Ymir: An Implementation in LISP

8.

The Ymir architecture described in the last chapter has been implemented in Common Lisp [Steele 1990] and its object-oriented extension, CLOS (Common Lisp Object System) [Lawless & Miller 1991, Steele 1990, Keene 1989]. It allows for the desing of rules, modules and control structures to create a character that can interact face-to-face with a human. This chapter details the implementation of the "bare-bones" foundation for such character design: object classes, methods, and other software constructs, as well as the hardware setup.

8.1 Overview of Implementation

First we will give a short overview of the implementation and then go into further detail, showing particular algorithms and examples of software.

8.1.1 Simplifications

In this implementation of Ymir, which we will refer to as "Alpha", several simplifications have been made from the general model presented in the last two chapters. The main ones are:

1. Dialogue automatically takes precedence over any other task the agent may be involved with.
2. Only two conversing parties are assumed (computer character and person).
3. The dialogue is centered around a task where the computer character is the expert; the interaction is driven solely by the human.

Other smaller simplifications of the Ymir architecture will become apparent as we get into the details of the implementation.

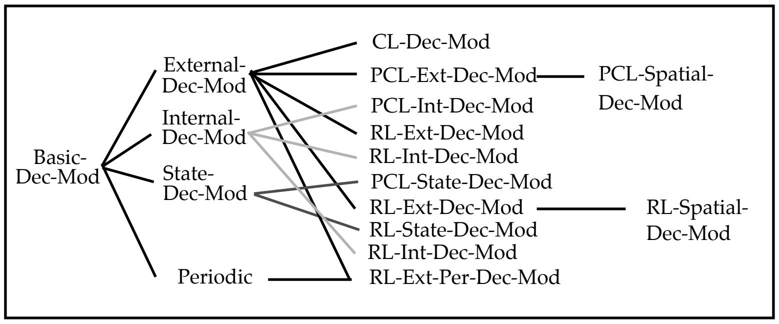


FIGURE 8-1. Types of behavior modules used in Ymir. Each level of behavior module inherits characteristics from the ones above and adds some of its own. (Dec-Mod = decision module.) See text for details on eac type.

8.1.2 Hardware Overview

Two computers are used for implementation Alpha:

1. A Digital Equipment Corporation 3000/300 runs most of Ymir: Reactive Layer, Process Control Layer and Knowledge Base).
2. A Digital Equipment Corporation 5000/240 runs the Action Scheduler.

In addition, six peripheral computers are used for data collection and graphics; these are presented in the next chapter. (More detail on the hardware and software used can be found in Appendix A2 on page 213.)

8.1.3 Software Overview

The main elements of Ymir are implemented in Lisp [Steele 1990]. Supporting software, such as graphics [Thórisson 1996, see Appendix A1] and body tracking [Bers 1996], is implemented in C [Kernigan & Ritchie 1988] and C++ [Stroustrup 1991].

Perceptual and decision modules have been implemented as object classes, using the features of CLOS. Blackboards are simply lists of sublists—each sublist being a posting containing information on the form [MSGS STATE TIMESTAMP], where STATE is either TRUE or FALSE, MSGS is the name of a message from the perceptual modules and TIMESTAMP comes from a global clock, timed in centiseconds.

Several types of decision modules in the RL and PCL have been implemented (Figure 8-1). Modules inherit characteristics from their superclass and add some of their own. Basic operators such as POST and UPDATE are implemented as CLOS methods and specialized for each object type.

The Action Scheduler in Ymir Alpha runs on its own UNIX machine with a socket connection to the PCL. A prioritizing scheduler on the PCL side is used to ship out actions to the Action Scheduler. The AS uses an identical prioritizing scheme to send motor commands to the animation system (animation system is described in Appendix A1 on page 203). The AS also has a scheme for selecting between behavior morphologies and an interrupt feature which will allow it to provide output even if it hasn't looked at all the possible morphologies for a particular behavior. This happens if the expected lifetime of a behavior has been reached (see "A Notation System for Face-to-Face Dialogue Events" on page 108).

8.1.4 Top-Level Loop

Although Ymir is intended for a distributed implementation, the current version runs the RL, PCL and the CL on the same processor. Pseudocode for the top-level loop on this machine is shown in Algorithm 8-1, along with processes next-level down. Let's now take a closer look at each of the layers.

8.2 Reactive Layer

The RL contains perceptual modules (Virtual Sensors and Multimodal Descriptors) and Decision Modules.

8.2.1 Perceptual Modules

Perceptual modules have been implemented as a collection of Virtual Sensors and Multimodal Descriptors, also refered to as a Logic Net, because of their use of logic gates and Boolean output.

