State machine models of timing and circuit design

Victor Yodaiken*

FSMLabs Inc. yodaiken@fsmlabs.com
http://www.yodaiken.com

Abstract. This paper illustrates a technique for specifying the detailed timing, logical operation, and compositional circuit design of digital circuits in terms of ordinary state machines with output (transducers). The method is illustrated here with specifications of gates, latches, and other simple circuits and via the construction of devices starting with a SR latch built from gates and then moving on to more complex devices. Circuit timing and transients are treated in some detail. The method is based on "classical" automata and recursive functions on strings. No formal methods, extended state machines, or process algebras are involved but a reference is made to potential applications of the Krohn-Rhodes theorem and other group/monoid based algebraic techniques.

Key words: transducer, Moore machine, primitive recursion, composition, parallel

1 Introduction

Both the time sensitive behavior and circuit architecture of digital circuits can be effectively modelled as ordinary state machines using recursive functions for specifications. State machines are already widely used to model sequential circuits[13], and for circuit synthesis and automata modelling. This paper describes a novel method for specifying complex circuits, composition, timing, and partially specified circuits in terms of classical Moore type transducers. In the computer science literature, it is often argued that state machines need to be "augmented" with clocks or otherwise to model timing (see for example [9, 12, 1]) but we will do without such additives here. Retaining the mathematical basis of un-augmented, "organic" state machines has both practical and theoretical advantages discussed in section 5. This note starts with the example of logic gates and latches constructed from cross-coupled gates, where timing and transient signals must be taken into account. Section 3 makes the case that the methods used in section 2 actually are equivalent to ordinary automata. Section 4 carries us into proofs and more examples from simple digital circuit design.

The focus in this paper is on the by-hand methods of specification and verification, but the methods described here seem to be well suited to automation.

* Reformatted and edited Jan-18-10, March 16 2010

2 Describing circuits

2.1 Behavior

Behavior should encompass the possibility of transient outputs and inputs, conditions that leave output undetermined, and "don't care" inputs and outputs. The basic idea here is to model circuits as state machines with output, possibly state machines constructed as products of simpler state machines. The recursive function technique permits convenient specification of enormous state machines, state machines with parameters, and quite arbitrary interconnect.

A state machine with output (a transducer) M can be treated as a map from sequences of inputs to output values with $M(w)$ corresponding to the output of M in the state reached by following the sequence w from M's initial state. If we put $M(\Lambda) = x_0$ where Λ is the empty sequence, we have defined the initial state output of M — something that is usually unspecified in state machines modelling circuits. If we then put $M(wa) = f(M(w), a)$ where wa is the sequence obtained by appending input a to sequence w, we have totally determined the input/output behavior of M.

For state machines modelling circuits, inputs can be "samples" of the input signals applied to the input pins during the shortest interval of interest (time units may be picoseconds or microseconds or femtoseconds, depending on the devices being studied). A sample assigns signal levels to pins. To start, we can suppose signal levels are limited to $\{0, 1\}$. Samples are then maps $InputPins \rightarrow \{0, 1\}$ where the set of input pins depends on the device in question. Then $Time(w)$ defined by $Time(\Lambda) = 0$ and $Time(wa) = 1 + Time(w)$ is a (not finite) state machine that simply counts the number of time intervals that passed since the initial state. But $High(w, p)$ defined by $High(\Lambda, p) = 0$ and $High(wa, p) = (High(w, p) + 1) * a(p)$ tells us how long pin p has been kept high in consecutive preceding time intervals.

Say that G is an OR gate with input pins P and propagation delay t only if $High(w, p) \geq t$ for some $p \in P$ implies that $G(w) = 1$. There are an infinite number of state machines that satisfy this constraint — corresponding to the process variability of actual gates.

Instead of defining Low to correspond to $High$, define a single function to see how long a pin has been *held* at a level.

$$\text{For } b \in \{0,1\} \text{ and pin } p$$

$$H(\Lambda, p, b) = 0 \tag{1}$$

$$H(wa, p, b) = \begin{cases} H(w, p, b) + 1 & \text{if } a(p) = b \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Or more compactly $H(wa, p, b) = (H(w, p, b) + 1) * (1 - |a(p) - b|)$.

