CS2135: The State Machine Simulation Language

Kathi Fisler, WPI

February 16, 2004

1 Introduction: What's a State Machine Simulator?

1.1 Example 1: Traffic Lights

Imagine that an engineering firm is designing new software for controlling the sequence of lights displayed by a traffic light. Given that a malfunctioning light could result in accidents, they want to monitor whether the new design is producing an expected (and acceptable) sequence of light colors. Suppose they sampled the current light color at short, regular intervals. A sequence that cycled through the colors in order, such as

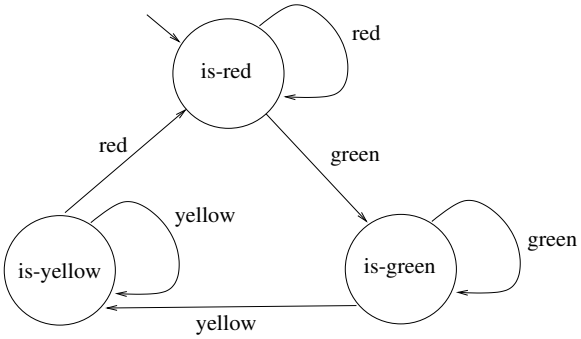
red red green green yellow red red green yellow ...

matches the expected behavior of a traffic light. If, however, the monitor detected an out-of-order sequence, such as

red yellow red yellow yellow red ...

then the monitor should report that the design contains an error.

Our goal in this lecture is to create programs for defining and running monitors. First, how might we define a monitor (forget how to run it for the moment)? Here's a common graphical notation (known as a *state machine*—you may have heard the equivalent term *finite automaton*):



What does this notation mean?

- The circles are known as *states*: they name the different valid configurations that a monitor could be in. For the traffic light, the monitor can be seeing one of the three light colors (red, yellow, or green); the states indicate the color of the light at the last sampling.
- The arrows are known as *transitions* (or sometimes *edges*): they show relationships between valid configurations. Notice that each arrow has a label on it. That label is known as a *guard*. A transition indicates the conditions under which the monitor can change configurations.

  For example, the transition from *is-red* to *is-green* has a guard *green*. This indicates that if the monitor is in the *is-red* configuration and sees a green light on the next sampling, then the monitor enters the *is-green*

configuration. Notice, however, that this is no transition leaving *is-red* with the guard *yellow*. That means that if the monitor is in the *is-red* configuration and the light sampling shows yellow, then the monitor should report an error (because something unexpected happened). The lack of a transition matching the current sample always indicates an error.

- The stubby arrow with nothing at its source marks the *starting configuration*. This example says that all traffic lights start displaying a red light.

This example shows how we can use state machines to write down monitors. Given a monitor as a state machine, how do we run it to watch for errors? In addition to the monitor, we need a list of the data sampled at each time. Assume we had a way to remember the current configuration. We take the first sample off the list, change configurations by matching the sample against the guards, and repeat on the rest of the list. For example, assume we had the state machine shown above and the sample list (*list* 'red 'green 'green 'yellow). Then we would progress through the samples and configurations as shown in the following table (in each line other than the first, the current configuration is the next configuration from the previous line):

Sample list	Current Configuration	Next Configuration
(*list* 'red 'green 'green 'yellow)	*is-red*	*is-red*
(*list* 'green 'green 'yellow)	*is-red*	*is-green*
(*list* 'green 'yellow)	*is-green*	*is-green*
(*list* 'yellow)	*is-green*	*is-yellow*
empty	*is-yellow*	OKAY (stop monitor)

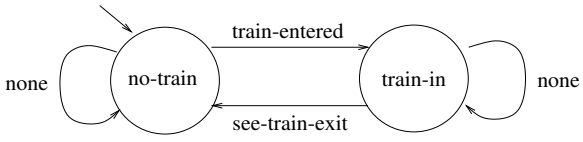
Here's an example of how the monitor progresses on an erroneous sequence of samples:

Sample list	Current Configuration	Next Configuration
(*list* 'red 'red 'yellow 'green)	*is-red*	*is-red*
(*list* 'red 'yellow 'green)	*is-red*	*is-red*
(*list* 'yellow 'green)	*is-red*	ERROR

Our task is to come up with a language for describing state machine monitors and to write an interpreter that runs the monitors against a list of samples, mimicing the tables above. The result of each run is either *okay* or *error*, accordingly.

