp`.

O ficheiro `ruba.lsp` principia com o cabeçalho e algumas declarações preliminares (apenas são comentadas as funções mais relevantes, sendo as restantes para uso interno do programa):

```
;;;
;;; RUBA II -- Rule Based Agents
;;; ============================
;;; version 0.5        01-Oct-96
;;;
;;; History:
;;; --------
;;;   0.1 (19-Jul-96) -- Initial version; plain rules only.
;;;   0.2 (18-Aug-96) -- Added integrated state machine.
;;;   0.3 (31-Aug-96) -- Added rule keywords and "else" part;
;;;                      More primitives, mostly from examples.
;;;   0.4 (21-Sep-96) -- More philosophical keywords renaming 8-)
;;;                      Meta-agents support and more primitives.
;;;   0.5 (01-Oct-96) -- Language keyword :elsestate added;
;;;                      Agent internal state dumping;
;;;                      Blackboard method "clear" added;
;;;                      Primitive "rem-post" added.
;;;


;;
;; Prologue
;; --------

(use-package 'clos)
(defparameter *ruba* t)
(defun unruba () (makunbound '*ruba*))
```

Note-se que a variável global `*ruba*` especifica que o `ruba.lsp` já foi carregado.

De seguida define-se o caracter que prefixa todas as variáveis do agente: '`/`', e alguns selectores para o formato de mensagens, utilizado no exemplo das bolas e no problema da alocação de recursos. Estes selectores são implementados como *macros* para permitir a sua utilização como primeiro argumento de um `setf`.

---

potencialidades também em outras áreas científicas. Este facto é também consequência de a IA ser uma disciplina interdisciplinar.

O desenvolvimento do RUBA não termina com este trabalho. Os seus autores desejam continuar no desenvolvimento de ferramentas para a implementação de agentes. Os autores gostariam, por fim, de encorajar o desenvolvimento do RUBA, não só em termos de aplicações, como também na continuação da própria ideia. Quaisquer comentários sobre o RUBA, ou sobre este trabalho em geral, serão extremamente bem vindos.

- Como já foi referido anteriormente, os agentes adequam-se particularmente à área de simulação. A representação de um ambiente complexo usando agentes (dividindo o problema pelos agentes) apresenta algumas potencialidades na implementação de sistemas de simulação. Tal como no caso da teoria dos jogos, pode-se fazer uma análise de como aplicar a metodologia orientada por agentes com um certo grau de generalidade.

- Ao longo deste trabalho tem-se usado agentes para representar partes de um problema. Mas no problema dos horários, um agente é mais do que uma parte do problema. Um agente representa e, de certa forma, emula o comportamento de um docente na negociação das aulas. Este aspecto da emulação do comportamento humano (ou de outro ser, em geral) é também uma área que importa explorar. Na realidade os grupos de trabalho constituídos por pessoas funcionam desta forma. Cada elemento do grupo pode desenvolver trabalho mais ou menos individual, mas há sempre reuniões em que o grupo se encontra para debater os aspectos comuns do trabalho em questão. Note-se a semelhança com as sociedades de agentes.

- Neste trabalho foi dado alguma atenção a um modelo genérico de negociação. A sua viabilidade foi ainda testada em alguns exemplos práticos. No entanto faltou fazer uma sistematização e formalização do processo de negociação. A sua formulação actual não permite flexibiliza-la ou extende-la sem por em perigo o seu funcionamento. Com esta formalização seria possível estabelecer uma base a partir da qual processos de negociação poderiam ser sintetizados.

16 Reflexão conclusiva

Este trabalho começou com pouco ou nenhum conhecimento prévio sobre a área dos agentes. A organização deste relatório segue em grande parte a ordem cronológica do trabalho efectuado. A primeira coisa que se fez foi uma recolha bibliográfica sobre os trabalhos mais significativos. Após esse levantamento, procedeu-se a uma sistematização e clarificação da definição do que é um agente. Após alguma experimentação em problemas concretos, decidiu-se construir uma linguagem com a qual se podem implementar agentes. Foram ainda implementados alguns exemplos da utilização desta linguagem, em campos distintos, procurando salientar a independência do domínio desta metodologia baseada em agentes. A parte nuclear deste trabalho é portanto a apresentação dos agentes como nova metodologia.

Os autores deste trabalho congratulam-se com o facto de não terem explorado o assunto a ponto de esgotarem as possibilidades da linguagem desenvolvida. Os mesmos estão ainda convictos que o paradigma dos agentes vai constituir um dos conceitos mais importantes no campo da IA. Este paradigma atravessa horizontalmente todas as áreas da IA, e não só. A metodologia baseada em agentes não tem aplicação exclusiva na IA, apresentando

área da simulação, os agentes podem suportar sistemas de simulação da grande complexidade, em que cada componente da simulação pode incorporar comportamentos muito próprios.

- O planeamento é uma área clássica da IA. Os agentes podem também aqui trazer uma outra forma de olhar o problema. Uma hipótese de abordar a questão seria por exemplo, usar simulação para ensaiar alguns planos, antes de o sistema se decidir por um plano final. Outra abordagem possível é a idealizada por Minsky [17]. Um plano é representado por um agente, que cria um conjunto de agentes, cada um com um objectivo atribuído e podendo gerar mais sub-agentes, e assim sucessivamente. Para a execução de uma tarefa, gera-se uma árvore de agentes, cada um com uma tarefa específica, baseando-se ou não nos seus sub-agentes. Cada agente representa aqui uma tarefa, em vez de uma parte do problema, como anteriormente.

15 Pistas para trabalho futuro

Nesta parte do relatório apresentaram-se alguns exemplos da aplicação da metodologia orientada por agentes. Estes exemplos foram escolhidos mais ou menos ao acaso para mostrar a viabilidade desta metodologia em várias áreas. Ficam no entanto alguns pontos por explorar:

- Um destes pontos é uma análise mais sistemática da aplicação dos agentes na teoria dos jogos. Há uma classe de jogos constituídos por um conjunto de peças, cada um com um conjunto bem definido de regras. Pode-se representar cada peça por um agente, havendo uma estratégia individual, e uma estratégia global acordada entre todos os agentes. A teoria dos jogos é uma área onde os algoritmos de procura são usados extensivamente. Como já foi referido neste relatório, os agentes podem ser vistos como uma nova forma de fazer procura, e portanto estas duas áreas dos agentes e da teoria dos jogos terão certamente muito em comum.

- A arquitectura usada nos agentes implementados pelo RUBA seguem uma arquitectura fixa. Na verdade, tomou-se mais em consideração a programação dos agentes, do que a arquitectura. A construção de agentes a partir da arquitectura é uma área que merece alguma atenção. Ao longo do segundo capítulo de [18], a apresentação de alguns tipos de agentes é acompanhada das respectivas arquitecturas. Ao longo do capítulo são apresentados agentes puramente de reflexos, com estado interno, orientados por um objectivo, e finalmente orientados por uma função utilitária. Estes tipos são apresentados sucessivamente, incrementando módulos à arquitectura do agente. A arquitectura é uma forma clara e bastante abstracta de definir agentes. O problema associado a esta representação é o facto de se abstrair do funcionamento de cada módulo, e portanto faltar uma definição precisa do baixo nível.

peça do jogo por um agente. Cada peça está identificada com um conjunto bem definido de regras, de movimento e de tomada de peças adversárias. Adicionalmente pode-se incorporar uma estratégia de jogo aos próprios agentes. Por um lado, cada agente poderia ter a sua estratégia individual, mais uma estratégia global acordada por negociação com outro agentes. Cada "equipa" teria um *blackboard* próprio sobre o qual se discutia a estratégia. A questão que se põe é saber se um sistema construído com base nestas ideias, conseguiria jogar xadrez razoavelmente (pelo menos em relação ao nível de dificuldade do jogador ocasional). Os autores não têm uma resposta a esta pergunta, mas estão convictos que poderão dar resultados interessantes. Uma crítica comum aos programas de xadrez para computador é a "frieza" calculista dos mesmos. Ou seja, a procura ignora (em princípio) a estratégia, ou outra qualquer abstracção. O que importa é determinar jogadas que maximizem uma determinada heurística. Esta abordagem por agentes poderia tornar o jogo muito mais interessante, adicionando alguma faceta "humana" às peças do adversário[17].

- A geração de frases sintácticamente (em termos linguísticos) correctas pode ser efectuada usando agentes. A linguagem escrita tem um certo número de regras sintácticas, por exemplo a precedência de um verbo pelo sujeito, etc. Considere-se a representação de uma classe de palavras por um agente, por exemplo, adjectivos. Este agente teria conhecimento das regras associadas a adjectivos. A frase seria construída progressivamente no *blackboard* (à semelhança do exemplo do `FRUITCAKE`). Cada agente apenas colocava uma palavra da sua classe, se esta colocação satisfizesse as regras associadas a essa palavra. É claro que desta forma apenas se constroem frases sintácticamente correctas, ignorando por completo o significado. Se as palavras fossem escolhidas ao acaso, dentro de cada agente, então as frases seriam ridiculamente desprovidas de qualquer sentido. Considere-se agora o que seria necessário adicionar ao sistema de forma, a gerar também frases com algum sentido. Podia-se por exemplo afixar no *blackboard* a indicação do que se pretende da frase, em termos de significado. Assim, os agentes, para além de satisfazerem as regras sintácticas, poderiam escolher criteriosamente as palavras a usar. Estamos perante uma nova forma de encarar o problema clássico da geração de linguagem natural.

- A exploração do problema do *jobshop* levantou algumas pistas em relação à possibilidade de fazer simulação usando agentes. A possibilidade de incorporar informação específica sobre um componente da simulação em cada agente, permite construir complexos sistemas de simulação de uma forma modular. Da mesma forma que as metodologias de programação orientadas por objectos desempenham um papel importante na

[17] O *Battle Chess* é um jogo clássico de xadrez, mas onde as peças do jogo são "humanizadas", e o movimento de uma peça é acompanhado por uma animação. Imagine-se um jogo de xadrez, onde estas animações exibissem o real estado "emocional" do agente... Este aspecto lúdico poderia tornar o jogo muito mais estimulante.

13.4.3 Melhoramento da solução

A segunda fase da resolução do problema do *jobshop* tal como foi abordado, consistiria na utilização dos mecanismos de negociação entre os agentes para melhorar a solução factível encontrada, um pouco à semelhança daquilo que em métodos clássicos de optimização se chama "procura em vizinhança". Porém, e porque se adoptou tempos de leitura de duração fixa, seria necessário proceder ainda antes da negociação, à variação desses tempos para cada agente. Esse aumento de complexidade integra-se na política de desenvolvimento incremental dos programas escritos em RUBA, segundo a qual, a implementação de estruturas com o objectivo de melhorar a solução factível só se justifica se elas forem também aplicáveis ao problema completo, *i.e.*, com tempos variáveis. Apesar de não se ter passado a essa fase por manifesta falta de tempo (real), presume-se que se poderia chegar a um resultado bastante satisfatório com uma estratégia semelhante à apresentada no exemplo anterior, com o título de "O problema de alocação de recursos", dada a sua afinidade com o problema presente.

14 Discussão sobre outras aplicações da linguagem

Tal como a esmagadora maioria das metodologias, em qualquer área de estudo, a apresentação de novas formas de abordar problemas, gera eventualmente novos problemas. A área da aplicação de uma qualquer metodologia nunca ou raramente é total. O domínio preferencial de aplicação dos agentes está no tipo de problemas, para os quais uma representação distribuída do conhecimento é a mais apropriada. Mas para que esta distribuição se faça facilmente é necessário que os parâmetros e variáveis atribuídas a cada agente sejam pouco dependentes entre si. Por exemplo, esta metodologia não é de todo apropriada para a resolução de equações diferenciais (no caso geral), onde o número de restrições entre variáveis é em geral muito elevado. No entanto, para o problema dos horários, em que é possível trocar aulas entre um grupo de docentes, independentemente dos restantes, e as restrições de cada docente são individualizadas, este método parece adaptar-se totalmente.

Os exemplos que foram explorados na secção anterior, procuraram ilustrar áreas muito distintas onde este tipo de metodologia pode ser aplicado. Há, no entanto, muitos outros problemas onde esta metodologia poderia ser aplicada com sucesso. Apresenta-se de seguida uma breve discussão sobre um conjunto de áreas em que a aplicação dos agentes poderia ser interessante e frutuosa.

- O xadrez é um campo já exaustivamente explorado pela IA, aplicando quase exclusivamente procura. Grande parte do trabalho nesta área centra-se em aumentar por um lado a eficiência dos algoritmos de procura, reduzindo as configurações a considerar, e por outro melhorar a qualidade das heurísticas. O que se propõe aqui é abordar o problema do xadrez de uma outra forma. Considere-se por exemplo representar cada

```
Got a new one : NIL
Agent #3 has token!
 ----->Putting back ... S
Seeking DN
Got a new one : DN
Inserting new paper DN
```

Os agentes #2 e #3 acabaram a leitura dos respectivos jornais, restituindo-os ao *blackboard*, tendo aquele último agente conseguido, adicionalmente, encontrar o jornal "DN" que pretendia ler a seguir. Por ter tido o azar de receber o *token* antes do agente #2 devolver o jornal "FT" ao *blackboard*, o agente #1 não conseguiu apanhá-lo desta vez. No entanto, ser-lhe-á dada mais uma oportunidade.

```
Agent #1 has token!
Seeking FT
Got a new one : FT
Inserting new paper FT
Agent #3 has token!
Agent #2 has token!
Seeking S
Got a new one : S
Inserting new paper S
```

No estado it just-pick da segunda jogada, o agente #1 e o agente #2 retiram do *blackboard* os jornais que pretendiam ler.

No final, as listas de leitura são mostradas, exibindo-se assim a solução final, que se verifica ser factível dadas as "restrições tecnológicas" (nome técnico das restrições que cada agente tem de obedecer, neste caso a ordenação e a duração da leitura de cada jornal) impostas de início:
Para as *wish-list*'s:

```
Wish list for agent #1 is (FT DN S)
Wish list for agent #2 is (FT S DN)
Wish list for agent #3 is (S DN FT)
```

Obteve-se a seguinte solução:

```
Reading list for agent #1 is (NIL FT NIL DN S)
Reading list for agent #2 is (FT S DN)
Reading list for agent #3 is (S DN FT)
```

Como se esperava de início, não se chegou a uma solução óptima (como seria p.ex. #1:(FT, NIL, DN, S); #2:(NIL, FT, S, DN); #3:(S, DN, FT)).

13.4.2 "*Faites vos jeux!*"

Uma vez provido cada agente da sua *wish-list*, e das regras que governam o seu comportamento, restava dar-lhes vida e esperar que entre eles chegassem a uma solução. Da execução do programa *Jobshop* listado em apêndice, resultou o seguinte *output* (abreviado e comentado, como nos exemplos anteriores, por motivos de maior legibilidade):

```
**** TIME : 1 ****
Agent #2 has token!
Seeking FT
Got a new one : FT
Inserting new paper FT
Agent #1 has token!
Seeking FT
Got a new one : NIL
Agent #3 has token!
Seeking S
Got a new one : S
Inserting new paper S
```

Acabada a fase de *drop-and-pick*, tem-se os agentes #2 e #3 a ler os jornais pretendidos. Segue-se a fase de *just-pick*:

```
Agent #3 has token!
Agent #1 has token!
Seeking FT
Got a new one : NIL
Inserting new paper NIL
Agent #2 has token!
```

O agente #1 não conseguiu encontrar no *blackboard* o jornal "FT" (a ser lido presentemente pelo agente #2), o que o leva a dar como perdido este *slot* de tempo por via da inserção de *NIL* na lista de leitura.

```
**** TIME : 2 ****
Agent #1 has token!
Seeking FT
Got a new one : NIL
Agent #2 has token!
 ----->Putting back ... FT
Seeking S
```

```
            (get-time /time)
            (get-next-paper /paper))
    :then (do (pick-paper-from-bb /paper /new-slot)
              ;; * It is now that NIL is inserted if no suitable
              ;;   paper is found on bb.
              (insert-new-paper /time /new-slot)))))
[... snip ...]
```

Para que, por um lado o agente tenha a oportunidade de "jogar", e por outro se efective a transição de estado, é essencial o *Meta-Agent* proceder à passagem do "testemunho" e à mudança do valor da variável *state* de cada agente.

```
[... snip ...]
(:onstate (running)
 :rules (:if (not (all-agents-happy))
         :then (do (increase /elapsed-time)
                   (remove-time)
                   (post-time)
                   (set-state drop-and-pick)
                   (run-every-agent /sons)
                   (set-state just-pick)
                   (run-every-agent /sons)
                   (check-for-happy-agents)
                   (rerun))
         :elsestate ending))
[... snip ...]
```

5. No ponto anterior são visíveis no código as chamadas à função *insert-new-paper* destinadas a registar a leitura de um jornal. Note-se que só no estado *just-pick* é que o agente, como que desesperado por nem neste, nem no estado *drop-and-pick* ter sido capaz de encontrar o jornal pretendido, assinala com *NIL* a impossibilidade de o ler nesta jogada.

6. Fazemos mais uma vez referência ao código apresentado no ponto 4, em especial às linhas em que consta o seguinte trecho:

```
;; * If there are no more papers to read ...
   (:if (not (get-next-paper /paper))
    :then (post-happy /already-read-list)))
```

O *Meta-agent* verifica regularmente, no fim de cada jogada, se todos os agentes se encontram satisfeitos (*happy*) o que significa que já todos leram os jornais que pretendiam.

```
                :rules (:then (do (let /sons (make-agents))
                                  (fill-blackboard)
                                  (post-time)
                                  (run-every-agent /sons)
                                  (rerun))
                              :gostate running))
[... snip ...]

(defun fill-blackboard ()
   (mapcar #'(lambda (post)
               (put-post agent '* 0 'to-pick post))
      '(FT S DN)))
```

4. Para distinguir cada uma das rondas existentes em qualquer jogada, deu-se os nomes de *drop-and-pick* e de *just-pick*, identificando-as com dois estados homónimos da máquina de estados incorporada em cada agente. Por se achar que os comentários contidos no código são suficientemente esclarecedores, transcreveu-se *verbatim* o trecho responsável pela implementação dessa máquina de estado:

```
[... snip ...]
;;*** First state of each round
(:onstate (drop-and-pick)
 :rules
   ;; * Whenever he's sick of reading -> drop
   (:if (and (get-time /time) (am-I-reading) (sick-of-reading /time))
    :then (put-away-paper))
   ;; * Just in case there's something interesting on bb -> pick
   (:if (and (get-time /time)
             (get-next-paper /paper)
             (pick-paper-from-bb /paper /new-slot))
    :then (insert-new-paper /time /new-slot))
   ;; * If there are no more papers to read ...
   (:if (not (get-next-paper /paper))
    :then (post-happy /already-read-list)))

;; *** Second state of each round.
(:onstate (just-pick)
 :rules
   ;; * If he didn't get a paper while in drop-and-pick state.
   (:if (and (not (am-I-reading))
```

13.4.1 Da teoria à prática

Convinha uma vez mais lembrar que a linguagem RUBA não foi escrita como solução de nenhum problema particular. Da mesma forma, este problema de *jobshop scheduling* não foi inventado pelos autores deste trabalho como um exemplo destinado a comprovar os pontos de vista destes em relação àquela nova linguagem. Daí que seja de certa forma surpreendente e compensador observar, como se poderá ver já a seguir, que o programa escrito em RUBA se apresenta bastante escorreito, por serem facilmente identificáveis as regras do "jogo" expostas acima.