A NAND gate can be specified as follows.

Definition 21 *N is a NAND-gate with propagation delay $t > 0$ on pins P only if*

$$\textit{Whenever } H(w, p, 0) \geq t \textit{ for any } p \in P \textit{ then } N(w) = 1 \tag{3}$$

$$\textit{and}$$
$$\textit{Whenever } H(w,p,1) \geq t \textit{ for all } p \in P \textit{ then } N(w) = 0 \tag{4}$$

The idea here is to minimally specify what we can expect from a gate and not constrain for example transient behavior unless we have some reason to be able to specify in more detail.

Given a function f, let's have $\#f$ count how long the output of f has been stable.

$$\#f(\Lambda) = 0 \tag{5}$$
$$\#f(wa) = \begin{cases} \#f(w) + 1 \text{ if } f(wa) = f(w) \\ 0 \qquad\qquad \text{otherwise.} \end{cases} \tag{6}$$

So it's useful to note that if N meets the requirements for a NAND-gate with delay t then holding an input pin low for $t + k$ time units causes output to be kept stable for at least k time units. This is a pretty obvious fact, but let's prove it just to show a style of recursion based proof that seems eminently automatable.

$$\text{For any } k \geq 0[\#N(w) \geq H(w,p,0) - (t+k)] \tag{7}$$

Note that $\#N(w) \geq 0$ by definition. If $H(w,p,0) \leq t$ then $H(w,p,0) - (t + k) \leq 0$ so there is nothing to prove. But if $H(w,p,0) > t$ then the NAND gate constraints require that $N(w) = 1$ and the rest follows by induction on w. For $w = \Lambda$ there is nothing to prove since $H(\Lambda,p,0) = 0$ by definition. Suppose that $H(wa,p,0) \geq t$. If $H(wa,p,0) = t$ then $H(wa,p,0) - (t+k) = 0$ and there is still nothing to prove since $\#N(w) \geq 0$ by definition. If $H(wa,p,0) > t$ then it must be that $H(w,p,0) \geq t$ since by construction $H(wa,p,0) - H(w,p,0) \leq 1$. Thus $N(w) = 1$ and $N(wa) = 1$ so $\#N(wa) = \#N(w) + 1$ and if $\#N(w) \geq H(w,p,0)$ as assumed by recursion, the inequality must still hold in the wa state.

I'll postpone multi-output circuits to section 4.

2.2 Composition

Now consider how two NAND-gates could be cross-coupled to create a latch. In this case, the inputs to the composite circuit *induce* inputs for the two component gates. Suppose that N_1 and N_2 are solutions to constraints 3 and 4 with input pins $\{1,2\}$. We want to construct some L that contains copies of N_1 and N_2 and that accepts input samples that are maps $\{set, reset\} \to \{0,1\}$. When an input a is applied to L an input c_1 will be generated for the copy of N_1 and an input c_2 for the copy of N_2. Input $c_1(1) = a(reset)$ and input $c_2(1) = a(set)$ as the two inputs are connected directly to the input pins of the components. But $c_1(2)$ will be set to the output of N_2 in the current state and $c_2(2)$ to the output of N_1 in the current state.

$$\text{Input:}w \rightarrow \begin{pmatrix} \text{Input:}u_1 \rightarrow \text{Machine:}N_1 \rightarrow \text{Output:}x_1 \\ \text{Input:}u_2 \rightarrow \text{Machine:}N_2 \rightarrow \text{Output:}x_2 \end{pmatrix} \rightarrow \rightarrow \text{Output:}(x_1, x_2)$$
$$\Uparrow \qquad \Downarrow$$
$$\Leftarrow \quad \Longleftarrow\!\!=\!\!=\!\!= \text{ feedback } \Longleftarrow\!\!=\!\!=\!\!= \quad \Longleftarrow$$

Define $L(w) = (N_1(u_1), N_2(u_2))$ where u_1 and u_2 are the induced input sequences that are themselves functions of L and w. We need to define a mapping $w \mapsto (u_1, u_2)$ and we do so recursively. Note that when $w = \Lambda$ we must have $u_1 = u_2 = \Lambda$ so that the components do not see any input until the composite system sees input.

