

The internal procedure `accept-action-procedure!` defined in `make-wire` specifies that when a new action procedure is added to a wire, the procedure is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the system's response would differ if we had defined `accept-action-procedure!` as

Answer.

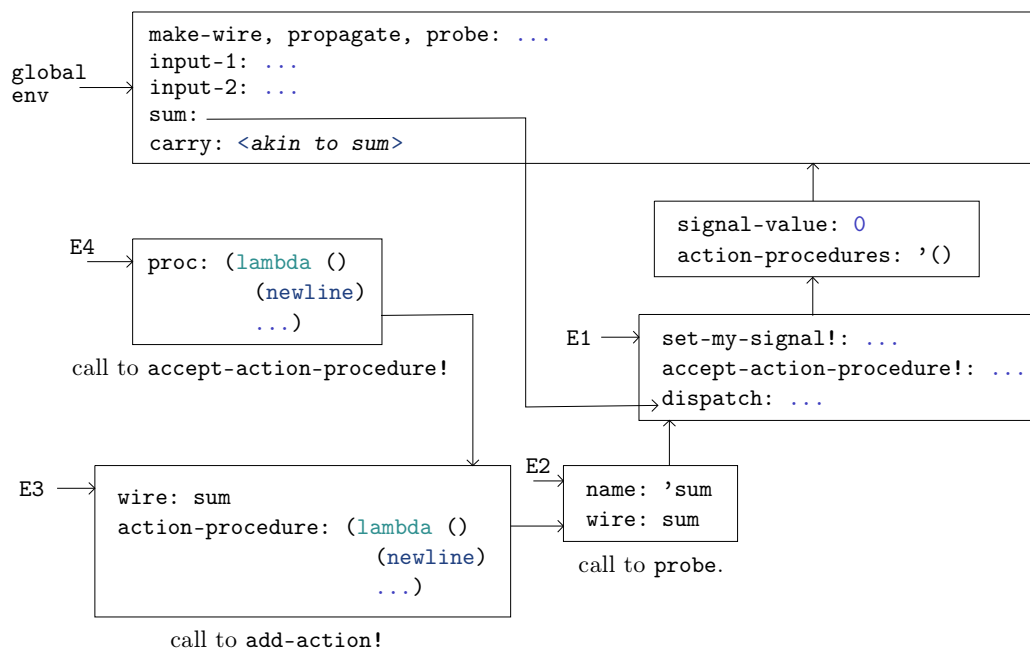
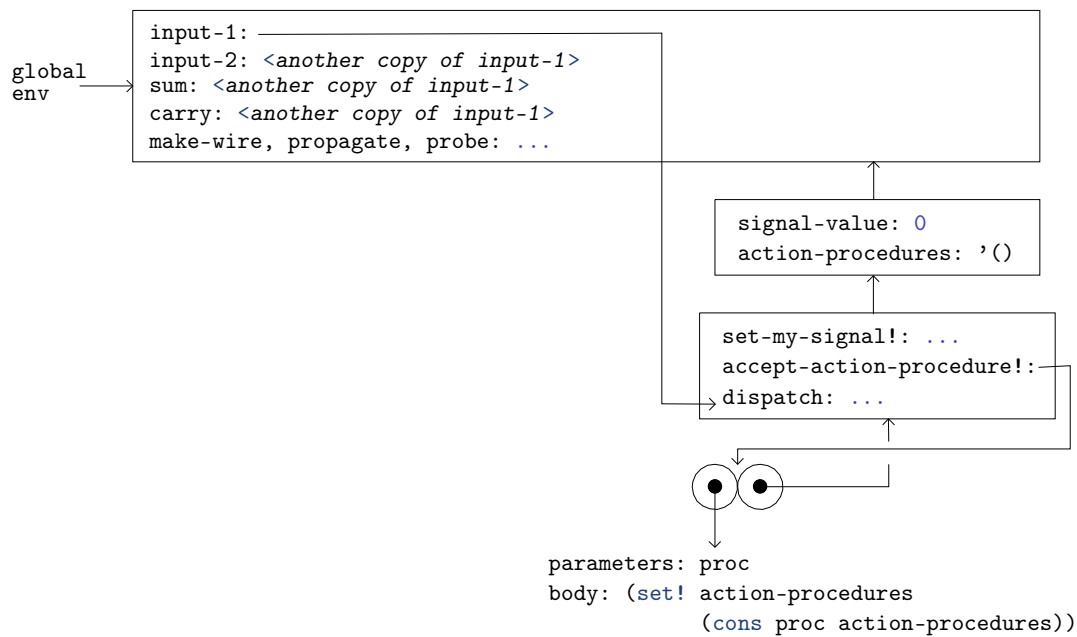
Let's try to trace through the half-adder example in the paragraph above in the environment model to see how the system's response would differ if we had eliminated `(proc)` from the body of `accept-action-procedure`.

