Paper covers Rock:
An Apex tutorial

Contents

1 Roshambo

Roshambo is another name for the game of "Rock, Scissors, Paper." In Roshambo two players first "prime" the game by pumping their arms, and then make hand gestures of a closed fist (rock), forked fingers (scissors) or a flat hand (paper) for their play. The rules are "Rock breaks scissors; scissors cut paper; paper covers rock," that is, rock beats scissors; scissors beat paper; paper beats rock; and if both players choose the same gesture, it is a tie.

In this tutorial, we'll use Apex to develop successively more interesting versions of a simulated Roshambo match.

2 Practice makes perfect: The basics of Apex

Eventually, we'll have two players playing multiple rounds of Roshambo. But let's start small. Every Apex application needs:

1. a `defapplication` form,
2. a Lisp initialization function that creates the application's agents, and describes each agent's initial goal,
3. at least one procedure for achieving the goal.

In our first application, we'll just write the basic code for a player–the basics of what a player needs to play Roshambo. Figure 1 shows some Apex code for a one-player version of Roshambo–we'll call it Roshambo practice.

Notice that the `defapplication` form gives our application a name *Roshambo 1*, and tells Apex what initialization function to run to start the simulation.

The function is a Lisp function that does the basic requirements of any Apex application:

- It creates an agent `(make-instance 'agent ...)`,
- And places the agent in a locale `(make-instance 'locale ...)`,
- And gives the agent an initial task `:initial-task '(practic roshambo)`.

Notice, by the way, that the `:init-sim` parameter to `defapplication` tells Apex the create a simulation application. Real time applications are created using the `:initapp` parameter. Since we're building simulated Roshambo agents, we want to use `:init-sim`.

The code for this first version of Roshambo can be found in the `Apex:examples;-roshambo;roshambo1.lisp` file. Start Sherpa, and load this application. If you take the following steps:

1. Press the "Step" button.
2. Select "Jill" in the object hierarchy (the pane to the left).
3. Press the "Agenda" button.

```
(defapplication "Roshambo 1"
    :init-sim (initialize-sim))

(defun initialize-sim ()
  (make-instance 'agent
    :name "Jill"
    :locale (make-instance 'locale :name 'gameroom)
    :initial-task '(practice roshambo)))

(procedure :sequential
  (index (practice roshambo))
  (step (prime))
  (step (make game gesture)))

(procedure :seq
  (index (make game gesture))
  (step (gesture rock)))

(procedure :seq (index (prime)))

(primitive (index (gesture ?gesture))
  (profile hand)
  (duration (500 ms))
  (on-completion (inform `(gestured ,+self+ ,?gesture))))
```

Figure 1: Roshambo, Version 1.

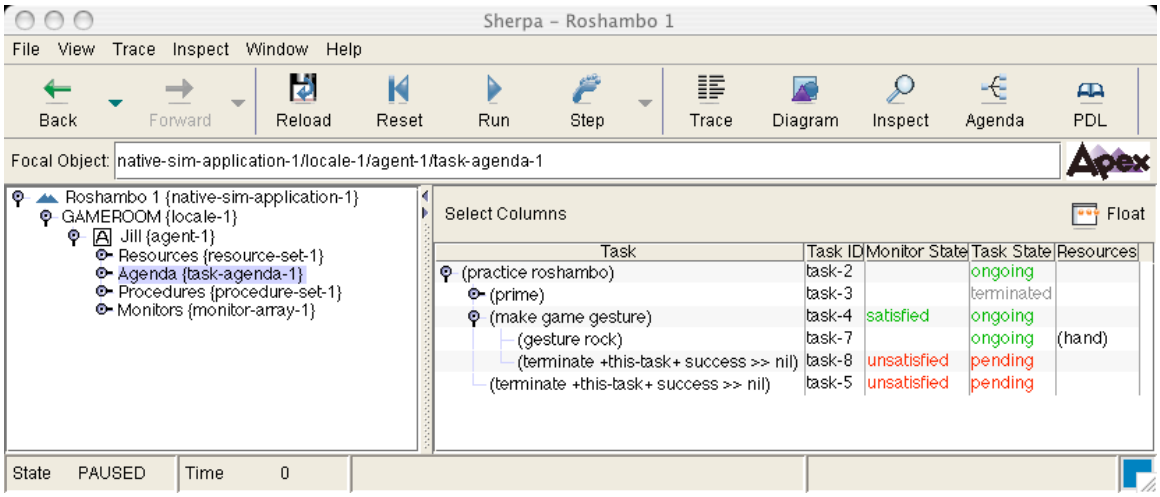

Figure 2: Agenda View of Roshambo, v1, after one step.

After following these steps, you should see something like Figure 2, which shows the state of the Agenda as the agent, Jill, takes her "priming" step.

