Discrete state variables

Victor Yodaiken
POB 1822
Socorro NM 87801 USA

yodaiken@fsmlabs.com

Abstract

Moore machines[6] provide a mathematically and intuitively clear model of computing systems — both devices and software. In this note, I show recursive functions on finite event sequences to define or specify properties of the very large scale Moore machines corresponding to non-trivial computer systems. Composition is given by a form of simultaneous recursion to allow arbitrary interconnection.

1 Introduction

If a set A contains names of events that can change the state of a discrete state system then each finite string of events w describes a path from the initial state of the system to some resulting state. The externally visible behavior of the system can be specified by a function on strings so that $\mathsf{f}w$ is the output of the system in the state reached by following w from the initial state. In this paper, I will show how to use such string functions to specify the behavior of complex discrete state systems - including those that change state in real-time - and also how to use string functions to describe the construction of complex systems from simpler ones. String functions are mathematically equivalent to Moore machines [6] but turn out to be easier to work with when the underlying state systems are large, partially specified, and constructed from simpler parts.

1.1 Basics

Write ϵ for the empty sequence and wa for the sequence obtained by appending event a to sequence w. Let's look at a few simple definitions.

- Define a counter C by $\mathsf{C}\ \epsilon = 0$ and $(\mathsf{C}\ wa) = 1 + (\mathsf{C}\ w)$ to count the number of events.
- Say any f is a *storage cell* over set V only if $\mathsf{f}\ wa = a$ for any $a \in V$ and $\mathsf{f}\ wa = \mathsf{f}\ w$ for $a \notin V$.
- The behavior of one function may be specified in terms of the behavior of others. Given a temperature sensing boolean signal $\mathsf{Boiling}$ we might require that $\mathsf{Boiling}\ w \leq \mathsf{ValveOpen}\ w$.
- There are several ways to model real-time state system, but to me the most natural method is to treat events as discrete samples of signals applied to inputs. Each sequence of samples then represents an interval or duration. Starting simply, consider a circuit with a single input pin and single output pin. Events can be from the alphabet $\{0, 1\}$ to represent the low (0) and high (1) logical signals over one time unit. Define $\mathsf{High1}\ \epsilon = 0$ and $\mathsf{High1}\ wa = (\mathsf{High1}\ w + 1) * a)$ and $\mathsf{Low1}\ \epsilon = 0$ and $\mathsf{Low1}\ wa = (\mathsf{Low1}\ w + 1) * (1 - a)$ so $\mathsf{High1}\ w$ is the *duration* of the interval in which the input has been held at high and $\mathsf{High1}\ w = 0$ when the most recent event drove the input signal low. Specify that f models a *wire* with delay t only if $\mathsf{High1}\ w > t$ implies $\mathsf{f}\ w = 1$ and $\mathsf{Low1}\ w > t$ implies $\mathsf{f}\ w = 0$.
- To make circuits a little more flexible consider events that are maps from a set P of pins to $\{0, 1\}$. Then a drives pin p to $a(p)$. So $(\mathsf{High}\ w)$ would be a map from pins to integers with $(\mathsf{High}\ w)(p)$ being the duration that the input pin p has been held high. Define $(\mathsf{High}\ \epsilon)(p) = 0$ and $(\mathsf{High}\ wa)(p) = ((\mathsf{High}\ w)(p) + 1) * a(p)$. Say that G is an AND-GATE with delay t only whenever $(\mathsf{High}\ w)(p) > t$ for every pin p then $\mathsf{G}\ w = 1$. I'll return to gates with more realistic models below.

- Consider a memory bank with events consisting of pairs $\langle l, v\rangle$ assigning a value v to a storage location l. Specify that the output $\mathsf{Mbank}\ w$ must be a map from L to V and that $(\mathsf{Mbank}\ wa)(l) = v$ if $a = \langle l, v\rangle$ and $(\mathsf{Mbank}\ wa)(l) = (\mathsf{Mbank}\ w)(l)$ otherwise.

