Discrete state variables

Victor Yodaiken
POB 1822
Socorro NM 87801 USA

yodaiken@fsmlabs.com

Abstract

Some recursive function techniques for describing large-scale compositional state machines encountered in computer engineering.

1 Introduction

Moore machines[6] provide a mathematically and intuitively clear model of computing systems — both devices and software. In this note, I show how to use recursive equations to apply Moore machines to large systems, to systems composed from multiple layers of connected components, and to partially constrained (incompletely specified) systems.

A Moore machine is usually given as a sextuple $M = (A, X, S, s_0, \delta, \lambda)$ where A is a set of events, X is a set of "outputs", S is a set of states with $s_0 \in S$ the "start state", $\delta : S \times E \to S$ is the state transition function and $\lambda : S \to X$ is the output map. Obviously, each Moore machine determines a map from finite sequences of events to output values $\lambda(\Delta(w))$ where Δ is the extension of δ to sequences. If w is a finite sequence of event symbols write $w.\mathbf{M}$ for the output of Moore machine $\mathbf{M}$ in the state determined by w — the state reached by following w from $\mathbf{M}$'s initial state. We may have function valued Moore machines, for example a Moore machine representing a circuit might output a map from wires to signals so that $w.\mathbf{Circuit}(v)$ would be the signal output on wire v in the state determined by w. And we can have multiple Moore machines all "seeing" the same events. For example,

$$w.\mathbf{Temperature} > 100 \to w.\mathbf{ValveControl} = \mathrm{OPEN}$$

is a nice property to have in a boiler. In general, I will use a bold font for Moore machines to indicate dependency on an event sequence parameter.

System *behavior* can be specified via recursive equations on output. Let ϵ be the empty sequence and let $w\mathfrak{a}$ be the sequence obtained by appending event $\mathfrak{a}$ to sequence w. Then $\epsilon.\mathbf{C} = 0$ and $w\mathfrak{a}.\mathbf{C} = 1 + w.\mathbf{C}$ defines a simple counter. A more elaborate counter $\mathbf{E}$ of *edges* can be defined with the use of a second state machine. Suppose we are concerned with circuits where events are maps $\mathfrak{a} : \mathrm{Pins} \to \{0, 1\}$ so that event $\mathfrak{a}$ sets input pin p to $\mathfrak{a}(p)$. Define **Polarity** by

$$\epsilon.\mathbf{Polarity}(\mathrm{pin}) = 0 \tag{1}$$

$$w\mathfrak{a}.\mathbf{Polarity}(\mathrm{pin}) = \begin{cases} 1 & \text{if } \mathfrak{a}(\mathrm{pin}) = 1; \\ 0 & \text{if } \mathfrak{a}(\mathrm{pin}) = 0; \\ 2 & \text{otherwise.} \end{cases} \tag{2}$$

and then define $\mathbf{E}$ by

$$\epsilon.\mathbf{E}(\mathrm{pin}) = 0 \tag{3}$$

$$w\mathfrak{a}.\mathbf{E}(\mathrm{pin}) = \begin{cases} w.\mathbf{E}(\mathrm{pin}) + 1 & \text{if } \mathfrak{a}(\mathrm{pin}) = 1 - w.\mathbf{Polarity}(\mathrm{pin}); \\ w.\mathbf{E}(\mathrm{pin}) & \text{otherwise.} \end{cases} \tag{4}$$

I am not assuming our state machine are finite — and $\mathbf{E}$ is not. Even though, actual computer systems and processes are finite state it can be useful to use infinite state machines as imaginary measuring devices.

Partially constrained Moore machines are also useful. For example, we can specify that only process p_0 can change an output signal without having any detailed specification of how processes change.

$$w.\mathbf{ControlSignal} \neq w.\mathbf{ControlSignal} \rightarrow w.\mathbf{CurrentProcess} = p_0$$

Many researchers in "formal methods" use the concept of *non-determinism* to approach *partially-specified* systems, but I don't see any advantages and it makes it impossible to model systems with functions from event sequences to outputs.