Definition 22 *L is a constructed RS-Latch with base propagation delay t on pins $\{r, s\}$ only if*

$$L(w) = (N_1(u_1), N_2(u_2)) \quad (8)$$
$$\textit{where each } N_i \textit{ is a NAND-gate with delay } t \textit{ on pins}\{1, 2\} \quad (9)$$
$$w \mapsto (u_1, u_2) \quad (10)$$
$$\textit{and when } w = \Lambda \textit{ then } w \mapsto (\Lambda, \Lambda) \quad (11)$$
$$\textit{and when } w \mapsto (u_1, u_2) \textit{ then } wa \mapsto (u_1c_1, u_2c_2) \quad (12)$$
$$\textit{where } c_1(1) = a(reset), c_1(2) = N_2(u_2), c_2(1) = a(set), c_2(2) = N_1(u_1) \quad (13)$$

Define $C(w) = N_1(u_1)$ to extract the q output from the latch.

2.3 Abstraction

Stepping back from the construction, consider an abstract behavioral specification of an RS-latch.

$$latched(\Lambda, b, t_{latch}) = 0 \quad (14)$$
$$latched(wa, b, t_{latch}) \quad (15)$$
$$= \begin{cases} 1 \text{ if } b = 1 \text{ and } H(wa, reset, 0) \geq t_{latch} \text{ and } H(wa, set, 1) \geq t_{latch} \\ \quad \text{or if } b = 0 \text{ and } H(wa, reset, 1) \geq t_{latch} \text{ and } H(wa, set, 0) \geq t_{latch} \\ \quad \text{or if } a(set) = a(reset) = 1 \text{ and } latched(w, b, t_{latch}) > 0 \\ 0 \text{ otherwise} \end{cases}$$

We want to say C is a SR-latch with delay t_{latch} if and only if whenever $latched(w, b, t_{latch}) = 1$ we must have $C(w) = b$. Without proof, if L is constructed from NAND-gates of delay t as specified above, and $C(w) = N_1(u_1)$ then C is a RS-latch with delay $3t + 2$. The $+2$ is an artifact of the model, which imposes a single time unit delay for a signal to propagate between the output and input of any connected components[1]. Of course, this is not a surprising fact, since engineers have been using latches since the 1950s and, by all indications, they work as intended.

Just for fun, consider a NAND-gate constructed from 3 gates. Suppose N_1 and N_2 and N_3 are all NAND-gates with delay t on pins $\{1,2,3\}$ and we want to build a NAND gate with input pins $\{1,2,3,4,5,6,7\}$ from them. Put $N'(w) = (N_1(u_1), N_2(u_2), N_3(u_3))$ and let $N(w) = N_3(u_3)$. Define $w \mapsto (\Lambda, \Lambda, \Lambda)$ when $w = \Lambda$. Then if $w \mapsto (u_1, u_2, u_3)$ define the mapping $wa \mapsto (u_1c_1, u_2c_2, u_3c_3)$ where

$$c_1(1) = a(1), c_1(2) = a(2), c_1(3) = a(3) \tag{16}$$

$$c_2(1) = a(4), c_2(2) = a(5), c_2(3) = a(6) \tag{17}$$

$$c_3(1) = a(7), c_3(2) = N_1(u_1), c_3(3) = N_2(u_2) \tag{18}$$

The glaring difference between N' and the constructed L is that the mapping from w to the u_i involves recursive feedback for L but not for N'.

3 Reduction to Automata

3.1 Representations

We're representing state machines (transducers) with functions on strings.

Correspondence between a transducer M and a string function f.
Input:w $\Rightarrow$ Machine:M $\Rightarrow$ Output:x $f(w) = x$

A Moore machine or transducer is usually given by a 6-tuple

$$M = (A, X, S, start, \delta, \gamma)$$

where A is the alphabet, X is a set of outputs, S is a set of states, $start \in S$ is the initial state, $\delta : S \times A \to S$ is the transition function and $\gamma : S \to X$is the output function. While it is usual to limit Moore machines to finite state sets, and such finite state machines are sufficient to represent any constructable digital system, it's convenient to have infinite state machines available for specifications. For example, a hypothetical infinite counter may be useful in describing the periodic behavior of a finite state device.