There are just three procedures and one primitive in our first version of Roshambo. In Apex, a *primitive* define the basic actions that an agent can take. This is true for both simulations and real-time applications. The one primitive in this version of Roshambo, `gesture`, uses the agent's (simulated) hand to make a gesture. We tell Apex that gesturing requires the `hand` resource in the `profile` clause. We tell Apex how long this action will take using the `duration` clause. In simulations, the only way that time passes is by including durations in primitive forms, so it's a good idea to include a `duration` for every primitive–it can be difficult to debug a simulation if many things are happening "at the same time," which can occur if primitive actions don't take any time to transpire. The action taken by this primitive is to call the `inform` function with the Lisp back-quoted list `` `(gestured ,+self+ ,?gesture) ``. The special variable `+self+` is always bound to the agent executing the task, and the variable `?gesture` will be bound when the primitive `(gesture ?gesture)` is called with a particular value. By the way, if we write `` `(gestured ,+self+ ,<?gesture>) ``–*i.e.*, with the variable surrounded by angle brackets–we'll get a warning if the variable is not yet bound. We'll use this convention in later versions.

The top-level goal for our agent is `(practice roshambo)`. You may notice that Apex procedures and primitives look different from ordinary function calls in usual programming languages such as Lisp or C or Java. Apex uses *pattern matching* to find procedures and primitives to execute, so Apex can happily have `index` clauses (which name the procedure or primitive) with spaces in them, and a name such as `play roshambo` is different from a name such as `play roshambo once`. Different people prefer different styles of naming–there's no particular reason not to name a procedure `play-roshambo-once` as if it were a Lisp function. Some people think names like `play roshambo once` make the code more readable.

The procedure `(play roshambo)` is declared to be `:sequential`. (Later, we use a shortened form, `:seq`, to declare the same thing.) This means the steps in `(play roshambo)` are to be carried out sequentially–that is, the second step will not be enabled until the first step terminates, etc. Because sequential execution is the norm for most programming languages, it may seem odd to declare a procedure to be sequential. But the model of task execution underlying Apex is fundamentally based on the idea of multi-tasking, and so, unless specified, tasks are executed as soon as they are enabled.

"Enabled," and "terminated" describe different states that an Apex task can be in. Figure 3 shows all of the possible task states a task can enter. When a task is created, it becomes "pending." When its preconditions are met, it becomes "enabled." When all resources it needs are available, it becomes "ongoing"–that is, is it running. The task may be interrupted, in which case it becomes "suspended." When it finishes, it becomes "terminated." There are other ways for a task to become terminated. For example, if its parent is terminated, it will become terminated, so there is a line from every other state to terminated in Figure 3. Understanding how tasks can transition is very helpful in understanding what is going on in general. For example, it helps explain the "task state" column in the Agenda View in Figure 2; the "pending" tasks are waiting for their preconditions to be fulfilled–in these cases, the completion of other steps.

Now, back to our Roshambo procedures. The procedure `(practice roshambo)` has two steps: `(prime)` and `(make game gesture)`. The `(prime)` procedure is very simple–there

