

# Implementation of Strategies as Distributed Agents

The clear analogy between a trading strategy and a robot opens the way to think further and visualize a set of trading agents coexisting at any point in time in the computing environment. The trading agents consume events, update their states, and communicate internally and with the external world.

For each trading agent an efficient top-level code can be designed on the basis of Figure 4.1. Although additional code is given in the Appendix, this section provides the detailed explanation of its most important features, and focuses on the top-level implementation of the chain:

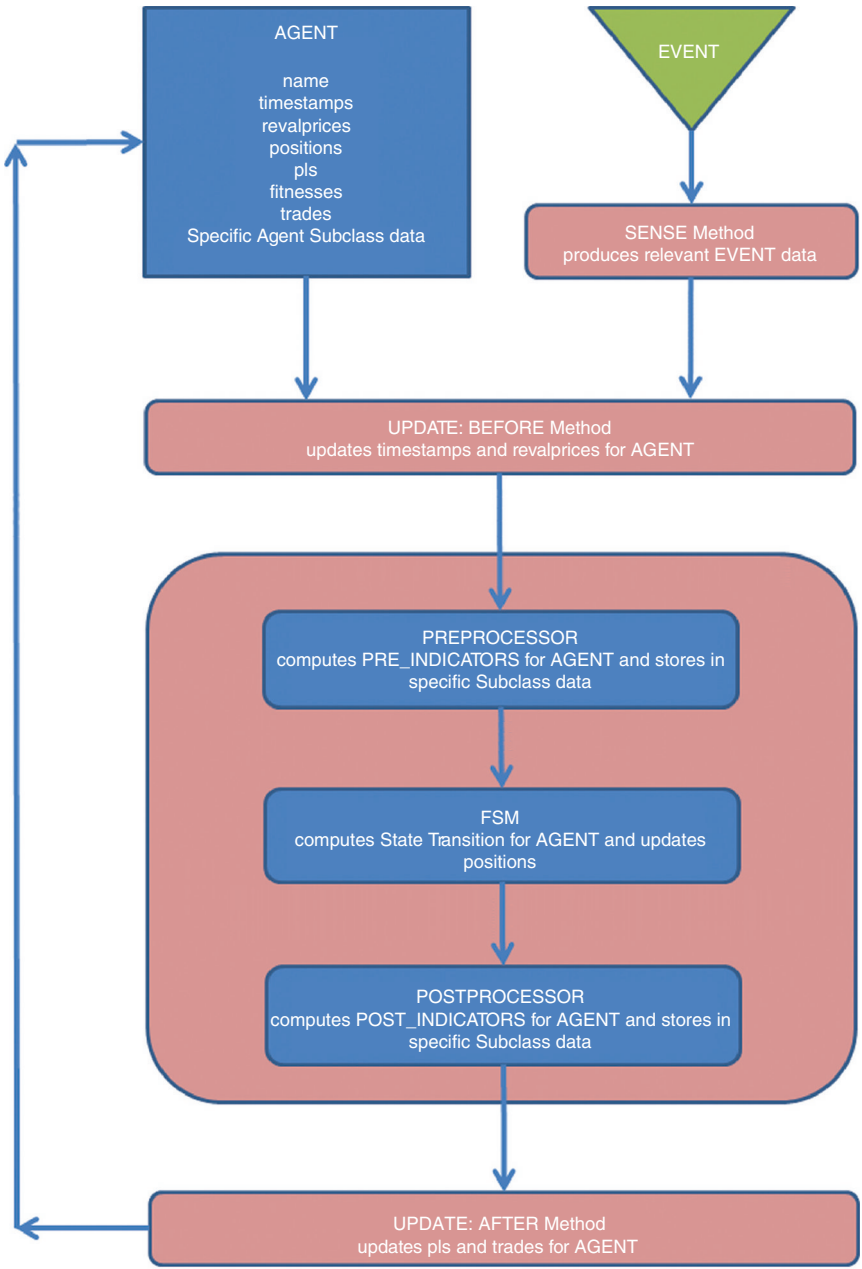
$$\text{Event} \rightarrow \text{Sensor} \rightarrow \text{Preprocessor} \rightarrow \dots \\ \dots \rightarrow \text{Control System} \rightarrow \text{Postprocessor}$$