[illegible]

frame. Using these we see how the four wires: `input-1`, `input-2`, `sum` and `carry` were defined, as figure 2 shows.

```
(set! action-procedures ;; the value of action-procedures here is '()
      (cons (lambda ()
                (newline)
                ...
                (display (get-signal sum))))
```

1



```
action-procedures))
```

```
(probe 'sum sum)  
;Value: ()
```

```
(probe 'carry carry)
;Value: ()
```

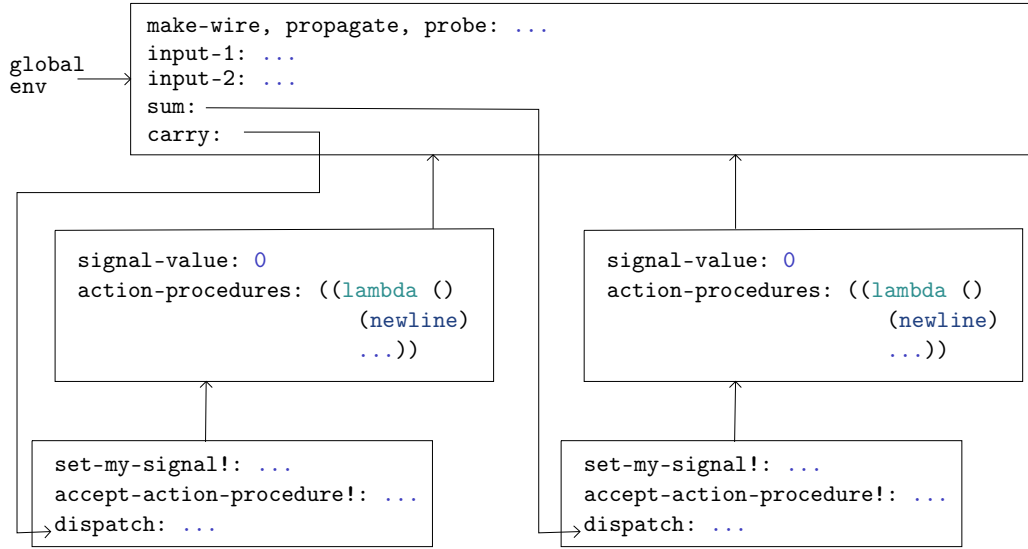


Figure 4. The resulting environment structure in the completion of evaluating `(probe 'sum)` and `(probe 'carry)`.

Next we connect the wires in a half-adder circuit, by evaluating the expression `(half-adder input-1 input-2 sum carry)`. As shown in figure 5, this created the procedure object `half-adder` whose environment was E5 where the local variable `d` and `e` are initialized. Then the arguments `input-1`, `input-2`, `sum` and `carry` were bound onto the formal parameters of `half-adder` in E6 and evaluated the body of `half-adder`, that is, things like `(or-gate input-1 input-2 d)` etc.

In applying `or-gate` to `input-1`, `input-2` and `d`, we set up a new environment E7 whose enclosing environment was E6 and evaluated the body of `or-gate` in E7. To get its job done, `or-gate` in turns called to `add-action!` twice with `input-1` and `input-2` passed as their arguments respectively. The `add-action!` procedure at this point simply added the `or-action-procedure` to the `action-procedures` of `input-1` and `input-2` without running them, thus nothing yet has been added into the agenda. In other words, the `or-gate` was plugged into the half-adder without initializing its output. Figure 6 presents us the environment structure in evaluating `(or-gate input-1 input-2 d)`. The