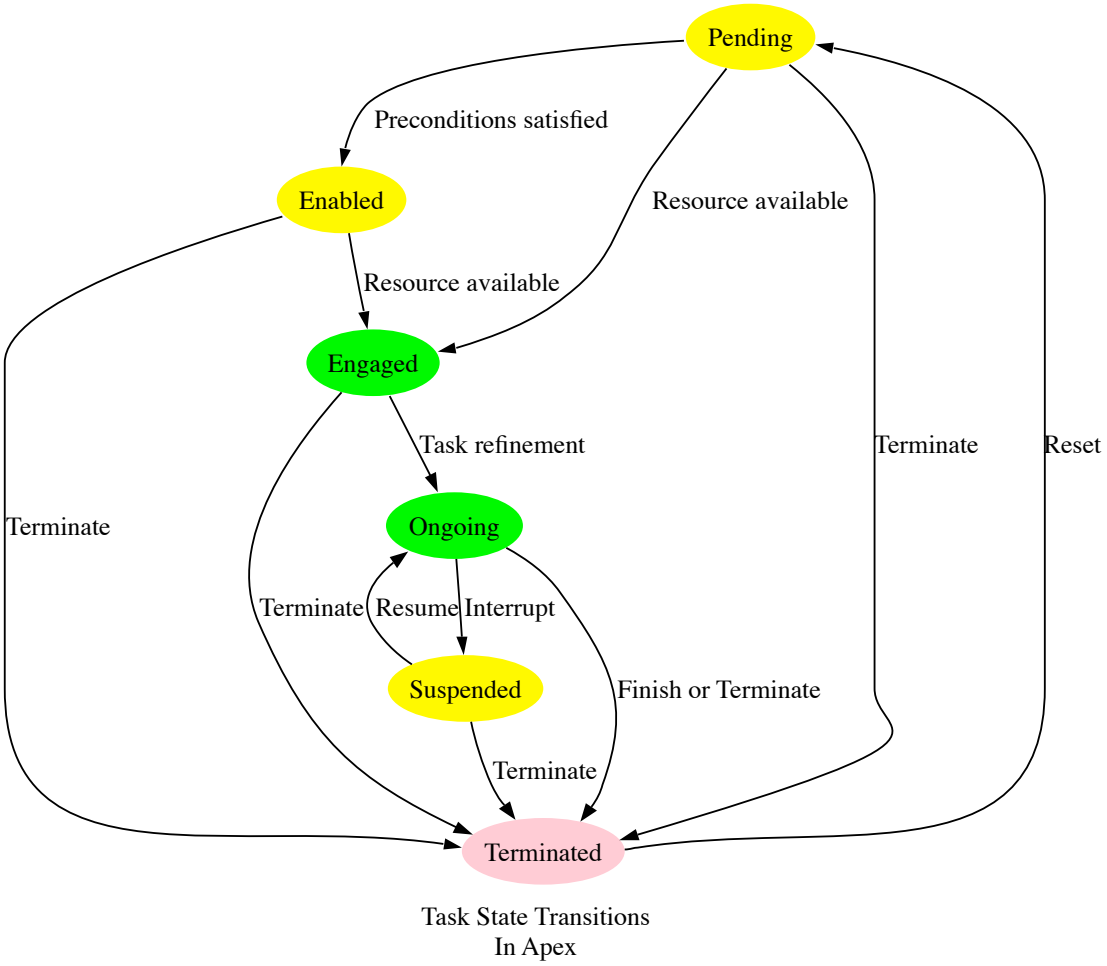

Figure 3: Task transitions in Apex.

```
(procedure :sequential (index (make game gesture))
  (profile hand)
  (step (gesture rock)))

(procedure (index (make game gesture))
  (profile hand)
  (step s1 (gesture rock))
  (step s2 (terminate) (waitfor ?s1)))

  (procedure (index (make game gesture))
  (profile hand)
  (step s1 (gesture rock))
  (step s2 (terminate) (waitfor (state ?s1 = terminated)))
```

Figure 4: Three equivalent procedures.

aren't even any steps! The `(make game gesture)` procedure just has one step. We still include a `:sequential` form to ensure they terminate. Note that the procedures in Figure 4 are equivalent. The second step in the second version explicitly terminates the overall task when the first step terminates. In actually fact, the first version of the procedure is essentially shorthand ("syntactic sugar") for the second version of the procedure.

But the second version of `(make game gesture)` in Figure 4 perhaps requires additional explication. Steps can be named or "tagged"; the tags for the steps in the second version are `s1` and `s2`, respectively. The `(terminate)` action in step `s2` terminates the enclosing task; *i.e.*, `(make game gesture)`. The `waitfor` clause describes the preconditions for step `s2`'s enablement.

This brings up another bit of syntactic sugar. Placing a variable with the same name as a step–as in `(waitfor ?s1)`–is a shorthand for that step's being terminated, as in the `(waitfor (state ?s1 = terminated)` of step `s2` of the *third* version of the procedure in Figure 4. You'll see (and use) this shortcut time and time again in Apex procedures.

3 Tell me about it: Communicating information between procedures

Our second version of Roshambo is nearly the same. It seems a bit bogus that the `(play roshambo once)` procedure should always gesture `rock`. Figure 5 shows a new version of Roshambo, in which a primitive is called to choose a gesture. This can be found in `roshambo2.lisp`.

We introduce a new resource, simulating the use of the brain, in the gesture choosing primitive. Our agent still always chooses to gesture "rock," but we have isolated the choosing to a separate task.

The other thing of note in version 2 is how primitive communicates to a procedure that calls it. The `return` clause sends a value, in this case, `'rock`, to the calling procedure. THe

```
(procedure :sequential
  (index (practice roshambo))
  (step (prime))
  (step (choose gesture => ?my-gesture))
  (step (gesture <?my-gesture>)))

(primitive (choose gesture)
  (profile brain)
  (duration (500 ms))
  (return 'rock))
```

Figure 5: Roshambo, version 2. Changed procedures only.

```
(defun initialize-sim ()
  (let ((locale (make-instance 'locale :name 'gameroom)))
    (make-instance 'agent
      :name "Jill"
      :locale locale
      :initial-task '(practice roshambo))
    (make-instance 'agent
      :name "Jack"
      :locale locale
      :initial-task '(practice roshambo))))
```

Figure 6: Roshambo, version 3. Creating multiple agents.

value is captured into a pattern variable by using `=>`; in this case, `(choose gesture => ?gesture)`.

We also used the `<?my-gesture>` form to help ensure that the variable `?my-gesture` is bound before we use it.

4 Two can play at this game: Multiple agents

Of course, Roshambo is a game between two players, and currently we have only one player. Version 3 (`roshambo3.lisp`) adds a second player, "Jack" to challenge our first player, "Jill." At first, we'll have Jack and Jill choose gestures in exactly the same way, *i.e.*, always choosing to play rock. Figure 6 shows that adding multiple agents is easy to do: we change the initialization function to do so.