Operators for Multimodal Descriptors and Virtual Sensors

Two operators are defined for Virtual Sensors, `UPDATE` and `POST`. The `UPDATE` operator simply feeds the module with the required data. Each perceptual module has a destination for posting its state, `msgs-dest`. Multimodal Descriptors have four operators: `UPDATE`, `POST`, `ACTIVATE` and `DEACTIVATE`. If a module is ACTIVE and all conditions in its pre-conditions are TRUE (these are `AND`ed), then it is `POST`ed as TRUE to `msgs-dest`. When it becomes FALSE (one or more of its conditions are FALSE), it is `POST`ed as FALSE. Decision Modules in the Proces Control Layer determine when descriptors are `ACTIVATE`d and `DEACTIVATE`d.

```
TOP-LEVEL-LOOP
  UPDATE {ALL SENSORS}
  UPDATE {ALL ACTIVE DESCRIPTORS}
  UPDATE {ALL ACTIVE RL DECISION MODULES}
  UPDATE {ALL ACTIVE PCL DECISION MODULES}
  SEND {ALL ACTION-REQUESTS IN *BEHAVIOR-REQUESTS*}
     TO ACTION SCHEDULER

UPDATE [SENSOR]
  Evaluate-Raw-Input {SENSOR}
  If {different (CURRENT-STATE {SENSOR})(LAST-STATE {SENSOR})}
     then POST {SENSOR} TO FUNCTIONAL SKETCHBOARD

UPDATE [DESCRIPTOR]
  Evaluate-Conditions {DESCRIPTOR}
  If {different (CURRENT-STATE {DESCRIPTOR})(LAST-STATE {DESCRIPTOR})}
     then POST {DESCRIPTOR} TO FUNCTIONAL SKETCHBOARD

UPDATE [RL-EXTERNAL-DECISION-MODULE]
  Evaluate-Conditions {RL-EXT-DEC-MOD}
  If {different (CURRENT-STATE {RL-EXT-DEC-MOD})(LAST-STATE {RL-EXT-DEC-MOD})}
     then POST {EXT-RL-DEC-MOD} TO   *BEHAVIOR-REQUESTS*

UPDATE [PCL-EXTERNAL-DECISION-MODULE]
  Evaluate-Conditions {PCL-EXT-DEC-MOD}
  If {different (CURRENT-STATE {PCL-EXT-DEC-MOD})(LAST-STATE {PCL-EXT-DEC-MOD})}
     then POST {DEC-MOD} TO   *BEHAVIOR-REQUESTS*

UPDATE [RL-INTERNAL-DECISION-MODULE]
  Evaluate-Conditions {RL-INT-DEC-MOD}
  If (different (CURRENT-STATE {RL-INT-DEC-MOD})(LAST-STATE {RL-INT-DEC-MOD}))
     then EXECUTE {INTERNAL-ACTION {RL-INT-DEC-MOD}}

UPDATE [PCL-INTERNAL-DECISION-MODULE]
  Evaluate-Conditions {PCL-INT-DEC-MOD}
  If {different (CURRENT-STATE {PCL-INT-DEC-MOD})(LAST-STATE {PCL-INT-DEC-MOD})}
     then EXECUTE {INTERNAL-ACTION {PCL-INT-DEC-MOD}}
```

ALGORITHM 8-1. Top-level loop that handles all events in the Reactive, Process Control and Content layers. Notice that this loop is a serial implementation of a largely parallel system. *BEHAVIOR-REQUESTS* contains all actions that should be sent to the Action Scheduler.

Virtual Sensors

Virtual Sensors consist of two elements: {1} a custom-made function that pre-digests the data needed for the sensor, and {2} the module itself, a CLOS object, with pointers to where to find the necessary data, a pointer to the custom-made function which takes that data as arguments, its own state (TRUE or FALSE), as well as a time stamp for when the module was last POSTed (Figure 8-2).

```
(defclass body-sensor (var-ref)
 ((msgs :initform nil)
  (func :initform nil)
  (data1 :initform (vector 1 1 1))
  (data2 :initform (vector 1 1 1))
  (state :initform nil)))
```

FIGURE 8-2. A *body-sensor* class. The content of its data slots will be refreshed in each call to UPDATE.