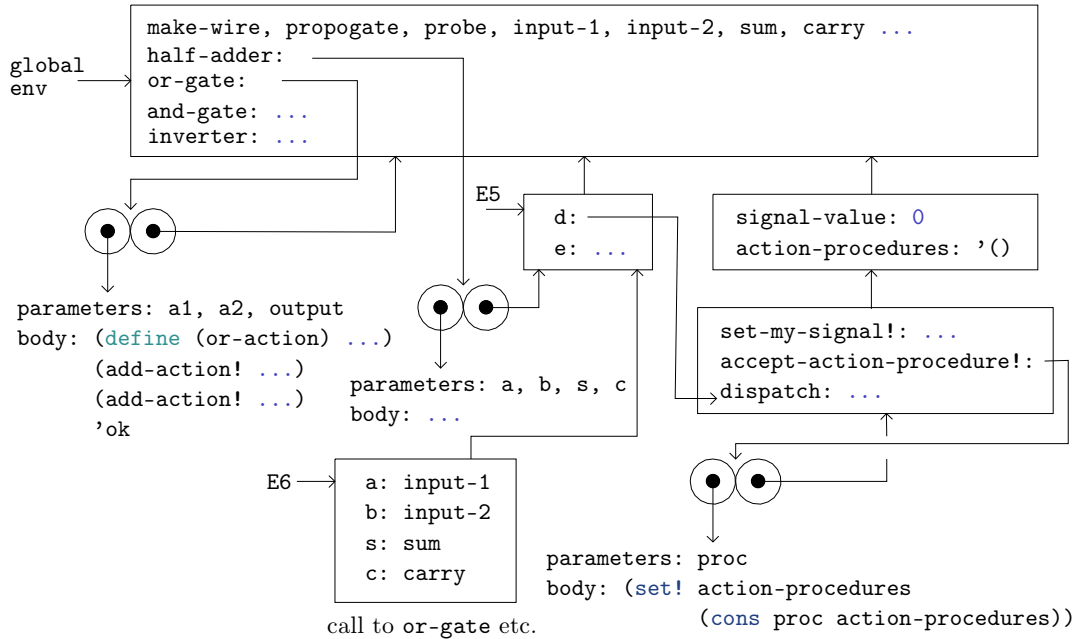


Figure 5. Environments created by applying `half-adder` to `input-1`, `input-2`, `sum` and `carry`.

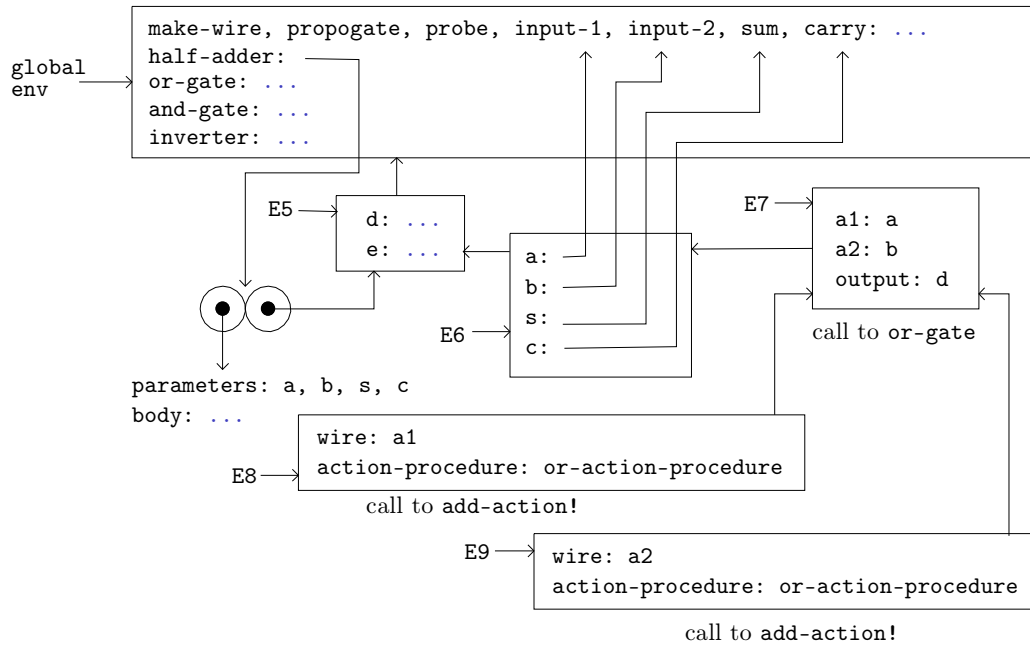


Figure 6. Environments built in evaluating `(or-gate input-1 input-2 d)`.

subsequent expression inside `half-adder` was evaluated similarly and figure 7 together with figure 8 shows the situation after the call to `half-adder`.

