State equations for discrete state systems

Victor Yodaiken

August 4, 2016

Abstract

Techniques for describing the behavior and architecture of computer programs and devices.

1 Introduction

How can we describe the behavior and design architecture of interesting discrete state systems such as computer programs and digital devices? Techniques introduced here involve simple algebra, classical state machine theory [1, 3], including the general product [2] and primitive recursion on sequences[4] although it doesn't look like it involves state machines. The approach is unusual enough that I have tried to make the presentation in small steps. Section 2 covers basic techniques and section 3 covers composition of components that can change state concurrently or in parallel. There is an example of a computer network in section 4 and the final section is a brief note on the relationship to automata theory.

2 Basics

1. Start with a set E of events called an event alphabet. This describes all the discrete events that can change system state.

2. The set E^* consists of the finite sequences over E including the empty sequence Nil. Any $s \in E^*$ defines a path for the system from the initial state to the state determined by s.

3. A *state variable*, y is defined by an equation $y = f(\sigma)$ where σ is a free variable over E^* and $f : E^* \to X$ (for some X) is called a sequence map.

4. Sequence maps extract information about state from event sequences[1]. Multiple state variables each associated with a different sequence map can describe different aspects of system state.

[1] I am assuming deterministic systems, but this is not anywhere near as limiting as some claim.

5. If y is a state variable, $y = f(\sigma)$, and $s \in E^*$, then $y(s) = f(s)$ is "the value of y in the state determined by s. Assume $y = f(\sigma)$ from now on.

6. In particular, $y(Nil)$ is the value of y in the initial state, before any events.

7. Writing se to append event e to sequence s on the right, $y(\sigma e)$ is the value of y in the next state, the state after e because $y(\sigma e) = f(\sigma e)$.

   Example: Let L be a state variable so that $L(\sigma e) = e$, then L is the most recent event (except in the initial state where value is not specified).

8. The variable σ is always a free variable over E^* and all state variables depend on σ unless specified otherwise. So $y = y(\sigma)$ is a true statement for any state variable y.

9. If e is a free variable over E, c is some constant, and h is some well-defined map, then the pair of equations

   $$y(Nil) = c \quad y(\sigma e) = h(e, y)$$

   defines y in every state by defining f for all σ .

   To see how it works, rewrite $y(\sigma e) = h(e, y)$ as $y(\sigma e) = h(e, f(\sigma))$since $y = f(\sigma)$. So from the pair of equations, $f(Nil) = c$, $f(\sigma e) = h(e, f(\sigma))$.

   Example: $C(Nil) = 0$ and $C(\sigma e) = C + 1$ defines C to be the length of σ. (Assume e is a free variable over the appropriate event alphabet from now on.)

10. There may be multiple state variables describing a single system with different sequence maps extracting different aspects of state information.

    Example: If z is a state variable and $z = g(\sigma)$, then $(z < y)$ can be rewritten as $(g(\sigma) < f(\sigma))$ which means: z is less than y in any state.

    Example: If y_1, y_2, y_3 are all state variables, each defined by $y_i = f_i(\sigma)$, then $y_1 > y_2 * y_3$ with no constraints on the sequence parameter, means "in any state the value of y_1 is greater than the product of the values of y_2 and y_3. The inequality can be rewritten $f_1(\sigma) < (f_2(\sigma) * f_3(\sigma))$.

11. The general form of the recursive equation for non-trivial state variables often refers to multiple state variables: $y(\sigma e) = \gamma(e, y, z_1, \ldots z_n)$ where each z_i is a previously or simultaneously defined state variable.

3 Composition and concurrency

12. If $z = y(u)$ where u is a sequence valued state variable, z does not depend directly on σ but on the value of u where $u = r(\sigma)$ for some r. The expression is meaningful only if u is sequence valued and the sequences are over the event alphabet of y.

13. Say state variable $u = r(\sigma)$ is a sequence translator from E to B for some event alphabet B, iff for all $s \in E^*$, $u(s) \in B^*$ if $u(s)$ is defined. A sequence translator translates sequences that belong to E^* to ones in some other alphabet. These sequences do not have to be the same length.

14. If $u = r(\sigma)$ is a translator and $z = y(u)$, then $z = y(u) = y(u(\sigma)) = y(r(\sigma)) = f(r(\sigma))$.

15. Suppose $z = y(u)$ and $u(Nil) = Nil$ and $u(\sigma e) = u(\sigma)h(e, z)$ where juxtaposition means append on the right. Require that $h(e, z)$ will be an event in B. Rewriting to show how it works: $u(\sigma e) = u(\sigma)h(e, z) = r(\sigma)h(e, y(u(\sigma)) = r(\sigma)h(e, f(r(\sigma)))$. Since we know $r(Nil) = Nil$, we have defined r for all σ.