[1] If that requirement is a problem the model can be adjusted. For example, we could require that for any circuit element E, $E(wa) = E(wc)$ for any samples a and c — that is, there is at least one unit of delay between the application of a signal and a response. A sensible limitation in any case. Because the output does not depend on input, we can calculate the output using a dummy input. So for example make c be any appropriate sample and let, $wa \mapsto (u_1c_1, u_2c_2)$ where $c_2(2) = N_1(u_1c)$ — eliminating the interconnect delay.

Given M, use primitive recursion on sequences to extend the transition function δ to A^* by:

$$\delta^*(s, \Lambda) = s \text{ and } \delta^*(s, wa) = \delta(\delta^*(s, w), a). \tag{19}$$

So $\gamma(\delta^*(start, w))$ is the output of M in the state reached by following w from M's initial state. Call $f_M(w) = \gamma(\delta^*(start, w))$ the *representing function* of M. A representing function is "finite" if and only if it represents a finite state machine.

The transformation from string function to transducer is via the well known Nerode equivalence relation[2, 11, 10]. Given $f : A^* \to X$ define $f_w(u) = f(w \circ u)$ where $\circ$ indicates string concatenation. Note that it is possible for $w \neq u$ but $f_w = f_u$ meaning that $f(w \circ z) = f(u \circ z)$ for all $z \in A^*$. Intuitively $f_w = f_u$ when both lead to the same state. Let $S_f = \{f_w : w \in A^*\}$.. Say f is finite if and only if S_f is finite. Define $\delta_f(f_w, a) = f_{wa}$ and define $\gamma(f_w) = f_w(\Lambda) = f(w)$. Then with $start_f = f_\Lambda$ we have a Moore machine

$$\mathcal{M}(f) = \{S_f, start_f, \delta_f, \gamma_f\}$$

and, by construction f is the representing function for $\mathcal{M}(f)$. See section 5 for a short discussion of the next step of abstraction: the monoid induced by the state machine.

Any M_2 that has f as a representing function can differ from $M_1 = \mathcal{M}(f)$ only in names of states and by including unreachable and/or duplicative states. That is, there may be some w so that $\delta_1^*(start_1, w) \neq \delta_2^*(start_2, w)$ but since $f_w = f_w$ it must be the case that the states are identical in output and in the output of any states reachable from them. If we are using Moore machines to represent the behavior of digital systems, these differences are not particularly interesting and we can treat $\mathcal{M}(f)$ as *the* Moore machine represented by f.

3.2 Products

Gecseg[4] describes a general automata product suitable to composition of digital circuits that is used in a simplified way here and in a more general way in [16]. A more recent and more algebraic treatment is found in Domosi[3].

Suppose we have a collection of (not necessarily distinct) Moore machines $M_i = (A_i, X_i, S_i, start_i, \delta_i, \lambda_i)$ for $(0 < i \leq n)$ that are to be connected to construct a new machine with alphabet A using a connection map g. The intuition is that when an input a is applied to the system, the connection map computes an input for M_i from the input a and the outputs of the factors (*feedback*).

Definition 31 General product of automata for digital circuits
Given $M_i = (A_i, X_i, S_i, start_i, \delta_i, \gamma_i)$ and h and g where $g(i, a, \boldsymbol{x}) \in A_i$, define the Moore machine: $M = \mathcal{A}_{i=1}^n[M_i, g, h] = (A, X, S, start, \delta, \gamma)$

- $S = \{(s_1 \ldots, s_n) : s_i \in X_i\}$ and $start = (start_1 \ldots, start_n)$

– $X = \{h(x_1 \ldots, x_n) : x_i \in X_i\}$ and $\gamma((s_1 \ldots, s_n)) = h(\gamma_1(s_1) \ldots, \gamma_n(s_n))$.
– $\delta((s_1 \ldots, s_n), a) = (\delta_1(s_1, g(1, a, \gamma(s))) \ldots, \delta_n(s_n, g(n, a, \gamma(s))))$.