1. Na declaração de cada agente é feita uma inicialização explícita da sua lista de jornais a serem lidos:

```
(defun make-agents ()
   (list (make-agent reader-agent :id 1 :blackboards (list bb)
            :knowledge '((/wish-list (FT DN S))))
         (make-agent reader-agent :id 2 :blackboards (list bb)
            :knowledge '((/wish-list (FT S DN))))
         (make-agent reader-agent :id 3 :blackboards (list bb)
            :knowledge '((/wish-list (S DN FT))))))
```

2. Aquilo a que se chamou anteriormente de cronograma não é mais do que uma lista própria de cada agente, em que a posição de um título reflecte o instante no tempo em que o agente leu o jornal respectivo. Por exemplo, o resultado

```
(NIL FT NIL DN S)
```

   obtido para o agente 1 declarado acima, interpreta-se como "Na jogada 1 o agente não leu nenhum jornal; na jogada 2 leu o seu primeiro jornal FT; na jogada 3 voltou a não conseguir ler o jornal pretendido; nas jogadas 4 e 5 leu os seus segundo e terceiro jornal, dando-se como satisfeito".

3. Como nalguns outro jogos, neste "jogo" há uma entidade representativa da "banca", a que chamámos de *Meta-Agent*, que tem a particularidade de ser "pai" dos agentes jogadores, e que tem a função de arbitrar as suas jogadas. Adicionalmente, inicializa o *blackboard* dispondo nele um exemplar de cada jornal a ser lido. No seguinte excerto do código do *Meta-Agent* são visíveis essas características:

```
[... snip ...]
:behaviour '((:onstate (init)
```

por uma certa ordem, reduzindo ao mínimo o tempo que cada leitor espera entre jornais. A distribuição óptima é aquela que leva a que se termine mais cedo a leitura dos jornais por todos os agentes. Como o problema é por natureza NP completo, não se pode ter a pretensão (recorrendo aos meios computacionais actuais) de encontrar a solução óptima em tempo real. Daí que se opte geralmente por encontrar soluções sub-óptimas.

Nesta modesta abordagem do problema, resolveu-se dividi-lo em duas partes (1. encontrar uma solução factível, 2. melhorar essa solução), tendo criado agentes que resolvem a primeira, e esboçado algumas ideias com vista a resolver a segunda. Como a factibilidade de uma solução não depende dos tempos de leitura de cada jornal, tomou-se a opção de, numa primeira fase, fazer com que esses mesmos tempos fossem todos iguais a um intervalo fixo definido mais adiante.

Como o título acima prenunciava, encarou-se o problema como se de um jogo de cartas se tratasse, em que cada leitor é um jogador, e cada jornal uma carta. As regras são as seguintes:

1. Cada jogador tem uma tabela com a ordem segundo a qual pretende ler os jornais, assim como um pequeno cronograma em que vai registando os períodos no tempo em que aqueles jornais são lidos;

2. Convenciona-se que cada leitor lê o seu jornal durante o tempo que medeia entre duas jogadas;

3. De início todos os jornais estão na mesa (*blackboard*), e os jogadores de mãos vazias;

4. Cada jogada é constituída por duas rondas, em que na primeira todos os jogadores podem pôr e tirar jornais da mesa, e na segunda só os podem tirar. Em cada ronda os jogadores intervêm uma só vez, sendo chamados por ordem aleatória. Desta maneira torna-se possível a cada um deles ver os jornais postos por todos os outros em cima da mesa (isto se ninguém os tirar de lá antes ...).

5. Se um jogador, numa qualquer ronda de qualquer jogada, encontrar o jornal que lhe interessa ler nessa altura, tira-o da mesa e regista no seu cronograma o nome desse jornal; caso não o encontre, assinala que nesse intervalo de tempo não conseguiu ler nenhum jornal;

6. O "jogo" termina, quando dos cronogramas de todos os agentes constarem todos os jornais que cada um deles se propunha a ler.

Uma vez definidas as regras deste pequeno jogo, interessa agora explicar de que forma é que, aproveitando as facilidades oferecidas pela linguagem RUBA, se implementou um sistema que fosse capaz de chegar ao resultado pretendido, ou seja, a uma solução factível do problema.

```
  Knowledge:
    /ALLOC = (2 3 1)
    /FREE = 0
    /TOTAL = 6
Agent ID #2 (#<AGENT #x07C75C5C>) Profile
  Blackboards: (#<BLACKBOARD #x07C73416>)
  Knowledge:
    /ALLOC = NIL
    /FREE = 6
    /TOTAL = 6
Agent ID #3 (#<AGENT #x07C75E1C>) Profile
  Blackboards: (#<BLACKBOARD #x07C73416>)
  Knowledge:
    /ALLOC = (2 1 3)
    /FREE = 0
    /TOTAL = 6
Agent ID #4 (#<AGENT #x07C75FDC>) Profile
  Blackboards: (#<BLACKBOARD #x07C73416>)
  Knowledge:
    /ALLOC = (1 2 3)
    /FREE = 0
    /TOTAL = 6
```

Como se pode observar pelo valor da variável `/alloc` para cada agente, a alocação de recursos corresponde à configuração acima referida.

Mais uma vez sublinha-se o facto de esta metodologia não garantir nem pretender obter a solução óptima. Neste exemplo apenas se usou a solução óptima para aferir da qualidade da execução do sistema. O que importa reter deste exemplo é por um lado, o facto de esta linguagem poder resolver problemas deste tipo, e por outro toda a flexibilidade que é possível incorporar em cada agente. Por flexibilidade entende-se a possibilidade de acrescentar restrições ao problema (limitar os produtos alocáveis a um recurso, por exemplo), adicionar indicadores de qualidade (por exemplo, penalizar um elevado número de produtos num só recurso), etc.

13.4 Um jogo de cartas chamado *jobshop*

Na tentativa de emular o comportamento do humano quando se põe um problema de distribuição de recursos escassos, considerou-se o problema clássico de *jobshop scheduling* [8]. Um dos *toy problems* que costuma servir como primeira introdução a esta área, é o de encontrar a distribuição óptima de jornais por um determinado número de leitores que exigem lê-los

```
                        (rem-my-complaint)
                        (do-trade /proposal))
                   :gostate idle)

              (:if /proposal  ; Implicit: a proposal not accepted but recv
                   :then (post-reject /proposal)))

    (:onstate (proposed)
     :rules (:if (accepted /trade :remove t)
                 :then (do-trade /trade)
                 :gostate idle)

              (:if (rejected /trade :remove t)
                   :gostate idle)

              (:then (reject-proposals))))

  :initstate 'idle)
```

Os agentes são iniciados sem nenhum produto alocado. Os produtos pretendidos são colocados *blackboard*, de onde os agentes os vão buscar. A variável do agente `/total` estabelece a dimensão total do recurso. Os produtos alocados a um determinado agente são coleccionados numa lista atribuída à variável `/alloc`, e a variável `/free` contém o espaço livre. As mensagens trocadas seguem o formato habitual, onde o conteúdo das queixas é (`<left>` . `<have>`), em que (`<left>` é o espaço que o queixoso ainda tem livre, e `<have>` uma lista dos produtos que tem alocados). O conteúdo das propostas e respectivas aceitações/rejeições é (`<give> <take>`) (`<give>` e `<take>` são respectivamente a dimensão do produto que se oferece e do que se recebe em troca). Note-se que há casos em que não é necessário fazer uma troca, mas apenas passar um produto de um recurso para o outro. Se assim for, um dos campos `<give>` e `<take>` toma o valor `nil`, conforme o caso.

Experimentou-se um problema de teste que consistiu em quatro recursos de dimensão 6, e em três produtos de cada uma das dimensões 1, 2 e 3. A solução óptima é atribuir um produto de cada uma das dimensões 1, 2 e 3, de modo a perfazer o limite de 6, para três recursos, deixando o quarto totalmente livre. Após correr o sistema durante algum tempo (o suficiente para se chegar a um estado estacionário, onde deixa de haver trocas de produtos), a expressão `(dump agents)` devolveu os seguintes resultados (o *output* foi editado por motivos de brevidade):

```
Agent ID #1 (#<AGENT #x07C75A9C>) Profile
  Blackboards: (#<BLACKBOARD #x07C73416>)
```

forma clara e que as soluções sejam razoavelmente boas. É importante manter em mente que esta abordagem por agentes não procura necessariamente soluções óptimas para problemas estereotipados, mas sim soluções razoavelmente boas (sub-óptimas) para problemas, privilegiando a representação, a modularidade, e portanto a flexibilidade.

A solução adoptada para o comportamento dos agentes é muito semelhante à gestão da negociação usada no problema das bolas. O código fonte foi simplesmente copiado, e alterado a partir daí, tornando-o específico a este problema. Os agentes negoceiam entre si produtos, segundo o seguinte critério (que tenta assegurar a convergência para uma solução): melhorar (*i.e.*, diminuir o desperdício) dos recursos com menor margem livre. Desta forma, estamos a manter o desperdício ao mínimo, à custa do aumento de sobras nos recursos menos utilizados, que é precisamente o que se pretende.

O código fonte, apresentado abaixo, encarrega-se de definir os agentes utilizados:

```
(defagent 'resource
  :behaviour
  '(
    (:onstate (idle complained)
     :rules (:if (and (find-product /product)
                      (feasible /product))
                 :then (do (rem-post /product)
                           (use-product /product)
                           (rerun))))

    (:onstate (idle)
     :rules (:if (and (has-products) (has-free))
                 :then (do (make-complaint /complaint)
                           (post-complaint /complaint))
                 :gostate complained))

    (:onstate (complained)
     :rules (:if (and (get-complaint /complaint)
                      (good /trade /complaint))
                 :then (do (post-proposal /trade /complaint)
                           (rem-my-complaint))
                 :gostate proposed)

            (:if (and (get-proposal /proposal :remove t)
                      (acceptable /proposal))
                 :then (do (post-accept /proposal)
                           (reject-proposals)
```

```
--- Agent #8 has token! ---
--- Agent #4 has token! ---
--- Agent #2 has token! ---
Going to state IN-BOARD
--- Agent #1 has token! ---
--- Agent #2 has token! ---
--- Agent #
*** - PRINT: User break
1. Break> (print-board theboard)
OOOOOO*O
*OOOOOOO
OO*OOOOO
OOOOOOO*
OOOOO*OO
OOO*OOOO
O*OOOOOO
OOOO*OOO
NIL
1. Break>
```

Neste exemplo não foi implementado um meta-agente, pelo que é necessário interromper manualmente a execução do sistema. No fim do extracto pode-se encontrar uma representação do tabuleiro (os '*' representam as rainhas).

13.3 O problema da alocação de recursos

Neste exemplo, considera-se uma formulação simplificada do problema da alocação de recursos. Considera-se um certo número de recursos, de natureza unidimensional (barras de metal, por exemplo), e que pretende-se cortá-las de forma a minimizar o desperdício, maximizando as sobras. Entende-se por desperdício aqueles bocados de recurso que, por serem muito pequenos, não poderão ser utilizados na prática. Pretende-se aproveitar ao máximo os recursos disponíveis, procurando poupar o máximo de material (sobras). Temos assim um conjunto finito de recursos, cada um de uma certa dimensão, e um conjunto finito de produtos pretendidos, cada um com uma dimensão. O objectivo é aproveitar ao máximo o menor número possível de recursos a utilizar.

Há pelo menos duas formas de abordar este problema usando agentes. Uma delas é considerar os recursos como agentes, e a outra considerar os produtos como agentes. Decidiu-se pela primeira opção por motivos de simplicidade. Note-se que é muito mais fácil implementar o controlo do desperdício a nível de recursos, do que a nível de produtos. Note-se que não se está à procura da solução óptima, mas apenas que se consiga representar o problema de uma

```
(defagent 'queen-wb
  :behaviour '((:onstate (in-board)
                :rules (:if (recv-warn)
                            :then (move-to (pick-place)))

                       (:if (attacking /someone)
                            :then (send-warn /someone)))

               (:onstate (out-board)
                :rules (:then (move-to (pick-place))
                        :gostate in-board)))

  :initstate 'out-board)
```

Após testar cada um dos agentes, verificou-se que este último nunca chegava a uma solução em tempo útil, ao contrário do primeiro, que mais tarde ou mais cedo acabava por a encontrar. De seguida, apresenta-se um extracto (editado por motivos de brevidade) de um diálogo com o sistema (em CLISP):

```
> (main)
--- Agent #4 has token! ---
Going to state IN-BOARD
--- Agent #3 has token! ---
Going to state IN-BOARD
--- Agent #6 has token! ---
Going to state IN-BOARD
--- Agent #7 has token! ---
Going to state IN-BOARD
--- Agent #1 has token! ---
Going to state IN-BOARD
--- Agent #5 has token! ---
Going to state IN-BOARD
--- Agent #1 has token! ---
--- Agent #3 has token! ---
--- Agent #5 has token! ---
--- Agent #4 has token! ---
--- Agent #5 has token! ---

[... snip ...]
```

deste problema usando o RUBA revelou-se extremamente simples. Simplicidade esta que não impediu de se ter atingido uma solução, ultrapassando largamente as expectativas dos autores.

Usando uma metodologia orientada para agentes, cada rainha é representada por um agente (8 agentes). Este agente pode estar num de dois estados: no tabuleiro (`in-board`) ou fora dele (`out-board`). Inicialmente os agentes são todos colocados fora do tabuleiro. Um agente que está for ado tabuleiro, procura neste uma posição vaga. Se encontra-la, então passa a ocupa-la. Desta forma os agentes vão entrando no tabuleiro, sempre para posições validas (não se atacando). No entanto vai haver uma altura em que mais nenhum agente fora encontre posições vagas. Nesta situação envia uma mensagem para o *blackboard* pedido que se "agite" o tabuleiro (predicado `shake`). Um agente que esteja no tabuleiro, e que receba esta mensagem, procura outra posição vaga para onde se possa deslocar, retirando a mensagem do *blackboard* (ou seja, um `shake` apenas atinge um agente de cada vez). O agente que implementa este comportamento extremamente simples (dois estados e uma regra por estado!) é definido da seguinte forma:

```
; A conservative vacancy-based Queen
(defagent 'queen-vb
  :behaviour '((:onstate (in-board)
                :rules (:if (and (recv-shake)
                                 (find-vacancy /position /vacancy))
                            :then (move-to /vacancy)))

               (:onstate (out-board)
                :rules (:if (find-vacancy nil /vacancy)
                            :then (move-to /vacancy)
                            :gostate in-board
                            :else (send-shake))))

  :initstate 'out-board)
```

É curioso como algo tão simples consegue resolver um problema típico de procura (com uma taxa de ramificação elevada). É preciso não esquecer que falta a definição de todas estas primitivas. Mas o que é relevante é a representação de alto nível da linguagem, abstraindo-se dos pormenores de baixo nível das primitivas.

Podemos caracterizar este agente (`queen-vb`, em que "vb" significa *vacancy based*) como sendo conservativo, ou seja, apenas se desloca para posições válidas, não tomando o risco de se expor ao raio de acção dos outros agentes (*i.e.*, rainhas). Foi ainda testado um outro comportamento que ignorava o raio de acção dos outros agentes, procurando uma solução através do envio de mensagens de aviso aos agentes sob o seu raio de acção (predicado `warning`). Este agente (denominado `queen-wb`, de *warning based*) é definido como:

```
                                    (rerun)))))

  :knowledge '((/total-agents 3) (/happy-agents 0))
  :initstate 'init)
```

Quando um agente consegue obter uma bola da cor desejada, emite uma mensagem com o predicado `happy` destinada a este meta-agente. Quando este meta-agente recebe três mensagens destas, uma por cada agente, então dá o processo como terminado. Abaixo apresenta-se um fragmento (editado por motivos de brevidade) do *output* do sistema[16]:

```
--- Agent #0 has token! ---
Going to state RUNNING
--- Agent #3 has token! ---
Posting complaint GREEN
Going to state COMPLAINED
--- Agent #2 has token! ---
Posting complaint BLUE
Going to state COMPLAINED

[...snip...]

--- Agent #2 has token! ---
--- Agent #3 has token! ---
--- Agent #1 has token! ---
Posting HAPPY
Going to state DONE
```

Neste excerto pode-se observar a atribuição de testemunhos, tal como algumas indicações sobre mudanças de estado, envio de mensagens, etc. Em particular, note-se que o primeiro agente a obter testemunho é o agente 0, ou seja o meta-agente. Este agente atribui testemunhos aos agentes 1 a 3 por si criados, terminando o processo após todos os agentes estarem satisfeitos (ou seja, terem enviado uma mensagem `happy` ao meta-agente).

13.2 O problema das 8 rainhas

Este problema consiste na colocação de 8 rainhas num tabuleiro de xadrez de forma a que não se ataquem, ou seja, que nenhuma rainha esteja no raio de acção de outra. Embora seja apenas um "quebra-cabeças", a solução não é trivial, mesmo para o humano. A resolução

[16]Todos os exemplos apresentados foram corridos em CLISP num Amiga A3000. O RUBA corre também com sucesso no CL *Allegro* para o *Windows*.

```
                :then (do (post-accept /prop)
                          (reject-proposals)
                          (rem-my-complaint)
                          (do-trade /prop))
                :gostate idle)
          (:if (and /prop (dislike (object /prop)))
                :then (post-reject /prop)))

   (:onstate (proposed)
    :rules (:if (accepted /trade :remove t)
                :then (do-trade /trade)
                :gostate idle)
          (:if (rejected /trade :remove t)
                :gostate idle)
          (:if t :then (reject-proposals))))
  :initstate 'idle)
```

Um agente é colocado inicialmente no estado `idle`, transita para o estado `complained` após ter apresentado uma queixa, passa para o estado `proposed` quando efectua uma proposta, e retorna ao estado `idle` após ter recebido a resposta à sua proposta. A primitiva `like` é utilizada para aferir se a cor de uma bola corresponde à desejada, e a primitiva `dislike` permite recusar bolas que não agradam de todo ao agente. Na prática, esta última primitiva retorna sempre `t`, dado que apenas retornaria `nil` se o agente já estivesse satisfeito, mas nunca é chamada quando o agente fica satisfeito. A existência desta primitiva é somente um reflexo conceptual da diferença entre "gostar" e "desgostar".