    Example: To construct a simple k element shift register from the L variables defined above in 7

    - set $L_i = L(u_i)$ for $0 < i \leq k$
    - Let $u_1(\sigma) = \sigma$ so that $L_i = L$.
    - Let

      $$u_{i+1}(Nil) = Nil \text{ and } u_{i+1}(\sigma e) = u_{i+1}L_i$$

      where the second equation appends L_i as an element to the sequence u_{i+1}.
      To see what this means suppose $u_i = r_i(\sigma)$ for some r_i (defined by the recursive definition of u_i) and expand: $u_{i+1}L_i = u_{i+1}(\sigma)L_i(\sigma) = r_{i+1}(\sigma)L(r_i(\sigma))$.

    Example: Modify the definition of $u_i(\sigma e)$ above to allow for a reverse operation that shifts backwards in a cycle.

    $$u_i(\sigma e) = \begin{cases} u_i(\sigma)e & \text{if } e \neq REVERSE \text{ and } i = 1 \\ u_i(\sigma)L_{i-1} & \text{if } e \neq REVERSE \text{ and } i > 1 \\ u_i(\sigma)L_{i+1} & \text{if } e = REVERSE \text{ and } i < n \\ u_i(\sigma)L_1 & \text{otherwise} \end{cases}$$

16. Say sequence translator u is recursive iff $u(Nil) = Nil$ and $u(\sigma)$ is a prefix of $u(\sigma e)$ (it may be that $u(\sigma) = u(\sigma e)$). (From here on I'm going to assume any translators are recursive unless specified otherwise).Note that non-recursive translators have an orwellian ability to rewrite past history.

17. Any u defined as above is necessarily recursive in the sense of 16 above.

18. A recursive translator does not need to append only one element to the translated event sequence on each event. We could have

    $$u(\sigma e) = concat(u, \gamma(e, ...)).$$

This preserves the key property that the previously produced sequence must be a prefix (perhaps improper) of the new sequence but permits the components to advance by more or fewer steps (since the postfix sequence may be empty).

19. The general form is similar to the general form in 11 above. Suppose $y, y_1, \ldots y_n$ are previously or simultaneously defined[2] state variables then:

$$z = y(v) \quad v(Nil) = Nil \text{ and } v(\sigma e) = concat(v, \gamma(e, z, y_1, \ldots y_n))$$

4 An example

I'm going to construct a simple model of a computer network that could be used to look at protocols for distributed consensus and reliable message transfer. The network assumptions are that message delivery is unreliable but messages are either delivered intact or not at all. Messages may be delivered out of order or duplicated but not corrupted. We can accomodate both multicast/broadcast and point-to-point at the same time.For realism, I'm going to assume messages "time out" at some point so they can't stay in the network indefinitely. I'll keep the model simple, but discuss some possible refinements on the way. First, I'll specify what a network medium (the communications medium itself) should do and then define network devices before connecting both in a system.

Suppose we have a set M of messages and a set D of device identifiers. Let $\mathcal{P}(X)$ be the set of subsets of set X (the powerset). We have maps $dest : M \to \mathcal{P}(D)$ that defines the set of destinations for a message and a map $source : M \to D$ that defines the source device of a message. We'll need some other maps to extract additional information from messages, but not yet.

The network medium is the actual communications link and associated routers and switches. The event alphabet E_{medium} consists of subsets of M to represent the set of messages accepted into the network in parallel in a single step. This set can be the empty set. We need a state variable *Deliver* with values that are subsets of M to represent the messages the network is ready to deliver to devices in the current state. To impose the constraints, define a state variable *Inflight* that tracks all the messages accepted and "when" they were accepted. A more detailed model would incorporate some notion of a clock or real-time, but for first approximation the event counter defined above in 9 can serve as a kind of marker of "when". Messages in flight in the network can expire after some constant $k_{timeout}$ events. So *Inflight* can be a set of pairs (c, m) where c is the event count at the moment the message arrived.

$$Inflight(Nil) = \emptyset \tag{1}$$

$$\begin{aligned} Inflight(\sigma e) = \{(c, m) : ((c, m) \in Inflight \text{ and } c + k_{timeout} < C) \\ \text{or } m \in e \text{ and } c = C\} \end{aligned} \tag{2}$$

[2]In the sense of simultaneous recursion.

Now all we have to do is specify the relationship between *Inflight* and *Deliver*:

$$Deliver \subset \{m : (c, m) \in \mathit{Inflight}\} \tag{3}$$

The devices have an event alphabet $E_{device} = M \cup \{Nil\}$ where event m denotes that a message m has been delivered to the device and Nil is for an event that just advances device state without any message being delivered. For the devices, we can start with a single state variable for the message being sent by the device in any state:

$$\begin{aligned} tx \text{ describes output for a device id } d \in D \text{ iff } tx \in M \cup \{Nil\} \\ \text{and if } tx \in M \text{ then } Source(tx) = d \end{aligned} \tag{4}$$

where $tx = Nil$ means the device does not want to transmit a message.