The specification of an AND-GATE or memory bank above can be satisfied by an infinite number of functions on sequence. This is the usual state of affairs - we rarely can define every possible system behavior precisely. Some researchers approach under-specification by treating systems as inherently *non-deterministic*, but this seems unnecessary complication. If we specify that whenever $\mathsf{Temp}\ w > 100$ it must be the case that $\mathsf{ValveOpen}\ w > 0$ we have no need to pretend that valves "choose" what to do non-deterministically. Similarly, if we specify that $\sum_{p \in Processes}(\mathsf{Running}\ w)(p) \leq 1$ we don't have to pretend that the operating system scheduling mechanism is unknowable. The locus of indeterminacy can be conveniently and more realistically placed in the specification instead of in the model or the system itself. We can use the simplifying assumption that G and $\mathsf{ValveOpen}$ and $\mathsf{Running}$ are a total functions on sequences and then let specifications of and-gates, valve control, and scheduler admit to a number of solutions so that the functions are nowhere near totally determined by the specifications.

1.2 Construction

Some researchers model composition on *concurrent threads* or some other programming language model but those are neither mathematically well-behaved or universal. The model used here is drawn from Gecseg's "general product" of state machines [2] which turns out to have a natural representation in terms of recursion. The intuition is that a composite system can be constructed by connecting a collection of simpler systems so that composite events generate sequences of events for the components. The component events are a function of the composite system events and the *feedback* - the outputs of the components. For example, suppose f and g are both wires with delay t and we want E to be constructed by connecting the wires in series. We can define that any composite event is passed directly to f and generates an event for g that is equal to the output of f. That is define E as the product of f as component 1 and g as component 2, so $(\mathsf{E}\ w)(i)$ is the output of component i and so that in the state determined by w, the event a generates sequence $\langle a\rangle$ for component 1 and $\langle b\rangle$ where $b = (\mathsf{E}\ w)(1)$ for component 2.

More formally, $\mathsf{F} = \prod_i[\phi]\mathsf{f}_i$ defines F and F^* simultaneously so that $(\mathsf{F}\ w)(i) = \mathsf{f}_i w_i$ where $w_i = (\mathsf{F}^* w)(i)$ and $(F^*\epsilon)(i) = \epsilon$ and $(\mathsf{F}^* wa)(i) = concatenate(w_i, \phi(i, a, \mathsf{F}w))$.

Here are some illustrative examples.

- A wire constructed by connecting two wires together at one point can be specified as $\mathsf{E} = \prod_{i=1}^{i=2}[S]\mathsf{f}_i$ where f_i is a wire with delay t_i and $S(i, a, \mathsf{F}w) = \langle a\rangle$ if $i = 1$ and $\langle(\mathsf{F}w)(i-1)\rangle$ otherwise.

- A shift register constructed from storage cells $\mathsf{s}_0 \ldots \mathsf{s}_n$ where we have a special event "rotate" and all other events are just data to be pushed into the register.

$$\mathsf{Shift} = \prod_{i=1}^{n}[R]\mathsf{s}_i$$

where

$$R(i, a, \mathsf{Shift}w) = \begin{cases} \langle a\rangle & \text{if } a \neq \text{rotate and } i = 0; \\ (\mathsf{Shift}\ w)(0) & \text{if } a = \text{rotate and } i = 0; \\ (\mathsf{Shift}w)(i-1) & \text{if otherwise ;} \end{cases}$$

- For a more complex construction consider a uni-processor executing some collection of processes that access a shared memory bank. The intuition is that on each step one process advances by reading and writing memory. One of the memory locations "current" is used to hold the index of the process to run. We can reuse the memory bank and hold off on the process specification except to require that there is a map h on process output that produces a sequence of writes to memory. Define $\mathsf{Z} = \prod_{c \in Processes, m}[K]\mathsf{f}_c$ where

each $\mathsf{f}_p : p \in Processes$ satisfies the to be defined specification for processes and f_m is a memory bank. Define $(\mathsf{Running}\ w)(p) = 1$ if and only if $(\mathsf{Z}\ w)(m)(current) = p$ and $(\mathsf{Running}\ w)(p) = 0$ otherwise. Now define $K(i, a, \mathsf{Z}\ w)$ so that for $i \in Processes$ if $(\mathsf{Running}\ w)(i) = 0$ then $K(i, a, \mathsf{Z}\ w) = \langle\rangle$ and if $(\mathsf{Running}\ w)(i) = 1$ then $K(i, a, \mathsf{Z}\ w) = \langle(\mathsf{Z}\ w)(m)\rangle$. So the non-running processes don't change state and the running process gets the contents of memory as an input. As for memory, define $K(m, a, \mathsf{Z}\ w) = h((\mathsf{Z}\ w)(p))$ for the one p so that $(\mathsf{Running}\ w)(p) = 1$. Note that a process can't instantly react to memory contents - as with real computer systems, processes need to read in one state and then write in a second state.