The next trick is composition. We can specify a circuit state machine **C** constructed from *gate* state machines so that $w.\mathbf{C}(i)$ is the output of gate i within the circuit. First, we need a collection of gate state machines $\mathbf{G}_1, \ldots \mathbf{G}_n$. Then we need a *connection function* γ so that $\gamma(a, i, w.\mathbf{C})$ is the sequence of events induced for gate i when event a is applied to the composite circuit. Note that feedback circuits can be modeled by having $\gamma(a, i, w.\mathbf{C})$ depend on $w.\mathbf{C}(j)$ for some set of gates $0 < j \leq n$. Formally, a product is a simultaneous definition of two state machines: the composite system **M** and, $\mathbf{M}^*$ an (infinite) state machine determining the connections. Let $w \odot z$ indicate concatenation of sequences.

$$\begin{aligned} \text{If } \mathbf{M} &= \prod_i^n \mathbf{M}_i[\phi] \\ \text{then } w.\mathbf{M}(i) &= w_i.\mathbf{M}_i \text{ where } w_i = w.\mathbf{M}^*(i) \\ \text{for } \mathbf{M}^*(i).\epsilon &= \epsilon \\ \text{and } wa.\mathbf{M}^*(i) &= w.\mathbf{M}^*(i) \odot \phi(a, i, w.\mathbf{M}) \end{aligned}$$

One useful property of these products is that $\mathbf{M}^*$ can be used look at the outputs of arbitrary state machines given the inputs seen by a component. That is, if $w_i = w.\mathbf{M}^*(i)$ then $w_i.\mathbf{K}$ is the output of **K** in the state reached by following the input sequence seen by component i. For example if **C** is a composite circuit and **E** is the edge counting state machine defined above then $w_i.\mathbf{E}(pin)$ for $w_i = w.\mathbf{C}^*(i)$ tells us how many edges appeared on the input pin pin for gate i within **C**.

The next section is an example drawn from operating systems that is covered without any use of products. Section 3 looks at gates and constructing a latch as a product of gates.

2 Operating systems: Mutual exclusion

NOTE: This example uses a very old method of mutual exclusion. If any reader can point me to a reference for this method, I would appreciate it very much.

The main state machine we need is **Mem** where $w.\mathbf{Mem}(addr)$ is the contents of memory cell $addr$. Let's also have **Pc** for the current program counter (instruction pointer) and **ProcessPc** so that $w.\mathbf{ProcessPc}(p)$ is the current program counter of process p. Finally **Current** is the active current process and $w.\mathbf{Instruction}$ is true if and only if the last system state change was due to execution of an instruction.

Let $\mathcal{C}$ be a set of addresses or memory locations that constitute a critical region. The mutual exclusion property we want to prove is:

$$(*)\text{There is at most one } p \text{ so that } w.\mathbf{ProcessPc}(p) \in \mathcal{C}.$$

We now need to specify the computer system in which our processes execute.

First, the program counter is the program counter of the current process and only the "current" process can advance by instruction execution (processes can change state due to other events, like interrupts).

$$w.\mathbf{ProcessPc}(w.\mathbf{Current}) = w.\mathbf{Pc} \tag{5}$$

$$\text{If } wa.\mathbf{Instruction} = 1 \text{ then } w.\mathbf{ProcessPc}(p) \neq wa.\mathbf{ProcessPc}(p) \text{ only if } p = w.\mathbf{Current} \tag{6}$$

The opcode for the current instruction is $w.\mathbf{Mem}(w.\mathbf{Pc})$ — the contents of the memory cell pointed to by the program counter. In this example, I'll only need two types of instructions $\mathtt{jump} : z$ (jump to location z) and $\mathtt{move} : x : j$ (store data x in location j). First, jumps don't change memory.

$$\text{If } w.\mathbf{Mem}(w.\mathbf{Pc}) = \mathtt{jump} : j \text{ and } wa.\mathbf{Instruction} \text{ then } w.\mathbf{Mem} = wa.\mathbf{Mem} \tag{7}$$