## 4.1 TRADING AGENT

---

The core class for a trading agent is **AGENT**:

```
(defclass AGENT ()
  ((name
    :accessor name
    :initarg :name)
   (timestamps
    :accessor timestamps
    :initform NIL)
   (revalprices
    :accessor revalprices
    :initform NIL)
   (orders
    :accessor orders
    :initform NIL))
```



**FIGURE 4.1** Top-Level Agent Architecture

```

(positions
 :accessor positions
 :initform NIL)
(pls
 :accessor pls
 :initform NIL)
(fitnesses
 :accessor fitnesses
 :initform NIL)
(trades
 :accessor trades
 :initform NIL)
(tradestats
 :accessor tradestats
 :initform NIL)
(incomingmessages
 :accessor incomingmessages
 :initform NIL)
(outgoingmessages
 :accessor outgoingmessages
 :initform NIL)
(recipientslist
 :accessor recipientslist
 :initarg :recipientslist
 :initform NIL)))

```

This class is just a data repository for each individual agent. At inception, when the class is created, the only input field required is the agent's name:

```

(defparameter *a* (make-instance 'AGENT
                                :name "MyFirstAgent"))

```

The data lists appearing in the agent class are divided in three main categories:

1. Data received from market update events: **timestamps** and **reval-prices**
2. Data calculated through trading activity: **orders**, **positions**, **pls**, **fitnesses**, **trades**, and **tradestats**
3. Data pertaining to communication with other agents: **incomingmessages**, **outgoingmessages**, and **recipients**, the latter being the list of agents that are declared as receivers of the agent's potential communications.

## 4.2 EVENTS

---

Events can be of different sources and have different natures. They all have the commonality that they carry information and that this information had been timestamped by some universal clock when emitted. The **EVENT** class reflects that abstract generality and contains slots for a timestamp and a value that can be anything:

```
(defclass EVENT ()
  ((timestamp
    :initarg :timestamp
    :accessor timestamp)
   (value
    :initarg :value
    :accessor value)))
```

It is specialized for market update events by the subclass **MARKETUPDATE** that contains the name or identifier of the security:

```
(defclass MARKETUPDATE (EVENT)
  ((security
    :initarg :security
    :accessor security)))
```

This class has child classes **PRC** for a single quote, **TICK**, **BOOK**, and **BAR** that are all discussed in Chapter 6, “Data Representation Techniques.” The generic function **price** has methods defined to extract the price information from each type of market update object. This function also optionally contains the slippage in the context of a simulation environment as discussed later.

This section focuses on explaining the mechanics of the consumption by agents of market update events. More general events that contain the communication between agents is covered in the next section.

## 4.3 CONSUMING EVENTS

---

The top-level function that implements the reaction of an agent to an event is **consume**:

```
(defun consume (a e)
  (when (observe a e)
    (update a e)))
```

The **observe** generic function is the primary sensor of the agent and acts as a filter. The agent is only interested in certain events, for example market price updates for a particular security or communications from a particular other agent. Hence events that do not meet the “observable” criterion are simply not sensed by the agent. The example below of communicating agents will show how this method can be implemented. The most generic method of *observe* is simply all eyes open:

```
(defmethod observe ((a AGENT) (e EVENT))
  T)
```

Only relevant events are passed to **update**, which is the central and most important method, responsible for the bulk of the logic of event processing.

#### 4.4 UPDATING AGENTS

This section explains the *update* method for market update events. The next section handles inter-agent communication events for which similar methods apply. The method *update* is composed of three parts, the **:before**, **:main**, and **:after** methods that correspond to the preprocessing, main calculation, and postprocessing stages.

As soon as the agent starts listening to market update events, the timestamps and prices of the events received are recorded and stored in the **timestamps** and **revalprices** lists. These lists are kept in reverse-chronological order because their handling is greatly simplified by the use of the **push** and **pop** LISP functions and yields more efficient methods to compute indicators, especially when using recursion. When the above generic function is called it first calls the **:before** method:

```
(defmethod update :before ((a AGENT) (e MARKETUPDATE))
  (when (null (timestamps a))
    (push 0 (pls a))
    (push 0 (fitnesses a)))
  (push (timestamp e) (timestamps a))
  (push (price e) (revalprices a))
  (preprocess a e)
  (format T ":BEFORE completed for agent ~A and event ~A~%" a e))
```

At the initial phase when the agent is receiving its first market update the **pls** and **fitnesses** are initialized to zero. Those lists have the same length as the **timestamps** and **revalprices**.