Proving properties of composite systems is, of course, key to getting any real use out of this method and so it's worth pointing out a sort of distributive law that makes such proofs easier. If $\mathsf{f}w$ implies $\mathsf{g}w$ then $\mathsf{f}w_i$ will imply $\mathsf{g}w_i$. Furthermore, we can treat sequence functions as virtual instruments, dropping them down into the states of components by evaluating them in terms of the sequence seen by that component. Here's an example where we might want to know whether one memory location was changed before or after a second memory location. Define $(\mathsf{T}\epsilon)(l) = 0$ and $(\mathsf{T}wa) = 1 + (\mathsf{T}w)(l)$ if $a \neq \langle l, v\rangle$ for some v, and to be 0 otherwise. In our uni-processor system let $w_m = (\mathsf{Z}^* w)(m)$ - the sequence of events seen by the memory bank, and then $(\mathsf{T}w_m)(l) > (\mathsf{T}w_m))(l')$ means l was last written before l' was written.

In the next two sections, I'll look at some more detailed examples and some proofs. The second example covers the construction of a latch from two nand-gates – one of the most amazing and fundamental constructions in computer engineering. The first example proves the correctness of a venerable and beautiful mutual exclusion algorithm based on self-modifying code. In the final section, I'll take a very brief look at the relationship between these functions on sequences, state machines and monoids.

2 Operating systems: Mutual exclusion

NOTE: This example uses a very old method of mutual exclusion and I just can't find the reference. If any reader can help me out on this,I would appreciate it very much.

Take the composite system definition above as a starting point:

$$\mathsf{Z} = \prod_{c \in Processes} [K] f_c \Big/ \bigcup \{m\}$$

where $Processes$ is a set of functions on sequences that all obey a, to be detailed, Process specification, and m is a memory bank on a set $L = \{0, \ldots, N, current\}$ of labels and a set $V = \{0, \ldots N \ldots N'\} \cup Processes$ so that $(f_m w)$ is a map from L to V. Note I've added a special memory location $current$ for storing the index of the current (running) process. Define $\mathsf{Running}\ w = (\mathsf{Z}\ w)(m)(current)$ and define $(\mathsf{Mem}\ w)(l) = (\mathsf{Z}\ w)(m)(l)$ for convenience. Now suppose we also have $(\mathsf{PC}\ w) : Processes \to L$ that gives us the program counter (instruction pointer) for each process. Our processes read instructions from the shared memory and the program counter is the address of the next instruction. Define $\mathsf{Op}\ \ w = (\mathsf{Mem}\ w)((\mathsf{PC}\ w)(\mathsf{Running}\ w))$ for the contents of the memory location pointed to by the program counter of the running process. Finally, we need $(\mathsf{Advance}\ w) = 1$ if and only if the last composite event marked the completion of an instruction and $(\mathsf{Advance}\ w) = 0$ otherwise. Instruction execution is only one of the ways the composite system can change state (I/O events and interrupts should not be ignored although they are not important in this particular exercise.) Define Here are some rules (I'll use the convention from Graham and Knuth that $[condition]$ has value 1 if true and 0 otherwise):

- On instruction execution, at most one process will change PC.

$$((\mathsf{Advance}\ wa) * \sum_p [(\mathsf{PC}\ wa)(p) \neq (\mathsf{PC}\ w)(p)]) \leq 1 \tag{1}$$

- If the pc of the current process points to an instruction $move\ v\ l$ then execution of that instruction sets memory location l to v, advances the program counter of the running process by 1 and has no other effects.