Notice that the signal on the wire `e` had gone wrong to stay at 0, which was supposed to be 1 at this point. Because when the `inverter` was connected to the `carry`, it simply added the `invert-input` procedure to the `action-procedures` of the latter one without running. This was precisely the same case we encountered in hooking the `or-gate` up to the `half-adder`. We shall see the problems arising by this non-initialization wiring soon, although nothing exceptional appears when we interact with the interpreter at present:

```
(half-adder input-1 input-2 sum carry)
;Value: ok
```

So far, the agenda still remains empty.

As we change the signal on `input-1` in figure 7 from 0 to 1:

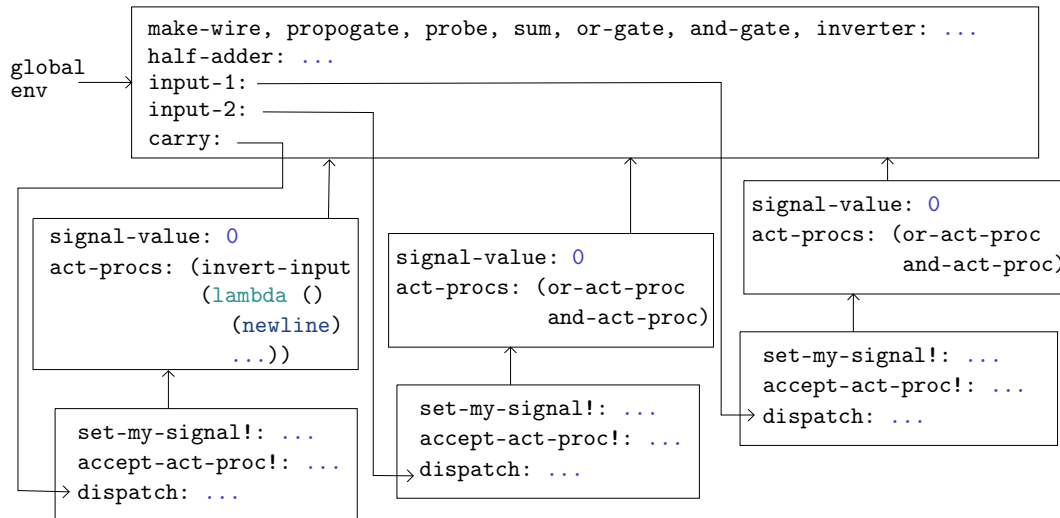


Figure 7. Environments after the call to `half-adder`. (continued on next figure)