Once this basic housekeeping has been done, the method calls the preprocessor that is specific to the agent and has to be defined separately. The preprocessor is responsible for computing indicators to pass to the agent's control system.

The agent's control system is also defined separately and constitutes the specific **main** method of the **update** generic function. Before those details are discussed, let us first complete the logical loop. Assume that instead of a robot, a human trader is sitting and watching the price updates. The trader decides on whether to trade and its **main** update method would simply be an input request of the following sort

```
(defmethod update ((a AGENT) (e MARKETUPDATE))
  (format T "Enter New Position for T= ~A and P= ~A ~%"
    (timestamp e) (price e))
  (let ((newposition (read)))
    (push newposition (positions a))))
```

that simply records the trader's new desired position in the market (no error checking is performed—this is just an example).

Once the **positions** is updated, the control system has done its job. Now the process needs to complete in order to be open to receiving a new event. The **:after** method of the **update** generic function takes care of that:

```
(defmethod update :after ((a AGENT) (e MARKETUPDATE))
  (let* ((L (length (timestamps a)))
    (lastposition (first (positions a)))
    (prevposition (if (< L 2) 0 (second (positions a))))
    (tradequantity (- lastposition prevposition))
    (lastprice (first (revalprices a)))
    (prevprice (if (< L 2) 0 (second (revalprices a))))
    (pl (if (< L 2)
      0
      (* prevposition (- lastprice prevprice)))))
    (push pl (pls a))
    (unless (zerop tradequantity)
      (push (make-TRADE :timestamp (timestamp e)
        :price (+ (price e)
          (slippage a e tradequantity))
        :quantity tradequantity)
        (trades a))
      (push (compute-tradestats (trades a)) (tradestats a)))
    (postprocess a e)
    (format T ":AFTER completed for agent ~A and event ~A ~%"
      a e)))
```

This method starts by computing the incremental PL on this price update and appends it to the **pls** list. Here it is clear that the reverse-chronological representation is optimal as only the **first** and **second** elements of a list are traversed, which bears a low computational overhead.

The resulting trade from the change in position is computed and stored as a structure in the **trades** list, only if the resulting trade quantity is nonzero. In that case, the function **compute-tradestats** is also invoked to compute trade-by-trade statistics (percent profitable, win-to-loss ratio, etc.) and the resulting structure is appended to the **tradestats** list. The **slippage** generic function is by default 0 but could be set to simulate frictional trading costs. The details of slippage and trade statistics calculations are discussed later in this part.

Finally the specific postprocessor to the agent is called. It may calculate another set of indicators that can only be defined once the position has changed. It also may or may not include a specific fitness calculation and update the **fitnesses** list. The concept of fitness will be relevant to Part Two and will be revisited there.

## 4.5 DEFINING FSM AGENTS

Having explained the top-level workings of the **update** method, it is now time to specialize the agent to contain a finite-state machine (FSM) representation of its control system. In the Common Lisp Object System it is a simple matter thanks to multiple class inheritance:

```
(defclass FSMAGENT (FSM AGENT)
  ((states
    :accessor states
    :initform NIL)))
```

The class **FSMAGENT** inherits all the slot definitions from **AGENT** and **FSM** and additionally will contain the history of its states that is updated for each event processed. Here the class **FSM** is defined to mirror the definition of the FSM given at the beginning of the chapter:

```
(defclass FSM ()
  ((currentstate
    :accessor currentstate
    :initarg :currentstate
    :initform NIL)
   (transitions
    :accessor transitions
    :initarg :transitions
    :initform NIL)))
```

One notices that only the **currentstate** and **transitions** are relevant. The finite set of  $N$  possible states is implicit in the complete list of  $N * N$  transitions so it is redundant. The disciplined designs of the FSMs presented in this book ensure this completeness, hence the redundancy of the list of states. Each transition is represented as an instance of the **TRANSITION** class:

```
(defclass TRANSITION ()
  ((initialstate
    :accessor initialstate
    :initarg :initialstate)
   (finalstate
    :accessor finalstate
    :initarg :finalstate)
   (sensor
    :accessor sensor
    :initarg :sensor
    :initform #'(lambda (x) x))
   (predicate
    :accessor predicate
    :initarg :predicate
    :initform #'(lambda (x) NIL))
   (actuator
    :accessor actuator
    :initarg :actuator
    :initform #'(lambda (x) NIL))
   (effected
    :accessor effected
    :initform NIL)))
```