I'll state without proof a theorem proved elsewhere that should be reasonably obvious[16].

Theorem 1 If each f_i represents M_i and $f(w) = h(f_1(u_1) \ldots, f_n(u_n))$
and when $w = \Lambda$, $u_i = \Lambda$
and if when $w \mapsto (\ldots u_i \ldots)$, then $wa \mapsto (\ldots u_i c_i \ldots)$ where $c_i = g(i, a, f(w))$.
and $M = \mathcal{A}_{i=1}^n[M_i, h, g]$ **then** f represents M

4 More examples

Let's step up the complexity of the examples to multi-output-pin circuits. In this case, we just add another parameter to select which output pin is being referenced. A single bit adder might output have output pins $\{sum, carry_{out}\} \to \{0, 1\}$ and accept input samples $\{v1, v2, carry_{in}\}$.

$$\begin{aligned} &D \text{ is a delay } t \text{ adder iff and only if} \\ H(w, carry_{in}, b_0) \geq t \text{ and } H(w, v1, b_1) \geq t &\text{ and } H(w, v2, b_2) \geq t \to \\ &D(w, carry_{out}) = \lfloor (b_0 + b_1 + b_2)/2 \rfloor \quad (20) \\ &\text{and } D(w, sum) = (b_0 + b_1 + b_2) \bmod 2 \end{aligned}$$

An n bit adder then has inputs that are maps from $\{carry_{in}, v1_1 \ldots v1_n, v2_1 \ldots v2_n\}$ to $\{0, 1\}$. The output pins are $\{r_1, \ldots r_n, c_{out}\}$. In contrast to a gate, there are never "do not care" inputs. Define $Last(wa) = a$. Define $Stable(\Lambda) = 0$ and

$$Stable(wa) = \begin{cases} 1 + Stable(w) & \text{if } a = Last(w) \\ 0 & \text{otherwise.} \end{cases}$$

Let $LastIn1(w) = (Last(w)(carry_{in}), Last(w)(v1_1), \ldots Last(w)(v1_n))$ and $LastIn2(w) = (Last(w)(v2_1), \ldots Last(w)(v2_n))$ Given a binary n-vector $\boldsymbol{b} = (b_1, \ldots b_n)$, define projection $\boldsymbol{b}[i] = b_i$ and then $unsigned(\boldsymbol{b}) = \Sigma_{i=1}^n \boldsymbol{b}[i] * 2^{i-1}$ assuming both that the lower indexed bits are the lower order bits and that the length of the vector is known. What we want of an n-bit adder V is that when inputs have been stable for long enough $Stable(w) > t_{adder}$.

$$(*) \quad unsigned(LastIn1(w) + unsigned(LastIn2(w)) = \Sigma_{i=1}^n V(w, r_i) 2^{i-1} + 2^n V(w, carry_{out})$$

We can construct a ripple carry adder as follows.

$$\begin{aligned} &V \text{ is a constructed ripple carry adder with } n \text{ bits if and only if} \\ &V(w) = (D_1(u_1, sum), \ldots D_n(u_n, sum), D_n(u_n, carry_{out})) \quad (21) \\ &\text{where each } D_i \text{ is a delay } t \text{ single bit adder} \quad (22) \\ &\text{and if } w = \Lambda \text{ then } w \mapsto (\Lambda, \ldots \Lambda) \quad (23) \end{aligned}$$

$$\text{and if } w \mapsto (u_1 \ldots, u_n) \text{ then}$$
$$(wa) \mapsto (u_1 c_1 \ldots, u_n c_n) \text{ where}$$
$$c_1(carry_{in}) = a(carry_{in}), c_{i+1}(carry_{in}) = D_i(u_i, carry_{out}) \tag{24}$$
$$c_i(v1) = a(v1_i), c_i(v2) = a(v2_i) \tag{25}$$

If t is the delay for the component adders, then we want to require that $\mathit{Stable}(n * t + n)$ should imply $(*)$. I'm just going to present proof sketches since the idea should be clear and the main claim here is that these proofs should be reasonably easy to automate. I want to particularly show how component properties can be pushed up the composition ladder because if $P(f(w))$ tells us that $f(w)$ has a property P and we can show it for all w, then $P(f(u_i))$ must be true. However, there are additional constraints on u_i because of the definition of the map from w to u_i.