As mensagens utilizadas seguem o mesmo formato descrito na parte anterior deste relatório, tal como as primitivas aí descritas.

Existe ainda definido um meta-agente que controla a execução do sistema, atribuindo os testemunhos e verificando quando a solução é encontrada:

```
(defagent 'meta-agent
  :behaviour '((:onstate (init)
                :rules (:then (do (let /sons (make-agents))
                                  (rerun))
                              :gostate running))

               (:onstate (running)
                :rules (:if (not (all-agents-happy))
                            :then (do (run-agents /sons :times 1)
                                      (check-for-happy-agents)
```

uma determinada cor. Inicialmente cada agente possui uma bola de cor diferente daquela que pretende. O objectivo do sistema é que os agentes negoceiem e troquem bolas entre si, de modo a satisfazer os desejos de cada um. Esta aplicação serviu para implementar alguns mecanismos básicos de negociação, tal como foi apresentada em fases anteriores deste relatório. Estes mecanismos são nomeadamente a afixação de queixas e troca de propostas.

A negociação processa-se da seguinte forma: se a cor da bola que um agente tem não corresponde à sua cor pretendida, então afixa uma queixa no *blackboard* indicando que se quer ver livre dela. Um outro agente, ao ver esta queixa no *blackboard* e se estiver interessado na bola correspondente, emite uma proposta para o *blackboard*, destinada ao emissor da queixa, propondo a troca da sua bola pela do agente queixoso. Note-se que esta proposta é dirigida a um agente específico. O agente queixoso responde a esta proposta de uma de duas formas: ou aceita a proposta ou rejeita-a. Se a aceita, então efectua logo a troca internamente. A mensagem de aceitação segue logo para o *blackboard*, dirigida ao outro agente, efectivando a troca. O outro agente, que está exclusivamente à espera de uma resposta à sua proposta, efectua a troca se a proposta foi aceite.

Para garantir o sincronismo da bola de cada agente, ou seja, não haver bolas criadas ou destruídas espontaneamente, é preciso garantir que, quando um agente emite uma proposta, este está incondicionalmente comprometido a efectivar uma troca se esta for aceite. Note-se a semelhança com o mecanismo de limitação a uma única negociação usada no ABSS. No entanto, esta aplicação permite mais do que uma negociação de cada vez.

A definição de um agente é a que se segue:

```
(defagent 'balls-agent
  :behaviour
  '(
    (:onstate (idle)
     :rules (:if (not (like /have))
                 :then (post-complaint /have)
                 :gostate complained)
            (:if (like /have)
                 :then (post-happy /have)
                 :gostate done))

    (:onstate (complained)
     :rules (:if (and (get-complaint /comp) (like (object /comp)))
                 :then (do (post-proposal /comp)
                           (rem-my-complaint))
                 :gostate proposed)
            (:if (and (get-proposal /prop :remove t)
                      (not (dislike (object /prop))))
```

12.4 Do ABSS para o RUBA

Como já foi referido anteriormente, a concepção do RUBA foi em grande parte baseada neste sistema. No entanto, este sistema ainda contém alguma complexidade excessiva que poderia comprometer a flexibilidade da linguagem. Nesta medida foram feitas algumas simplificações:

- A comunicação directa e síncrona entre agente compromete conceptualmente a ideia de os agentes serem pró-activos. No RUBA a comunicação é toda ela efectuada pelo *blackboard*. Os agentes têm a iniciativa de consultar ou afixar mensagens. O código de cada agente apenas pode correr quando este detém o testemunho.
- No ABSS o *blackboard* está dividido em áreas dependentes do domínio e do tipo de mensagens. Na concepção do RUBA uniformizou-se o conceito de mensagem e o *blackboard* é totalmente independente do domínio de aplicação (não poderia ser de outra forma).

Como nota marginal, o programa ABSS foi implementado em C++, usando naturalmente objectos para representar os agentes e o *blackboard*. No entanto, verificou-se na prática que o nível baixo do C++, em relação a outras linguagens, é demasiado inconveniente para este tipo de aplicações (senão para todas). Por este motivo, a implementação do RUBA foi feita em linguagem LISP. Esta linguagem permitiu uma maior flexibilização do código tal como um muito menor tempo de desenvolvimento, do que seria possível com o C++. Código LISP é em geral muito mais claro e melhor organizável do que em C++. O LISP é de facto a linguagem por excelência para a implementação das ideias da IA.

13 Aplicação da linguagem

Esta secção apresenta quatro exemplos da aplicação da linguagem. O primeiro corresponde a um problema de negociação de bolas, muito simples. De seguida resolve-se (com sucesso) o problema clássico das 8 rainhas ([18], pág. 64) usando agentes. Uma simplificação do problema da alocação de recursos é posteriormente explorado, usando negociação e algum código do problema das bolas. Por fim, faz-se alguma exploração sobre o problema do *jobshop*, embora bastante simplificado. Estes exemplos procuram explorar algumas potencialidades da linguagem, em campos bastante distintos, desde os "quebra-cabeças" até aos problemas de *scheduling*.

Pode-se encontrar em apêndice o código fonte completo destes exemplos.

13.1 O problema das bolas

Considere-se três agentes, cada um possuindo uma bola de uma determinada cor. Cada bola tem cor diferente das outras duas. Cada agente tem a intenção de obter uma bola de

Agente 0		
	dias	
horas	0	1
0		-
1		-
2		-
3	-	-

Agente 1		
	dias	
horas	0	1
0	-	-
1	-	-
2		
3		

Agente 2		
	dias	
horas	0	1
0		
1		
2	-	-
3	-	-

Figura 4: Nestas tabelas mostra-se (com um '-') as horas nas quais cada docente não pode dar aulas. Estas são portanto as restrições a que cada agente está sujeito.

Este fragmento do *output* do ABSS mostra cada uma das trocas feitas, indicando o agente A_0 (`A0`), o último agente da cadeia (`AN`) e a profundidade da negociação (`depth`, o número de agentes envolvidos na troca é `depth+1`).

Os horários globais, inicial e final, podem ser encontrados na figura 5. Note-se que todas as aulas problemáticas (que não podem ser leccionadas pelo docente) foram resolvidas.

Horário inicial		
	dias	
horas	0	1
0	0	*1*
1	*1*	2
2	*2*	*0*
3	*0*	1

Horário final		
	dias	
horas	0	1
0	0	**2**
1	**0**	2
2	**0**	**1**
3	**1**	1

Figura 5: Horário inicial, com as horas problemáticas em itálico, e o horário final, com as horas alteradas a cheio.

Embora este problema seja extremamente simples, e até de resolução fácil por inspecção, o sistema conseguiu resolvê-lo apenas com 4 iterações. os autores acreditam que este sistema é suficientemente poderoso e flexível para conseguir resolver problemas de maior dimensão e mais complexos, sem muito esforço adicional de programação. Por esforço adicional de programação, entende-se a implementação da complexidade necessária sem alterar os aspectos mais básicos da negociação.

1. Horário global reunindo todas as aulas de todos os docentes. Inicialmente todos os agentes consultam esta tabela obtendo daí o seu horário. Este horário, tal como os horários de cada agente são todos coerentes entre si.

2. Tabela de queixas contendo todas as queixas activas. De cada vez que um agente emite uma queixa, esta é colocada aqui.

3. Zona de troca de propostas onde os agentes afixam as propostas destinadas a todos os agentes.

A comunicação entre agentes é não só efectuada através do *blackboard*, mas também por comunicação directa entre agentes. Todas as trocas de propostas (privadas) destinadas a um agente específico são trocadas directamente e sincronamente (resposta imediata) entre agentes.

Fizeram-se algumas simplificações em relação à concepção original do sistema. Uma delas é a limitação de cada agente só poder estar envolvido numa única negociação. Como cada negociação pode envolver vários agentes, considerou-se desnecessariamente complicado a um agente sincronizar todas as propostas em que poderia estar envolvido. A segunda limitação é a de o sistema apenas admitir um único processo de negociação de cada vez. Esta limitação está relacionada com a anterior. A outra limitação é a de apenas ter sido implementada a avaliação estática de cada dia/hora. Ou seja, com esta arquitectura é imediato implementar uma avaliação da possibilidade de dar uma certa aula a um dia/hora que seja dependente do contexto (por exemplos das aulas vizinhas). No entanto, apenas foi implementada uma avaliação estática de cada dia/hora, baseada na tabela em causa, que é carregada logo que o agente é criado. Para evitar bloqueios no sistema, devido por exemplo a uma negociação envolvendo muitos agentes, implementou-se um mecanismo de terminar automaticamente uma negociação, por iniciativa dos próprios agentes. Este mecanismo é baseado na idade da proposta no *blackboard*.

Testou-se o sistema, primeiro com várias dimensões de horários para aferir da sua capacidade de resolver horários complexos, de lidar com negociações extensas, estabilidade, etc. Para efeitos ilustrativos apresenta-se de seguida a resolução de um horário pequeno, com dois dias e quatro horas, envolvendo três agentes. As preferências de cada agente constam da figura 4. Neste exemplo apenas se considerou preferências de valor 0 e -1, correspondendo respectivamente à possibilidade e impossibilidade de dar uma aula no dia/hora correspondente.

Um horário problemático para todos os agentes (figura 5) foi alimentado ao sistema. Após apenas 4 iterações (atribuição de testemunho) uma solução foi atingida. Neste período foram efectuadas apenas três trocas:

```
> Trade DONE: A0=0 AN=1 depth=1
> Trade DONE: A0=1 AN=2 depth=1
> Trade DONE: A0=0 AN=1 depth=1
```

até pode ser denominada de agente, e que tem a responsabilidade de pôr termo a negociações muito grandes, do ponto de vista de número de agentes envolvidos.

Uma solução muito mais interessante é a de haver uma meta-negociação sobre o término de um processo de negociação. Assim, cada agente teria possibilidade de intervir na resolução de saber se vale a pena ou não, continuar um dado processo de negociação. Esta meta-negociação poderia inclusive ser feita nos mesmos moldes da negociação normal, explicitando desta forma a validade conceptual da utilização desta metodologia baseada em agentes.

Retomando um ponto em aberto, importa referir que todas as decisões (afixar ou não nova proposta, afixar mais queixas, etc.) estão completamente encapsuladas dentro do agente, podendo não ser estáticas. Ou seja, durante o funcionamento do sistema, estas decisões podem ser tomadas em função do passado, e inclusive de estados emocionais dos agentes. Nesta perspectiva, estamos a tratar de agentes não só inteligentes, mas também emocionais, com capacidade de tomar decisões com base nessas emoções! Trata-se por fim de mais um passo na emulação do comportamento humano.

12.3 Implementação

A implementação deste sistema é anterior à construção do RUBA. No entanto podem - se notar bastantes semelhanças. O ABSS serviu para testar algumas hipóteses e ideias, e como se poderá notar, é em alguns aspectos mais complexo. Para o RUBA aproveitou-se o essencial, de forma a se conseguir um sistema simples e poderoso.

Cada agente representa um docente negociando aulas do horário com os restantes. No interior de cada um destes agentes temos:

1. Tabela de preferências indicando para cada dia e cada hora o grau de possibilidade que esse agente tem de dar uma aula. Convencionou-se que valores negativos significam dificuldade, proporcional ao seu valor absoluto, em dar uma aula a essa hora, valores positivos indicam que é favorável dar uma aula, e o valor 0 denota indiferença.

2. Horário local contendo apenas as aulas dadas pelo docente representado.

3. Outros dados relacionados com a negociação em curso.

A tabela de preferências é usada para determinar sobre a possibilidade de um agente aceitar ou rejeitar uma aula nessa hora/dia. No entanto o encapsulamento desta avaliação em cada agente permite construir avaliações muito mais complexas. Por exemplo, dinâmicas ao longo do tempo ou sensíveis a aulas vizinhas. Esta última possibilidade seria usada por exemplo para desencorajar a criação de furos entre as aulas.

A difusão de mensagens por todos os agentes baseia-se num *blackboard* comum a todos estes, estando dividido em três áreas:

12.2.2 Processo de negociação

Uma das características que definem um agente é a sua capacidade de tomar a iniciativa (intenções). Cada processo de negociação é assim iniciativa de um dado agente. O funcionamento do sistema é to tipo *token ring*, onde cada agente apenas pode ter a iniciativa de produzir uma acção quando é detentor de um testemunho. A forma como esse testemunho é passado não é por enquanto relevante, podendo ser cíclico (*round robin*) ou de natureza estocástica.

Um processo de negociação é desencadeado quando um agente tem uma aula a uma hora inconveniente, e tem portanto necessidade de mudar a sua localização temporal, trocando essa hora com outro agente, por exemplo. Assume-se que as alterações a efectuar no horário são todas deste tipo, ou seja, trocas de horas, entre dois ou mais agentes. Generalizando este conceito, verifica-se que o simples deslocamento da hora é uma troca dessa hora com um furo. O agente em causa afixa no *blackboard* essa hora que pretende alterada, sob a forma de uma *queixa*. Chamemos a este agente A_0.

Quando um outro agente toma poder do testemunho e encontra a referida queixa, analisa - a. Se há possibilidade de ter uma aula a essa hora, então significa que está disposto a trocar uma das suas aulas com o agente autor da queixa — A_0. Sendo assim, este agente (a que chamamos A_1) propõe a troca ao agente A_0. Esta proposta é formada não por uma, mas eventualmente por um conjunto de horas que este agente A_1 está disposto a dar em troca. Se a resposta deste agente for positiva, ou seja, uma das horas que A_1 propõe interessa a A_0, a troca é imediatamente efectuada, e a queixa de A_0 retirada do *blackboard*. Se, caso contrário, A_0 não aceitar nenhuma das horas propostas por A_1, uma de duas coisas pode acontecer :

1. O agente A_1 desfaz-se do testemunho, passando-o a outro agente, ou

2. o agente A_1 envia uma proposta ao *blackboard* para afixação.

O objectivo desta última opção é permitir trocas entre mais do que dois agentes, ou seja, um terceiro agente, digamos A_2 interessa-se por uma das horas propostas por A_1. Neste caso, A_2 efectua uma sua proposta ao agente original A_0, à semelhança do que A_1 fez anteriormente. Caso A_0 aceite a troca com uma destas horas, o processo de troca entre os três agentes é desencadeado.

Este processo é naturalmente extensível a um número arbitrário de N agentes. A troca consiste em o agente A_n ficar com uma hora proposta por A_{n-1}, em troca de uma hora que passa a ser ocupada por A_{n+1}. Para N agentes, temos a condição fronteira de A_0 ficar com uma hora de A_{N-1}.

Por motivos práticos é necessário haver um mecanismo de impedir cadeias de negociação muito grandes. Numa primeira abordagem, podemos limitar N a um número pré-determinado. A monitorização deste valor pode ser efectuada por uma entidade externa, que

informação a ser consultada por um conjunto de entidades) correspondem ao termo utilizado pela restante literatura. Esta arquitectura assemelha-se em muito à usada no RUBA, com excepção da comunicação directa entre agentes.

Um dos argumentos mais relevantes a favor do uso dos agentes é o de ser possível e vantajoso, do ponto de vista prático, condensar em cada agente a especificação das restrições referidas no último parágrafo da secção (2). Note-se que as chamadas restrições estruturais do problema não estão incluídas neste grupo, pois a sua satisfação é imperativa e de carácter imutável. Pretende-se que apenas as restrições de carácter ajustável sejam individualizadas em cada agente.

Como ponto de partida e de inspiração, foi usada a metodologia normalmente usada pelos humanos para a resolução deste problema. Na prática, é muito rara a construção de um horário a partir do nada (excepto o caso do primeiro horário). Normalmente usa-se o horário anterior e efectua-se sobre ele as alterações necessárias de modo a acomodar as novas necessidades a satisfazer. Por outro lado, cabe aos interessados levantar problemas e propostas de modo a resolver os seus problemas. As alterações normalmente efectuadas consistem em trocas de horas, entre dois ou mais intervenientes, nomeadamente os próprios docentes.

Estas são as ideias base que foram utilizadas na construção do sistema. Em poucas palavras, pretende-se que o sistema *emule* o comportamento dos humanos na resolução do problema dos horários.

O sistema proposto funciona efectuando processos de negociação e eventualmente concretizando as trocas necessárias de uma forma iterativa, partindo de um horário inicial já existente. Este processo de negociação envolve dois ou mais agentes, e no caso de cada um deles estivar satisfeito com uma proposta de troca, esta é efectuada.

Note-se que as restrições estruturais, sendo satisfeitas no horário inicial, como é assumido à partida, devem ser incondicionalmente satisfeitas após cada alteração, durante todo o período de funcionamento. Em termos formais, isto significa que as transições de estado devem manter a validade da solução. Ao nível da implementação, este requisito é cumprido pela própria forma como os dados são representados, e consequentemente os operadores que neles actuam.

12.2.1 Arquitectura

O sistema é formado por um conjunto de agentes e por um *blackboard*. Este último é responsável pelo armazenamento de informação comum aos agentes, como o horário actual e as propostas da autoria dos agentes, como será tratado adiante.

Cada agente representa um docente, contendo o seu horário, contendo apenas as suas aulas, e todo o mecanismo que permite a avaliação do horas inconvenientes.

Uma das vantagens desta metodologia é a possibilidade de adicionar posteriormente outros tipos de agentes, representando por exemplo as turmas ou outra entidade envolvida.

12.1 Enunciado do problema

Assume-se que um horário curricular é um conjunto em que cada elemento é constituído por

1. dia (valores discretos representando de *segunda* a *sexta*)
2. hora (valores discretos da hora do dia, podendo ou não ser intervalados de uma hora mas estando limitados inferior e superiormente)
3. sala (valores discretos representativos de cada sala)
4. docente (*mutatis mutandis*)
5. cadeira (*mutatis mutandis*)
6. turma (*mutatis mutandis*)

Cada elemento representa portanto uma *aula*. Este conjunto está sujeito a algumas restrições de natureza física, de modo a que seja factível na realidade:

1. Para cada terno $(dia, hora, sala)$ não pode haver mais do que uma aula atribuída;
2. A cada docente e a cada turma não pode estar atribuída mais do que uma aula por cada par $(dia, hora)$.

Para além das referidas restrições estruturais, há mais um variado conjunto de requisitos (ou interesses) que se pretendem satisfeitos. Embora não sejam tão inflexíveis como as anteriores, não deixam de, por vezes, serem de cumprimento obrigatório. Como exemplo podemos referir a necessidade de garantir uma ou mais horas de almoço para alunos e docentes, o inconveniente da existência de "furos" nos horários (horas antecedidas e sucedidas por aulas de inactividade), restrições de tempo, horas inconvenientes ou mesmo impraticáveis para docentes, etc. Estas restrições caracterizam-se por dois factores e que são argumento relevante para a adopção da metodologia usada (sociedade de agentes). Estas duas características são: (1) o facto de poderem não ser inflexíveis, ou seja, o seu não cumprimento não invalida a solução; (2) a diversividade destas, sendo difícil encontrar um suporte formal comum, e podendo ser incrementado sem que para isso seja necessário repensar o sistema.