The sensor classes implemented are:

1. Body-sensor-with-fixed-reference
2. Body-sensor-with-variable-reference
3. Prosody-sensor
4. Speech-sensor

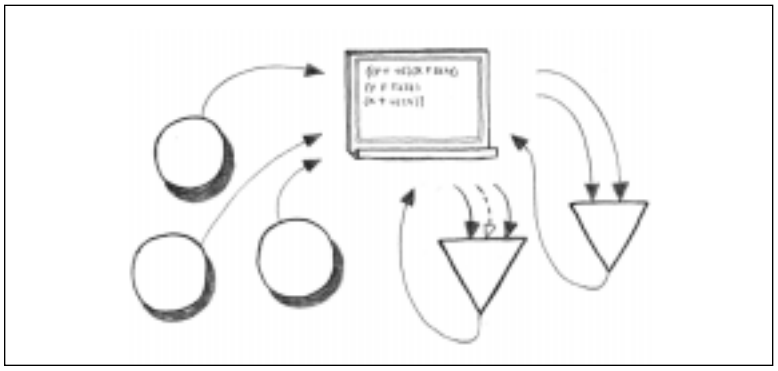


FIGURE 8-3. Virtual Sensors (circles) post their status on the Sketchboard, while Multimodal Descriptors read the Sketchboard to compute their own states, subsequently to be posted on the Sketchboard as well. A large collection of sensors and descriptors makes up a Logic Net (LN).

```
(defclass mm-descriptor ()
 ((msgs :initform nil)
 (pos-conds :initform '())
 (neg-conds :initform '())
 (state :initform nil)
 (stamp :initform 0)
 (active :initform T)
 (thresh :initform 1)))
```

FIGURE 8-4. The multimodal descriptor class. The `thresh` slot contains a value that determines how many of the conditions in neg-conds and pos-conds are needed to make the descriptor `POST` as TRUE to the blackboard.

Body sensors with a fixed reference track an object relative to a stationary object, such as the user's left hand in relation to the monitor. Those with variable reference track the relative spatial aspects of two objects in relation to each other, e.g. the position of the left hand in relation to the trunk. As explained in the last chapter, prosody sensors track some aspect of speech that has nothing to do with its propositional content, such as speech-on-off, intonation, etc.; speech sensors are related to the pragmatic and semantic aspects of speech. We will see example implementations of these sensor classes in the next chapter.

Multimodal Descriptors

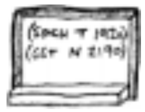
Multimodal integration is handled by what can be thought of as a net of first-order, Boolean logic gates (see "Appendix: Logic Net" on page 126), which I will simply refer to as a logic net (LN, see last section in this chapter). The system uses a special syntax developed for easy construction and modification (by the agent designer as well as the run-time environment). The basic element of this net is the Multimodal Descriptor.

```
(looking-at-hands (pos-conds
 (looking-at-r-hand 0.5)
 (looking-at-l-hand 0.5))

(addressing-me (pos-conds
 (turned-to-me 1.0)
 (facing-me 1.0)
 (facing-domain 1.0)))
```

FIGURE 8-5. Examples of POS-CONDS lists of two multimodal descriptors. The first is a simple aggregator of two virtual sensors; the second is a heuristic for determining if the user's utterance is meant for the agent. The condition's score is added up and compared to the module's threshold to determine if the module is posted as true.

Only static descriptors have been implemented in Ymir Alpha. The descriptors have a set of positive and negative pre-conditions (Figure 8-4). Each condition has a value associated with it (Figure 8-5). When the descriptor is updated, each of the values for those conditions that are TRUE are added up; if they add up to more than the descriptor's pre-set threshold, the descriptor is set to TRUE; otherwise it is FALSE. There are two functionally distinct groups of descriptors, *static* and *dynamic*. Static descriptors simply respond to a static situation, whereas dynamic descriptors detect patterns over time intervals, such as a specific gaze pattern or a combination of arm and eye movements for a given interval. The operators UPDATE, POST, ACTIVATE, DEACTIVATE are implemented as CLOS methods. To check a blackboard for a specific state being TRUE or FALSE, the operator Call-BB is used. It takes a single condition and returns TRUE if that condition was last posted as TRUE, and FALSE otherwise.



Communication via the Sketchboard

The sensors and descriptors communicate with each other via the Functional Sketchboard, where their states are POSTed with a time stamp every time they change. The messages, contained in the MSGS slot of the sensor or descriptor, its state (TRUE or FALSE) and a time stamp are included (e.g. [ADDRESSING-ME T 35415]). The Sketchboard thus accumulates a history of node states, which can be related to other time stamped events in the system, such as turn state or words spoken.

8.2.2 Decision Modules

The general model of Decision Modules is this: Each Decision Module has an associated intention and a condition list. If the conditions become true, the intention "fires". In terms of Ymir, this means that it either results in some internal process running or some outward behavior being executed. A module can be ACTIVE or INACTIVE. If it is ACTIVE, it will fire when the conditions are met; if it is INACTIVE it cannot fire.