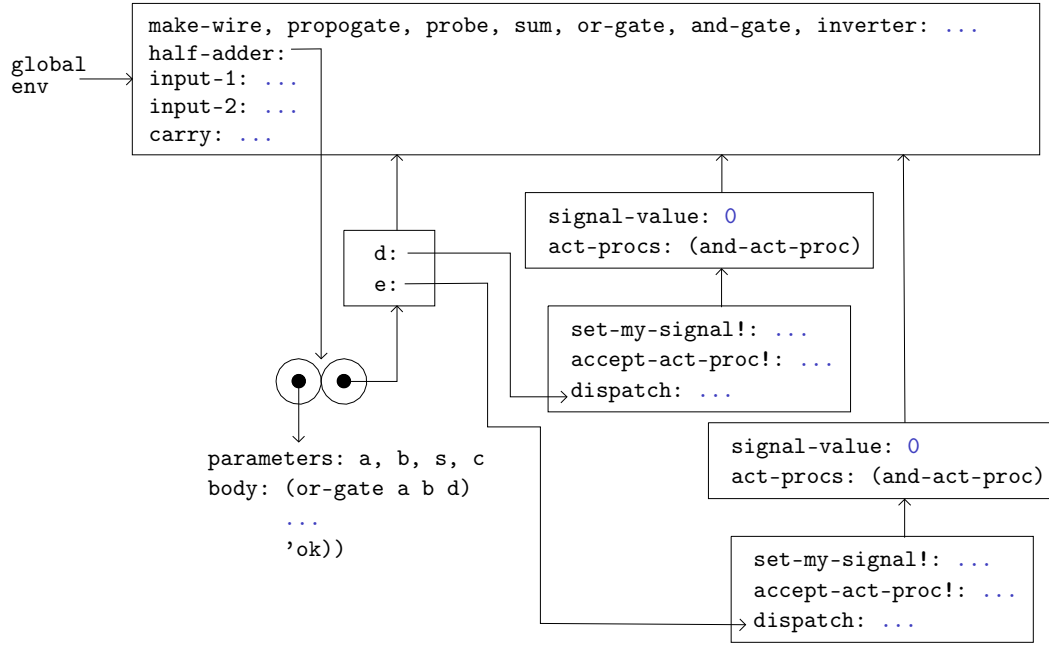


Figure 8. Environments after the call to `half-adder`. (continued)

```
(set-signal! input-1 1)
;Value: done
```

All the action-procedures it possessed were run, due to what stated by `set-my-signal!`, one of the internal procedures of `make-wire`. Note that it was at this juncture that the actions of setting the signal on `d` and `carry` were added into the agenda. The signal on `carry` would stay at 0 even its signal was reset by the `and-gate` after an `and-gate-delay`, for the value of `input-2` remained to be 0. Hence, the inverter would not be triggered and the signal on `e` stayed to be 0. However, `D` would change its signal from 0 to 1 at time 5 for the sake of the `or-gate`. This mutation further triggered `sum` to reset its signal at one `and-gate-delay` later, that is, at time 8, since the signal on `e` remained 0, the signal on `sum` stayed at 0. Figure 9 shows the states of wires after evaluating `(set-signal! input-1 1)` and figure 10 shows contents of agenda at this point. Since neither the `sum` nor the `carry` changes its value during the propagation, nothing more than `'done` is prompted when we run the simulation:

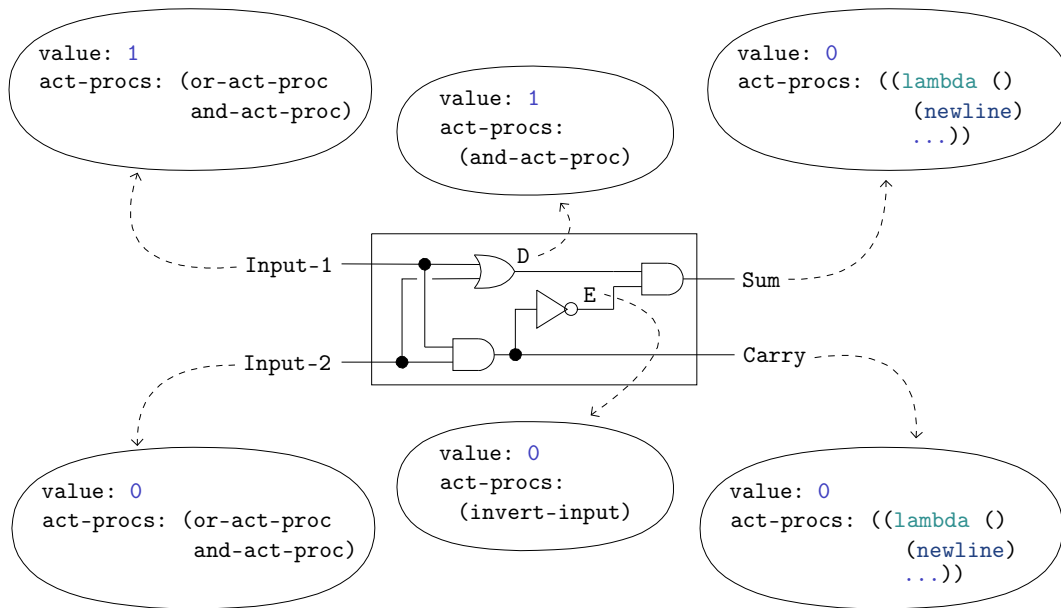


Figure 9. States of wires after evaluating `(set-signal! input-1 1)`.