Because we want Jack and Jill to share the same location, we first create a `locale` object before creating the agents. Because they have the same top level goal, they both prime and gesture "rock," and their done. Figure 7 shows a trace of their actions. Here we are tracing the `gestured` events–notice that the `inform` primitive is informing all the agents of gestures, so that when Jack makes a "rock" gesture, both Jill and Jack receive a `(gestured #agent-2`

```
[0 Jill] (task-started #{task-2 (practice roshambo)})
[0 Jack] (task-started #{task-4 (practice roshambo)})
[0 Jill] (task-started #{task-5 (prime)})
[0 Jill] (task-started #{task-6 (choose gesture)})
[0 Jack] (task-started #{task-9 (prime)})
[0 Jack] (task-started #{task-10 (choose gesture)})
[500 Jack] (task-started #{task-11 (gesture rock)})
[500 Jill] (task-started #{task-7 (gesture rock)})
[1000 Jack] (gestured #{agent-1 Jill} rock)
[1000 Jill] (gestured #{agent-1 Jill} rock)
[1000 Jack] (gestured #{agent-2 Jack} rock)
[1000 Jill] (gestured #{agent-2 Jack} rock)
```

Figure 7: Jack and Jill play version 3 of Roshambo.

`Jack rock)`[1]. The forms in the left hand column show the time stamp and the agent–notice that choosing a gesture takes 500 ms., and gesturing takes 500 ms., so the "run" ends after 1 second (1000 ms).

5 Bundles of joy: Providing different procedures for different agents

Of course, it is a bit silly for Jack and Jill to have the same strategy of always playing "rock." We can give agents separate libraries of procedures and primitives, called *bundles* by using the `:in-apex-bundle` form. In the initialization function, we specify which bundles an agent has access to. In version 4 (`roshambo4.lisp`), we'll specify that Jill always plays "paper," and Jack always plays "rock," by giving them different `(choose gesture)` primitives. Any procedures or primitives defined before bundles are defined are shared by all agents. The changed code is in Figure 8.

6 Fair play: Inter-agent communication

Now we have multiple agents, but they don't know who has won the game. We need a way for agents in Apex to communicate to one another. The normal way for this to occur in Apex is through the use of *routers*. A router is a "publish-and-subscribe" mechanism for agents; that is, agents subscribe to a router, and, when a message is published to the router, each subscriber receives the message. Agents can publish messages to routers they are subscribed to. Figure 9 shows an example. As we said previously, agents are automatically subscribed to a default router. The Apex Lisp function `inform` sends messages to routers. When agents are created, they can be assigned a list of routers. Figure 10 shows the new

[1]To show the `gestured` events in Sherpa, select the Trace view, press the Settings button on the Trace view toolbar, and select the `gestured` checkbox under 'App' (Application) events.

```
(defun initialize-sim ()
  (let ((locale (make-instance 'locale :name 'gameroom)))
    (make-instance 'agent
      :name "Jill"
      :locale locale
      :use-bundles '(:paper-strategy)
      :initial-task '(practice roshambo))
    (make-instance 'agent
      :name "Jack"
      :locale locale
      :use-bundles '(:rock-strategy)
      :initial-task '(practice roshambo))))

(in-apex-bundle :rock-strategy)    ;; always choose rock
(primitive (choose gesture)
  (profile brain)
  (duration (500 ms))
  (return 'rock))

(in-apex-bundle :paper-strategy)   ;; always choose paper
(primitive (choose gesture)
  (profile brain)
  (duration (500 ms))
  (return 'paper))
```

Figure 8: Roshambo, version 4. Using Bundles. Changed procedures only.

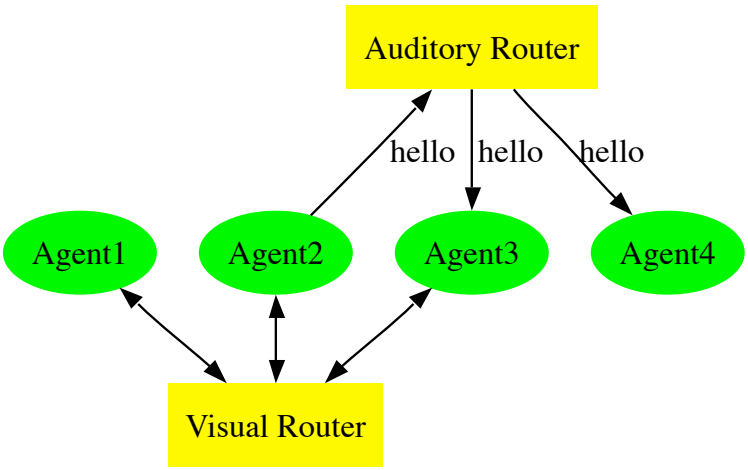

Figure 9: Publish and Subscribe Routers. The message Agent 2 published to the Auditory Router is passed to Agents 3 and 4. Agent 2 also receives the message. Agent 1, not a subscriber, does not receive the message.

initialization code–note that the Lisp class for a router is *ps-router*. We'll give our agents a visual router (for gestures) and an auditory router (for spoken messages). Figure 10 shows how the primitives `gesture` and `say` use `inform` to send messages to their respective routers. The code can be found in the file `roshambo5.lisp`. Notice that the `say` primitive calculates the duration based on the length of the message (400 ms. per word).