Operators for Decision Modules

Four decision module operators are defined: UPDATE, POST, ACTIVATE, and DEACTIVATE. UPDATE supplies a module with access to all data it needs to make its decision. If a module is ACTIVE and its state is TRUE (i.e. all conditions in its FIRE-CONDS lists are met—these are ANDed), then its messages is POSTed to msgs-dest and the module subsequently DEACTIVATEd, and its STATE reset to FALSE. If the module is INACTIVE, the conditions in its reset lists (POS-RESTR-CONDS and NEG-RESTR-CONDS) are checked, and, if all of them are met (these are also ANDed), the module is ACTIVATEd.

```
(defclass basic-dec-mod ()
 ((msgs :initform nil)
 (state :initform nil)
 (active :initform nil)
 (el :initform 100)
 (stamp :initform 0)
 (pos-conds :initform '())
 (neg-conds :intiform '())
 (pos-restr-conds :initform '())
 (neg-restr-conds :initform '())
 ))
```

FIGURE 8-6. The decision-module class. (See also Figure 8-1.)

Decision Module Structure

Decision Modules have been implemented as CLOS classes. Specializations for decision modules are (Figure 8-1):

1. Internal Decision Modules
2. External Decision Modules
3. State Decision Modules

4. Periodic Decision Modules

Each decision module is associated with one intention (I), whose name is kept in the MSGS slot (Figure 8-6). A module has a STATE slot that shows whether it has been fired or not, i.e. whether all the conditions in the POS-CONDS and NEG-CONDS lists have been met simultaneously. If they have, the module's STATE is set to TRUE. In this state, the module is not executable—the conditions in POS-RESTR-CONDS and NEG-RESTR-CONDS LISTS determine whether we restore the module's state to executable (Figure 8-6).

The following general model captures the design philosophy of the decision modules: A module can have four boolean states: ACTIVE or INACTIVE (mutually exclusive), and TRUE or FALSE (also mutually exclusive). It has seven slots: [1] POS-FIRE-CONDS and [2] NEG-FIRE-CONDS, two lists of conditions that, when all conditions in the first are TRUE and those in the second FALSE, will make the module fire (turn its own state to TRUE), [3] POS-RESET-CONDS and [4] NEG-RESET-CONDS, two lists of conditions that, when all conditions in [3] are TRUE and all in [4] are FALSE will make the module active, [5] STATE, containing the state of the module (either TRUE or FALSE), [6] MSGS, containing the message that is posted when the module changes state, and [7] MSGS-DEST, containing a pointer to `msgs-dest`, the destination for its messages (either a blackboard or the Action Scheduler).

To deal with unpredictable time delays from the time when a decision is made to execute an act until it reaches its final stage of being sent to the "muscles", the Action Scheduler uses an action's expected lifetime (A_{el}). This time-out specifies the total time the behavior can stay in the system before reaching the motors (or muscles).

Time stamping is performed whenever a module's action is `POST`ed, and can be used by any of the other decision modules for activating behaviors based on the age of messages reported on the blackboard. The operator `Call-BB-Time` gives the last posting time for a given module, and can be put in any of a decision module's condition lists.

Periodic Decision Modules

Periodic modules simply `POST` their MSGS at a fixed frequency whenever their conditions are met. This is useful for recurring behaviors such as blinking.[1] One could, for example, define conditions that "make the character sleepy", and during these conditions it blinks at a slower frequency than when engaged in conversation.

Internal and External Decision Modules

External behaviors POST messages to a buffer, subsequently to be shipped to the Action Scheduler; internal behaviors execute various internal-acting procedures that are run inside the top-level loop (see above). Behaviors on the action request list are therefore visible actions; internal ones are invisible to the user.

External modules contain the name of a behavior in their MSGS slot, which gets put on a *BEHAVIOR-REQUESTS* cue list and sent to the Action Scheduler. The fact that these are all of type reactive makes their priority in the Action Scheduler (and on the cue list for getting shipped to the AS) the highest.

Internal modules contain a *function* name in their MSGS slot; when the module is POSTed this function is Funcalled. This is implemented by using two separate methods for the UPDATE operator, which take each kind of module.

State Decision Modules

```
(defclass state-dec-mod
               (basic-dec-mod)
 ((next :initform '())
 (active-mm-descrs :initform '())
 ))
```

FIGURE 8-7. The *state* specialization of the basic decision module.