The-agenda		
	Time	Action
current-time →	0	<no action procedure>
	3	(lambda () (set-signal! carry 0))
	5	(lambda () (set-signal! d 1))
	8	(lambda () (set-signal! sum 0))

Figure 10. Contents of the-agenda after evaluating (set-signal! input-1 1).

```
(propagate)
;Value: done
```

Currently, we are 8 time units from the beginning of the simulation.

Finally, let's come to analyse how the process evolves when we set the signal on input-2 to 1. Figure 11 gives us a clear view of the evolution of process after we changed the signal on input-2.

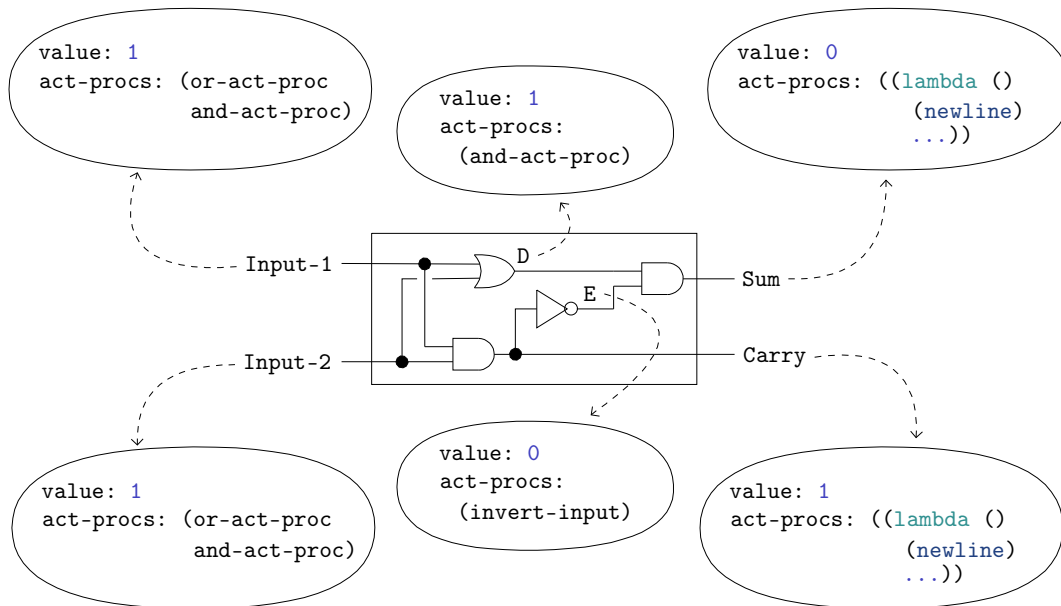


Figure 11. States of wires after evaluating (set-signal! input-2 1)

See that when the signal on input-2 was altered, the action-procedures it contained would be run immediately. Evaluating or-action-procedure added (lambda () (set-signal d 1)) into the agenda, and by scheduling, it set the signal on d to be 1 after an or-gate-delay. Likewise, evaluating and-action-procedure caused the agenda to extend with an action procedure (lambda ()

(set-signal carry 1)) which changed the signal on carry from 0 to 1 after an and-gate-delay. In other words, carry changed its signal to 1 at time 11 and d regenerate the signal of 1 at time 13. The mutation of the signal on carry further trigger the inverter to reset its output, e, to be 0 also at time 13. Since neither of d and e changes its value in this process, the and-gate connected to sum won't be triggered. Figure 12 shows the contents of the agenda just before we run the simulation. To confirm

The-agenda		
	Time	Action
current-time →	8	<no action procedure>
	11	(lambda () (set-signal! carry 1))
		(lambda () (newline) (display 'carry) ...)
	13	(lambda () (set-signal! d 1))
		(lambda () (set-signal! e 0))

Figure 12. Contents of the-agenda before running the simulation for a second time.

our prediction, we just set the signal of input-2 and allow the value to propagate:

```
(set-signal! input-2 1)
;Value: done

(propagate)

carry 11 New-value = 1
;Value: done
```