12.2 A solução proposta

Como já foi referido anteriormente, a solução proposta consiste na utilização de uma sociedade de agentes inteligentes. A arquitectura do sistema é formada pelos agentes em si, com capacidade de inter-comunicação e por uma estrutura que acumula informação de interesse comum. Designamos esta estrutura por *blackboard*, pois as suas características (afixação de

Parte IV
Exploração

Nesta parte do relatório apresenta-se a exploração de algumas ideias sobre os agentes, em particular usando a linguagem RUBA. O trabalho constante desta parte não é cronologicamente posterior à concepção do RUBA. Na verdade a construção de uma linguagem para a implementação de agentes sempre foi o desígnio dos autores.

Uma simplificação do problema da construção de horários curriculares foi explorada. Desta forma foi possível obter ideias sobre que características deve a linguagem suportar em termos de agentes. Este exemplo mostra que a metodologia dos agentes é suficientemente abstracta para ser aplicada em qualquer linguagem. De seguida apresentam-se alguns exemplos da aplicação da linguagem RUBA à resolução de problemas concretos. A ideia aqui não é a de somente apresentar novas formas de resolver problemas antigos, mas sim ilustrar as potencialidades de uma representação baseada em agentes.

12 O problema dos horários

O ABSS (*Agent Based Schedule Solver* consiste num programa com o intuito de auxiliar a resolução do problema dos horários curriculares utilizando uma metodologia baseada em agentes inteligentes. Este programa foi desenvolvido anteriormente ao RUBA, sendo portanto uma implementação directa de agentes e do processo de negociação. Este projecto insere-se num processo de recolha de ideias com vista à construção da linguagem (RUBA). Portanto a abordagem seguida foi a de implementar um sistema baseado em agentes, aplicado a um problema concreto, e a partir daí inferir sobre que características e propriedades deveria ser baseada a linguagem RUBA.

O problema dos horários é extremamente difícil de resolver devido à sua forte natureza combinatória. O espaço de estados associado a este problema é portanto impraticável por meio dos métodos usuais de procura. Para a resolução deste problema é necessário recorrer a métodos que restrinjam o espaço da procura, sem com isso degradar significativamente a qualidade da solução.

De entre as variadas formas de utilizar agentes, escolheu-se adoptar a da sociedade de agentes. Esta alternativa consiste em ter um conjunto de agentes independentes, mas comunicando entre si, forma a resolver o problema em causa em conjunto. O modelo adoptado da sociedade de agentes é baseado na negociação entre os agentes. Cada agente representa "interesses" a serem satisfeitos, total ou parcialmente pela solução final. O termos "interesse" é aqui usado factor de qualidade do ponto de vista do agente em causa. O conjunto dos interesses de todos os agentes constitui uma heurística. A qualidade da solução final é função desta heurística.

como no caso dos agentes, esta última letra é especificada na altura em que o agente é criado (via `make-agent`), pela variável `/lastchar`.

Estes três exemplos serviram para ilustrar algumas ideias-base da linguagem, tal como as suas potencialidades. Na próxima parte deste relatório são apresentados alguns exemplos da aplicação desta linguagem a problemas concretos.

11 Análise crítica da linguagem e perspectivas para o seu desenvolvimento

Já foi discutido neste trabalho o quão vaga é a noção de agência. Actualmente verifica-se algum frenesim em chamar a quase tudo de agente, só porque se "mexe". Uma implementação de uma linguagem que pretenda suportar esta noção de agente, corre dois riscos: por um lado, se for construída a partir do nada, reduz-se a re-inventar uma linguagem convencional mesmo que com mecanismos específicos para agente, e por outro lado, se se pretende ser suficientemente genérico, corre-se o perigo de resultar uma linguagem demasiado vaga. A opção de implementar o RUBA num ambiente aberto como o LISP mostra que a linguagem possui inerentemente todas as potencialidades do LISP, e portanto não há necessidade de reinventar parte da linguagem.

Uma observação global do RUBA revela que os agentes criados são intrinsecamente racionais. Ou seja, o comportamento de cada agente é definido por um conjunto de regras. Embora a linguagem tenha sido criada com a ideia de ser o mais geral possível, esta apresenta algumas limitações, nomeadamente no que respeita à capacidade de aprender, à manifestação de emoções (no sentido referido nas primeiras partes deste relatório) e à reactividade (no sentido de Brooks). Ou seja, embora a linguagem permita implementar estas propriedades, seria útil uma extensão de forma a permitir a implementação destes conceitos de uma forma declarativa. À data da redacção deste relatório não há modelos suficientemente gerais de forma a implementar as referidas três capacidades, pelo menos sem comprometer a linguagem a qualquer um desses modelos. Qualquer uma destas capacidades mereceria um estudo *per se*, pelo que a sua análise escapa ao âmbito do presente trabalho.

desta vez da à máquina de estados integrada. Cada agente é inicializado com o estado `ready`. Neste estado, o agente aguarda expectante pela sua vez de participar no "jogo". Logo que as condições estejam reunidas, este coloca a sua letra no *blackboard*, e transita para o estado `done`. Neste último estado, o agente não faz rigorosamente mais nada. Uma implementação possível desta ideia é a seguinte:

```
(defagent 'fruitcake-agent
  :behaviour '((:onstate (ready)
                :rules (:if (or (on-bb /prevchar)
                                (equal /prevchar nil))
                            :then (put-bb /mychar) :gostate done)))
  :initstate 'ready)
```

Note-se a ausência premeditada de regras associadas ao estado `done`.

Na opinião dos autores esta segunda implementação para além de ser mais breve, é também mais elegante, e ilustra bem a motivação que levou à integração de uma máquina de estados no agente.

Um programa tradicional de IA para resolver um problema, termina a sua execução com a apresentação de uma solução. Nos anteriores exemplos a solução aparece no *blackboard* mas não há nenhum mecanismo de paragem do processo. Ou seja, o sistema global não tem consciência que o problema foi resolvido (ou neste caso, que a tarefa foi completada). Uma forma de contornar isto é usar um *meta-agente*, já referido quando se apresentou a atribuição controlada de testemunhos. Neste caso pretende-se um agente responsável por atribuir testemunhos e ter consciência que a tarefa foi executada. Um meta-agente é simplesmente um agente. O prefixo *meta-* denota apenas que se trata de um agente que lida com outros agentes. É uma abstracção que mostra alguma recursividade da arquitectura, e consequentemente o seu potencial acrescido.

O próximo exemplo que se apresenta consiste apenas na adição de um meta-agente ao exemplo anterior. Este meta-agente é responsável pela atribuição (estocástica) de testemunhos até que a palavra `FRUITCAKE` esteja completamente afixada no *blackboard*. Note-se que não é necessário fazer qualquer tipo de alteração aos agentes previamente definidos! O novo agente tem a seguinte definição:

```
(defagent 'meta-agent
  :behaviour '((:rules (:then (do (let /sons (make-list-of-agents))
                                  (run-agents /sons
                                              :until (on-bb /lastchar)))))))
```

Basicamente há apenas uma regra sem antecedente (e portanto de execução obrigatória) que atribui à variável `/sons` uma lista de agentes, e que de seguida faz atribuição estocástica de testemunhos até que a última letra da palavra `FRUITCAKE` seja afixada no *blackboard*. Tal

conjunto de variáveis iniciais. São estas variáveis que associam cada agente a uma letra da palavra `FRUITCAKE`:

```
(defparameter agents (list
  (make-agent fruitcake-agent :id 1 :blackboards (list bb)
              :knowledge '((/prevchar nil) (/mychar f)))
  (make-agent fruitcake-agent :id 2 :blackboards (list bb)
              :knowledge '((/prevchar f)   (/mychar r)))
  (make-agent fruitcake-agent :id 3 :blackboards (list bb)
              :knowledge '((/prevchar r)   (/mychar u)))
  (make-agent fruitcake-agent :id 4 :blackboards (list bb)
              :knowledge '((/prevchar u)   (/mychar i)))
  (make-agent fruitcake-agent :id 5 :blackboards (list bb)
              :knowledge '((/prevchar i)   (/mychar t)))
  (make-agent fruitcake-agent :id 6 :blackboards (list bb)
              :knowledge '((/prevchar t)   (/mychar c)))
  (make-agent fruitcake-agent :id 7 :blackboards (list bb)
              :knowledge '((/prevchar c)   (/mychar a)))
  (make-agent fruitcake-agent :id 8 :blackboards (list bb)
              :knowledge '((/prevchar a)   (/mychar k)))
  (make-agent fruitcake-agent :id 9 :blackboards (list bb)
              :knowledge '((/prevchar k)   (/mychar e)))))
```

A variável `agents` contém uma lista dos 9 agentes, com ID's de 1 a 9, fazendo todos referência ao mesmo *blackboard*, criado por:

```
(defparameter bb (make-blackboard))
```

Resta saber como se põe o sistema em marcha. A função `(run <list-agents>)` desempenha essa tarefa, usando a atribuição estocástica de testemunhos.

Posto o sistema em funcionamento, este não tem nenhum mecanismo explícito de paragem, continuando a atribuição de testemunhos, mesmo após a palavra ter sido completada no *blackboard*. Normalmente os ambientes LISP dispõem de um mecanismo (ex: `Ctrl-C`) para interromper a execução, e a partir daí podemos examinar o conteúdo do *blackboard*:

```
1. Break> (get-board bb)
(F R U I T C A K E)
```

No entanto, observe-se que o agente se comporta como se admitisse dois estados, e a colocação da sua letra no *blackboard* fosse consequência da transição de um estado para o outro. Sendo assim, propõe-se uma implementação da mesma ideia, mas fazendo recurso

necessário 9 agentes, um por cada letra. No primeiro exemplo, a classe de agentes a que estes pertencem, pode ser definida como:

```
(defagent 'fruitcake-agent
  :behaviour
  '((:rules (:if (and (on-bb /prevchar) (not (on-bb /mychar)))
             :then (put-bb /mychar))

            (:if (and (equal /prevchar nil) (bb-empty))
             :then (put-bb /mychar)))))
```

As primitivas de que este programa faz uso são as seguintes:

```
(defprim 'on-bb (obj)
  (not (null (find obj (get-board (nth-bb 0 agent))))))

(defprim 'put-bb (obj)
  (put-msg (nth-bb 0 agent) obj))

(defprim 'bb-empty ()
  (null (get-board (nth-bb 0 agent))))
```

Este programa é constituído por duas regras, não estando condicionadas por nenhum estado (e portanto são de avaliação incondicional). O agente possui duas variáveis pré-definidas, uma das quais indica qual a letra por que é responsável a sua colocação (`/mychar`) e a outra a letra que a precede (`/prevchar`). A primeira regra é activada caso a letra precedente conste no *blackboard* (`(on-bb /prevchar)`) e simultaneamente a sua letra ainda não esteja lá (`(not (on-bb /mychar))`). Se esta última condição não fosse considerada, cada agente repetiria mais do que uma vez a colocação da sua letra no *blackboard*. Quando esta conjunção se verifica, o consequente da regra é executado, o que consiste na colocação da letra do agente no *blackboard* (`(put-bb /mychar)`).

Observe-se que, se o programa do agente ficasse limitado a esta regra, no início, quando não há nenhuma letra no *blackboard*, nenhum agente teria a iniciativa de colocar a sua. Há um agente cuja letra é a primeira da palavra (`F`), e este deve ter a iniciativa de a colocar no *blackboard*. Isto é conseguido por via da segunda regra do programa. Aquele agente caracteriza-se por não ter nenhuma letra que preceda a sua, pelo que se convenciona que a variável `/prevchar` tome o valor `nil` para indicar este facto. Sendo assim, caso o `/prevchar` seja `nil` e o *blackboard* esteja vazio (`(bb-empty)`), o consequente da regra é executado, e a sua letra é colocada no *blackboard*.

De seguida, criam-se os 9 agentes encarregues de executar a tarefa proposta, todos eles pertencendo à mesma classe de agentes atrás definida, mas cada um com o seu próprio

9.6.3 Variáveis do agente

Cada agente possui algumas variáveis especiais que estão relacionadas com o funcionamento do próprio sistema. Uma delas já foi referida — `/state` — que embora se possa usar outra variável qualquer para designar o estado, é mais cómodo usar esta. A variável `/myself` é atribuída com o próprio agente, sendo uma forma elegante (e normalmente usada por linguagens orientadas por objectos) de as regras do agente terem acesso "por fora" ao próprio agente. O mecanismo de reavaliação de todas as regras pode ser despoletado pela primitiva `(rerun)`, mas o que esta primitiva realmente faz é pôr a variável do agente `/rerun` com o valor `t` ("verdadeiro"). Esta reavaliação apenas ocorre se no fim da avaliação de todas as regras, esta variável for "verdadeira". Assim se se colocar esta variável com o valor `nil` ("falso"), o efeito de um `(rerun)` é anulado.

9.6.4 Criação de um *blackboard*

A criação de um *blackboard* é concretizada pela função `make-blackboard`, que não toma nenhum argumento e que devolve um objecto correspondente ao *blackboard* criado.

9.6.5 Métodos do *blackboard*

O *blackboard* apenas admite dois métodos. Esta simplificação extrema do *blackboard* foi propositada, de forma a delegar aos agentes toda a "inteligência" do sistema. Como é habitual, o primeiro argumento destes métodos corresponde ao objecto em causa[15], que neste caso é o *blackboard*.

`(put-msg <bb> <msg>)`

`(rem-msg <bb> <msg>)` — Estes dois métodos adicionam e removem mensagens `<msg>` do *blackboard* `<bb>`, respectivamente.

`(get-board <bb>)` — Todas a mensagens constantes num *blackboard* são armazenadas numa lista. Este método devolve a lista completa.

10 Alguns exemplos ilustrativos

Nesta secção apresentam-se dois exemplos de aplicação da linguagem, ilustrando duas formas de realizar a mesma tarefa. O objectivo é colocar no *blackboard*, por ordem, as letras da palavra `FRUITCAKE`. Cada agente é responsável por colocar uma só letra, sendo portanto

[15]Ao contrário de outras linguagens orientadas por objectos, onde os métodos são "propriedade" dos objectos, o CLOS extende este conceito em que um método é escolhido em função da classe a que pertence um ou mais argumentos, a especificar pelo método.

O cabeçalho desta mensagem contém o ID `<source-id>` do agente (identificador inteiro definido pela palavra chave `:id` quando o agente é criado) que origina a mensagem, o ID `<destination-id>` do agente a quem se destina a mensagem em causa (ou '`*`' se se destinar a todos os agentes), a referência `<blackboard-id>` do *blackboard* onde esta mensagem é afixada (assume-se que todos os agentes utilizam a mesma indexação dos *blackboards*, mas de qualquer forma este parâmetro não é utilizado correntemente, constituindo apenas alguma redundância adicional), e um predicado `<predicate>` que exprime o tipo da mensagem (por exemplo, no caso da negociação exposta na parte seguinte, usa os predicados `trade`, `proposal` e `complaint`). O parâmetro `<contents>` precedido por "`.`" denota que todos os restantes elementos da lista constituem o conteúdo da mensagem, que é dependente da aplicação.

Resta a descrição dos métodos do agente que fazem uso deste formato de mensagem:

`(put-post <agent> <dest> <bb-id> <pred> <contents>)` — o envio de uma mensagem para o *blackboard* com índice `<bb-id>` destinada ao agente com ID `<dest>`. A mensagem caracterizada pelo predicado `<pred>` transporta `<contents>` como conteúdo (normalmente uma lista).

`(rem-post <agent> <bb-id> <post>)` — este método remove a mensagem `<post>` do *blackboard* com índice `<bb-id>`.

`(find-post <agent> <bb-id> [:predicate <pred>] [:from-me t] [:to-me t] [:remove t])` — este método procura no *blackboard* de índice `<bb-id>` a primeira mensagem que satisfaça todas as condições especificadas pelos argumentos opcionais: o `:predicate` requer que a mensagem tenha `<pred>` como predicado, o `:from-me` obriga a que a mensagem tenha origem no agente `<agent>`, e o `:to-me` especifica que a mensagem é destinada a este agente (tendo em conta que se o ID de destino for '`*`', esta condição é satisfeita). Se alguma mensagem é encontrada nestas condições, é devolvida a primeira, caso contrário devolve `nil`. A palavra-chave `:remove` controla se após uma mensagem ter sido encontrada, é automaticamente removida do *blackboard*.

`(pick-post <agent> <bb-id> [:predicate <pred>] [:remove t])` — este método é em tudo semelhante ao método anterior, com a excepção que neste caso uma de todas as mensagens que satisfazem as condições requeridas é escolhida aleatoriamente. Note-se que, embora apenas o argumento opcional[14] `:predicate` seja indicado, está implícito que a mensagem deve também ser destinada ao agente `<agent>`. Tal como no método anterior, se o valor do argumento da palavra-chave `:remove` for `true`, a mensagem devolvida é automaticamente removida do *blackboard*.

[14]Uma pequena nota na terminologia: em termos de LISP estes argumentos não são realmente denominados como opcionais, mas sim como *keyword arguments*. Os argumentos opcionais em LISP são diferentes destes. (ver por exemplo [10].)

É importante notar que esta função também cria um objecto CLOS, à semelhança da função `defagent` acima descrita, mas desta vez da classe `agent`. Esta classe é derivada da `agent-class`, possuindo portanto todas as variáveis e métodos desta. No momento da criação do agente, a função `make-agent` trata de copiar todos os dados comuns à classe de agentes, para o agente recém-nascido, nomeadamente o programa e as variáveis pré-definidas.

9.6.2 Métodos do agente

Expõem-se de seguidas os métodos mais importantes do agente:

`(run-agent <agent>)` — a atribuição de um testemunho a um agente é concretizada com este método. O agente que recebe este método executa uma passagem pelo seu programa. (Existe ainda um método de nome `give-token` mas que se limita a chamar `run-agent` e a imprimir uma mensagen indicativa da passagem de testemunho.)

`(agent-eval <agent> <expression>)` — este método avalia uma expressão RUBA, definida nos termos anteriormente referidos, como a chamada a primitivas, avaliação de argumentos, substituição de variáveis do agente, etc.

`(apply-prim <agent> <prim> <args>)` — este método aplica uma primitiva de nome `<prim>` a uma lista de argumentos `<args>` (que não serão avaliados), no contexto do agente `<agent>`.

`(set-var <agent> <variable> <value>)` — a atribuição de um valor `<value>` a uma variável `<variable>` do agente `<agent>` (não dispensa que o nome comece por '`/`', tanto neste método como em todos os outros que façam referência a variáveis do agente) é efectuada por meio deste método.

`(get-var <agent> <variable>)` — complementando o método anterior, este devolve o valor actual da variável `<variable>` do agente `<agent>`.

`(nth-bb <index> <agent>)` — este método devolve o *blackboard* com índice `<index>` da lista interna do agente `<agent>`.