$$\begin{aligned} \text{If } & ((\mathsf{Advance}\ wa) * [(\mathsf{Running}\ w) = p] * [(\mathsf{Op}\ w) = move\ v\ l]) = 1 \\ \text{then } & (\mathsf{Mem}\ wa)(l) = v \text{ and for } l' \neq l, (\mathsf{Mem}\ \ wa)(l') = (\mathsf{Mem}\ \ w)(l') \\ & \text{and } (\mathsf{PC}\ \ wa)(p) = (\mathsf{PC}\ \ w)(p) + 1 \end{aligned} \tag{2}$$

- If the pc of the current process points to an instruction $jump\ l$ then execution of that instruction sets the program counter of that process to l and has no other effects

$$\begin{aligned} \text{If } & ((\mathsf{Advance}\ wa) * [(\mathsf{Running}\ w) = p] * [(\mathsf{Op}\ w) = jump\ l]) = 1 \\ & \text{then for every} l', (\mathsf{Mem}\ wa)(l') = (\mathsf{Mem}\ w)(l') \\ & \text{and } (\mathsf{PC}\ \ wa)(p) = l \end{aligned} \tag{3}$$

Now suppose we have a subset C of L consisting of *critical* locations and we want to ensure that at most one process is in the critical section in any state:

$$(*) \quad \sum_p [(\mathsf{PC}\ w)(p) \in C] \leq 1$$

Let's pick out two locations $entry$ and $exit$ with $entry \notin C$ and $entry+1 \in C$ and $exit \in C$ with $exit+1 \notin C$. Let's require that processes only enter and exit the critical region by executing instructions in the entry and exit locations and that, initially, no process is in the critical section.

$$\text{If } [(\mathsf{PC}\ wa)(p) \in C] > [(\mathsf{PC}\ w)(p) \in C] \text{ then } (\mathsf{PC}\ w)(p) = entry \text{ and } (\mathsf{Advance}\ wa) = 1 \tag{4}$$

$$\text{If } [(\mathsf{PC}\ wa)(p) \in C] < [(\mathsf{PC}\ w)(p) \in C] \text{ then } (\mathsf{PC}\ w)(p) = exit \text{ and } (\mathsf{Advance}\ wa) = 1 \tag{5}$$

$$\sum_p [(\mathsf{PC}\ \epsilon)(p) \in C] = 0 \tag{6}$$

Now we require that the contents of $entry$ should only be changed by execution of the instructions at entry or exit and that the instruction at exit remains unchanged and we will be ready to specify the mutual exclusion algorithm itself.

$$\begin{aligned} & \text{If } [(\mathsf{Mem}\ wa)(entry) \neq (\mathsf{Mem}\ \ wa)(entry)] \\ \text{then } & (\mathsf{PC}\ w)(\mathsf{Running}\ \ w) \in \{entry, exit\} \text{ and } (\mathsf{Advance}\ wa) = 1 \end{aligned} \tag{7}$$

$$[(\mathsf{Mem}\ wa)(exit) = (\mathsf{Mem}\ \ w)(exit)] \tag{8}$$

The key to the algorithm is in the contents of $exit$ and the initial contents of $entry$. Let $z_0 =$ "$jump\ entry$ and $z_1 = move\ z_0\ entry$. Then

$$(\mathsf{Mem}\ \ \epsilon)(entry) = z_1 \tag{9}$$

$$(\mathsf{Mem}\ \ \epsilon)(exit) = move\ z_1\ entry \tag{10}$$

is the algorithm! The basic idea is that the first process to execute an instruction at $entry$ executes $move\ z_0\ entry$ and drops through into C (since $entry + 1 \in C$) and changes $entry$ to hold $jump\ entry$. Any other process trying to enter will execute the jump instruction at $entry$ and this will jump it back to $entry$. When the first process leaves the critical region by executing the instruction at $exit$ it will drop out of the critical region (since $exit + 1 \notin C$) and open the gate again by resetting the value of $entry$ to $move\ z_0\ entry$

To prove that these requirements are sufficient to assure $(*)$ we prove a stronger property that other properties A and B both hold. Note that A implies $(*)$.

$$(A) \quad (\sum_p [(\mathsf{PC}\ w)(p) \in C]) + [(\mathsf{Mem}\ w)(g) = z_1] = 1$$

$$(B) \quad [(\mathsf{Mem}\ w)(entry) = z_0] \leq \sum_p [(\mathsf{PC}\ w)(p) \in C])$$