A problem with the above modules is that they can't cause internal conditions as a function of being in a particular state. This would be useful for conditions that are mutually exclusive, such as Dialogue-On/Dialogue-Off, yet come with different requirements for perceptual activities and sub-processes. To solve this, state decision modules are made for keeping track of things such as dialogue state, turn state, etc., and switching on and off the right kinds of perceptual processing. They can be thought of the transition rules in an ATN (augmented transition network) with the difference that they can lead to more than one new state (Figure 8-6). To take an example, the module Dialogue-Off can be made to transition to two states, Dialogue-On and User-Has-Turn. State modules switch multimodal descriptors between being ACTIVE and INACTIVE (the ones that are to be active during the state are stored in a list in the ACTIVE-MM-DESCR slot). Since not all descriptors should be ACTIVE during all states, this provides a mechanism for a sort of "narrowing of attention" for the agent, as well as freeing up processor cycle time.

1. A more intelligent solution to controlling blinking is representing the moisture of its eyes with an evaporation constant and a decision module that reads this simulated moisture and decides to blink when it goes below a certain threshold. This would also be more in keeping with the philosophy of the whole system, the "event-driven" model of control, but is not necessary for the purposes of communicative dialogue.

8.3 Process Control Layer

The PCL is for the most part identical to the Reactive Layer, except that no special perceptual processes are implemented at this level. Interesting future candidates for advanced perceptual processes would be ones that monitor the performance of groups of decision modules and, in conjunction with other decision modules, can modify the ones that don't perform well. Such systems have been called B-Brains [Minsky 1985].

8.3.1 Decision Modules

Decision modules in the PCL are the same types as those in the Reactive Layer. The only difference is that these can look for conditions in both the Functional Sketchboard and the Content Blackboard.

8.3.2 Communication via the Content Blackboard

In the current implementation, messages from the PCL to the CL and from the CL to the PCL are posted to the Content Blackboard. Currently the Content Layer posts more messages to the PCL than vice versa (Figure 8-8). As we established in the last chapter, the PCL posts messages to the Content Layer about the status of the interaction. These are messages about the turn status, for example: A short utterance would be more likely to be a back channel if the agent was speaking—this knowledge could be used in real-time by the CL by weighting the vocabulary toward back channel feedback utterances, thus increasing the probability of correct recognition. Such a setup has not been tested yet in Ymir, but is being investigated.

FROM KB TO PCL
Rcv-Speech
Speech-Data-Avail
KB-Succ-Parse
KB-Exec-Act
CL-Act-Avail
KB-Exec-Act
TKB-Act-Avail
TKB-Exec-Speech-Act
TKB-Exec-World-Act
DKB-Exec-Act
Exec-Done

FIGURE 8-8. A basic set of communication primitives between the Process Control Layer and a Topic Knowledge Base.

Rcv = received, Act = action; Exec =executing; succ = succsessful.

8.4 Content Layer

The Content Layer contains what can be thought of as a combined DKB and TKB. No experiments have yet been done with multiple knowledge bases or multiple topics. The Content Blackboard, which is used by the CL and the PCL, was discussed above.

8.4.1 Dialogue Knowledge Base

A very minimal knowledge for interaction has been implemented. It only contains knowledge about greetings and good-byes. Its small size made it simplest to integrate directly with a topic knowledge base in the case of Gandalf. Gandalf's topic knowledge (section 9.1, page 105),

```
(defclass behavior ()
  ((name   :accessor name   :initarg :name   :initform nil)
   (acts   :accessor acts   :initarg :acts   :initform nil)
   (delay  :accessor delay  :initarg :delay  :initform 0)
   (execution-time :accessor exec-time  :initform nil)))
```

FIGURE 8-9. The generic *behavior* class.

revolves around simple facts about the solar system, such as how big planets are, how many moons they have, etc. It will be described in the next chapter.

8.4.2 The Topic Knowledge Base

The outcome of any interpretation-response generated by a TBK is sent directly to the virtual world (initiated by decision modules in the PCL), without going through the agent's motor mechanisms. This makes it very easy to accommodate multiple knowledge bases for diverse virtual environments (e-mail, graphics, audio, movie clips, etc.) without having to encode these skills in terms of complex end-effector actions such as would be the case if we wanted an all-purpose physical robot

Communication between the TKB and the PCL is handled with a set of pre-determined communication primitives (Figure 8-8) as mentioned above.

8.5 Action Scheduler

The AS keeps track of the facial state and uses knowledge about which layer initiated the action request, as well as the age of the action request, to compose a viable motor scheme for satisfying it. The process of going from a high-level "intention" to an actual motor act is called "morphing" for lack of a better term. Because there is no feedback mechanism between the AS and the decision making layers in the current implementation, feedback about the decision's success has to be gotten by sensing the state of the user. This scheme seems to work well with very simple knowledge bases, but will probably break down for more complex characters.

8.5.1 Behaviors

Behaviors are implemented as CLOS objects (Figure 8-9). When designed, a list of these is created in a general format (Figure 8-10 on page 123), and a make function then called on the list. We refer to the collection of behaviors generated as CLOS objects as the agent's *Behavior Lexicon* (Figure 8-10).