Os métodos que se seguem fazem uso de um formato das mensagens (inspirado no KQML [7]) que transitam entre os agentes. Uma *mensagem* é representada no RUBA como uma lista de elementos, e o formato que será utilizado daqui para a frente é:

```
(<source-id> <destination-id> <blackboard-id>
 <predicate> . <contents>)
```

9.5 Definição de uma classe de agentes

A função[13] `defagent` define uma classe de agentes que partilham o mesmo programa `<program>`, um mesmo conjunto de variáveis iniciais e um estado inicial `<state>`:

```
(defagent '<name>
          [ :behaviour '<behaviour> ]
          [ :knowledge '( { (<variable> <initial-value>) }* ) ]
          [ :initstate '<state> ] )
```

O comportamento é especificado por `<behaviour>`, as variáveis iniciais são inicializadas aos pares (`<variable>`, `<initial-value>`) e o estado inicial pode ser especificado por `<state>`. Neste último caso, utiliza-se a variável de estado por defeito — `/state`. De outra forma, uma variável de estado poderia ser inicializada sob a palavra-chave `:knowledge`.

Após a execução desta função, um objecto do tipo `agent-class` é criado, representando a nova classe de agentes definida. Chama-se de novo a atenção para a distinção entre classes de agentes e classes de objectos do CLOS. Ou seja, embora a função `defagent` instancie um objecto da classe CLOS `agent-class`, este objecto *representa* uma classe de agentes.

9.6 Criação de agentes e de *blackboards*

9.6.1 Instanciação de um agente

A criação de agentes propriamente dita é efectuada por via da função `make-agent`:

```
(make-agent <agent-class>
            [ :blackboards (list { <blackboard> }* ) ]
            [ :id <id> ]
            [ :knowledge '( { (<variable> <initial-value>) }* ) ]
            [ :initstate '<state> ] )
```

Os *blackboards* com os quais o agente troca mensagens, são especificados pela lista de `<blackboard>`. No interior do agente estes *blackboards* são indexados por via do método `nth-bb` (ver adiante) a partir de 0, inclusive (0, 1, 2, 3...). Cada agente possui um inteiro que o identifica, especificado por `<id>`. Os restantes argumentos funcionam de forma idêntica aos da função `defagent`. A função `make-agent` devolve o objecto correspondente ao objecto criado, e portanto, deve ficar referenciado por alguma variável ou lista para posterior uso (caso contrário o mecanismo de *garbage collection* trata de lhe dar um destino menos desejado — morte súbita).

[13] Podia ser implementada como uma *macro*, mas não se vislumbrou nenhuma vantagem significativa.

```
(defprim <name> <args> . <body>)
```

que é sintácticamente idêntica a um `defun`[12]. Este tipo de primitivas implica implicitamente a avaliação dos argumentos previamente à chamada da função. O `<body>` constitui uma função LISP comum (uma expressão *lambda* para ser mais preciso), com a adição da passagem de um argumento escondido denominado `agent`. Este argumento é passado automaticamente para a primitiva, não figurando na lista de argumentos `<args>`. A definição de uma primitiva* é em termos de sintaxe idêntica:

```
(defprim* <name> <args> . <body>)
```

As primitivas são guardadas numa *hash-table* (indexada naturalmente pelo nome da primitiva) associadas ao símbolo global `*primitives*`. Este símbolo é definido por meio de um `defvar`, o que lhe confere algumas propriedades particulares (*special symbol*) mas que são irrelevantes no presente contexto.

9.4.1 Algumas primitivas básicas

Como já foi referido anteriormente, as expressões usadas nas regras da linguagem não suportam *macros* nem formas especiais. Por isso foi necessário criar algumas primitivas para desempenharem a mesma função, nomeadamente as primitivas `and` e `or`. Para agrupar várias expressões para execução sequencial (à semelhança da forma `prog` do LISP) usa-se a primitiva `do`. Descrevem-se de seguida outras primitivas exclusivas do RUBA:

`(let <variable> <value>)` — A atribuição de um valor `<value>` a uma variável `<variable>` faz-se por uso desta primitiva, que funciona de forma idêntica a um `setf` do LISP.

`(run-agents <agent-list> [:while <while-expr>] [:until <until-expr>] [:times <num>])` — Esta primitiva atribui testemunhos estocasticamente até uma das três seguintes condições se verificar:

- Ser usada a palavra chave `:while` e a expressão `<while-expr>` for "falsa";
- Ser usada a palavra chave `:until` e a expressão `<until-expr>` for "verdadeira";
- Ser usada a palavra chave `:times` e terem sido atribuídos `<num>` testemunhos.

`(rerun)` — A execução desta primitiva força o agente a re-executar a sua programação. É equivalente a uma atribuição de um testemunho ao mesmo agente, com a vantagem de não encher a *stack* (se um agente atribuísse o testemunho a ele próprio, indefinidamente, então mais tarde ou mais cedo a execução rebentava com um *stack overflow*.)

[12] A primitiva é identificada pelo nome `<name>` tomando os argumentos `<args>` (uma lista) e sendo definida por `<body>`. O ponto antes do `body` indica que todas as expressões LISP que sucedem ao ponto são consideradas como o corpo da função. Esta notação é extensivamente usada em [10].

sistema[11]. Pode-se ainda considerar uma distribuição não uniforme das probabilidades de cada agente receber o testemunho, podendo retratar um importância acrescida que se dá a certos agentes na sua contribuição para a solução final.

- *atribuição controlada* — este é o caso mais interessante e realmente mais útil. Considera-se atribuição controlada ao mecanismo de atribuir o testemunho de uma forma dependente do contexto. Uma possibilidade seria a existência de um agente (denominado *meta-agente*) que saiba analisar a situação do momento, atribuindo o testemunho a um agente particular que, na sua perspectiva, fosse mais relevante. Uma outra alternativa seria que esta gestão estivesse distribuída por todos os agentes, sendo por exemplo o actual possuidor do testemunho a determinar qual o próximo agente a recebê-lo. Pode-se aqui considerar uma disputa e/ou negociação de testemunhos entre agentes.

- *atribuição mista* — uma combinação dos métodos anteriores, simultaneamente ou, em face dos resultados, o método de atribuição ser alterado dinamicamente.

Quando um agente recebe um testemunho, este avalia as declarações (cada uma contendo uma ou mais regras) sequencialmente pela ordem com que foi escrito. Este começa pela primeira declaração, e para cada uma verifica se esta especifica algum estado (no contexto da máquina de estado integrada). Se tal não acontecer ou se se verificar que o agente está nesse estado, as regras são avaliadas sequencialmente. Se durante a avaliação destas regras, o estado do agente muda (por virtude da palavra chave `:gostate`), então esta avaliação termina, passando-se imediatamente para a próxima declaração. Para cada regra verifica-se o valor lógico do antecedente. Se este for "verdadeiro" o consequente é executado (e se um estado destino for declarado, o agente efectua a mudança de estado), caso contrário o anti-consequente (`ELSE`) é executado.

Por defeito, o comportamento do agente é avaliado uma só vez. Contudo há casos (nomeadamente na implementação de um meta-agente) em que é necessário que certas regras sejam executadas mais do que uma vez, durante um mesmo testemunho. Para o efeito existe uma variável do agente, denominada `/rerun`, que determina se após uma passagem pelas declarações, o agente deve repetir o processo do princípio. A inclusão da primitiva `rerun` (sem argumentos, que torna esta variável "verdadeira") força que o agente reavalie as suas declarações mais uma vez.

9.4 Definição de uma primitiva

Uma primitiva é criada por meio da *macro*

[11]Segundo Brooks [2] a complexidade do meio ambiente pode implicar comportamentos globais complexos, mesmo com agentes simples. A atribuição aleatória do testemunho não é mais do que um acrescentar de complexidade ao meio ambiente que os agentes habitam.

assim automaticamente acesso às estruturas internas do agente. Caso o símbolo do primeiro elemento da lista não corresponder a uma primitiva, então é aplicado aos restantes elementos da lista a função LISP correspondente a esse símbolo.

No LISP a chamada a uma função é precedida pela avaliação dos seus argumentos, com excepção das formas especiais. Quando uma expressão do RUBA chama uma função do LISP, esta tem de ser obrigatoriamente uma função pura, não sendo admitida a chamada de formas especiais ou *macros*. No contexto da linguagem, o mesmo acontece com as primitivas. Mas para permitir esta flexibilidade de poder ou não avaliar os argumentos, existem dois tipos de primitivas. As primitivas normais avaliam os seus argumentos antes da chamada à primitiva. No entanto, os argumentos das denominadas primitivas* não são avaliados automaticamente (podendo cada argumento ser avaliado explicitamente no interior da definição da primitiva). A necessidade da inclusão desta facilidade deve-se à criação de primitivas de `and` e `or`, cujos argumentos são avaliados até se encontrar um que resulte em "falso" ou "verdadeiro", respectivamente. Outra aplicação das primitivas* é a definição de primitivas que tomam como argumento um símbolo de uma variável que se deseja alterar (passagem por referência). Na verdade seria possível resolver o problema obrigando o uso da forma especial `(quote ...)` (ou da *read-macro* "'"), mas é convicção dos autores que o código final resulta mais claro recorrendo ao método utilizado, bastando lembrar que seria necessário ao programador o conhecimento de quais os argumentos e quais as primitivas que não devem ser avaliadas. Cabe à linguagem e não ao programador (como utilizador da linguagem) este conhecimento.

9.3 Execução

A execução dos agentes é controlada por um testemunho (*token*). Somente durante o tempo em que o agente possui o testemunho, corre o seu programa. Após o seu programa ter sido percorrido, o agente retorna o controlo a quem lhe deu o testemunho, e assim sucessivamente. Este testemunho é passado de agente a agente de forma a que a solução global seja resultado da contribuição de todos os agentes. Há varias regras de passagem de testemunho que se podem adoptar:

- *atribuição sequencial* — o testemunho é atribuído sequencial e ciclicamente em anel. Este método tem a desvantagem de poder viciar os resultados, dependendo da programação dos agentes. Intuitivamente pode-se imaginar agentes muito simples, mas que com este tipo de atribuição de testemunho, redundem num comportamento global repetitivo.

- *atribuição estocástica* — neste caso o testemunho é atribuído aleatóriamente (uniformemente ou não) pelos agentes. Ao contrário do caso anterior, esta aleatoriedade pode eventualmente estimular uma maior complexidade no funcionamento global do

separados por espaços. Um comportamento é definido como uma sequência de declarações[10] (*statements*):

```
( { <statement> }* )
```

onde cada `<statement>` é composto por:

```
( [ :onstate ( {<state>}* ) [<state-var>] ]
  [ :rules   ( {<rule>}*  ) ] )
```

As regras `<rule>` definidas abaixo serão avaliadas, se a variável do agente que representa o estado assumir um dos valores `<state>` da lista. Se a palavra-chave `:onstate` não for especificada, então as regras são avaliadas incondicionalmente. A variável que representa o estado pode ser opcionalmente especificada por `<state-var>`, tendo o valor por defeito `/state` (note-se que as variáveis locais ao agente começam por `/`). Cada regra `<rule>` tem o seguinte formato:

```
( [ :if <if-condition> ]
  [ :then <then-action> ]
  [ :gostate <then-state> [<state-var>] ]
  [ :else <else-action> ] )
```

onde `<if-condition>` representa a expressão correspondente à pré-condição da regra. Se esta expressão devolver "verdadeiro", a expressão `<then-action>` é avaliada. Caso contrário será a expressão `<else-action>` a ser avaliada. Uma mudança de estado (caso a pré-condição se verificar) pode ser especificada pela palavra-chave `:gostate`, sendo o novo estado `<then-state>`. Tal como anteriormente, a variável que representa o estado pode ser especificada por `<state-var>`, sendo por defeito a variável `/state`.

9.2 Expressões

As expressões indicadas anteriormente seguem a sintaxe geral do LISP com algumas excepções. Uma expressão é constituída por uma lista LISP onde o primeiro elemento denota a função a ser aplicada aos restantes elementos da lista. Globalmente ao sistema existe especificado um conjunto de funções denominadas *primitivas* e que constituem os elementos cujo nível é imediatamente abaixo das regras. Estas primitivas são funções a partir das quais as regras são construídas. Estas primitivas são implementadas como funções LISP (*lambda expressions*) mas com a particularidade de correrem no contexto do agente em causa, tendo

[10] Usa-se aqui a notação habitual para definir uma sintaxe: `{X}*` significa zero ou mais repetições de `X`, `[X]` denota que `X` é opcional, `X|Y` significa `X` *ou* `Y`, os `<X>` simbolizam elementos sintácticos. Tudo o resto são literais.

2. A especificação do comportamento do agente. Como foi referido na secção anterior, este comportamento é definido declarativamente por meio de um conjunto de regras.

Cada regra possui um antecedente (*IF*) que, depois de avaliado, caso seja "verdadeiro" executa o consequente (*THEN*). Caso contrário, executa um "anti-consequente" (*ELSE*). As regras podem ser agrupadas de forma a que o agente se transforme numa máquina de estados. Ou seja, integrada com a programação de cada agente está uma máquina de estados. Uma variável pode assim definir um estado. Conforme o estado em que estiver num dado momento, um certo conjunto de regras a ele associado é analisado. Mas se se condiciona a avaliação das regras pelo estado desta máquina de estados, então é necessário adicionar os mecanismos que permitam a mudança destes estados. Na secção seguinte descreve-se em mais pormenor como isto funciona.

A motivação que levou à integração desta máquina de estados prende-se essencialmente com a necessidade de um agente assumir vários estados ao longo de um processo de negociação.

O antecedente e os consequentes de uma regra são constituídos por expressões LISP, com duas extensões. A primeira é que as funções utilizadas nestas expressões podem ser *primitivas* do sistema. Ou seja, paralelamente aos agentes é definido um conjunto de primitivas, que são idênticas a funções usuais, mas que funcionam no contexto de um agente particular, tendo acesso às estruturas internas do agente em que são executadas. De cada vez que uma função é encontrada, primeiro procura-se se o seu nome designa uma primitiva, e se for este o caso, a primitiva é chamada. Caso contrário, é executada a função LISP homónima. A segunda extensão às expressões LISP, é semelhante à anterior mas refere-se às variáveis do agente. Estas variáveis são distinguidas das variáveis do LISP por serem precedidas do carácter '`/`'.

No contexto da definição dos agentes faz sentido falar em "classes" de agentes, ou seja, em agentes que partilham o mesmo comportamento. Os agentes são assim *definidos* por classes. A *criação* de um agente é um processo distinto, e é feita a partir de uma classe de agentes previamente definida. Pode-se assim falar de um agente como instância de uma "classe" de agentes (tendo o cuidado de não confundir este processo com a instanciação de objectos do sistema CLOS).

9 Definição da linguagem

9.1 Comportamento

A parte fundamental da sistema consiste na definição do comportamento de cada agente. A definição de um comportamento segue a sintaxe habitual do LISP, no que respeita a *S-expressions*: cada expressão é delimitada por parêntesis '`(`' e '`)`' e os elementos da lista

orientada por objectos).

8.1 Implementação

As razões que levaram à escolha do LISP (e o CLOS) para a implementação da presente linguagem prendem-se sobretudo com o nível superior de abstracção que esta linguagem oferece, em relação a outras linguagens mais usuais. De todas as características do LISP, aquelas que são mais proveitosas para o presente trabalho são: fraca verificação de tipos (*weak type-checking*), não sendo necessário saber *a priori* de que tipo são os objectos que se manipulam; operações de fecho (*closures*) que permitem a funções terem como resultado outras funções (*lambda*); e as *macros* que permitem praticamente definir uma nova linguagem em cima do LISP. No entanto, convém também referir que em contraponto a estas características, temos alguns pontos negativos do LISP, nomeadamente o facto de ser normalmente de execução mais lenta[9] e de a ligação da linguagem com o ambiente (interface gráfica, etc.) deixar ainda bastante a desejar. Contudo estes problemas são considerados como secundários para os objectivos do presente trabalho.

As metodologias orientadas por objectos adequam-se bem ao contexto dos agentes, nomeadamente no que respeita à capacidade de encapsulamento de dados e funções. Isto porque um agente pode ser considerado um objecto pró-activo e não passivo. Não sendo o LISP uma linguagem nativamente orientada por objectos, foi necessário recorrer ao sistema CLOS (*Common Lisp Object System*). O CLOS funciona como uma *package* no ambiente LISP e fornece mecanismos para definir classes, objectos e métodos no contexto das linguagens orientadas por objectos.

8.2 O ambiente

O sistema é baseado numa sociedade de *agentes* que comunicam entre si por meio de um ou mais *blackboards*. Um ponto importante é que toda a comunicação entre agentes é feita por intermédio do *blackboard*. Cada *blackboard* é simplesmente um local onde mensagens provindas dos agentes são afixadas e acessíveis a todos eles.

8.3 Anatomia de um agente

Do ponto de vista anatómico, cada agente é constituído por duas partes:

1. Um conjunto de *variáveis* que constitui o conhecimento do agente. Cada variável é acessível por meio de um símbolo que a identifica. Estas variáveis são locais a cada agente.

[9] Em termos de eficiência de algoritmos, a linguagem com que se implementa um algoritmo não afecta a ordem de crescimento do tempo de execução, em função da dimensão da instância.

proposta de troca, este compromete-se a efectuar a troca se o outro aceitar. Ou seja, o acto de aceitar uma proposta é irreversível, e o agente emissor da proposta deve estar preparado para essa irreversibilidade. Esta particularidade é essencial para garantir o sincronismo entre os vários agentes, que neste caso significa que as bolas são efectivamente trocadas, não havendo criação espontânea nem "evaporação" de bolas.

Esta é a formulação mais simples do processo de negociação, onde é possível construir uma série de refinamentos, como por exemplo a possibilidade da formulação de contra-propostas, ou até de envolver mais do que dois agentes no processo de negociação. Na parte seguinte deste relatório apresenta-se uma aplicação prática deste conceito da negociação, com o refinamento da negociação envolvendo mais de dois agentes. Esta aplicação é anterior à criação do RUBA, sendo uma experiência que permitiu recolher bastantes ideias, principalmente para o desenho da linguagem.

Pode-se ainda referir mais duas possibilidades de extensão deste paradigma de negociação. Uma delas é a inclusão de "emoções", no sentido em que foram tratadas nas primeiras secções deste relatório. Imagine-se por exemplo um agente que se "irrita" durante o processo de negociação e decide não tomar mais parte deste, ou um outro agente que repara no impasse que uma negociação estar-se-ia a tornar, e portanto decide ser mais benevolente na aceitação de propostas, mesmo que menos favoráveis para ele. Os autores acreditam que este conceito de "emoções" é bastante promissor, pois é o incrementar do nível de inteligência do agente sem fazer uso da racionalidade pura. Pode haver problemas concretos onde a extracção de mais informação específica do problema não é prática, e uma procura baseada em agentes traria maior robustez na resolução do problema, fazendo apenas uso a conceitos como este das "emoções". A outra possibilidade consiste na suspensão de uma negociação em curso, para proceder a uma *meta-negociação*. Esta meta-negociação consistiria numa negociação para determinar algum assunto relativo à própria negociação (e por isso a utilização do prefixo *meta-*). Esta seria uma outra forma de adicionar robustez e até alguma inteligência ao sistema, sem recorrer a conhecimento específico do problema.