Second, jumps do set the program counter.

$$\text{If } w.\mathbf{Mem}(w.\mathbf{Pc}) = \mathtt{jump} : j \text{ and } wa.\mathbf{Instruction} \text{ then } wa.\mathbf{Pc} = j \tag{8}$$

Third, move instructions assign a value to exactly one memory location.

$$\text{If } w.\mathbf{Mem}(w.\mathbf{Pc}) = \mathtt{move} : z : j \text{ and } wa.\mathbf{Instruction} \text{ then } wa.\mathbf{Mem}(i) = \begin{cases} z & \text{if } i = j; \\ w.\mathbf{Mem}(i) & \text{otherwise.} \end{cases} \tag{9}$$

Finally, in our simplified computer, a move instruction causes the process to advance the program counter by one (1).

$$\text{If } w.\mathbf{Mem}(w.\mathbf{Pc}) = \mathtt{move} : z : j \text{ and } wa.\mathbf{Instruction} \text{ then } wa.\mathbf{Pc} = w.\mathbf{Pc} + 1 \tag{10}$$

Fix some location g (the gateway to the critical section) with $g \notin \mathcal{C}$ but $g+1 \in \mathcal{C}$ and location q (the exit from the critical section) with $q \in \mathcal{C}$ and $q+1 \notin \mathcal{C}$.

$$g \notin \mathcal{C}, g+1 \in \mathcal{C},\ q \in \mathcal{C}, q+1 \notin \mathcal{C} \tag{11}$$

We require that entry to $\mathcal{C}$ come via instruction execution at g and exit via instruction execution at q. And let's also insist that no process starts off in the critical region.

$$\text{If } wa.\mathbf{Pc} \in \mathcal{C} \text{ then } w.\mathbf{Pc} \in \mathcal{C} \text{ or } \{wa.\mathbf{Instruction} \text{ and } w.\mathbf{Pc} = g\} \tag{12}$$

$$\text{If } wa.\mathbf{Pc} \notin \mathcal{C} \text{ then } w.\mathbf{Pc} \notin \mathcal{C} \text{ or } \{wa.\mathbf{Instruction} \text{ and } w.\mathbf{Pc} = q\} \tag{13}$$

$$\text{For all process ids } p, e.\mathbf{ProcessPc}(p) \notin \mathcal{C} \tag{14}$$

This algorithm relies on self-modifying code and the existence of "reflexive" instructions. We have three specific instruction of interest $B = \mathtt{jump} : g$, $L = \mathtt{move} : B.g$ and $U = \mathtt{move} : L.g$ where B is the instruction executed by a *B*locked process, L is the instruction executed by an entering process that is *L*ocking the entry behind it, and U is the *U*nlock. The algorithm is specified by requiring that g initially contain L and only be changed by execution of instructions either at g or at q, and that q contains U. If memory location g contains L, execution of L will move the current process into the critical region (by specification 10) and also lock the door behind by putting B into memory location g. Any process that tries to enter once the first process is in the critical region will find that the instruction at g jumps back to g — preventing entry. When a process executes the instruction at q, it sets g back to L.

$$e.\mathbf{Mem}(g) = L \tag{15}$$

$$w.\mathbf{Mem}(q)) = U \tag{16}$$

$$\text{If } wa.\mathbf{Mem}(g) \neq w.\mathbf{Mem}(g) \text{ then } w.\mathbf{Pc} \in \{q, g\}\} \text{ and } wa.\mathbf{Instruction} \tag{17}$$

Before proving $(*)$, consider why this algorithm does not work on many modern computers. The reason is that our constraints 9 and 7 are false for processors with non-coherent instruction caches. In modern processors, $w.\mathbf{Mem}(w.\mathbf{Pc}) = z$ and $wa.\mathbf{Instruction}$ do not assure that the last instruction to be executed was z.