Rather than have the agents merely practice Roshambo, Jack's goal is now to `(play roshambo with jill)`, and Jill's goal is now to `(play roshambo with jack)`. The basic plan for playing a game of Roshambo against an opponent is:

1. Prime and choose a gesture,
2. Make the play gesture,
3. When you see the opponent's gesture, determine the winner,
4. Declare the winner.

In previous versions, we had our agents first prime, and then choose a gesture. But of course, there's no reason that our agents can't be thinking of a gesture as they are priming. Our simulation now has a bit of multi-tasking in it: they can potentially think and make a priming motion at the same time. (Well, the `prime` procedure doesn't do anything yet, but we'll fix this soon). Figure 11 shows the rest of the Version 5 code that has changed.

The interesting new things in Figure 11 are (1) using routers to communicate between agents and (2) adding a bit of multi-tasking to our agents. Our procedure `(play roshambo with ?opp-name)` is not strictly sequential, so we have to annotate steps so they don't begin until logically previous steps are completed. Figure 11 shows yet another syntactic shortcut:

```
(defun initialize-sim ()
  (let ((locale (make-instance 'locale :name 'gameroom))
        (auditory-router (make-instance 'ps-router :name 'auditory-router))
        (visual-router (make-instance 'ps-router :name 'visual-router)))
    (make-instance 'agent
      :name 'jill
      :locale locale
      :use-bundles '(:paper-strategy)
      :routers (list auditory-router visual-router)
      :initial-task '(play roshambo with jack))
    (make-instance 'agent
      :name 'jack
      :locale locale
      :use-bundles '(:rock-strategy)
      :routers (list auditory-router visual-router)
      :initial-task '(play roshambo with jill))))

(primitive (index (gesture ?gesture))
  (profile hand)
  (duration (500 ms))
  (on-completion
   (inform `(gestured ,+self+ ,?gesture)
     :router (router-named 'visual-router))))

(primitive (index (say . ?something))
  (profile voice)
  (duration (list (* (length ?something) 400) 'ms))
  (on-completion
   (inform `(said ,@?something)
     :router (router-named 'auditory-router))))
```

Figure 10: Roshambo, version 5, showing initialization code and primitives for routing messages.

```
(procedure
  (index (play roshambo with ?opp-name))
  (step (find agent ?opp-name => ?opponent))
  (step (prime)
        (waitfor <?find>))
  (step (choose gesture => ?mine))
  (step (gesture ?mine)
        (waitfor <?prime> <?choose>))
  (step (determine winner +self+ <?mine> <?opponent> <?other>
         => (?winner ?winning))
(waitfor ?gesture (gestured <?opponent> ?other)))
  (step (say winner is <?winner>)
(waitfor <?determine>))
  (step (terminate)
(waitfor <?say>)))

(primitive
  (index (determine winner ?person1 ?gesture1 ?person2 ?gesture2))
  (profile brain)
  (duration (300 ms))
  (return
   (determine-roshambo-winner ?person1 ?gesture1 ?person2 ?gesture2)))

(defun determine-roshambo-winner (person1 gesture1 person2 gesture2)
  (cond
   ((eq gesture1 gesture2)
    (list '|a tie| gesture1))
   ((or (and (eq gesture1 'rock)
             (eq gesture2 'scissors))
        (and (eq gesture1 'scissors)
             (eq gesture2 'paper))
        (and (eq gesture1 'paper)
             (eq gesture2 'rock)))
    (list person1 gesture1))
   (t (list person2 gesture2))))
```

Figure 11: Roshambo, version 5, playing procedures and primitives.

```
[0 jill] (task-started #{task-2 (play roshambo with jack)})
[0 jill] (task-started #{task-5 (find agent jack)})
[0 jill] (task-started #{task-6 (prime)})
[0 jill] (task-started #{task-19 (gesture priming)})
[0 jill] (task-started #{task-7 (choose gesture)})
[500 jill] (gestured #{agent-2 jack} priming)
[500 jill] (gestured #{agent-1 jill} priming)
[500 jill] (task-started #{task-8 (gesture paper)})
[1000 jill] (gestured #{agent-1 jill} paper)
[1000 jill] (task-started #{task-9 (determine winner #{agent-1 jill}
              paper #{agent-2 jack} priming)})
[1000 jill] (gestured #{agent-2 jack} rock)
[1300 jill] (task-started #{task-10 (say winner is #{agent-2 jack})})
```

Figure 12: Jill loses to Jack who seems to have played "priming".

if there is no tag name for a step, the tag name is created from the first form in the step's activity (so, for example, the default step name for `(choose gesture => ?mine)` is `choose`. (If multiple steps are created with the same first form, Apex will generate a name plus a number, *e.g.*, `choose-1`. It is probably best not to rely on steps automatically generated with numbers–give the steps names of your own choosing).