1.2 Example 2: A Train-Safety Protocol

As a second monitor example, consider a simple messaging protocol for a railway signalman (human). The signalman stands at one end of a very long tunnel. His input samples contain three kinds of messages: one from the operator at the other end of the tunnel saying that a train has entered the tunnel (*train-entered*), an observation that the train exits the tunnel on his end (*is-train-exit*), and a message *none* if neither of other two messages are active. The operator should call an error if he ever gets a *train-entered* message while he is still waiting for a train to exit the tunnel. The following state machine shows the monitor for this operator's protocol:



(Note: This example is part of a slightly larger example of a real protocol that people believed was working until it resulted in a massive train crash in England back in the 1860's—look for details on the Clayton Tunnel accident if you're interested. Protocol validation remains an important area of research in Computer Science, as a means of detecting flaws before accidents occur in practice.)

2 A Language for Monitors

Now that you've seen two examples of monitors, let's develop a (non-graphical) language for writing them down and an interpreter for running them. We'll use the traffic light as the running example throughout these notes. Take a few minutes, and try to develop the data definitions that you need for monitors. Do this before reading on (when I'll present two different languages you might have proposed):

2.1 Language 1: The Structures Approach

As in the past, we can start by identifying the pieces that go into describing a monitor, then write data definitions for those pieces. What pieces do we have here? When we introduced the state machine notation, we identified two kinds of information: states and transitions (with guards). How might we turn these into data definitions?

```
;; A state is a (make-state symbol list[transition])
(define-struct state (name trans-out))

;; A transition is a (make-trans symbol symbol)
(define-struct trans (guard next-state))

;; A monitor is a (make-monitor symbol list[state])
(define-struct monitor (init-state states))
```

Using this approach, our traffic light would look like:

```
(define TL-monitor
  (make-monitor 'is-red
                (list (make-state 'is-red (list (make-trans 'red 'is-red)
                                                (make-trans 'green 'is-green)))
                      (make-state 'is-green (list (make-trans 'green 'is-green)
                                                  (make-trans 'yellow 'is-yellow)))
                      (make-state 'is-yellow (list (make-trans 'yellow 'is-yellow)
                                                   (make-trans 'red 'is-red))))))
```

How would the interpreter work? It would take the monitor and the list of samples and the current state as input. Given the current state and the first sample, it would determine the next state (using a series of filters) and call the interpreter recursively with the rest of the sample and the the next-state. In other words, the code skeleton would look like:

```
;; interp-monitor : monitor list[symbol] → symbol
;; run monitor on samples, returning 'okay or 'error
(define (interp-monitor a-monitor samples)
  (run-monitor (monitor-init-state a-monitor)
               samples
               (monitor-states a-monitor)))

;; run-monitor : symbol list[symbol] list[states] → symbol
;; run monitor on samples from current state, returning 'okay or 'error
(define (run-monitor curr-state samples all-states)
  (cond [(empty? samples) 'okay]
        [(cons? samples)
         (let ([next-state (find-next-state curr-state (first samples) all-states)])
           (cond [(boolean? next-state) 'error]
                 [else (run-monitor next-state (rest samples) all-states)]))]))

;; find-next-state : symbol symbol list[state] → symbol or false
;; finds name of next-state in transition from given state (first arg) on given input/guard (second arg)
```

Exercise: Write *find-next-state* (hint: use *filter*)

2.2 Language 2: The Functions Approach

We can also capture monitors using functions to represent states. Why might this make sense? We can view each state as a delayed computation that's just waiting for an input (the current sample). Each state waits for an input (the sample), then calls the next state (a function) on the rest of the samples.

What might the language definition look like in this framework?

```
;; A run-output is either 'error or 'okay

;; A state is a function (list[symbol] → run-output)
;; (where the input is a list of samples)

;; A monitor is a state (the initial state)
```

Using this approach, we could define our traffic light as:

```
(define TL-monitor
  (local [(define (is-red samples)
            (cond [(empty? samples) 'okay]
                  [(cons? samples)
                   (cond [(symbol=? (first samples) 'red) (is-red (rest samples))]
                         [(symbol=? (first samples) 'green) (is-green (rest samples))]
                         [else 'error])]))
          (define (is-yellow samples)
            (cond [(empty? samples) 'okay]
                  [(cons? samples)
                   (cond [(symbol=? (first samples) 'yellow) (is-yellow (rest samples))]
                         [(symbol=? (first samples) 'red) (is-red (rest samples))]
                         [else 'error])]))
          (define (is-green samples)
            (cond [(empty? samples) 'okay]
                  [(cons? samples)
                   (cond [(symbol=? (first samples) 'green) (is-green (rest samples))]
                         [(symbol=? (first samples) 'yellow) (is-yellow (rest samples))]
                         [else 'error])]))]
    is-red))
```