Note-se a semelhança destas ideias com a concepção de uma estratégia de controlo para o sistema, mas a diferença importante está no facto de se distribuir esta estratégia de controlo por todos os agentes. Na verdade, aquilo a que vulgarmente se chama estratégia de controlo, as emoções e o agente "meta-negociador" não são senão meta-mecanismos que visam dirigir um processo que se encontra no nível superior. A perspectiva de Damásio [3] corrobora este ponto de vista.

8 Aspectos constitutivos

Nesta secção descreve-se a camada de *software* que suporta a linguagem, e que consequentemente define o contexto do seu funcionamento. A linguagem de programação escolhida para a implementação foi o LISP (usando o CLOS para obter alguns mecanismos de linguagem

agentes. No entanto em certas situações, a negociação pode degenerar num processo de complexidade exponencial. De facto existem duas classes de problemas de procura a distinguir: por um lado, aqueles cujo número de soluções é reduzido (único, em certos casos) e aqueles que possuem um número elevado de soluções possíveis, consideradas as respectivas restrições. Tipicamente os primeiros são problemas da espécie "quebra-cabeças", de complexidade exponencial, enquanto que os segundos são problemas pragmáticos, por vezes de solução complicada, mas não complexa. Considere-se por exemplo, o problema do caixeiro viajante. Se se pretender uma *solução óptima*, esta situa-se no primeiro caso e é extremamente complexo, dado que pressupõe um espaço de estados a ser visitado exaustivamente: não passa pois de um "quebra-cabeças" interessante. Se o objectivo é encontrar uma solução pragmática (satisfatória e não óptima como refere Simon), não é difícil encontrá-la, mesmo com algumas restrições relevantes para o domínio (não consta que caixeiros viajantes andem pelo país a resolver problemas NP completos...). A abordagem proposta visa sobretudo esta classe de problemas, parecendo pois particularmente adaptada aos objectivos da IA.

Resta agora explicar como funciona este processo de negociação. A ideia base deste processo reside na *emulação* do comportamento humano em negociação. Sendo assim, a negociação tem início quando um agente apresenta uma *queixa* no *blackboard*. O significado desta queixa corresponde a uma situação que o agente pretende resolver, por exemplo a posse de um objecto que lhe é desfavorável. Tomemos como exemplo o problema das bolas (que será concretizado na próxima parte do relatório). Este problema consiste em dois ou mais agentes, cada um possuindo uma bola de uma determinada cor. O objectivo deste problema consiste em os agentes trocarem entre si bolas, de forma a que cada um tenha um bola da cor que pretende. Ou seja, cada agente sabe a cor da bola que quer e possui uma bola não necessariamente da mesma cor. Note-se que não há nenhuma estratégia global de controlo, ou seja, não há nenhum mecanismo que resolveria trivialmente o problema, dando a cada agente a bola que este quer. Os agentes apenas têm "consciência" da bola que têm e da cor da bola que querem, para além do mecanismo de comunicação com os outros agentes por via do *blackboard*.

Tome-se um destes agentes e denomine-se de A_0. Se a cor da bola que possui não corresponde aos seus desejos, então envia para o *blackboard* a referida queixa. Esta queixa tem apenas a indicação que este agente pretende-se livrar dessa bola. Eventualmente um outro agente (A_1) toma conhecimento desta queixa pela consulta do *blackboard*. Dependendo de um critério configurável, este agente pode ou não aceder ao desejo de A_0, formulando-lhe uma *proposta* de troca. Esta proposta inclui a cor da bola que A_1 oferece ao A_0 em troca da bola que A_0 possui. Esta proposta é colocada no *blackboard* e tem a indicação que o seu destinatário é o A_0. O agente A_0, ao receber a proposta pode fazer uma de duas coisas: ou aceita a proposta, efectuando a troca, ou rejeita a proposta. A resposta correspondente é dirigida ao A_1 (via *blackboard*, como sempre). Note-se que quando um agente apresenta uma

representação de conhecimento, dependendo dos problemas em si. Na opinião dos autores este facto caracteriza os pontos fortes de uma abordagem utilizando agentes.

Em termos formais, se tomarmos o estado global do sistema como a colecção do estado de cada agente, podemos ver a negociação como um processo de *procura*. Este segundo ponto reveste-se igualmente da maior importância. É claro que é sempre possível argumentar que afinal os agentes não trazem nenhuma novidade, pois no fundo fazem apenas procura. Na opinião dos autores esta ideia é incorrecta e perigosa. É incorrecta porque uma linguagem de programação, antes de constituir um meio de implementar algoritmos, é uma forma do ser humano exprimir as suas ideias — um meio de comunicação entre pessoas (como refere Sussman). E é perigosa por ser redutora e simplista. Por exemplo, progressos na área das linguagens de programação (como o caso da febre da programação orientada por objectos) seriam totalmente fúteis segundo este ponto de vista, pois no fim resume-se tudo a sequências de *bits* alimentadas a um circuito combinatório (o microprocessador). O que está em causa não é o que resulta deste processo todo, seja procura ou não, mas sim as vantagens de pensar em termos de agentes. Seria obviamente pretensioso afirmar que esta metodologia é adequada para todos os problemas. Esta metodologia deve ser avaliada pelos seus méritos face a cada aplicação que se considere. E estas vantagens são nomeadamente:

1. A distribuição do problema pelos vários agentes, da forma que se achar mais adequado. Este ponto revela o seu potencial, nomeadamente em problemas complexos com muitas restrições fracamente relacionadas[8], como por exemplo a representação das disponibilidades de cada docente no problema da construção de horários curriculares (este problema será tratado na próxima parte deste relatório). Este problema é particularmente interessante porque permite ilustrar a distinção entre a abordagem clássica e a orientação por agentes. Quem tente resolver o problema de encontrar um horário satisfatório, está perante um problema de decisão multi-critério extremamente complexo. Numa perspectiva distribuída, existem, de facto, vários problemas a serem resolvidos (tantos quantos são os agentes envolvidos) procurando-se um resultado final que satisfaça (e não optimize) cada um dos pontos de vista em confronto.

2. Evitar, por um processo de negociação, a necessidade de efectuar procura num espaço de estados que cresce exponencialmente com as dimensões em análise (número de disciplinas, salas, etc.). A forma habitual de controlar a complexidade da procura é recorrer a heurísticas que caracterizem a qualidade das soluções ou dos estados considerados. Tal como foi explicado no ponto anterior, em problemas complexos é por vezes muito difícil construir uma função heurística apropriada para conduzir a procura. Adoptando uma metodologia baseada em agentes, esta medida heurística da qualidade fica distribuída pelos vários agentes, sendo algo que emerge da própria sociedade de

[8]O conceito de restrições fracamente relacionadas é usado aqui num sentido intuitivo, significando que estas restrições podem ser convenientemente separadas e atribuídas a agentes diferentes, por exemplo.

Nas aplicações exploradas neste trabalho apenas se fez uso de um *blackboard*, mas o sistema suporta estruturalmente mais do que um *blackboard*, não só por motivos de flexibilidade como também porque em aplicações mais complexas esta facilidade pode ser muito útil.

Os *blackboards* são na verdade o meio ambiente básico em que estes agentes habitam, embora seja sempre possível aos agentes comunicarem com o exterior de outra forma, pela escrita de primitivas para o efeito. No entanto, note-se a elegância de basear o sistema todo nas mesmas estruturas simples e independentes da aplicação.

Ao longo do desenvolvimento da linguagem e dos exemplos da sua aplicação surgiu a ideia de adicionar, explicitamente uma máquina de estados. Usando variáveis e condicionais *IF-THEN* seria sempre possível implementar uma máquina de estados, mas esta conduta levaria a que o código do agente fosse confuso e pouco claro. Por esta razão foi incorporada na linguagem uma máquina de estados, ou seja, é claro e explícito no código o que o agente deve fazer em cada estado, tal como as condições que o levam a transitar de estado. Esta necessidade surgiu da prática, pois verificou-se a conveniência do comportamento do agente ser condicionado por estados e pelas suas transições.

Note-se que os referidos *blackboards* não são forçosamente globais, ou seja, definidos globalmente a todo o sistema. O RUBA suporta por exemplo *blackboards* locais a um agente, ou a grupos de agentes. Outra possibilidade interessante é a criação de agentes dentro de agentes, ou seja, um agente ter a iniciativa de criar no seu seio uma outra sociedade de agentes, partilhando por exemplo um *blackboard* local criado para o efeito, e trocar informação com esta nova sociedade. Esta ideia foi de facto implementada revelando-se extremamente útil, como se verá na parte deste relatório dedicada à exploração da linguagem. No entanto, pode-se adiantar que a aplicação deste conceito se traduziu na criação de um agente central (denominado de meta-agente) que é responsável pela criação dos outros agentes. Este meta-agente é responsável pelo controlo da execução (a atribuição dos testemunhos, que será explicado adiante) e pela eventual detecção da chegada a uma solução final. Este tipo de flexibilidade é de facto oferecida pelo RUBA.

7.2 Negociação entre agentes como procura

A ideia de agentes efectuarem *negociação* entre eles esteve na base da criação do RUBA. Um processo de negociação pressupõe a ideia de *acordo* entre todos eles. Ou seja, ao longo do processo de negociação, são efectuadas operações sobre o estado global do sistema, ou seja, à sua representação distribuída pelos agentes. Podemos tomar o estado global do sistema como a colecção do estado de cada agente (não confundir com a máquina de estados). É claro que poderá haver alguma redundância nesta representação, tendo por exemplo as restrições do problema, e portanto a dependência entre os estados de cada agente.

Uma particularidade deste sistema, e das metodologias orientadas para agentes em geral, é o facto de o problema em resolução pelos agentes estar *representado de uma forma distribuída* por todos estes. Isto constitui um ponto importante, pois pode facilitar enormemente a

ideia de negociação entre agentes, ideia esta que será desenvolvida adiante.

Há dois paradigmas básicos de programação: O paradigma da *linguagem procedimental* fundamenta-se na programação explícita de algoritmos, ou seja, são especificados todos os passos que o programa deve seguir, mesmo correndo o risco que em tarefas relativamente simples, nem sempre o código fonte revele essa simplicidade. O outro paradigma consiste na *linguagem declarativa*, que permite especificar um comportamento por meio de declarações, por exemplo regras. Destas duas abordagens, a escolhida foi a *declarativa*. As razões que levaram à escolha foram o maior nível de abstracção que esta linguagem permite, e a maior clareza do código fonte. No entanto, não se pode considerar a linguagem que de seguida se descreve como 100% declarativa, pois parte do processo da aplicação da linguagem pode passar pela especificação procedimental de algumas partes (as *primitivas* da linguagem).

7.1 Um agente por dentro

Uma questão básica que se coloca quando se pretende criar um agente, é a de saber o que é que ele irá conter, ou seja, quais os componentes que um agente deverá ter, de modo a conseguir funcionar como desejado. O problema complica-se no presente caso da criação de uma linguagem de aplicação geral.

Um componente do agente já foi referido — o programa. Este termo não é talvez o mais apropriado, não só porque se está a fazer uma descrição de alto nível de um agente, como também o facto de o termo estar demasiado ligado à programação convencional. Não se pretende especificar um programa, mas sim definir um *comportamento*. Este comportamento é definido por meio de regras do tipo *IF-THEN-ELSE*.

De seguida, é necessária uma forma de definir o estado do agente, ou se se preferir, as variáveis de estado do agente. Estas variáveis definem o estado dinâmico do agente, seja uma representação dos objectos que possui, do conhecimento que tem do exterior, etc.

Resta agora estabelecer os mecanismos necessários de forma a ligar estes dois componentes e a garantir a comunicação do agente com o exterior, *i.e.*, percepção e acção. As primitivas são funções definidas no contexto dos agentes e que servem este propósito. Pode-se encarar estas primitivas como os predicados que entram nas regras *IF-THEN-ELSE* acima referidas.

Estes são os três componentes básicos da linguagem. A comunicação entre agentes é garantida por um ou mais *blackboards*. No âmbito da linguagem o termo *blackboard* é entendido numa perspectiva minimalista, ou seja, ao contrário de algumas formulações que deram origem ao termo (no contexto da IA) que entendiam o *blackboard* com uma estrutura mais ou menos activa. Neste caso o *blackboard* é apenas um local para afixação de mensagens entre agentes. Este mecanismo permite afixar mensagens para todos os agentes, para algum agente em particular (simulando portanto comunicação directa entre agentes) ou até grupos de agentes que possuem alguma propriedade em comum. Esta abordagem minimalista justifica-se, dado o facto de se procurar distribuir a inteligência por todos os agentes. Cada agente tem conhecimento (no acto da sua criação) dos *blackboards* com que vai interactuar.

Parte III
RUBA — Linguagem para implementação de agentes

Nesta parte do relatório apresenta-se uma linguagem destinada à implementação de agentes a que se denominou RUBA (RUle Based Agents). Na primeira secção explicam-se os motivos que levaram à criação da linguagem. Na secção que se lhe segue, apresenta-se informalmente a linguagem tal como o ambiente em que os agentes criados funcionarão. De seguida completa-se a apresentação com uma definição mais formal da linguagem. Na secção seguinte, mostra-se a utilização da linguagem com alguns exemplos da sua aplicação. Por fim, as duas ultimas secções tratam da análise crítica da linguagem e de perspectivas para o seu desenvolvimento futuro, respectivamente.

7 Motivação para a criação da linguagem

As soluções são habitualmente consequências de problemas. Mas por vezes acontece que soluções particulares para um problema contêm ou sugerem metodologias capazes de resolver classes mais abrangentes de problemas. No caso dos agentes, que não surgiram directamente da resolução de um problema concreto, faz sentido perguntar, para que é que esta metodologia serve. Na opinião dos autores deste trabalho, os agentes continuam a ser em grande parte uma solução à busca de problemas. A investigação que actualmente se faz em torno dos agentes, comprova esta ideia. Parte-se normalmente do conceito de agência, e procuram-se problemas que poderão ser mais eficazmente resolvidos com base nesta metodologia. O presente trabalho não escapa a esta regra. Sendo assim, procurou-se desenvolver um projecto na sequência do estudo dos agentes, em vez de imaginar um problema concreto para o qual esta abordagem fosse adequada. Problemas artificialmente criados podem comprometer a validade das suas soluções.

O presente trabalho consiste na criação de uma linguagem para a implementação de agentes. Pretende-se que esta linguagem permita desenvolver agentes capazes de resolver uma grande variedade de problemas concretos, evitando compromete-la com algum nicho de problemas. Os autores crêem que esta é a melhor forma de demonstrar a utilidade da ideia dos agentes.

Embora o objectivo desta linguagem seja a capacidade de se aplicar à maior variedade possível de problemas, a sua construção foi motivada e inspirada na abordagem da sociedade de agentes. Tem-se assim um ou mais agentes, cada um especificado por esta linguagem, com um ou mais *blackboards*, que servem de meio de comunicação entre eles. A programação consiste em definir para cada agente como se comportar. Esta arquitectura foi baseada na

Linguagens deste tipo são sem dúvida promissoras e de extrema utilidade, no entanto sofrem de alguma excessiva complexidade. Não faz muito sentido, por enquanto construir agentes cuja complexidade do núcleo seja insignificante com o esforço para garantir uma comunicação que respeite o KQML.

6.5 Integração do AOP e do KQML

O denominado "Agent-K" [4] é o produto da integração do sistema "Agent0" anteriormente referido, e da linguagem de comunicação KQML. Um agente não pode funcionar isolado, pelo que é necessário adicionar-lhe um sistema que lhe permita comunicar com o exterior. Este sistema foi construído extendendo o "Agent0" de forma a suportar o KQML. Esta extensão permite ainda efectuar esta comunicação via rede (TCP/IP ou HTTP).

6.6 Agentes que se movem pela rede

O *Telescript* é um projecto comercial para a implementação de agentes. Segundo a empresa que o desenvolveu (*General Magic*), não se trata apenas de uma ferramenta, mas de uma tecnologia. Esta tecnologia baseia-se na mobilidade dos agentes por uma rede de computadores, nomeadamente a *Internet*. O *Telescript* é constituído por um *engine* em cada máquina, que suporta esta mobilidade dos agentes entre computadores. Esta abordagem contrasta com o RPC (*Remote Procedure Call*), que se baseia na chamada de funções remotamente. Em vez disto, pode-se caracterizar este paradigma do *Telescript* como RP (*Remote Programming*). Presentemente há um grande crescimento de ligações à *Internet* ocasionais (acesso público, via *Internet Service Providers*). Estes utilizadores, não estando permanentemente ligados à rede, poderão usufruir desta tecnologia. Os agentes móveis podem representar estes utilizadores durante a sua ausência, por exemplo na defesa dos seus interesses em alguma negociação. Os autores deste trabalho desejam salientar que os agentes são seres cujas características se adaptam muito bem a redes de computadores. A *Internet* constitui portanto uma direcção privilegiada para onde os agentes se poderão desenvolver.

terão emoções. Neste artigo são enumeradas algumas constrições de sistemas inteligentes, e uma arquitectura para a mente é proposta. Na opinião dos autores do artigo, as emoções estão fortemente relacionadas com os motivos. Este trabalho é filosoficamente interessante, mas sente-se a natural dificuldade de passar alguns destes conceitos à prática, dado o seu elevado nível de abstracção. Esta é de facto uma dificuldade comum encontrada neste domínio. Ou se consegue desenvolver ideias num alto-nível, sendo impossível concretiza-las num sistema inteligente a curto-prazo, ou então conseguem-se desenvolver sistemas aparentemente inteligentes, mas deparando-se com enormes dificuldades quando se pretende incrementar a inteligência destes sistemas.

6.3 Programação Orientada por Agentes

A AOP é uma linguagem desenvolvida por Yoav Shoham para a programação de agentes.Em [21] pode-se encontrar uma descrição desta linguagem, tal como referências a outras linguagens[7]. A AOP é baseada numa representação lógica de agentes, sendo o sistema de desenvolvimento composto por: um sistema lógico para definir o estado mental do agente, uma linguagem de programação interpretada para a programação do agente, e um processo para compilar o programa em código executável de baixo nível. Em termos da linguagem, esta permite definir três componentes: um conjunto de "capacidades", um conjunto inicial de "crenças", e um conjunto "regras de compromisso". Cada uma destas regras estabelece uma acção que é activada quando uma determinada mensagem é recebida e quando o agente está num determinado estado mental. A implementação deste sistema tem o nome de "Agent0".