We do a bit of Lisp programming in Version 5. In particular, we create a Lisp function to return the winner (or `|a tie|` plus the winning or tying gesture as a list. We also use the standard primitive[2] `(find agent ?name)` to convert the name of an agent to the underlying data structure for the agent–this will mean the bindings of `+self+` and `?opponent` will has the same underlying form in call to `(determine winner +self+ <?mine> <?opponent> <?other>)`.

7 Prime time: Understanding when things happen

Let's turn our attention to what seems like a minor thing, but will cause us a headache. We're going to break our application by adding one little seemingly innocuous line–understanding what goes wrong and why will make us better Apex developers.

Here's the line we're going to add: we'll make the `prime` procedure take one step, of making a priming gesture, like so:

```
(procedure :seq (index (prime))
  (step (gesture priming)))
```

The file `roshambo6-broken.lisp` contains this code. Running the code results in the trace shown in Figure 12.

It's apparent that the `determine winner` task is taking the `priming` gesture as a move–even though Jack gestured `priming` at 500 ms, and `rock` at 1000 ms., which is when the

[2]Found in `Apex:apexlib;default-apexlib.lisp`.

`determine winner` task begins. Recall that the `determine winner` procedure has the following waitfor condition: `(waitfor ?gesture (gestured <?opponent> ?other)`.

Sherpa allows you to examine the state of monitors. To look at the state of this monitor when its conditions succeeded, take the following steps (assuming that the broken version 6 has run to completion):

1. Click on "Jill" in the object hierarchy (the panel to the left),
2. Press the "Agenda" button.
3. Open up the agenda tree until the `determine winner` task is visible.

Notice that its monitor state column says the monitor has been satisfied. Hovering the cursor over the word "satisfied" shows a brief view on the monitor, but a more elaborate picture can be seen by:

1. Right-click[3] on the "satisfied" word in the monitor column.
2. Select "Monitor diagram."
3. In a little bit, you should see a diagram that looks something like Figure 13.

Apparently, "priming" is seen as the last gesture, even though we have set the steps up so that a game gesture occurs *after* a priming gesture. What is happening here?

Consider the five steps of procedure `(play roshambo with ?opp-name)`. When this procedure is selected, all of these steps are created and their underlying tasks initiated. The monitor on the `(determine winner)` step is created at the time the step gets created. Crucial to understanding the problem we have just encountered is this: *monitoring for a condition always occurs within an interval of time*, either explicitly or implicitly defined. The implicit interval is always this: an interval beginning at the creation time of the monitor, and ending at infinity. In other words, monitors by default look for conditions that occur at or after the monitor is created.

Because the monitor for `(gestured <?opponent> ?other)` is created at the same time that the `(prime)` step is created, and the priming gesture is the *next* gestured event that occurs, the monitor matches `(gestured <?opponent> ?other)` to `(gestured {agent-2 jack} priming)`, binding `?other` to `priming`.

8 A measured and orderly response: State variables and complex monitors

The solution to our problem with Version 6 of Roshambo will first require us to understand how Apex works with data, either coming from its internal changes (such as the changing state of a task from pending to enabled, for example) or from other sources (such as the gestures of other agents, in our examples), which we'll call *sensors*.

[3]For the Macintosh, this is Apple-click, if you have a one-button mouse.

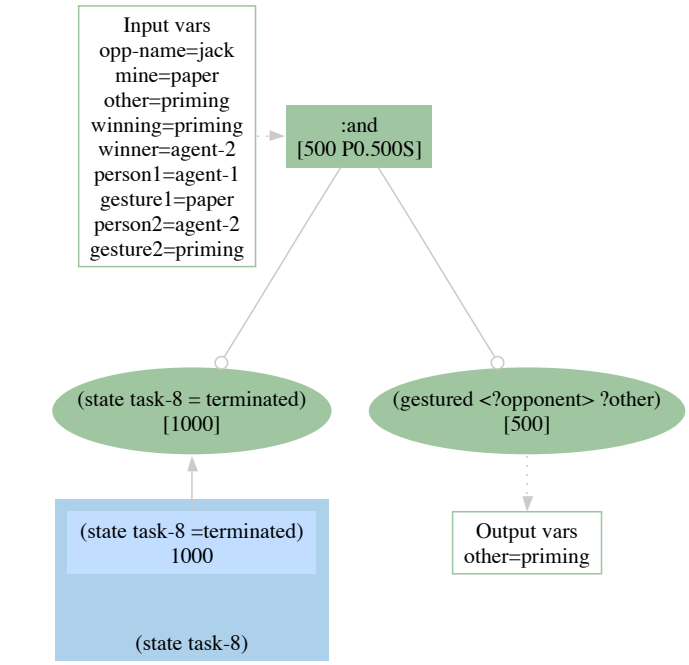

Figure 13: Visual representation of monitor from broken Version 6. Note that "priming" is an input value for variable `?other` caused by the first `gestured` event at 500 ms.