Proofs are usually by induction: one strings and on the component index. First, there is a lemma for how long the output of an adder is stable when the input is held stable. Recall the definition of $\#f$ above in equation 6 and consider $\#D(w, sum)$.

Ignoring the composite system, say that t delay adder D must have the property that:

$$\mathit{Stable}(w) \geq t + k \rightarrow \#D(w, sum) \geq k \text{ and } \#D(w, carry_{out}) \geq k \tag{26}$$

Note this w ranges over input sequences for single-bit adders - we'll turn to w for the n-bit adder below. Proof: Let $w = \Lambda$ and there is nothing to prove since $\#D(\Lambda) = 0$ and $\mathit{Stable}(\Lambda) = 0$. Suppose the lemma true for w and consider wa. Suppose $\#D(w) = j$ and consider $\#D(wa)$ where we extend the $\#f$ notation to allow for pin arguments. If $k = 0$ then all we have to prove is that $\#D(w) \geq 0$ which is trivially true. So consider $k > 0$ and $\mathit{Stable}(wa) \geq t + k' + 1$ for some $k' \geq 0$. Note that $\mathit{Stable}(w) = j$ implies that $H(w, p, b_p) \geq j$ for each input pin p for some value of b_p. So by the definition of an adder the outputs of D are determined when $\mathit{Stable}(w) \geq t + k$. From the definition of H we have $H(wa, p, b_p) \leq H(w, p, b_p)$ only when $H(wa, p, b_p) = 0$ which we know is false (or else $\mathit{Stable}(wa) = 0$. So $H(w, p, b_p) < H(wa, p, b_p)$ which means $H(wa, p, b_p) = 1 + H(w, p, b_p)$ so $H(w, p, b_p) = t + k'$ and $k' \geq 0$ so $H(w, p) \geq t$ which means that the outputs at state w were determined by the inputs which have not changed by state wa. QED.

Now switch back to the composite structure. By construction, $H(w, v1_i, b1_i) = H(u_i, v1, b1_i)$ and $H(w, v2_i, b2_i) = H(u_i, v2, b2_i)$ and $H(w, carry_{in}, b_0) = H(u_1, carry_{in}, b_0)$. So by the lemma at 26 and the initial assumption $\#D_1(u_1) \geq (n-1)t + n$ since the inputs have been stable for $n * t + n$. Now we note that for $i > 1$, we must have $\#D_{i-1}(u_{i-1}) + 1 \leq H(u_i, carry_{in}, b_i)$. The proof is by induction using the definition of the mapping from w to the u_i. This tells us that the inputs for $\#D_1(u_1) \geq t + k + 1 \rightarrow \#D_i(u_i) \geq k$. The 1 gets lost because of the 1 unit delay between output pins and input pins - see the note above. At this point it's a matter of putting the parts together to complete the proof.

5 Discussion

Earlier papers by the same author [14, 15] proposed the use of the general product for modelling composition and concurrency, but suffered from too much use of the "formal methods" approach. The proofs carried out here were not really feasible using formal logic. There is a good reason why mathematicians relegate formal logic to "meta mathematics" and why "working mathematics", the language of applied mathematics, science, and engineering, is not formalized in practice. Similarly, augmented state machines don't have the underlying mathematical coherence of ordinary state machines via Myhill/Nerode congruences and the well known connection to semi-group theory. Instead, for example, input/output automata [8, 9] take us into the realm of "process algebras" and similar constructs.