Proof. Let **Safe** be a boolean Moore machine defined as follows:

$$w.\mathbf{Safe} = \begin{cases} 1 & \text{if (there is exactly one process id } p \text{ so that } w.\mathbf{ProcessPc}(p) \in \mathcal{C}; \\ & \quad \text{and } w.\mathbf{Mem}(g) = B); \\ & \text{or (there is no process id } p \text{ so that } w.\mathbf{ProcessPc}(p) \in \mathcal{C}; \\ & \quad \text{and } w.\mathbf{Mem}(g) = L); \\ 0 & \text{otherwise.} \end{cases}$$

Of course ϵ.**Safe** since, condition 14 assures that there is no process in the critical region and 15 tells us that ϵ.**Mem**$(g) = L$. Suppose that w.**Safe** $= 1$ and consider wa.**Safe**. There are four cases:

- Suppose there are no processes in the critical region in the w state and none in the critical region in the wa state.

  In this case, constraint 17 tells us that either w.**Mem**$(g) = wa$.**Mem**$(g) = L$ (by induction - so there is nothing to prove) or w.**Pc** $\in \{g, q\}$ and wa.**Instruction**. If w.**Pc** $= q$ then, since by 5 w.**Pc** $= w$.**ProcessPc**$(w$.**Current**$)$ we would have **Current** in the critical region, violating the premise. So if w.**Pc** $= g$ then, by the induction premise we know that w.**Mem**$(g) = L$ and if wa.**Instruction** constraint 9 would make wa.**Pc** $\in \mathcal{C}$ which also violates the premise, so wa.**Instruction** $= 0$ and this means wa.**Mem**$(g) = w$.**Mem**$(g) = B$ by constraint 17. QED.

- Suppose there are no processes in the critical region in the w state and at least one in the critical region in the wa state.

  Following the same reasoning as the previous step w.**Pc** $= g$ and wa.**Instruction** $= 1$ must hold. By 9 we have wa.**Pc** $\in \mathcal{C}$ and wa.**Mem**$(g) = B$. Only **Current** can change so no other process can enter the critical region. That means in the wa exactly one process is in the critical region and **Mem**$(g) = B$ as required.

- Suppose there is exactly one process p_0 in the critical region in the w state and at least one in the critical region in the wa state.

  Let's index the process ids $p_0, \ldots p_k$. Let i be the highest index of any process id so that wa.**ProcessPc**$(p_i) \in \mathcal{C}$. By constraint 12 either w.**ProcessPc**$(p_i) \in \mathcal{C}$ or w.**ProcessPc**$(p_i) = g$ and wa.**Instruction** $= 1$. In the first case, by premise $i = 0$ since there was only p_0 in the critical region in the w state. In such a case we have that only p_0 is in the critical region and by the reasoning of the previous case, wa.**Mem**$(g) = B$. In the other case, the premise is that w.**Mem**$(g) = B$ so by 7 p_i cannot enter the critical region since wa.**Pc** $= g$.

- Suppose there is exactly one process p_0 in the critical region in the w state and there are none in the critical region in the wa state.

  Same idea.

3 A hardware example

Some fundamental notions of boolean functions are needed and these are adapted from [5].

Combinatorial functions. If you hold any input pin of an *and-gate* low (0) long enough or hold any input pin of an *or-gate* high (1) long enough, the output will stabilize at, respectively, 0 and 1. We need to formalize this property in order to be able to describe how a device as simple as a latch works. A circuit has a set of input pins and one or more output values — which may be physically the same pins. Events are maps $Pins \to \{0, 1\}$ which indicate a signal value for the set of input pins. Given any set of pins P, the boolean AND function is $AND(a) = \Pi_{p \in Pins} a(p)$ and the OR function is $OR(a) = max\{a(p) : p \in Pins\}$. The NAND function is $NAND(a) = 1 - AND(a)$. As McCluskey notes, both actual gates and logical functions sometimes can be steered or determined by a subset of pins. If a is an event assigning signals to the set of pins P write $\beta \subset a$ if $\beta : P' \to \{0, 1\}$ where $P' \subset P$. Note that for AND, β restricted to on pin p_0 so that $\beta(p_0) = 0$ determines the value of $AND(a)$ for any $\beta \subset a$. Similarly, if $\beta(p_0) = 1$ then β determines the value of $AND(a)$ for any $\beta \subset a$. Say that β forces b on F is $F(a) = b$ whenever $\beta \subset a$.