Sensors nominally create time-stamped data, with readings sometimes done at constant intervals, sometimes not. Each reading provides a time-stamped *measurement* of a *state variable*, that is, an *attribute* of some *object* (often the agent, or a component of the agent), which changes over time.

Syntactically, measurements in Apex have the form (*attribute object* = *value*), where *attribute object* define the state variable of interest. In fact, we have already seen state variables used in monitors: monitors for termination of tasks, that look like `(state` *task* `= terminated)`. Events like `(gestured` *agent* `priming)` look very much like a measurement–but note the lack of an equals sign (=). Other events–such as `(said winner is a tie)`–look less like a measurement. Event forms that are not measurements are called *atomic episodes* in Apex.

Since we are in control of the `gesture` primitive, it is trivial to convert its gestures into measurements: now, the gesture primitive will emit forms with an equal sign, that is forms like `(gestured` *agent* `=` *gesture*`)`.

We can monitor for many different conditions, especially when these conditions are based on measurements of state variables. As a bonus, Sherpa offers a useful *state variable view* that shows the changing values of state variables over time, and the monitor views in Sherpa will do the same thing. In the present case, what we would like is to monitor for two things to occur *in order*: first the priming, then the game gesture. Apex provides monitoring capability for this and other conditions. In particular, we can use an `:in-order` monitor to look for a priming gesture followed by a game gesture. Figure 14 shows the changed procedures; these can be found in file `roshambo6-fixed.lisp`.

Other solutions

There are some other possible ways to fix the `(play roshambo with ?opp-name)` procedure, which we will sketch out here.

1. An additional condition could be added to the gesture step, viz., `((gesture <?mine>) (waitfor <?prime> <?choose>` **`(gestured ?opponent = priming)`**. But will this work?

   It won't work. Both the `gesture` step and the `determine` steps are created at the same time, and the `(gesture` *opponent* `= priming)` measurement will be sent to both monitors.

2. Perhaps one can force the `gesture` step to be created later by creating a new subprocedure, something like Figure 15. **Unfortunately this doesn't work either**–the tasks of a procedure are created "at the same time," and the tasks of a subtask are created "at the same time." A little thinking will convince one that, for simulation applications at least, that these two times are, in fact the same time[4].

3. Perhaps one can use an `:or` monitor to monitor for just the right measurement. For example, `(:or (gesture ?opponent = rock) (gesture ?opponent = paper) (gesture`

[4]In a real-time application, there will be real time passing, so the tasks will not, in fact, be created at the same time).

```
(procedure (index (play roshambo with ?opp-name))
  (step (find agent ?opp-name => ?opponent))
  (step (prime) (waitfor <?find>))
  (step (choose gesture => ?mine))
  (step (gesture <?mine>) (waitfor <?prime> <?choose>))
  (step (determine winner +self+ <?mine> <?opponent> <?other> =>
          (?winner ?winning))
          (waitfor ?gesture
             (:in-order
               (gestured ?opponent = priming)
               (gestured ?opponent = ?other))))
  (step (say winner is <?winner>) (waitfor <?determine>))
  (step (terminate) (waitfor <?say>)))

(primitive (index (gesture ?gesture))
  (profile hand)
  (duration (500 ms))
  (on-start
   (inform `(gestured ,+self+ = ,?gesture)
   :router (router-named 'visual-router))))
```

Figure 14: Changes to Version 6 to add an `:in-order` monitor. Changes are highlighted.

`?opponent = scissors)`). This will work, but there's a problem: you can't capture the opponent's move![5]

But there is a way to do this: we can place general constraints on monitors.

```
(waitfor ?gesture
 (:measurement (gestured <?opponent> = ?other))
 :constraints (member '?other '(rock paper scissors))))
```

4. The pattern matcher allows us to define constraints directly on values, using the special `(?is` *`var predicate`*`)` form. So, we can define a Lisp function `(defun valid-p (x) (member x '(rock paper scissors)))`, and we can set the monitor in this way:

```
(waitfor ?gesture
 (:measurement (gestured <?opponent> = (?is ?other valid-p))))
```

9 Over and over and over again: Repeating tasks

It's not much fun to play just one game of Roshambo, so let's allow our agents to play any number of games. The step-level clause `repeating` provides a means to repeat a step while

[5]OK, I admit it, I tried to do this, and was chagrinned to see that I didn't have a binding for `?other`.

```
(procedure (index (play roshambo with ?opp-name))
  ...
  (step (det winner <?mine> <?opponent>)
   (waitfor (gestured <?opponent> = priming)))
  (step (terminate) (waitfor ?det)))

(procedure (det winner <?mine> <?opponent>)
  (step (determine winner +self+ <?mine> <?opponent> ...))
   (waitfor (gestured <?opponent> = <?other>))) ...
```

Figure 15: Putative changes to Version 6. Will this do the trick? (No).

or until some Lisp condition is met. Because the condition is a Lisp condition, we need to set up a Lisp data structure to record the number of games a player has played. We'll just use a Lisp property list to record this. In addition, we'll need a new Apex primitive to increment the game count, and be sure to invoke this primitive after a game is played. Since we're are already saying who the winner is after a result is determined, we'll create a new procedure to do all the post-game work. This is version 7 of our Roshambo code; you'll find it in `roshambo7.lisp`, and the changed procedures in Figure 16.