Before going into details as to how the FSM is initialized and maintained for a particular strategy class it is important to understand at the high level how the FSM is operated. First of all, the consumption of an event by a transition object is defined by the following **perform** method:

```
(defmethod perform ((tr TRANSITION) (e EVENT))
  (setf (effected tr) (funcall (predicate tr)
                              (funcall (sensor tr)
                                       e))))
```

This method first calls the transition's sensor function on the event (e.g., the **price** method discussed above). The output is passed to the



transition's predicate that either returns T or NIL. NIL is equivalent to False in LISP whereas True can be represented by T or any non-NIL expression. This Boolean value is stored in the transition's **effected** field.

The consumption of an event by the whole FSM is performed by the **operatefsm** method:

```
(defmethod operatefsm ((fsm FSM) (e EVENT))
  (let* ((applicable-transitions
          (remove-if-not #'(lambda (x) (equal (initialstate x)
                                              (currentstate fsm)))
                        (transitions fsm)))
        (effected-transition
          (car (remove-if-not #'(lambda (x) (perform x e))
                              applicable-transitions))))
    (funcall (actuator effected-transition)
              (funcall (sensor effected-transition)
                        e))
    (setf (currentstate fsm) (finalstate effected-transition))
    (format T "Transition $ -> S~%"
            (initialstate effected-transition)
            (finalstate effected-transition))))
```

This method works exactly as explained when the FSM concept was introduced initially. The **applicable-transitions** variable is initialized to the subset of potential transitions out of the FSM's current state. The **perform** method is applied to all these potential transitions and returns True or False. The transition for which it is True is stored in the **effected-transition** variable. The **effected-transition** is then made to perform some action (e.g., change of the agent's **position**) and for this task the transition's **actuator** function is called on the transition's sensor output. Finally the state of the FSM is changed to the final state of the effected transition.

To close the loop, here is finally the **main** method for the **update** generic function for the **FSMAGENT** case:

```
(defmethod update ((a FSMAGENT) (e MARKETUPDATE))
  (setfsm a)
  (format T "Set FSM completed for ~S~%" (name a))
  (operatefsm a e)
  (format T "Operate FSM completed for ~S~%" (name a))
  (push (currentstate a) (states a))
  (format T ":MAIN completed for ~S and new state ~S added ~%"
          (name a) (currentstate a)))
```

This method overrides the manual main **update** method discussed for the hypothetical human agent. Remember that this **main** method is called after the **:before** method that contains all the preprocessing, and before the **:after** method that contains all the postprocessing. It initially resets the agent's FSM with the **setfsm** method that will be discussed below. This implicitly changes the FSM's parameters (indicators) given the observation of the event. It then runs the FSM decision matrix that implicitly updates the agent's **positions** list. It appends the new state of the agent to the **states** list. Then it finally passes control to the **:after** method.

## 4.6 IMPLEMENTING A STRATEGY

The code discussed above implements the universal top-level process for consumption of events by FSM-endowed agents. To operate it concretely one needs to specialize the **FSMAGENT** class and the **setfsm** method to a particular trading strategy.

Here the simplistic trend-following strategy is explained in details. The next chapter discusses a series of real-world examples that are more complicated but the essentials of the code are the same as for the simple example here.

The strategy's class is defined as a subclass of *FSMAGENT*:

```
(defclass SIMPLEMODEL (FSMAGENT)
  ((L
    :accessor L
    :initarg :L)
   (COUNTER
    :accessor COUNTER
    :initform 0)
   (MA
    :accessor MA
    :initform 0)))
```

It contains the slot for the initially settable parameter **L** that is the length of the lookback period for the moving average calculation. The other slots are for the counter and moving average indicators that are computed by the process. To initialize a concrete class instance that has lookback value of 10 one would evaluate the following expression:

```
(defparameter *mod1* (make-instance
  'SIMPLEMODEL
  :L 10))
```

The instance of our class will be stored in the *\*mod1\** global variable.