Combinatorial circuits and time. Assignments can be considered to be discrete samples of the input values asserted on pins for some time unit. Counting samples measures duration. Define a state machine H to compute how long an assignment subset has been kept stable by

$$\epsilon.\mathbf{Held}(\beta) = 0 \tag{18}$$

$$w\alpha.\mathbf{Held}(\beta) = \begin{cases} w\alpha.\mathbf{Held}(\beta) + 1 & \text{if } \beta \subset \alpha \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

Now say **C** *implements a boolean function* F *with delay* t *if and only if* for any β that forces b on F: $w.\mathbf{Held}(\beta) \geq t \rightarrow w.\mathbf{C} = b$.

A latch. The circuits defined above are called *combinatorial* — output fluctuates as a time delayed function of input. For every such circuit there is some k so that the current output can be deduced from the last k events (if the output is forced). In contrast, a latch is a primitive memory device that can store a value indefinitely. There is no fixed k for a latch — the current output may have been stored at an arbitrarily long time in the past. The SR latch has two input pins 1 and 2 for *set* and *reset*. Let $\mathrm{SET}(1) = 1 = \mathrm{SET}(2) + 1$ and let $\mathrm{RESET}(1) = 0 = \mathrm{RESET}(2) - 1$. To specify a latch, let's first specify a state machine with more visible state.

$$\mathbf{D}.\epsilon = 0.5 \tag{20}$$

$$w\alpha.\mathbf{D} = \begin{cases} 1 & \text{if } w\alpha.\mathbf{Held}(\mathrm{SET}) > t; \\ 0 & \text{if } w\alpha.\mathbf{Held}(\mathrm{RESET}) > t; \\ w.\mathbf{D} & \text{if } \mathbf{D} \in \{0,1\} \text{ and } \alpha = \mathrm{HOLD}; \\ 0.5 & \text{otherwise} \end{cases} \tag{21}$$

Say **L** implements a SR latch with delay t if and only if **L** operates on $\{1,2\}$ assignments and $w.\mathbf{D} \in \{0,1\}$ implies $w.\mathbf{L}(1) = w.\mathbf{D}$.

A latch. An SR latch can be constructed by *cross-coupling* two nand-gates. Suppose that $\mathbf{G}_1$ and $\mathbf{G}_2$ implement nand on $P = \{1,2\}$ with delay t_G. If $i = 1$ let $i' = 2$ and if $i = 2$ let $i' = 1$ — i' is the other wire. Define $\mathbf{L} = \prod_{i=1}^{i\leq 2} \mathbf{G}_i[\mathrm{cross}]$ to operate on $\{1,2\}$ assignments and $\mathrm{cross}(\alpha, i, w.\mathbf{L}) = \langle c \rangle$ with $c(1) = \alpha(i)$ and $c(2) = w.\mathbf{L}(i')$ Now we want to prove that **L** implements an SR latch.

Proof Let $w_i = w.\mathbf{L}^*(i)$. Note that $w.\mathbf{Held}(\mathrm{SET}) \leq w_1.\mathbf{Held}((1,0))$ and $w.\mathbf{Held}(\mathrm{SET}) \leq w_2.\mathbf{Held}((1,1))$ and $w.\mathbf{Held}(\mathrm{RESET}) \leq w_1.\mathbf{Held}((1,1))$ and $w.\mathbf{Held}(\mathrm{RESET}) \leq w_2.\mathbf{Held}((1,0))$ from the definition of cross. It follows that $w.\mathbf{Held}(\mathrm{SET}) = t + 1 + n \rightarrow w_2.\mathbf{Held}((2,1) \geq n$. This is proved by induction on ni and w. It follows that $w.\mathbf{Held}(\mathrm{SET}) = 2t + 1 \rightarrow w_2.\mathbf{Held}((2,1) \geq t$ and thus It follows that $w.\mathbf{Held}(\mathrm{SET}) = 3t + 1 \rightarrow w_1.\mathbf{Held}((2,0) \geq t$ and And so on.