Of course, we need to change the top-level goals of Jill and Jack to be `(play roshambo 3 times with jack)` and `(play roshambo 3 times with jill)`, respectively–with however many games we want them to play.

10 Strategic decisions: Estimation monitors

Our agents' Roshambo strategies are rather lame–Jill always plays paper, and Jack always plays rock. Let's give Jill a little more intelligence, and have her base her choice on the last gaming gesture by Jack–if he played rock last time, she'll play paper this time, if paper, scissors. Of course, Jill will have to make a choice based on some other strategy the first time she plays, because she'll not have a history to base her decisions on.

As in the previous section, we'll have to be careful to distinguish between game gestures and priming gestures. It's a bit of a pain to distinguish between regular gestures and game gestures, so as part of the `(record game)` procedure, we'll have the agent tell itself the game gesture of the opponent as a measurement; that is, `(game-gesture opponent = gesture)`. We'll also take a slightly different tack–we'll view the values of the `(game-gesture oppoenent)` state variable as persisting. That is, we'll assume that once a player has made a gesture, that gesture remains relevant until another gesture comes along. In doing so, we treat the measurements of a state variable as a *fluent*, that is, a proposition that remains true until a new value comes along. In Apex's terminology, this kind of condition is called an *estimation*–that is, in this case, we are estimating that the value of a state variable at some time t was the last measurement made at or before t. Apex provides two kinds of estimation monitors: *persistence* monitors and *regression-based* monitors. What we want is a persistence monitor for Jill. In general, we can control how long a persistence monitor's estimation will

```
(defparameter *games* (list))
(defun games (player) (or (getf *games* player) 0))
(defun (setf games) (val player) (setf (getf *games* player) val))

(procedure :seq
  (index (play roshambo ?n times with ?opp-name))
  (step s2 (play roshambo with ?opp-name)
    (repeating :until (>= (games +self+) ?n))))

(primitive (increase game count)
  (duration 0)
  (return (incf (games +self+))))

(procedure
  (index (record game ?mine ?opponent ?other ?winner ?winning))
  (step (increase game count))
  (step (say winner is <?winner>))
  (step (terminate)
    (waitfor ?increase ?say)))
```

Figure 16: Version 7 of Roshambo: Playing multiple games; changed procedures.

be considered to hold; for Roshambo, we'll assume it will hold forever.

Figure 17 shows the changed code for Version 8 of Roshambo–in addition, of course, we have to initialize the Jill agent with this bundle. The entire application can be found in `roshambo8.lisp`.

The `choose gesture` procedure for Jill is as complicated a procedure as we have seen. Basically, it sets up two tasks that compete in parallel, and, given the nature of the underlying data, only one with succeed. The two cases are if the opponent hasn't made a previous game gesture, and if the opponent has made a previous game gesture. If the opponent hasn't made a game gesture yet, there will be no measurement of it, we check whether it is not the case that a measurement was made since "the beginning of time," (remember, we're assuming that game gestures persist). In this case, we'll choose a gesture randomly.

If the opponent has made a game gesture, there will be a measurement–and we use a `:persist` estimator to find it. In this case, we'll call an "expert" to decide the gesture based on that last gesture–this is a pretty simple expert, so we can encode it in a primitive clause.

11 Game for more: Next steps in Apex

We started with a simple practice game of Roshambo, and now we have two agents playing each other multiple times, using different strategies. Perhaps you can think of other strategies to implement. The Apex Manual contains the reference material you'll need. And if you come up with a killer new Roshambo strategy–let us know!

```
(primitive (remember (?attr ?obj) = ?value)
  (duration (500 ms))
  (on-start
   (cogevent `(,?attr ,?obj = ,?value) +self+)))

(in-apex-bundle :beat-last-strategy)

(defun random-elt (lst) (elt lst (random (length lst))))

(primitive (choose gesture randomly)
  (profile brain)
  (duration (500 ms))
  (return (random-elt '(rock paper scissors))))

(primitive (choose gesture by last ?last)
  (profile brain)
  (duration (500 ms))
  (return (ecase ?last (rock 'paper) (paper 'scissors) (scissors 'rock))))

(procedure (choose gesture)
  (step s1 (choose gesture randomly => ?gesture)
    (waitfor (:not (:measurement (game-gesture opponent = ?)
                    :timestamp (> 0)))))
  (step s2 (choose gesture by last ?last => ?gesture)
    (waitfor (:measurement (game-gesture opponent = ?last)
                :estimation (:persist))))
  (step (terminate >> ?gesture)
    (waitfor (:or ?s1 ?s2))))
```

Figure 17: Roshambo version 8, `:beat-last-strategy` bundle.