How would we write the interpreter for the second definition? The monitor is now just a function that expects to receive a list of samples. Given a list of samples, we simply pass them to the monitor function (the initial function), which in turn calls all the other functions for the other states until the list of samples becomes empty.

```
;; interp-monitor : monitor list[symbol] → run-output
;; run monitor on samples, returning 'okay or 'error
(define (interp-monitor a-monitor samples)
  (a-monitor samples))
```

This example illustrates how your choice of data definition for a language can dramamtically change the amount of work needed to run programs in the language.

2.3 Which Language Design is Better?

We've now seen two rather different language definitions: one based on structures and one based on functions. The two different definitions of *interp-monitor* suggest that these give rise to rather different language implementations, but let's try to characterize the differences more clearly:

Differences in Design Style

- The structure-based definition is purely *syntactic* – it captured the information in the monitor (states and transitions) as explicit structures, then left the interpreter to figure out what those structures mean (that's why it's called an interpreter). This is the same style that we used to capture slides (in class) and animations (in lab).
- The function-based definition exploits some knowledge about what a monitor *does* – it goes beyond what the notation looks like and also considers what the notation will be used for (in other words, it takes the *semantics* of the monitors into account, rather than just the *syntax*). This style leaves less work for *interp-monitor*, because much of the work gets buried in the definition of the monitor itself—the monitor *IS* already a (Scheme) program that runs itself!

This is a substantial distinction, one that you can see clearly in the different definitions of *interp-monitor*. For the function-based approach, *interp-monitor* doesn't really have much work to do, while that work is substantial in the structure-based approach.

Differences in Flexibility

What if we wanted to write additional software over monitors, such as a tool that uses a monitor to generate (acceptable or erroneous) sequences (test cases) rather than check them? We could write such a program over the structure-based definition. The function-based definition, on the other hand, is customized to the original problem, so it supports fewer new applications over monitors. Given the importance of flexibility in software design, why would anyone choose the function-based approach?

Differences in Performance

The function-based version will execute *much* faster than the structure-based version on a large example. Why? Because the interpreter for the structures has to do all the work of filtering through the states to find the next states and the transitions. In the function-based version, the only computational work lies in the cond, which will be much cheaper than the filters. In real-world practice, speed is of utmost importance in monitoring and testing software, so that makes a strong case for the function-based approach.

2.4 Summary of Comparison

There is no clear answer to the question of which design is better. It depends on the constraints of the real application you are building. In general, the structure-based version gives you flexibility at a cost penalty, while the function-based version gives you performance with a loss of reuse. You simply need to understand the requirements of your particular application to make this decision.

3 A Technical Note: Interpreters Versus Compilers

Technically, we can summarize the differences between the two styles as follows:

- The structure-based approach implements the monitor language through an *interpreter*. As a reminder, an interpreter is a program that consumes a program (in this case, in the monitor language) as input and returns the result of running that program (the 'error or 'okay).
- The function-based approach implements the monitor language through a *compiler*. A compiler is a program that consumes a program (in some language) and produces a program in another language; the produced (output) program is run to yield the result of running the original program. In this case, we served as the compiler: we manually represented the monitor language as a program in Scheme, then ran the Scheme program to get the result (the 'error or 'okay). If we'd designed a custom notation for writing down monitor programs, instead of relying on the graphical notation, the program that took that notation and produced the Scheme program would be called the compiler.

For those of you who have heard that compilation is faster than interpretation, our discussion of the differences in performance between the two approaches supports this claim.

Warning

Some of you have no doubt heard the phrase “X is a compiled language” or “Y is an interpreted language”. These phrases are *non-sensical*, and show a certain gap in your training. “Interpreted” or “compiled” are attributes of the *implementation*, not of the *language*. ANY language can be interpreted or compiled (we’ve seen one example here). It’s certainly true that some languages are more often implemented via interpreters as opposed to compilers, and vice-versa, but that decision is not intrinsic to the language (rather, it arises from the application for which the language was defined). Don’t make the mistake of using these phrases (unless of course you recant your WPI degree first ...).