CHAPTER 5

Inter-Agent Communications

The framework of handling market update events extends naturally to a more general class of communication events. Endowing trading agents with the ability to communicate with each other opens a whole new avenue in the design of trading strategies.

The subclass of communication events contains the reference to the originating entity (an agent) and the list of recipients for which that communication is addressed to:

```
(defclass COMM (EVENT)
  ((originator
    :accessor originator
    :initarg :originator)
   (recipients
    :accessor recipients
    :initarg :recipients)))
```

The message of the communication event would be contained in the subclass **value** field and it would also be timestamped like any other event.

5.1 HANDLING COMMUNICATION EVENTS

The agent should take into consideration all the communication events directed at it and ignore all the rest. If it is not one of the intended recipients, the event is not handled by the agent. Also, the agent does not talk to itself and, just in case, filters out all messages that it emits. Hence the *observe* method that implements that primary sensor is

```
(defmethod observe ((a AGENT) (e COMM))
  (and (member a (recipients e))
       (not (equal a (originator e)))))
```

In the most general case, the handling of the communication event by the **update** method goes by exactly the same design pattern as explained above for handling market events. No new top-level logic needs to be written but the **preprocess** and **postprocess** methods need to be implemented for a particular agent subclass. The FSM transitions need to handle communication events independently of market update events while maintaining logical completeness. Such a complete version is discussed in the context of handling some inter-market execution algorithms in Part Three.

Here a simpler example is given where one assumes that the handling of communication events only affects the parameters of the FSM but not the state of the agent. This reduced version already allows for testing an interesting range of strategies.

The events that are relevant to the agent are passed initially to the **update** preprocessing stage, that is, its **:before** method:

```
(defmethod update :before ((a AGENT) (e COMM))
  (push e (incomingmessages a))
  (preprocess a e)
  (format T ":BEFORE completed for agent ~A and COMM event
                                                ~A~%" a e))
```

The preprocessor interprets the meaning of the message that needs to be implemented for any particular agent subclass as is shown in the example later.

In this simplified implementation, the interpretation of the information contained in the message yields a change of internal parameters of the trading agent, but not an immediate change of its state. The state can only change when the next market update is consumed by the agent. Hence the **:main** update method for handling a communication event by an FSM agent is simply

```
(defmethod update ((a FSMAGENT) (e COMM))
  (setfsm a)
  (format T "Set FSM completed for ~S~%" a)
  (format T "MAIN method completed for ~S and COMM event
                                          ~S ~%" a e))
```

So the only thing that happens here is that the FSM of the agent is reset with the new parameters that the preprocessor has computed from the interpretation of the message. Finally some postprocessing may be done (e.g., saving the new agent's internal parameters in some external database for recovery from fault purposes):

```
(defmethod update :after ((a AGENT) (e COMM))
  (postprocess a e)
  (format T ":AFTER completed for agent ~A and COMM event
                                                ~A~%" a e))
```

In this simplified design pattern the **operatefsm** method is not called after the FSM had been reset to the new parameters. Hence the agent does not change its state when such a communication is handled. Quite a few situations can be dealt with in this way, in particular in the context of designing and simulating price-taking strategies (rather than strategies that use limit orders, like market-making). To perform a trade, the agent needs to wait for the next price or order book update and would act as soon as that information is received.

5.2 EMITTING MESSAGES AND RUNNING SIMULATIONS

The method that an agent uses to emit a message to its recipients is

```
(defmethod emit ((a AGENT) msg)
  (push (make-instance 'COMM
                       :originator a
                       :recipients (recipientslist a)
                       :timestamp (first (timestamps a))
                       :value msg)
        *events-queue*)
  (push (list (first (timestamps a)) msg)
        (outgoingmessages a)))
```

For simplicity, it uses the last timestamp available to the agent (timestamp of the last market update received) but this can obviously change to the system clock time in a real-time implementation. The *emit* method can be called from any point in the **update** chain (preprocessor, transitions, postprocessor).

The ***eventsqueue*** is a global list, seen by all the agents, that buffers all the events being broadcast. In a simulation environment it is initially populated by the history of market update events. When an agent emits a message, that message is placed at the top of the queue by the **push** function, so this will be the first message to go out unless other agents place their

messages on top of this. Once all the agents are done placing their messages on the queue, the events are broadcast one by one.

Here it is important to note that the handling of a communication event by an agent precludes the possibility of the agent emitting another communication event. An agent can only emit a communication event when handling a market update event. This is done to avoid spamming the broadcast with inter-agent communication at the expense of handling market price events. With this in mind, a convergent simulation process is given by the following function:

```
(defun run-simulation (events)
  (dolist (a *agents*)
    (initialize a))
  (setf *events-queue* events)
  (while *events-queue*
    (let ((e (pop *events-queue*)))
      (dolist (a *agents*)
        (consume a e)))))
```

After all the agents have been initialized, the **`*events-queue*`** is set to the list of (initially market update) events to handle. The first element is popped from the queue and distributed for the consumption of all the agents. **`(pop lst)`** is a destructive operation that returns the **`car`** of the **`lst`** and resets **`lst`** to **`(cdr lst)`**.