In the initial state, we have $\sum_p[(\mathsf{PC}\ \epsilon)(p) \in C] = 0$ by 6 and so 9 we have $\mathsf{Mem}\ \epsilon(entry) = z_1$ (A) is true initially, and since $(\mathsf{Mem}\ \epsilon)(entry) \neq z_0$ and no processes are in C, we also have B. Assume A and B are true in the state determined by w and consider wa. Let $n = \sum_p[(\mathsf{PC}\ w)(p) \in C]$ and $n_a = \sum_p[(\mathsf{PC}\ wa)(p) \in C]$. If $n = n_a$ then there is no process such that $[(\mathsf{PC}\ w)(p) \in C] \neq [(\mathsf{PC}\ wa)(p) \in C]$ because otherwise (by 4 and 5) that would be the unique process with this property and so n_a could not be the same as n. By 7 only an instruction execution by a process at $entry$ can change $entry$ and by the induction hypothesis that instruction is either z_1 or z_0. If the instruction was z_1 then after the instruction by 2 the executing process will be in C which contradicts $n = n_0$ and if the instruction was z_0 then after the instruction by 3 the contents of $entry$ don't change. So if $n_a = n$ the number of processes in C is unchanged and the contents of $entry$ is unchanged so A and B must hold in the wa state. If $n_a < n$ then $n_a = 0$ and $n = 1$ there is some unique p so that p leaves the critical region and p executes the instruction at $exit$ which by 2 and 5 means that $(\mathsf{Mem}\ wa)(entry) = z_1$ which means that in the wa state there will be no processes in the critical region and both A and B will be true. If $n_a > n$ then $n_a = 1$ and $n = 0$ and some p enters the critical region. By 4 process p must have program counter at $entry$ and since it does enter, we must have $entry$ containing z_1 and execution of the instruction will cause $entry$ to contain z_0 which means A and B will both be true.

3 A hardware example

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

The signals assigned to input pins can be represented by functions from pins to $\{0, 1\}$ and each such assignment event can be considered to be a discrete sample of the input values asserted on pins for some time unit. Counting samples measures duration. We also want to count *subsets* of samples when we only care about the signals on some pins and the others are "don't care". 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 gate 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 $Pins \rightarrow \{0, 1\}$ which indicate a signal value for the set of input pins. It's convenient to consider these event functions as *sets* of pairs - like $\{(pinA, 0), (pinB, 1)\}$. If we define Held so $(\mathsf{Held}\ w)(\beta)$ tells us how long, how many of the last consecutive samples have contained β as a subset, then an or-gate with delay of t should output 1 if any pin has been held to 1 for at least t time units: if $(\mathsf{Held}\ w)(\{p, 1\}) > t$ for some pin t.

$$(\mathsf{Held}\ \epsilon)(\beta) = 0 \tag{11}$$

$$(\mathsf{Held}\ wa)(\beta) = ((\mathsf{Held}\ wa)(\beta) + 1) * [\beta \subset a] \tag{12}$$

Now say C *implements a NAND gate with delay of* t *if and only if* for any β with some $(p, 0) \in \beta$ $[(\mathsf{Held}\ w)(\beta) \geq t] \leq (\mathsf{C}\ w)$. and for β where $\beta(p) = 1$ for every pin p: $[(\mathsf{Held}\ w)(\beta) \geq t] \leq (1 - (\mathsf{C}\ w))$.

Lets write $(\#\mathsf{C}\ w)(b)$ for the time that the output of C has been equal to b: $(\#\mathsf{C}\ \epsilon)(b) = 0$ and $(\#\mathsf{C}\ wa)(b) = [(\mathsf{C}\ wa) = b] * (1 + (\#\mathsf{C}\ w)(b))$ Note that $(\mathsf{Held}\ w)(\{(p, 0)\}) = t_G + n$ implies that $(\#\mathsf{C}\ w)(1) \geq n$ for a NAND-gate and if $\beta^1 p = 1$ for all the pins p for NAND-gate C then $(\#\mathsf{C}\ w)(0) \geq n$ whenever $(\mathsf{Held}\ w)(\beta^1) \geq t_G + n$.