It well known that every state machine induces a monoid, and there is a decomposition theory for state machines that parallels and extends group decomposition via the Jordon-Holder theorem. The researchers who first looked at algebraic automata theory such as Hartmanis[6] were focused on so-called "cascade" products in which there is no feedback[2, 7, 5]. For example, the composed nand-gate above has no feedback between the u_i — information moves in one direction only. The constructed SR latch, on the other hand, requires that the output of $N_1(u_i)$ be fed back into u_2. The standard digital architecture distinction between combinational and sequential circuits appears related to this distinction.

Recall S_f the state set induced by $f : A^* \to X$. Let $Mon(f)$ be the monoid consisting of the set of maps $\sigma_w : S_f \to S_f$ for $w \in A^*$ so that $\sigma_w f_u = f_{u \circ w}$ and the operation $\sigma_u \sigma_z = \sigma_{u \circ z}$. The set of maps will be a superset of S_f because $f_w = f_z$ does not imply that $\sigma_w = \sigma_z$ since $\sigma_w f_u = f_{u \circ w}$ which may be distinct from $\sigma_z f_u - f_{u \circ z}$. But if S_f is finite then $Mon(f)$ must be finite and certainly σ_Λ is the identity and

$$\sigma_w(\sigma_u \sigma_v) = \sigma_{w \circ u \circ v} = (\sigma_w \sigma_u)\sigma_v.$$

The product used here was ignored by the pioneers of algebraic automata theory for a variety of reasons, but perhaps one of them was that the practitioner interest was in state minimization which leads naturally to questions of decomposition. The full generality of feedback products also loses the analogy to the Jordon-Holder series that is key to the Krohn-Rhodes decomposition. But the product used here is quite constrained. Each $c_i(p) = a(p')$ or $c_i(p) = G_j(u_j, p')$ where G_i is a component factor because the communication mimics the connection of wires so that no computation can be done in the connection map. There are additional constraints imposed by fan-out and fan-in limits and we may want more constraints such as the $E(wa) = E(wa')$ constraint mentioned above that prevents instant response to inputs. We can conjecture that this constrained product also has structural implications for the construction of monoids. With the caution that what follows is extremely early conjecture let's look at some possibilities relating combinations versus sequence circuits to group structure.

What we call combinatorial circuits in digital circuit engineering seems to produce state systems with only trivial loops - where$f_w = f_{wa}$. This limits the complexity of subgroups within the underlying monoid. So products with non-trivial feedback may be the only ways to introduce longer loops into the state structure.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. Michael A. Arbib. *Theories of Abstract Automata.* Prentice-Hall, 1969.
3. Pal Domosi and Chrystopher L. Nehaniv. *Algebraic Theory of Automata Networks (SIAM Monographs on Discrete Mathematics and Applications, 11).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.
4. Ferenc Gecseg. *Products of Automata.* Monographs in Theoretical Computer Science. Springer Verlag, 1986.
5. A. Ginzburg. *Algebraic theory of automata.* Academic Press, 1968.
6. J. Hartmanis. Loop-free structure of sequential machines. In E.F. Moore, editor, *Sequential Machines: Selected Papers*, pages 115–156. Addison-Welsey, Reading MA, 1964.
7. W.M.L. Holcombe. *Algebraic Automata Theory.* Cambridge University Press, 1983.
8. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.
9. Nancy Lynch. *Input/Output Automata: Basic, Timed, Hybrid, Probabilistic, Dynamic,...*, volume 2761/2003 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2003.
10. J. Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson Air Force Base, 1957.
11. A. Nerode. Linear automaton transformations. In *Proc. AMS*, 1958.
12. Ramzi Ben Salah, Marius Bozga, and Oded Maler. On timed components and their abstraction. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 63–71, New York, NY, USA, 2007. ACM.
13. John F. Wakerly. *Digital Design Principles and Practices: 4th edition.* Prentice Hall, 2005.
14. Victor Yodaiken. The algebraic feedback product of automata. In *DIMACS Workshop on Computer Aided Verification*, 1989.
15. Victor Yodaiken. Modal functions for concise representation of finite automata. *Information Processing Letters*, Nov 20 1991.
16. Victor Yodaiken. Primitive recursive presentations of automata and their products. *CoRR*, abs/0907.4169, 2009.