During the consumption process loop on the **`*agents*`** list, any agent can emit one or more messages that are consecutively pushed onto the queue. These messages are handled one by one by all the agents in the next step, but as written above, no more communication events can be emitted at this stage. When all these communication events are processed, the next event will be a market update event and the whole process restarts until the **`*events-queue*`** gets depleted to the empty list **`NIL`**.

5.3 IMPLEMENTATION EXAMPLE

In this section, a concrete example of communicating agents built on the simplified design pattern is presented. Each agent trades one particular security and communicates to the other its state. The agents take each other's state into account so as to always be in a situation of opposite trading positions—either (Long,Short), (Short,Long), or (Flat,Flat). This simulates a trend-following pairs-trading model where only high conviction trades are allowed (with one security trading up while the other is trending down).

The strategy class can be defined as a subclass of **SIMPLEMODEL**:

```
(defclass SIMPLEMODELCOMM (SIMPLEMODEL)
  ((MKT
    :accessor MKT
    :initarg :MKT)
   (UNBLOCKSHORT
    :accessor UNBLOCKSHORT
    :initform -1)
   (UNBLOCKLONG
    :accessor UNBLOCKLONG
    :initform 1)))
```

The **MKT** field is the identifier of the security that a particular instance of the class is operating on. It is used to filter the market update events via the implementation of the **observe** method:

```
(defmethod observe ((a SIMPLEMODELCOMM) (e MARKETUPDATE))
  (equal (MKT a) (security e)))
```

Before the agent starts processing any events, it needs to be initialized. The **initialize** method emits a message communicating the agent's **:INIT** state. This "I'm alive" message will be automatically picked up by the other agent and vice versa.

```
(defmethod initialize ((a SIMPLEMODELCOMM))
   (with-slots (MKT L states name) a
    (when (null states)
      (push :INIT states)
      (setf name (concatenate 'string
                              "SIMPLE_MODEL_"
                              (format NIL "~A_~A" MKT L)))
      (emit a :INIT))))
```

The **preprocess** generic function now has two distinct methods to handle the different event types. The market updates are handled by the same method as the superclass, but for clarity it is as follows:

```
(defmethod preprocess ((a SIMPLEMODELCOMM) (e MARKETUPDATE))
  (with-slots (L COUNTER MA revalprices) a
            (setf COUNTER (length revalprices))
            (setf MA (avg-list (sub-list revalprices 0 L)))))
```

The communication events from the other agent are handled by:

```
(defmethod preprocess ((a SIMPLEMODELCOMM) (e COMM))
  (with-slots (UNBLOCKSHORT UNBLOCKLONG) a
              (case (value e)
                    (:INIT (setf UNBLOCKSHORT 0)
                           (setf UNBLOCKLONG 0))
                    (:LONG (setf UNBLOCKSHORT -1)
                           (setf UNBLOCKLONG 0))
                    (:SHORT (setf UNBLOCKSHORT 0)
                            (setf UNBLOCKLONG 1)))))
```

This function performs the interpretation of the message received from the other agent. If the other agent is in the **:INIT** state, it blocks its own long and short positions. If the other agent is in the **:LONG** state, it blocks its own long positions. If the other agent is in the **:SHORT** state, it blocks its own short positions. This ensures that the agents are either both flat or have opposite sign positions at all times.

The setup of the FSM explicitly takes these **UNBLOCKLONG** and **UNBLOCKSHORT** values to alter the market exposure of the agent:

```
(defmethod setfsm ((a SIMPLEMODELCOMM))
  (with-slots (L COUNTER MA UNBLOCKLONG UNBLOCKSHORT states
               currentstate revalprices transitions positions name) a
    (setf currentstate (first states))
    (setf transitions (list
                       (make-instance
                        'TRANSITION
                        :initialstate :INIT
                        :finalstate :INIT
                        :sensor #'price
                        :predicate #'(lambda (p)
                                       (<= COUNTER L))
                        :actuator #'(lambda (p)
                                      (push 0 positions)
                                      (format T
                                              "~S INIT->INIT ~%"
                                              name)))
                       (make-instance
                        'TRANSITION
                        :initialstate :INIT
                        :finalstate :LONG
                        :sensor #'price
                        :predicate #'(lambda (p)
                                       (and (> COUNTER L)
                                            (> p MA)))
```

```
 :actuator #'(lambda (p)
               (push UNBLOCKLONG positions)
               (emit a :LONG)
               (format T
                       "~S INIT->LONG ~%"
                       name)))
(make-instance
 'TRANSITION
  :initialstate :INIT
  :finalstate :SHORT
  :sensor #'price
  :predicate #'(lambda (p)
                 (and (> COUNTER L)
                    (<= p MA)))
  :actuator #'(lambda (p)
                (push UNBLOCKSHORT positions)
                (emit a :SHORT)
                (format T
                        "~S INIT->SHORT ~%"
                        name)))
(make-instance
 'TRANSITION
 :initialstate :LONG
 :finalstate :INIT
 :sensor #'price
 :predicate #'(lambda (p)
                NIL)
 :actuator #'(lambda (p)
               NIL))
(make-instance
 'TRANSITION
 :initialstate :LONG
 :finalstate :LONG
 :sensor #'price
 :predicate #'(lambda (p)
                (> p MA))
 :actuator #'(lambda (p)
               (push UNBLOCKLONG positions)
               (format T
                       "~S LONG->LONG ~%"
                       name)))
(make-instance
 'TRANSITION
  :initialstate :LONG
  :finalstate :SHORT
  :sensor #'price
  :predicate #'(lambda (p)
                 (<= p MA))
```

```
  :actuator #'(lambda (p)
                (push UNBLOCKSHORT positions)
                (emit a :SHORT)
                (format T
                        "~S LONG->SHORT ~%"
                        name)))
(make-instance
 'TRANSITION
 :initialstate :SHORT
 :finalstate :INIT
 :sensor #'price
 :predicate #'(lambda (p)
                NIL)
 :actuator #'(lambda (p)
               NIL))
(make-instance
 'TRANSITION
 :initialstate :SHORT
 :finalstate :LONG
 :sensor #'price
 :predicate #'(lambda (p)
                (> p MA))
 :actuator #'(lambda (p)
               (push UNBLOCKLONG positions)
               (emit a :LONG)
               (format T
                       "~S SHORT->LONG ~%"
                       name)))
(make-instance
 'TRANSITION
  :initialstate :SHORT
  :finalstate :SHORT
  :sensor #'price
  :predicate #'(lambda (p)
                 (<= p MA))
  :actuator #'(lambda (p)
                (push UNBLOCKSHORT positions)
                (format T
                        "~S SHORT->SHORT ~%"
                        name)))))))
```