6.4 Linguagem de comunicação entre agentes

Numa sociedade de agentes, a comunicação entre estes é um ponto fundamental. E para haver comunicação entre agentes, possivelmente heterogéneos, é necessário estabelecer uma linguagem comum. O projecto KQML desenvolvido pela DARPA foi criado com precisamente essa intenção. No artigo [7] esta linguagem é apresentada. A sintaxe desta linguagem é baseada no LISP, possuindo portanto toda a flexibilidade e recursividade que lhe é inerente. Para que uma linguagem deste tipo tenha sucesso, é necessário por um lado garantir que a sua definição seja suficientemente genérica, de forma a abraçar qualquer tipo de comunicação que se pretenda empreender. Por outro lado, é preciso precaver contra casos em que um agente comunique algo a outro agente, cada um de origem diferente, tendo apenas em comum o facto de respeitarem o KQML. Um exemplo de uma mensagem KQML:

```
(reply :language KIF :ontology motors :in-reply-to q1
       :content (scalar 10 kgf))
```

[7]Não foi possível aos autores encontrarem o artigo original do Shoham: Y. Shoham, Agent-oriented programming, Technical Report STAN-CS-1335-90, Dept. of Computer Science, Stanford Univ.

6. Planeamento de Produção e Gestão Administrativa (*Workflow and Administrative Management*)

7. Comércio Electrónico (*Electronic Commerce*)

8. Interfaces com o Utilizador Adaptáveis (*Adaptive User Interfaces*)

Tendo em vista a diversidade de aplicações que se abre aos agentes inteligentes, torna-se necessário desenvolver uma arquitectura aberta que sirva de base à sua criação, assim como uma língua comum que lhes permita interagir uns com os outros.

Ao contrário dos dois exemplos anteriormente apresentados, este projecto não compreende um projecto académico. Trata-se de uma aplicação da metodologia de agentes a uma área predominantemente comercial. É interessante observar ideias e conceitos de inteligência artificial a serem aplicados no mercado, o que aliás tem vindo a tornar-se cada vez mais comum nestes últimos anos. Estes agentes prestam ajuda a tarefas do tipo de gestão de sistemas informáticos de media e grande dimensão. É claro que neste campo deixa-se para trás conceitos mais académicos como reactividade, emoções, etc. mas somente temporariamente, na convicção dos autores deste relatório.

6 Outros projectos envolvendo agentes

6.1 Autómatos que procuram prazer

Em 1968 o termo "agente" ainda não era usado no contexto da inteligência artificial. No entanto é possível encontrar em [5] um estudo sobre um autómato (que de facto possui todas as características de um agente) que procura "prazer". Este conceito de "prazer" é especificado de alguma forma ao autómato, e este, através de um planeamento mais ou menos rudimentar, procura satisfazer esse conceito. É curioso notar estes dois aspectos: Em primeiro lugar o autómato opera num meio ambiente artificial, e toma decisões sobre a sua movimentação nesse meio por via da sua percepção. Ora, esta é exactamente a definição de agência. Por outro lado há a identificação de conceitos intrinsecamente humanos (ou biológicos no sentido lato), como o "prazer". À semelhança das emoções, trata-se de alargar termos usados no contexto das pessoas, para as máquinas. As emoções não são propriedade exclusiva dos humanos, mas somente um rótulo que se relaciona com um conjunto de comportamentos. E se máquinas artificiais são capazes de exibir comportamentos idênticos, faz todo o sentido em atribuir-lhes esses mesmos rótulos.

6.2 Robots que terão emoções

Aaron Sloman tem desenvolvido algum trabalho interessante no domínio da ciência cognitiva. Pode-se encontrar em [20] por exemplo uma dissertação interessante defendendo que robots

um utilizador humano ou outro programa, empregando para isso algum conhecimento ou representação dos objectos ou desejos do ente criador. Os agentes inteligentes são encarados como um modelo de *design* e não como uma nova tecnologia ou novo produto, permitindo consequentemente uma nova abordagem dos problemas suscitados pelas tendências atrás referidas, e cujas soluções pelos métodos convencionais se têm apoiado fundamentalmente na capacidade do operador de torná-los compreensíveis pela máquina, ao invés de investi-la de autonomia suficiente para lidar com eles no estado bruto. Esta forma pragmática de entender os agentes inteligentes estende-se ainda à forma como são descritos em termos de um espaço definido pelas três dimensões: *agência, inteligência e mobilidade*, e ilustrado na figura 3. Entende-se neste caso, por *agência* o grau de autonomia e autoridade investidas num agente, e que pode ser caracterizado pela natureza das interacções com outras entidades do sistema. A *inteligência* designa a capacidade de raciocínio e condicionamento do comportamento pela aprendizagem. Quanto à *mobilidade*, esta refere-se à maior ou menor facilidade de um agente se deslocar numa rede.

Figura 3: Abrangência dos agentes inteligentes, segundo o grupo de agentes da IBM.

Sendo porventura demasiadamente específica, na medida em que agente inteligente é definido como uma entidade de *software*, esta abordagem revela-se suficiente para o desenvolvimento de aplicações nas áreas de:

1. Gestão de Sistemas e Redes (*Systems and Network Management*)
2. Acesso e Gestão Móveis (*Mobile Access / Management*)
3. Recepção e envio de Correio e Mensagens (*Mail and Messaging*)
4. Acesso e Gestão de Informação (*Information Access and Management*)
5. Colaboração (*Collaboration*)

4 *Softbots*

No artigo [6] os autores identificam os *softbots* como um passo importante no domínio da inteligência artificial de modo a esta melhor enfrentar problemas reais. Estes *softbots* são agentes de *software* onde são utilizadas técnicas clássicas da IA (aprendizagem e planeamento) adaptadas ao mundo real (informação incompleta e meio ambiente dinâmico).

Os agentes descritos em [6] têm em vista a satisfação de propriedades que, do ponto de vista dos autores, definem os *softbots*. Na implementação descrita no referido artigo, apenas um pequeno (mas crescente) sub-conjunto de propriedades é efectivamente satisfeito; a saber: comportamento orientado por objectivos, capacidade de planeamento, execução e recuperação de erros, representação declarativa do conhecimento, aprendizagem e adaptação, operação contínua, processamento de linguagem natural, comunicação e cooperação, mobilidade e auto-reprodução.

A ideia subjacente a estes agentes é a existência de um planeador que funciona com informação incompleta. Em [16] pode-se encontrar um tratamento da teoria sobre a qual se estrutura este planeador. Este planeador (designado por SNLP) é baseado no STRIPS mas com a extensão de ser um planeamento não-linear. Denomina-se um plano de não-linear quando este não estipula uma ordem estrita de acções a seguir, em contraste com um plano linear que especifica uma ordem de acções. Ou seja, trata-se de um plano onde há algum grau de liberdade no encadeamento das acções.

Estes agentes são dedicados a tarefas específicas, normalmente associadas a ambientes do tipo sistemas operativos (ex: UNIX). À semelhança do que foi referido quanto ao projecto OZ, estes agentes também sofrem das mesmas limitações, pois o conhecimento e o comportamento é incorporado por meio de regras de planeamento.

5 Agentes da IBM — uma perspectiva pragmática

Em [9] encontram-se referidas algumas das tendências actuais da computação que vêm a pôr em causa a forma como temos vindo a resolver os problemas nesta área. São elas a sofisticação e crescente complexidade tanto do *hardware* como do *software*; os grandes volumes de informação com que o utilizador se vê confrontado; a mobilidade deste, que começa a exigir o acesso à informação independentemente do local onde se encontra; e a cada vez menor proficiência do utilizador comum dos computadores. A resposta a estas tendências encontra-se, segundo os investigadores da IBM, na utilização de *agentes inteligentes*, em quem serão delegadas algumas funções (tais como procura e filtragem de informação, interface personalizada, etc.) passando eles a desempenhar as tarefas desejadas de modo automático, sem controlo directo da parte do utilizador, mas seguindo tão somente as suas orientações.

A definição de agentes inteligentes, neste contexto resume-se assim a *entidades de software que executam determinado conjunto de operações, servindo com algum grau de autonomia*

emoções estão associados *comportamentos*. Estes comportamentos estão na origem dos objectivos a apresentar à arquitectura Hap. O exemplo apresentado em [1] denominado Lyotard, é um agente emocional com comportamentos semelhantes aos de um gato. Algumas emoções na implementação final do Lyotard são: medo, felicidade, tristeza, admiração, repreensão, gratidão, ingenuidade, amor e ódio[6]. Em [1] ilustra-se um exemplo de interacção do Lyotard com um humano. Como nesta fase de desenvolvimento ainda não há um módulo de linguagem natural, o Lyotard está limitado a acções mais simples, tal como num gato. Algumas das acções empreendidas por parte do agente neste exemplo de interacção são: a movimentação do agente pelo espaço virtual, saltar para objectos, observar o meio que o rodeia, "morder" o jogador e deixar-se acariciar pelo jogador.

Adicionalmente às emoções existe uma estrutura identificada com a questão de *sentimentos* em relação aos objectos com que interage. Os objectos do mundo exterior com que o agente toma contacto são representados internamente como um sentimento de *gosto* ou de *nojo*. Ao longo da vivência do agente, estes sentimentos são modificados em conformidade com a interacção do agentes com esses objectos.

A modularidade desta arquitectura constitui um dos seus pontos fortes. Em 1995 foi adicionada a esta arquitectura um módulo de linguagem natural [14] denominado *Glinda*. Este módulo extende o *Hap* de forma a este conseguir lidar com a gramática da linguagem natural. Outro exemplo que mostra a flexibilidade da arquitectura é a utilização destes agentes para implementar a denominada *negociação natural* [19], usando uma abordagem minimalista baseada na forma como as pessoas negoceiam.

O OZ é um projecto de dimensão considerável e que tem vindo a desenvolver-se nestes últimos anos. Na sua base está a aplicação de arquitecturas predominantemente reactivas, o que faz sentido dado o objectivo de os agentes exibirem comportamento natural, no sentido de não artificial. De facto, os mecanismos por detrás do comportamento de seres biológicos são reactivos, e a prática tem vindo a demonstrar exactamente isso. Seres artificiais, construídos com base em arquitecturas de controlo reactivas, mostram um comportamento notavelmente biológico. Sendo assim, faz todo o sentido que o projecto OZ tenha começado com a construção de um módulo de planeamento de carácter reactivo — *Hap*. Este módulo básico serve de suporte a todo o funcionamento do agente. Mas sendo este módulo basicamente um planeador, o agente vai estar condicionado por todas as limitações dos planeadores, nomeadamente no que respeita à representação de conhecimento. Ou seja, embora muitos comportamentos sejam facilmente postos em prática, usando uma adequada representação sobre a qual o planeador funcione, outros problemas há em que o contrário se passe. A opção de basear todo o sistema num planeador, mesmo que enriquecido pela reactividade, acaba por perder em generalidade.

[6] "fear", "happy", "sad", "admiration", "reproach", "gratitude", "angel", "love" e "hate", na literatura anglo-saxónica.

Parte II
Aplicações Ilustrativas

3 Projecto OZ — Agentes Emocionais

Por volta de 1990 teve início na *Carnegie-Mellon University* um projecto denominado OZ com o objectivo de construir agentes para aplicação em realidade virtual e ficção interactiva. Estes agentes são denominados, na literatura original, por *agentes credíveis*[3]. A *credibilidade* destes agentes provém da necessidade de que na interacção humano-agente deixe de existir o sentimento de artificialidade e que seja gerada no humano a sensação que o agente é *real*. Estas ideias estão intimamente ligadas à componente artística do comportamento dos agentes.

Para a obtenção de comportamentos mais "humanizados", o grupo de investigadores do projecto Oz optou por uma arquitectura (*Tok*, [1]) baseada em comportamentos reactivos [13], no sentido de Brooks [2]. Sobre esta camada seriam posteriormente construídos novos módulos, nomeadamente as emoções e a geração de linguagem natural [14].

Os primeiros passos do projecto começaram com a construção da referida camada reactiva, denominada por *Hap* [13]. Este módulo interage directamente com o meio ambiente, sendo todos os outros módulos construídos directamente sobre este. O Hap é basicamente um planeador que opera sobre uma árvore de objectivos[4]. Quando se pretende que o Hap cumpra um dado objectivo, este decompõe-no numa árvore de sub-objectivos. Este processo termina quando se obtêm sub-objectivos primitivos cuja execução é trivial. O Hap é formado por duas partes: a primeira corresponde à árvore de objectivos activos[5] e a segunda a uma memória de planos, que constitui a base de conhecimento do Hap. A reactividade do Hap provém do facto de ambas funcionarem em estreita relação com as percepções do agente. Ao longo da execução dos planos, o agente pode-se aperceber de ocorrências exteriores que condicionam tanto a evolução da árvore de objectivos como a memória dos planos. Note-se que está subjacente a esta arquitectura um sistema formal simples e eficiente, em contraste com alguns agentes baseados em sistemas formais complexos, pouco eficientes e pesados computacionalmente. O objectivo do Oz é apresentar comportamentos *credíveis*, em vez de uma coerência lógica absoluta (tão rara no comportamento humano comum, pois a realidade é demasiadamente complexa para se encaixar facilmente numa moldura matemática rígida).

As emoções são implementadas numa estrutura denominada *Em* [1] que funciona em estreita ligação com o Hap. Este módulo lida com um conjunto de representações de *emoções*. Com base nestas emoções é construído o chamado *estado emocional do agente*. A estas

[3] "Believable agents" na literatura original.
[4] "Goal tree" na literatura original.
[5] "Active plan tree" na literatura original.

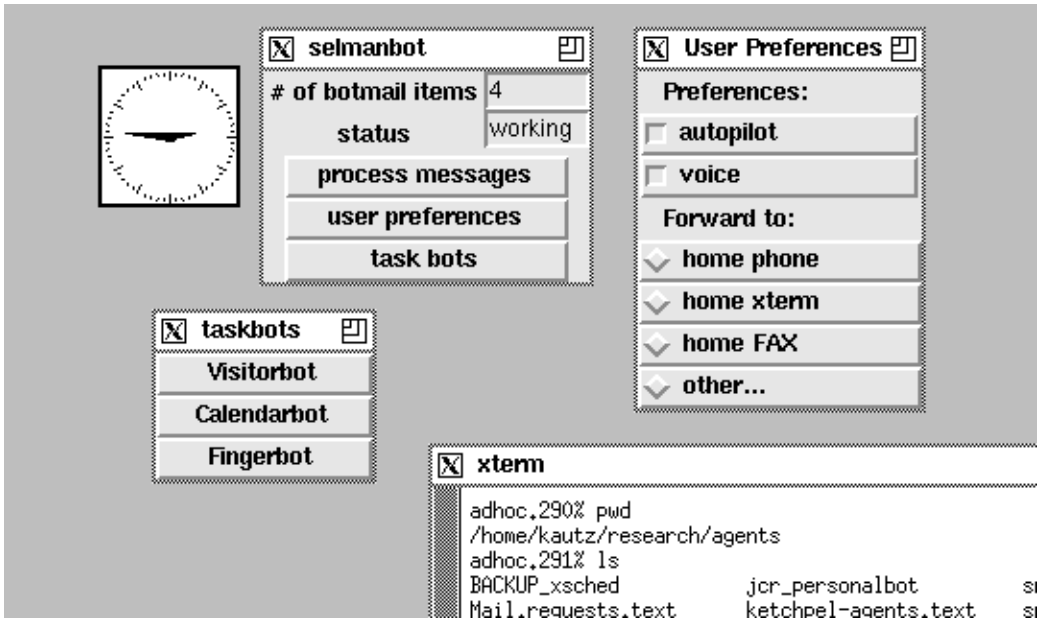

Figura 2: Interface gráfica com um agente *userbot* que representa um utilizador particular.

agentes representantes dos utilizadores e obtém as suas disponibilidades.

Figura 1: (a) Vista global do projecto ALIVE: o utilizador em frente ao ecrã gigante; (b) Imagem correspondente reproduzida no ecrã.

visitantes é um problema moroso de resolver, de cada vez que alguém deseja visitar os laboratórios. Por um lado há que satisfazer os pedidos do visitante (com quem o visitante se deseja encontrar), e por outro garantir que os visitados estejam disponíveis a essa hora. Trata-se portanto de um caso onde a automatização do processo seria desejável. Numa primeira abordagem construiu-se um programa (agente) monolítico que contactava via *email* as pessoas envolvidas e construía o horário a partir daí. No entanto optou-se posteriormente por desenvolver um sistema baseado num conjunto de agentes, em que o agente central (*visitorbot*), responsável pela construção final do horário, entrava em contacto com os agentes representantes dos utilizadores. Cada utilizador (potencial visitado) é representado por um agente (*userbot*, figura 2) que comunica com o utilizador humano por meio de uma interface gráfica. Através dessa interface o utilizador toma conhecimento dos visitantes e informa o agente da sua disponibilidade. O agente central recolhe (usando a rede) estes dados de cada agente representante, permitindo-o formar um horário que satisfaça as necessidades de todos.

Estes agentes são por construção radicalmente diferentes dos anteriores. Enquanto que nos agentes de entretenimento o fundamental era a *forma*, estes procuram obter informação precisa de uma forma metódica. A primeira diferença encontra-se logo no meio ambiente, que neste caso é constituído pela interface com o utilizador e pelos outros agentes. O núcleo corresponde ao algoritmo que entra em contacto com os

legítimo atribuir emoções a seres artificiais (serão os humanos constituídos somente por moléculas orgânicas?), pode-se pelo menos propor um agente que consiga *apresentar-se de forma a ser credível* em termos de emoções. Não se pretende essa coisa mágica, que se acredita pertencer apenas a humanos, mas sim criar a *ilusão* de um comportamento emocional.

A partir da combinação destes elementos é possível construir uma enorme variedade de agentes, seja do ponto de vista da arquitectura, do comportamento, do meio ambiente, etc. Dependendo de cada problema a resolver, estas características deverão ser combinadas. Algumas destas são condicionadas directamente pelas condições do problema, por exemplo o meio ambiente que os agentes irão habitar, e outras são deixadas ao critério de quem desenvolve os agentes. Como se verá ao longo deste relatório, um dos pontos mais fortes dos agentes, na opinião dos autores, é o facto de a representação de conhecimento ficar simplificada e flexível.

No que respeita aos agentes, é preciso notar que a sua formulação não pretende primariamente resolver novos problemas, mas simplificar a resolução de problemas. Não se trata de nenhuma panaceia universal, mas sim uma nova forma de olhar a resolução de problemas.