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's distinguish three special samples $SET = \{(1,1),(2,1)\}$, $RESET = \{(1,0),(2,0)\}$, and $HOLD = \{(1,1),(2,1)\}$ - the fourth combination is not permitted and its effects are unspecified. The idea here is that if $\mathsf{Held}\ w)(SET) > t$ will latch 1, $(\mathsf{Held}\ w)(RESET) > t$ will latch 0, $(\mathsf{Held}\ w)(HOLD) > 0$ will keep the latched value unchanged if a value has been latched. Let's define Lset and LReset to track whether something has been latched.

$$\mathsf{Lset}\ \epsilon = 0; (\mathsf{Lset}\ wa) = [([(\mathsf{Held}\ wa)(SET) \geq t] + (\mathsf{Lset}\ w) * [a = HOLD]) > 0] \tag{13}$$

$$\mathsf{LReset}\ \epsilon = 0; (\mathsf{LReset}\ wa) = [([(\mathsf{Held}\ wa)(RESET) \geq t] + (\mathsf{LReset}\ w) * [a = HOLD]) > 0] \tag{14}$$

Say L implements a SR latch with delay t if and only if L operates on $\{1,2\}$ assignments $\mathsf{L}\ w \in \{0,1\}$ and $(\mathsf{L}\ w)(1) \geq \mathsf{Lset}\ w$ and $(\mathsf{L}\ w)(1) \leq (1 - \mathsf{LReset}\ w)$.

A latch construction. An SR latch can be constructed by *cross-coupling* two nand-gates. Suppose that G_1 and 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 $\mathsf{L} = \prod_{i=1}^{i\leq 2} \mathsf{G}_i[cross]$ to operate on $\{1,2\}$ assignments and $cross(a, i, \mathsf{L}\ w) = \langle c \rangle$ with $c(1) = a(i)$ and $c(2) = (\mathsf{L}\ w)(i')$ Now we want to prove that L implements an SR latch for $t = 3t_G + 3$.

Proof Let $w_i = (\mathsf{L}^* w)(i)$ and let $a_i = cross(a, i, \mathsf{L}\ w)$. Note that since $cross(a, 1, z)(1) = a(1)$ we have $a_1(1) = a(1)$ and Note that since $cross(a, 2, z)(1) = a(2)$ we have $a_2(1) = a(2)$. It follows that
$(\mathsf{Held}\ w)(SET) \leq (\mathsf{Held}\ w_1)(\{1,0\})$
$(\mathsf{Held}\ w)(RESET) \leq (\mathsf{Held}\ w_1)(\{1,1\})$
$(\mathsf{Held}\ w)(SET) \leq (\mathsf{Held}\ w_2)(\{1,1\})$
$(\mathsf{Held}\ w)(RESET) \leq (\mathsf{Held}\ w_2)(\{1,0\})$. So if $(\mathsf{Held}\ w) > 3t_G + 3$ must imply $(\mathsf{Held}\ w_1)(\{1,0\}) > 3t_G + 3$ so $(\#\mathsf{G}_1 w_1)(1) > 2t_G + 2$ so $(\mathsf{Held}\ w_2)(\{(2,1)\}) > 2t_G + 1$. Thus $(\mathsf{Held}\ w_2)(\{(1,1),(2,1)\}) > 2t_G + 1$. It follows that $(\mathsf{Held}\ w_1)(\{(1,0),(2,0)\}) > t_G$. What we have shown is that if $(\mathsf{Held}\ w)(SET) > t$ then $(\mathsf{Held}\ w_2)(\{(1,1),(2,1)\}) > 2t_G + 1$ and $(\mathsf{Held}\ w_1)(\{(2,0)\}) > t_G$. Definitions of $cross$ and nand-gate show that if those conditions apply in the w state, they will be true in the wa state for $a = HOLD$. Thus $(\mathsf{L}\ w)(1) \geq \mathsf{Lset}\ w$. The RESET case is similar.

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 $\delta^*(s, wa) = \delta(\delta^*(s, w), ae)$. Say that M *implements* the function $M : A^* \to X$ so that $M(w) = \lambda_M(\delta_M^*(s_0, w))$. Given a map $f : A^* \to X$ 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 $F = \prod_i [g] h_i$ 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.