Behaviors in Ymir Alpha are of two kinds:

1. `act` behaviors and
2. `mot-lev` behaviors.

The former contain leaf nodes, i.e. motors, to be moved for generating a particular behavior. These have a direct mapping to a particular motor configuration (dynamic or static), e.g. a facial expression. The latter are abstractions of behaviors that generally have more than one way to be realized. Together, these form a tree. A behavior that subsumes a `mot-lev` behavior is always one level above the leaves of the tree.

Behavior modules above the motor-level (i.e. `act`) have a [1] NAME, and [2] OPTIONS—a list of behaviors that can be used in morphing the behavior; each option is a list of behaviors, which is a list of the form [NAME, EXEC-TIME, DEALY], where NAME is the behavior's name, EXEC-TIME is the execution time for that behavior, and DELAY is a time-delay that offsets this behavior's execution from the execution of the behavior that subsumes it.

Each `mot-lev` action has a [1] NAME—the behavior's unique name, [2] MOTOR-LIST—a list of the motors involved. Each motor in this list contains [1] MOTOR-NAME—the motor's unique name, [2] EXEC-TIME—its default execution time, and [3] REL-POS—the motor's goal position, relative to its range of motion.

`act` behaviors can contain replacement execution times for the behaviors they subsume. Thus, when the behavior `eyes-neutral` is executed as part of the higher-up behavior `face-neutral`, it takes 400 ms for the motors to get to their final position, but when `eyes-neutral` is called directly it takes 100 ms. When the execution times differ for each motor, the longest motor would take the *max* execution time (400 ms in the example above), while the others would be a percentage of that. Using this scheme, a behavior's EXEC-TIME could be recalculated when composing a final action, e.g. in light of time-constraints, but this feature has not been implemented.

8.5.2 Behavior Requests

In Ymir Alpha, behavior requests are received in the Action Scheduler over a socket connection. They are put on a buffer, *PENDING*[2], where

2. This buffer has the corresponding *BEHAVIOR-REQUESTS* output buffer on the PCL side.

```
(defun prioritize-incoming (PEND)
 (let*  ((return nil)
         (RL        1)
         (layer    RL)
         (satisfied nil))
  (loop while (not satisfied) do
   (dolist (act PEND)
     (if (eq (second act) layer)
         (setf satisfied t
           return act)))
   (setf layer (1+ layer)))
  (setf PEND
   (remove return PEND))
  return))
```

ALGORITHM 8-2. Lisp-code for scheduling actions in the Action Scheduler. The procedure `prioritise-incoming` receives the *PENDING* list, returns an act to work on, prioritized by the system that initiated it.

they are serviced based on who—RL, PCL or CL—created the request (Algorithm 8-2). The algorithm simply services all RL requests until there are none left, then it executes one PCL request, if there is one. If there are no RL or PCL requests to execute, it executes a content-related (CL) request.

Requests are received in the form [ACTION-NAME TIME-STAMP EXPECTED-LIFETIME WHO], where WHO is one of RL, PCL or CL. If the expected lifetime has been reached, and no `mot-lev` behavior has been found for it yet, the action is cancelled:

$$\textbf{Cancel}\ \{A\}\ \text{IF (TIME-NOW} > \text{TIME-STAMP}_A + \text{EL}_A). \qquad \{8.1\}$$

No feedback is sent back to the originator of the action whether this action was executed or not. This information is expected to flow back through the virtual sensors as a particular reaction of the user to the lack of behavior. Since there is a tight loop of functional analysis going in to the agent, any problem in the global aspects of dialogue should show up there instantaneously and a new action would be triggered.[3] Thus, the need for complex book-keeping protocols *within* the system—as opposed to through the *outer* feedback loop by way of the effect the agent's behavior has on the user's behavior—should be diminished, if not eliminated.

8.5.3 Generating Behavior Morphologies

When a behavior request is selected to be executed, its name is looked up in the Behavior Lexicon (Figure 8-10). The AS will look at the options available for an action and select one that interferes the least with the ongoing actions of the agent's communicative features such as brows, gaze, hands, mouth, etc. This recursive algorithm works as follows: First it checks in the lexicon if the behavior to be morphed has more than one option. If it has, it takes the first option and goes down one level, again checking for options. Once it reaches the leaves, it pops back up and finds the leaves (motors) for the next option. When it has found motor actions for two options, it selects. If there are more options for the current level, it repeats this until the best one is left. Then it pops up one more level and continues.