Ilustra-se de seguida algumas ideias atrás mencionadas com dois exemplo de concretização de agentes, adoptando abordagens completamente diferentes:

- Um deles corresponde ao projecto *ALIVE* [15] em desenvolvimento no *MIT Media Lab.* Estes agentes são vocacionados para o entretenimento, sendo normalmente designados por agentes de interface. O ambiente onde estes agentes animados vivem é um espaço virtual 3D. O objectivo do projecto é conseguir uma interacção com pessoas de uma forma credível. O termo credibilidade é usado neste contexto como a capacidade de seres artificiais exibirem comportamento humanizado. Uma das concretizações deste projecto é um ecrã da altura de uma pessoa, onde são mostrados seres virtuais animados, correspondendo a agentes. Algumas câmaras associadas focalizam o ser humano que pretende interagir com o sistema, determinando a sua posição e comportamento. O ecrã reproduz (como um espelho) a imagem do humano tal como os referidos seres virtuais, que respondem ao humano. Por exemplo, um cão virtual pode-se deslocar na direcção para onde o humano está a apontar o dedo (figura 1).

  Neste caso, os agentes são predominantemente *reactivos* ao que o humano faz. A *percepção* dos agentes corresponde à visualização do comportamento do humano. Este tipo de agentes é propício à expressão de emoções, dado o tipo de interacção que existe. O meio ambiente em que estes agentes funcionam é um espaço virtual 3D, habitado também pelo humano, por meio das câmaras.

- O artigo [11] descreve o desenvolvimento de um sistema baseado em agentes para a marcação de horas para visitas ao laboratório. A construção de um horário para

homogénea), por agentes distintos (sociedade heterogénea), ou mesmo por uma hierarquia de agentes. A relação entre os agentes pode ainda ser de cooperação ou baseada na competição (ou até mesmo em ambas). Marvin Minsky em [17] propôs que a camada de raciocínio do agente fosse composta por uma agenciação (decomposição em agentes) orientada para acções, ou seja, uma acção pode ser decomposta em várias sub-acções paralelas cuja execução fosse atribuída a diferentes agentes.

- A referência a sociedades de agentes sugere a importância da **comunicação** entre agentes. A comunicação entre agentes é normalmente baseada (sintaxe) num modelo de *mensagens*. Em termos de conteúdo (semântica) estas mensagens podem ser ordens a serem executadas pelo agente receptor, ou transmissão de *crenças* (conhecimento declarativo) ou de outros elementos. Alguns exemplos de tipos de mensagens [18]: informação, interrogação, resposta, ordem ou comando, promessa ou oferta, reconhecimento, e partilha.

- O **comportamento** de um agente pode ser desencadeado por meio de um conjunto de regras que determinam univocamente o seu comportamento perante cada situação. Na perspectiva oposta pode-se ter um agente cujo comportamento é determinado pela sua relação com o meio-ambiente. Neste caso diz-se que se trata de um *comportamento emergente*, pois não é previsível *a priori* qual o resultado da interacção do agente com o meio.

- O agente pode ser **reactivo** em relação a acontecimentos exteriores ou acreditar que somente o resultado das suas acções é relevante (o estado do meio ambiente é, num dado instante, somente consequência das acções do agente). O termo "reactivo" utiliza-se aqui no sentido de reagir em resultado directo de acontecimentos exteriores dos quais o agente tem percepção (mais uma vez relacionado com[2]).

- O **tempo de vida** dos agentes permite classificá-los como permanentes ou transitórios. Um agente a quem é confiada uma tarefa específica pode deixar de existir após a conclusão desta. A eliminação de agentes pode também ser efectuada por uma entidade exterior (um outro agente, por exemplo), quando a existência do agente deixa de ser necessária.

- Associada à ideia de tempo de vida está a forma de **criação** de um agente. O caso mais habitual é o da criação determinística, ou seja, a constituição do agente é bem conhecida *a priori*. Uma outra alternativa é a da criação de agentes com base num modelo, mas cuja concretização final não seja especificada, podendo mesmo ser aleatória.

- Um outro aspecto que despertou a atenção de muitos investigadores nesta área foi a possibilidade de um agente artificial possuir e expressar **emoções** — uma característica intrinsecamente humana. Pondo de parte a discussão filosófica de ser ou não

OOP pertence ao domínio da engenharia de *software*, o paradigma dos agentes pertence ao campo da IA.

Estas reflexões de carácter geral procuram definir de um modo suficientemente global o conceito de agente. De seguida, faz-se uma análise dos *items* (características, propriedades, etc.) que distinguem os vários tipos de agentes. O objectivo seguinte de análise é a caracterização dos pontos essenciais a considerar no projecto de um agente.

- O **meio ambiente** pode ser real ou artificial. Os agentes baseados em *software*, por exemplo, "habitam" num meio completamente artificial. A noção de agente é suficientemente abstracta para incluir praticamente qualquer tipo de ambiente que se pretenda imaginar. Este meio ambiente pode ser *estruturado* ou *não-estruturado*. Entende-se por um meio ambiente estruturado aquele cuja constituição ou a sua arquitectura seja conhecida. Se se tomar um meio ambiente como um sistema constituído por partes distintas, denomina-se estruturado um meio em que as características que distinguem as partes e a forma como interagem são bem conhecidas *a priori*. O ambiente pode ainda ser *acessível* por parte do agente (acesso completo ao estado do ambiente), *determinístico* (possibilidade de calcular a evolução futura do ambiente), *episódico* (divisão do tempo em episódios de percepção-acção, *estático* ou *dinâmico* (se o ambiente se mantém inalterável enquanto o agente delibera sobre as suas futuras acções), *discreto* ou *contínuo* (se o conjunto de possíveis percepções e acções é discreto ou contínuo). Note-se que o ambiente é em grande parte determinado por factores externos, não sendo parte integrante da arquitectura do agente. No entanto é possível criar e/ou estruturar o meio ambiente com vista a simplificar o projecto do agente. As características seguintes, pelo contrário, podem ser encaradas como os tópicos mais relevantes subjacentes a uma arquitectura de um agente.

- Um agente pode ser mais ou menos **racional**, dependendo da relevância dada ao núcleo em termos de IA. Neste ponto pode-se já adivinhar uma classificação em agentes inteligentes ou não-inteligentes. Note-se que não há uma relação directa entre inteligência e complexidade, podendo uma arquitectura relativamente pouco complexa apresentar inteligência na sua relação com o meio (tese defendida por R. Brooks em [2], por exemplo). A racionalidade de um agente pode ser classificada em [18] *perfeita* (as decisões tomadas pelo agente maximizam uma função utilitária pré-definida), *calculativa* (se as decisões tomadas são as melhores perante os dados que o agente dispõe) ou *optimalidade limitada* (se o agente toma as melhores decisões perante as limitações de recursos computacionais de que dispõe).

- O agente pode "viver" por si ou por interacção com outros agentes. O meio ambiente de alguns agentes pode até consistir somente na interacção com outros agentes. É usual denominar este aglomerado de agentes que comunicam entre si por **sociedade** de agentes. Estas sociedades podem ser compostas por agentes idênticos (sociedade

2 Definição de um agente

> **AGENTE,** *s. m.* (lat. *agente*). Tudo que actua ou opera. [...]
>
> *in "Dicionário Prático Ilustrado", 1964*

A designação *agente* aplica-se a uma grande variedade de contextos. No seguimento deste trabalho, será denominada como *agente* toda a entidade artificial que satisfizer as seguintes propriedades ([18]):

1. Estar inserida num *meio ambiente* com o qual interage;
2. Obter do meio, informação perceptual através de um conjunto de recursos denominados *sensores*;
3. Actuar no meio ambiente utilizando um conjunto de recursos a que se chama *actuadores*;
4. Entre os sensores e os actuadores haver uma camada (*núcleo*) responsável pela produção de acções a partir da percepção.
5. A existência do agente visa um *objectivo* e é consequência de uma *intencionalidade* por parte da entidade criadora. [1]

De uma forma perfeitamente genérica pode-se caracterizar uma arquitectura geral de um agente, constituído por um núcleo (4) que comunica com o exterior (1) por meio de sensores (2) e actuadores (3). Ao longo desta secção apresentam-se exemplos de concepções de agentes tal como formas diversas de entender e concretizar as características acima citadas.

O termo *objectivo*, usado nas propriedades acima indicadas, pode não ser totalmente especificado, podendo reduzir-se a, por exemplo, um objectivo básico de sobrevivência ou à satisfação de um índice de desempenho.

Estas características são parcialmente semelhantes à formulação da programação orientada por objectos (OOP[2]). Há no entanto duas diferenças importantes a ter em conta. A primeira é que se trata de níveis diferentes de abstracção. A OOP assenta sobre o código de baixo nível, sendo uma forma sistemática e elegante de organizar os dados e o código.A segunda diferença importante é que enquanto os objectos são entidades passivas, respondendo a métodos, os agentes são *pro-activos*, ou seja, têm capacidade de tomarem iniciativa em termos de acção. De facto a forma mais directa e cómoda de implementar agentes seria recorrer à OOP, encapsulando cada agente num objecto. Mas note-se que, enquanto que o

[1] Esta propriedade não está abrangida pela definição de agente em [18].
[2] Do inglês *"Object Oriented Programming"*.

muito sentido caracterizar a inteligência dos componentes dessas arquitecturas. Os sistemas funcionavam como um todo. A eliminação de um ou mais componentes tinha normalmente efeitos desastrosos. No entanto, foi necessário criar uma abstracção que incluísse em si a inteligência propriamente dita, e a forma como esta interagia com o exterior, ou seja, o ambiente. Sistemas inteligentes não vivem por si, mas integrados num meio ambiente. Outro factor a considerar é o inter-relacionamento entre agentes. Cada agente é um ser artificial individual, que se relaciona com o exterior, seja com o ambiente no qual está inserido, seja com outros agentes.

Como R. Brooks [2] tem defendido, comportamento inteligente e complexo não pressupõe sempre sistemas complexos. Comportamentos complexos podem emergir de uma interacção rica de um sistema simples com ambientes complexos. O sistema a considerar deixa de ser o agente em si, mas o agente integrado no meio ambiente.

A definição de agente é inerentemente vaga, pelo que o conceito se pulverizou. O termo agente é hoje em dia aplicado exaustivamente na área da IA. Os autores do presente trabalho crêem que se trata de um conceito fundamental nesta área, tendo tal facto sido uma forte motivação para dedicar o presente trabalho final de curso ao estudo dos agentes.

A abrangência do conceito de agentes é bastante diversa, indo desde os estudos mais teóricos envolvendo lógicas não-modais ou psicologia, até às concretizações mais práticas de agentes, como por exemplo filtragem de *email* e *news*, procura de informação na *World Wide Web*, etc.

Este relatório prossegue com uma análise e sistematização sobre os aspectos mais conceptuais dos agentes. De seguida, introduzem-se alguns exemplos de projectos baseados em agentes. Apresenta-se, a seguir, uma linguagem para a criação de agentes. Esta linguagem representa o culminar deste trabalho, e de todo o estudo efectuado sobre agentes. Posteriormente expõe-se alguma exploração das ideias dos agentes, utilizando os conceitos de negociação e alguns exemplos da utilização da linguagem desenvolvida na resolução de alguns problemas simples. O objectivo deste trabalho não é o de apresentar soluções novas para problemas antigos, mas sim a construção de uma base metodológica para a resolução de uma grande variedade de problemas.

A forma clássica de resolver problemas usando metodologias de IA é a de construir um modelo mais ou menos simplificado do problema, e desenvolver um sistema para a sua resolução. Se o enunciado do problema se tornar mais complexo, ou se novos factores deverem ser tomados em consideração, o sistema terá de ser repensado, mais ou menos profundamente, dependendo das alterações; no entanto, de um modo geral, o sistema está todo ele em causa. A metodologia que este trabalho propõe procura representar os dados do problema e o conhecimento associado, de forma a que seja imediato proceder a alterações no enunciado do problema. É claro que, em concreto, nem sempre será tão imediato proceder a estas alterações, mas os autores do presente trabalho são da opinião que a representação do conhecimento usando agentes é uma forma extremamente promissora de garantir esta flexibilidade.

Parte I
Introdução à Temática dos Agentes

1 O que é um agente?

A inteligência é uma característica primariamente identificada com o ser humano. É comum dividirem-se os animais em racionais e não racionais, sendo o Homem o único ser racional à face do planeta. A racionalidade está, assim, fortemente identificada com a inteligência. Estes conceitos nunca foram, no entanto, definidas com rigor absoluto, pois a definição de inteligência esteve sempre implícita à própria existência do Homem, único ser eventualmente capaz de pensar nestas coisas.

Tudo mudou quando a expressão "inteligência artificial" começou a ser utilizada, quando máquinas começaram a ser capazes de fazer coisas, no domínio do processamento de informação, anteriormente apenas executados por seres humanos. Foi por volta da década de 50 que esta revolução começou, tendo John McCarty, Alan Turing, Marvin Minsky e muitos outros, lugares destacados na discussão que então surgiu, sobre a legitimidade de chamar "inteligente" a algo que não era humano. Ao longo da história, o Homem tem mostrado esta tendência de se considerar dono exclusivo de algo. Já antes de Galileu, o Homem (planeta Terra) considerava-se o centro do universo. Da mesma forma, o Homem também não é o único ser possuidor de inteligência.

Mais de 40 anos volvidos desde as primeiras discussões sobre a artificialidade da inteligência, a área científica denominada de "inteligência artificial" (IA) ganhou forma, amadureceu e encontrou aplicações concretas no dia-a-dia. Hoje já ninguém se atreve a pôr em causa esta área. Os resultados estão à vista de todos.

Por muito que as mentes mais conservadoras procurem uma definição elitista de "inteligência", tem-se vindo a conseguir ao longo do tempo construir sistemas artificiais capazes de *emular* facetas do comportamento e da capacidade humana. Este termo "emular" é muito importante. A inteligência artificial pode ser definida em poucas palavras como a "arte" de construir sistemas artificiais capazes de exibir comportamento inteligente. O ser humano constitui portanto um ponto de referência. Por um lado o termo "inteligência" surge em primeiro lugar aplicado ao ser humano, mas também o objectivo final paradigmático e filosófico da área da IA é a construção de uma máquina capaz de emular o comportamento humano.

O termo "agente" teve origem relativamente recente no campo da IA, associado com a ideia de um ser autónomo inteligente. O termo agência foi pela primeira vez utilizado neste contexto por McCarty e Hayes em 1969, e catorze anos depois (1983), Jon Doyle propôs a noção de agente como sendo nuclear no campo da IA.

No princípio, os sistemas eram globalmente inteligentes. Ou seja, os sistemas inteligentes eram monolíticos, no sentido que embora fosse possível identificar arquitecturas, não fazia

Resumo

Este trabalho desenvolve algumas ideias sobre agentes inteligentes. Os agentes tratados foram motivados por *negociação* numa *sociedade* de agentes. No entanto, procede-se primeiro a uma sistematização sobre o que os agentes são, seguida de alguns exemplos de projectos relacionados com agentes. O núcleo deste trabalho trata o desenvolvimento de uma linguagem para a criação de agentes de *software*. Os agentes criados por esta linguagem habitam a sociedade comunicando entre si por meio de um ou mais *blackboards* comuns. É ainda desenvolvido um modelo genérico de negociação entre agentes. Apresentam-se ainda alguns exemplos de agentes para resolver problemas concretos, aplicando o paradigma de negociação e a linguagem desenvolvida, separadamente ou em conjunto.

Abstract

This report develops some ideas about inteligent agents. The motivation behind these agents was the *negotiation* in an *agent society*. First of all the agent idea is clearly defined, followed by some examples exploring some of these ideas. The kernel of this work deals with the development of a language for implementing software agents. The created agents live in society sharing one or more blackboard structures. It is also developed a generic agents-based negotiation model. A set of examples exploring these ideas is then presented. These examples use the negotiation model and the language.

Agradecimentos

Os autores deste trabalho desejam exprimir a sua profunda gratidão pelo estímulo, participação, inspiração , suporte e ajuda por parte do Prof. Carlos Alberto Pinto Ferreira, orientador do presente trabalho final de curso, sem o qual não teria sido possível levar este trabalho a bom porto.

A Código fonte do RUBA anotado **64**

B Código fonte dos exemplos de aplicação do RUBA **73**
B.1 O problema das bolas 73
B.2 O problema das 8 rainhas 76
B.3 O problema da alocação de recursos 80
B.4 O problema do *jobshop* 83

9.3 Execução 25
9.4 Definição de uma primitiva 26
9.4.1 Algumas primitivas básicas 27
9.5 Definição de uma classe de agentes 28
9.6 Criação de agentes e de *blackboards* 28
9.6.1 Instanciação de um agente 28
9.6.2 Métodos do agente 29
9.6.3 Variáveis do agente 31
9.6.4 Criação de um *blackboard* 31
9.6.5 Métodos do *blackboard* 31

10 Alguns exemplos ilustrativos **31**

11 Análise crítica da linguagem e perspectivas para o seu desenvolvimento **35**

IV Exploração **36**

12 O problema dos horários **36**
12.1 Enunciado do problema 37
12.2 A solução proposta 37
12.2.1 Arquitectura 38
12.2.2 Processo de negociação 39
12.3 Implementação 40
12.4 Do ABSS para o RUBA 43

13 Aplicação da linguagem **43**
13.1 O problema das bolas 43
13.2 O problema das 8 rainhas 46
13.3 O problema da alocação de recursos 49
13.4 Um jogo de cartas chamado *jobshop* 52
13.4.1 Da teoria à prática 54
13.5 "*Faites vos jeux!*" 57
13.5.1 Melhoramento da solução 59

14 Discussão sobre outras aplicações da linguagem **59**

15 Pistas para trabalho futuro **61**

16 Reflexão conclusiva **62**

Índice

I Introdução à Temática dos Agentes **1**

1 O que é um agente? **1**

2 Definição de um agente **3**

II Aplicações Ilustrativas **9**

3 Projecto OZ — Agentes Emocionais **9**

4 *Softbots* **11**

5 Agentes da IBM — uma perspectiva pragmática **11**

6 Outros projectos envolvendo agentes **13**
6.1 Autómatos que procuram prazer 13
6.2 Robots que terão emoções 13
6.3 Programação Orientada por Agentes 14
6.4 Linguagem de comunicação entre agentes 14
6.5 Integração do AOP e do KQML 15
6.6 Agentes que se movem pela rede 15

III RUBA — Linguagem para implementação de agentes **16**

7 Motivação para a criação da linguagem **16**
7.1 Um agente por dentro 17
7.2 Negociação entre agentes como procura 18

8 Aspectos constitutivos **21**
8.1 Implementação 22
8.2 O ambiente 22
8.3 Anatomia de um agente 22

9 Definição da linguagem **23**
9.1 Comportamento 23
9.2 Expressões 24

Trabalho Final de Curso
A Programação por Agentes

Licenciatura em Engenharia Electrotécnica e de Computadores

Ramo de Controlo e Robótica

Instituto Superior Técnico

Rodrigo Martins de Matos Ventura (36612)
Nuno Miguel Coelho Gonçalves (36563)
Nuno Miguel Garção Carvalho (33776)
([1])

23 de Outubro de 1996

[1]Os autores poderão ser contactados pelo email: `nuron@isr.ist.utl.pt`