The agents **`emit`** their state only when their state changes. This is handled in the particular transition **`actuator`** functions. This communication is parsimonious (necessary and sufficient).

Finally, there is no need for a **`postprocess`** method on a communication event as the acknowledgment that the event has been processed is

handled by the **update**'s **:after** method. The **postprocess** on a market update can be used (automatically) from the **SIMPLEMODEL** superclass.

The agents can be instantiated and their recipients lists set by evaluating the following set of expressions:

```
(defparameter *a1* (make-instance
                    'SIMPLEMODELCOMM
                    :MKT "AAPL"
                    :L 10))

(defparameter *a2* (make-instance
                    'SIMPLEMODELCOMM
                    :MKT "DELL"
                    :L 10))

(push *a2* (recipientslist *a1*))
(push *a1* (recipientslist *a2*))

(push *a2* *agents*)
(push *a1* *agents*)
```

Finally a simple simulation can be performed using the **run-simulation** function. For this one needs to create a data set with market update events

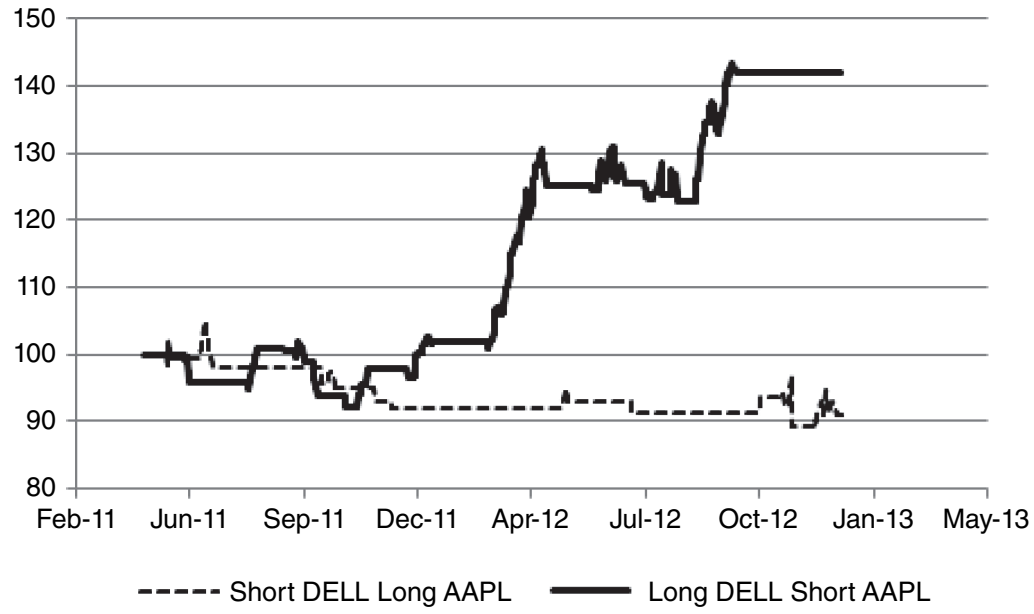

FIGURE 5.1 Communicating Agents Example

for the two securities. Figure 5.1 is an example of an output that shows the polar opposite market positions of the two agents.

In this simple example, each agent preserves its state independently of the other. The state is solely defined by the position of the price `P` relative to the `MA` and is independent of the agent's `position` in the market. This means that each agent is always in sync with its market but varies its position taking into account both agents' states.