Before the agent is able to consume any events it needs some basic initialization. The **initialize** method sets the FSM's original state to **:INIT**. It also reflects the value of **L** in the internal name of the class, which is handy when one runs several instances at the same time and wants to output results in a practical format.

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

Assume for simplicity that the **SIMPLEMODEL** agent consumes all market update events that are passed to it. Thus no specific method on the **observe** generic function needs to be defined and hence it will always return True and will immediately pass control to the **update** method.

The preprocessor method, however, needs to be defined for the **SIMPLEMODEL** class:

```
(defmethod preprocess ((a SIMPLEMODEL) (e MARKETUPDATE))
  (with-slots (L COUNTER MA revalprices) a
    (setf COUNTER (length revalprices))
    (setf MA (avg-list (sub-list revalprices 0 (- L 1)))))
```

It sets the counter to the length of the list of **revalprices** (which is the same as the length of the **timestamps** list). These lists are non-NIL because the preprocessor operates after the first event's price and timestamp had been added to them. The preprocessor computes the **MA** by invoking the **sub-list** function that returns the subset of the first **L** elements (or less if not available) of the **revalprices** list.

The **setfsm** method is the core of the strategy's decision making:

```
(defmethod setfsm ((a SIMPLEMODEL))
  (with-slots (L COUNTER MA 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 #'~S(lambda (p)
                  (and (> COUNTER L)
                      (> p MA)))
 :actuator #'(lambda (p)
                (push 1 positions)
                (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 -1 positions)
                (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 1 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 -1 positions)
               (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 1 positions)
              (format T
                    "~S SHORT->LONG ~%"
                    name)))

(make-instance
 'TRANSITION
 :initialstate :SHORT
 :finalstate :SHORT
 :sensor #'price
 :predicate #'(lambda (p)
                (<= p MA))
 :actuator #'(lambda (p)
                (push -1 positions)
                (format T
                    "~S SHORT->SHORT ~%"
                    name))))))

```

The **setfsm** method updates the FSM's state with the latest state of the agent (remember that the **states** list is in reverse chronological order like all the others). It then updates the **transitions** list with the parameters **COUNTER** and **MA** that have been computed and stored into the **SIMPLE-MODEL** class by the **preprocess** method above.

There are three states and nine possible transitions. However the *LONG* → *INIT* and *SHORT* → *INIT* transitions are not allowed and their predicate functions always return NIL (False). Hence, despite the fact that those transitions are declared for the sake of completeness, they will never happen in the course of the computation.

The sensor function for each transition is the **price** method on a **MARKETUPDATE** event. Each predicate would take that price as input if it is ever passed to it.

The actuator of a transition updates the **positions** list when that transition occurs, as per the **updatefsm** method explained above. So one and only one position update occurs when a market update event is processed. The actuator also prints which actual transition has occurred.

Finally a postprocessor is not really needed here but one could just use it for outputting the agent's data at each event consumption:

```

(defmethod postprocess ((a SIMPLEMODEL) (e MARKETUPDATE))
  (with-slots (name COUNTER MA states positions pls) a
    (format T "Event ~S ~S Consumed for Agent ~S :~%"
            (timestamp e) (price e) name)

```

```
(format T "Output: COUNTER= ~S MA= ~S State= ~S
          Position= ~S PL= ~S~%" COUNTER MA (first states)
          (first positions) (first pls)))
```

This finishes the explanation of the structure of the basic code that implements the event consumption cycle of an FSM-driven agent.

To run the code and do a simulation of the simple model, suppose a list of market update events is created and called **\*events\***. That list contains the consecutive individual event classes in chronological order.

Then to run the **\*mod1\*** strategy on that list, one would simply invoke

```
(dolist (e *events*)
  (consume *mod1* e))
```

and watch the outputs from the existing **format** calls on the console.

Also, assume we define a list of 100 simple strategies of the kind by varying the **L**, storing the result into the **\*agents\*** list:

```
(defparameter *agents* NIL)

(for (i 10 110)
  (push (make-instance
        'SIMPLEMODEL
        :L i)
        *agents*))
```

Then running all the agents at once on the events list is easy:

```
(dolist (e *events*)
  (dolist (a *agents*)
    (consume (a e)))))
```

Here we assume that for each **e**, the agents consume that event in a single thread (consecutively). However, it is easy to make each agent run the update process in a different concurrent thread and synchronize the results before the next event is consumed. This topic will be covered in Part Four.