4 Semi-final notes and semigroups

Given a Moore machine (A, S, δ, λ), the transitive extension δ^* is defined by $\delta^*(s, \epsilon) = s$ and $\delta^*(s, w\alpha) = \delta(\delta^*(s, w), \alpha e)$. Say that M *implements* the function $M : A^* \rightarrow X$ so that $M(w) = \lambda_M(\delta^*_M(s_0, w))$. Given a map $f : A^* \rightarrow X$ define a congruence over A^* so that $w \cong_f u$ if and only if for every $z \in A^*$ $f(wz) = f(uz)$. Say that f is *finite* if and only if $E^*/\cong_f$ is a finite set. Note that f is finite if and only if it can be implemented by a Moore machine with a finite state set. For proof it suffices to construct a finite state Moore machine from f using $E^*/\cong_f$ as the state set to show that if f is finite it can be implemented by a finite state Moore machine and and to show that if M implements f and $\delta^*(s_0, w) = \delta^*(s_u, u)$ then $w \cong_f u$ — so that f is finite if M has a finite number of states.

It's useful to note that if $h_1, \ldots h_n$ are finite and g has a finite image, then $f\epsilon = k_0$ and $fwa = g(a, h_1 w, \ldots h_n w, fw)$ must also be finite. Given the same assumptions, $fw = (h_1 w_1, \ldots h_n w_n)$ where $w = \epsilon$ implies $w_i = \epsilon$ and $(wa)_i \mapsto w_i a_i$ where $a_i = g(a, fw))$ must also be finite.

The recursive composition used here to model structure is a functional variant of the *general product* of state machines defined in [2]. There are some unexplored relationships between this product and the classical results of algebraic automata theory, described by Holcombe [4], Arbib [1] and Ginzburg [3]. Simply by refining our congruence, so that $w \sim_u$ iff for all $z, v \in E^*$ we have $f(vwz) = f(vuz)$ we get a monoid under concatenation of representatives. Classical algebraic automata theory investigates the relationship between the monoids induced by finite state machines and the monoids of their products. The focus was on cascade and wreath products with no feedback: in our context where e_i does not depend on fw. This leads at once to some questions about what happens when feedback is constrained, but not forbidden. For example, in modeling circuits, the type of connection map used in the latch example above seem generally useful. These have a couple of constraints including $length(a_i) = 1$. For most circuit technologies it is also required that there are strict limits on fan-in and fan-out. And, for most circuit models we will probably require that $wa.C = wa'.C$ for any a anda' because circuit output would otherwise change instantaneously: something not seen in nature. In general, when we connect components, the connection is a solder dot or a shared memory location or register or a network. That is, the connection is, in practice, constrained to simply copy data. It follows that connection maps, in accurate models of systems, will be very simple functions Would all these constraints tell us something about the structure of the composite state functions and their monoids?

5 Notes

A much earlier version of this work can be found in [8] with applications in [7] and [9]. Unfortunately, it took me 12 years to understand good advice from Professor George Avrunin that the formal logic notation was an impediment instead of an advantage.

References

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

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

[3] A. Ginzburg. *Algebraic theory of automata*. Academic Press, 1968.

[4] W.M.L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1983.

[5] Edward J. McCluskey. *Logic Design Principles*. Prentice-Hall, 1986.

[6] E.F. Moore, editor. *Sequential Machines: Selected Papers*. Addison-Welsey, Reading MA, 1964.

[7] V. Yodaiken and K. Ramamritham. Verification of a reliable broadcast algorithm. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 571 in LNCS. Springer-Verlag, 1992.

[8] Victor Yodaiken. Modal functions for concise representation of finite automata. *Information Processing Letters*, Nov 20 1991.

[9] Victor Yodaiken and Krithi Ramamritham. Specification and verification of a real-time queue using modal algebra. In *IEEE Real Time Systems Symposium*, 1990.