This process can only be terminated after at least one option has been traced down to the leaves. It terminates under two conditions: {1} if there are no more options to select between, and {2} if the expected lifetime of the action has been exceeded. The latter is checked every time a

3. By designing cascaded decision modules for reactive behaviors, failures in the interaction are met with "pre-compiled" reactions. See "Creating Behavior Classes with Cascaded Decision Modules" on page 104.

```
(setf *Behavior-Lexicon*
  ;ACT TEMPLATE:
   ;(name class (((act-name-of-option-1 delay exec-time)
   ;             (act-name delay exec-time) etc*)
   ;             (etc*)))
  ;MOTOR TEMPLATE:
   ;(motor-name class delay exec-time pos/data)
   '(
     ; MORPHOLOGICAL DEFINITIONS
       ;Features
         ;neutral
     (face-neutral act
                     (((mouth-neutral 100 400)
                       (eyes-neutral 0 300)
                       (brows-neutral 0 500))))
     (brows-neutral act
                     (((left-brow-neutral 0 400)
                       (right-brow-neutral 0 400))))
     (left-brow-neutral mot-lev
                     (((Bll 0 400 30)
                       (Blc 0 400 30)    ;Brow, left, central
                       (Blm 0 400 30)))) ;Brow, left, medial
     (right-brow-neutral mot-lev
                     (((Brm 0 400 30)
                       (Brc 0 400 30)
                       (Brl 0 400 30))))
     (eyes-neutral act
                     (((upper-lids-neutral 0 100)
                       (lower-lids-neutral 0 100))))
     . . . .
```

FIGURE 8-10. A short segment of the behavior lexicon for Gandalf (see next chapter). Behaviors that are above the leafs are marked as "act"; behaviors that contain only motor commands are marked "mot-lev". (Figure 8-11 shows the names of the facial motors.) A list like *Behavior-Lexicon* is given as an argument to a function that automatically creates CLOS behavior objects. (A full listing of Gandalf's behavior modules is given on page 153.)

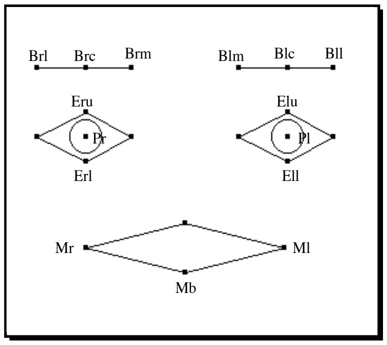


FIGURE 8-11. Movable control points—or motors—are coded as shown: Bll = brow, left, lateral; Blc = brow, left, central; Blm = brow, left, medial; Elu = eye, left, upper; Ell = eye, left, lower; Pl = pupil, left; Ml = mouth, left; Mr = mouth, right; Mb = mouth, bottom. Brow, pupil and eye are mirrored on the right side of the face. Head motion is coded as H. All motors are referenced with an absolute position from 0 to 100. Motors with two degrees of freedom are addressed by either h or v, for horizontal and vertical motion, respectively. All motors can be addressed and run in parallel. (See "Character Animation" on page 203.)

selection between two options has been performed. Upon termination, the leaves (motors commands) get sent to the animation unit, and the character behaves.

8.5.4 Motor Control in the Action Scheduler

The animation unit can be thought of as the character's muscles: it makes sure that the behavior takes the time it is supposed to take, as defined in the EXEC-TIME. Normally all motor specifications of a behavior get sent to the animation unit at the same time. An example of such a behavior is the behavior Smile: corners of the mouth are moved upward and outward, while the lower eyelids are moved slightly up from their resting position, all at once. For sequential actions, certain motor commands may have to wait for others to finish. The DELAY of a motor determines how long after the whole action started it should begin execution. An example of a behavior that uses this feature is the behavior Blink: first the eye is closed, then opened.

Speech

Just as motor commands are ballistic once they leave the AS, speech leaving the AS is also ballistic. It is therefore important that the speech is segmented correctly to allow for cancellations in case the user interrupts the agent. Currently, this is done at natural boundaries larger than the word but shorter than the sentence. Noun phrases, verb phrases and fillers are all sub-components that give useful (albeit not *always* appropriate) boundaries. How the AS controls the incremental execution of long actions, and what its communication with the Topic Knowledge Bases would be remains an issue of further research.

8.5.5 Motor Programs: Animation Unit

The animation unit provides the agent with muscles. It receives commands from Action Scheduler in the form [MOTOR, POSITION, TIME], where motor is the motor to move, position is the new (absolute) position it should move to and time is the absolute time it should take to get there. The commands received by this unit are ballistic (except for manual gesture—see below).