The event alphabet E of the composed system can be left abstract. All we need is to show how the components connect

$$Sending = Deliver(v) \tag{5}$$

$$\begin{aligned} \text{for all } d \in D,\ \mathit{Transmits}_d = tx_d(u_d) \\ \text{where each } tx_d \text{ describes output for device } d \end{aligned} \tag{6}$$

$$u_d(Nil) = v(Nil) = Nil \tag{7}$$

$$v(\sigma e) = v\ e_v \text{ where } e_v \text{ is a state variable and } e_v \in E_{medium} \tag{8}$$

$$u_d(\sigma e) = u_d\ e_d \text{ where } e_d \text{ is a state variable and } e_d \in E_{device} \tag{9}$$

$$m \in e_v \text{ only if for some } d,\ \mathit{Transmits}_d = m \tag{10}$$

$$m = e_d \text{ only if } m \in Sending \text{ and } d \in Dest(m) \tag{11}$$

Does this model act as anticipated? For example, can a device only receive a message m if device $Source(m)$ previously sent it? Define state variable R to be used with event alphabet E_{device} by $R(Nil) = \emptyset$ and $R(\sigma e) = R \cup e$. Define state variable S in the same context by $S(Nil) = \emptyset$ and $S(\sigma e) = S \cup tx$. Then our claim in the context of the system is that $m \in R(u_d)$ must imply $m \in S(u_{Source(m)})$.

Before the main proof, let's prove that $(c, m) \in \mathit{Inflight}(v)$ and $source(m) = d$ implies that $m \in S_d$. Proof is by induction on state. In the initial state we know that $(c, m) \notin \mathit{Inflight}(v(Nil))$ so there is nothing to prove. Suppose $(c, m) \notin \mathit{Inflight}(v)r$ for any c but for some c, $(c, m) \in \mathit{Inflight}(v(\sigma e)) = \mathit{Inflight}(ve_v)$. By the definition of *Inflight* this means $m \in e_v$ but by 10 this means for some d, $\mathit{Transmits}_d = m$. By definition of $\mathit{Transmits}_d$ this means $tx_d(ve_d) = m$ which means $d = source(m)$ and $S_d(m)$.

Same method for the main part. Since $R(u_d(Nil)) = \emptyset$ there is nothing to show. Assuming the condition is true for any $m \in R(u_d)$ consider $R(u_d(\sigma e))$. If $m \in R(u_d)$ there is nothing to prove by the inductive hypothesis. If $m \notin R(u_d)$ and $m \in R(u_d(\sigma e))$ then $u_d(\sigma e) = u_d\ m$. Which means $m \in Sending(\sigma e)$. By definition, $Sending(\sigma e) = Deliver(v\ e_v)$ so $m \in Deliver(v\ e_v)$ but by equation 3 that means for some c, $(c, m) \in \mathit{Inflight}(v\ e_v)$. QED.

5 Notes on machine theory and solutions to state variable equations

If we say y is a state variable, then there is an equation $y = f(\sigma)$. But is there a solution to this equation - is there a f that satisfies whatever other constraints we have on y? It's easy to find examples where there is no solution: $y < 0$ and $y > 0$. And generally there is not a unique solution. Another consideration is whether there is a *finite state* solution.

If we want y to be implemented by a real device we want f to be finite state in the sense of Myhill-Nerode equivalence. Define a relation R_f on E^* so that $(w,z) \in R_f$ if and only if $(\forall z \in E^*) f(concat(w,q)) = f(concat(z,q))$. Remember Nil is in E^*. It's easy to show R_f is an equivalence. Say f is finite state if and only if R_f partitions E^* into a finite number of equivalence classes[3].. The method for producing a classical state machine from such a relation is straightforward. So if y is a state variable with a finite state solution, then y is possibly something one could implement on a computer or in a digital device of some kind[4].

References

[1] M. A. Arbib. *Algebraic theory of machines, languages, and semi-groups.* Academic Press, 1968.

[2] F. Gecseg. *Products of Automata.* Monographs in Theoretical Computer Science. Springer Verlag, 1986.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Welsey, Reading MA, 1979.

[4] R. Peter. *Recursive Functions in Computer Theory.* Ellis Horwood Series in Computers and Their Applications, 1982.

[3]The *congruence* R'_f so that $(w,z) \in R'_f$ iff $(\forall q,r \in E^*) f(concat(q,w,r)) = f(concat(q,z,r))$ produces a submonoid of the free monoid E^*.

[4]The "possible" is indicated because e.g. some device with $10^{100^{100^{100^{\cdot\cdot}}}}$ states might not actually be feasible in our universe.