Manual Gesture Control

The ideal way to animate a hand would make use of a representation of the hand that incorporated motors in all finger joints, as well as those of the arm. This is in fact a serious research area in the robotics industry, gesture recognition and gesture generation [Cassell et al. 1994, Wexelblatt 1994, Sparrell 1993, Cutkosky 1992] and will not be dealt with here in any depth. The problem is simplified in Ymir Alpha by representing separately the hand's *position* and *shape*, and by giving the hand two states, *at-rest* and *active*. Whenever the animation module receives a command for a manual gesture it will execute the given type of gesture for the requested time period, after which it moves it back to its *at-rest* position.4

The gestures also have some controllable parameters, such as a *pitch* and *yaw* for deictic gestures, and *duration* for beat gestures. Gestural interruptability has been implemented: If a gesture is executing when a new hand gesture command arrives, the current action will be cancelled, and the new command will take over. The shape of the hand is interpo-

4. Thanks to Hannes Vilhjálmsson for his contributions to and implementation of the hand and the gesturing mechanism.

lated from its current state to the shape associated with the first position in the new gesture, while the hand is moved linearly from its current position to the first position of the new command. This scheme works surprisingly well considering its simplicity. (See also "Character Animation" on page 203.)

8.6 *Appendix: Logic Net*

8.6.1 Syntax

As mentioned above, the logic of the virtual sensors and multimodal descriptors are implemented in a custom-designed syntax. Figure 8-1 shows how common logic gates can be built from this syntax. The advantages to this syntax, as opposed to for example using the logical operands of Lisp, are mainly related to ease of manipulation, by the developer and the run-time system itself. Possible future enhancements that are facilitated by the approach are:

1. Weights could be used to change thresholds of both condition lists and nodes at run-time.

TABLE 8-1. Common logic gates and their equivalents in Logic Net Syntax. Values of the conditions are added; threshold is assumed to be 1.0.

LOGIC GATE	TRUTH TABLE	LOGIC NET SYNTAX
AND	A B | OUT L L | L L H | L H L | L H H | H	POS[A 0.5][B 0.5] NEG[]
NAND	A B | OUT L L | H L H | H H L | H H H | L	POS[] NEG[A 1.0][B 1.0]
OR	A B | OUT L L | L L H | H H L | H H H | H	POS[A 1.0][B 1.0] NEG[]
NOR	A B | OUT L L | H L H | L H L | L H H | L	POS[] NEG[A 0.5][B 0.5]
XOR	A B | OUT L L | L L H | H H L | H H H | L	POS[A 0.6][B 0.6] } NEG[A 0.4][B 0.4] } D1 POS[D1 0.4] NEG[A 0.6][B 0.6]
XNOR	A B | OUT L L | H L H | L H L | L H H | H	POS[A 0.6][B 0.6] } NEG[A 0.4][B 0.4] } D1 POS[A 0.6][B 0.6] NEG[D1 0.4]

2. Continuous-value sensors and descriptors could be integrated more easily, while the Boolean nature of the network could still be preserved at the highest descriptor level.
3. It is easier to implement a learning mechanism for number-based logic gates.
4. Time-based descriptors can more easily be added.

8.6.2 Logic Net: Any Alternatives?

A choice was made to use a custom-designed logic net (LN) as the backbone of the perceptual system. Other contenders for the task of the logic net include fuzzy logic [Kacprzyk 1992] and Fuzzy Cognitive Maps (FCMs) [Dickerson & Kosko 1994]. LNs do not make use of membership functions like fuzzy logic, or graded feature vectors like FCMs, and are therefore simpler to develop and modify. Whether LNs are a sufficiently powerful mechanism for extracting functional aspects of dialogue is an empirical question beyond the scope of this thesis, and will hopefully be settled in future psychological research. One important advantage of using Boolean nodes at the sensory stage is that we can use the states of the modules to track the history of the functional interpretive process: Each time a sensor or descriptor node changes, its state is posted to a blackboard where the other nodes can subsequently read its current state. This allows us to track the progress and path of a particular interaction sequence, as well as to summarize and store the history for future reference. These are key elements in allowing the system to learn over time and to allow the user or the agent to reference past dialogue events. With a continuous (non-Boolean) system, where features are detected to a certain degree and cannot be treated as crisp events that either happen or not happen, post-analysis of internal history becomes problematic, or at least much more complex.[5]

Another advantage of this approach is the possibility of a parallel implementation. Since the nodes are modeled as objects that communicate asynchronously via a common blackboard, parallel implementation could speed up the execution of the system and increase reliability of overall system performance, as well as make more complex sensors and descriptors a viable option, all without slowing computation.

A third advantage is the ease with which new sensors and descriptors can be developed and added without disrupting the ones that are already in place. But first and foremost, it is fast.

5. An interesting question here is how we remember events to have either occurred or not occurred—is a vague memory of an event an indication that continuous-scale memory indices are at work?