ABSTRACT

According to much of the formal methods literature, "classical" automata theory is too limited to model complex discrete state systems, particularly, concurrent ones constructed from communicating components. This paper takes a look at the critique of automata theory in the early process algebra literature[23, 4], to show how methods developed in automata theory in the 1950s and 1960s overcome the claimed limitations. These include methods of defining behavioral and structural equivalence and methods for characterizing composition, concurrency, and interaction. Since much of the classic automata theory literature is not well known anymore, the paper also provides brief historical literature survey.

***K*eywords** computer history, automata, state, Milner, process algebra, recursion, Moore machines, concurrency

FORMAL METHODS, STANDARD AUTOMATA THEORY AND PROCESS ALGEBRA

A PREPRINT

Victor Yodaiken

March 17, 2026

Somewhere in the 1970s and 1980s, the formal methods research community reached a consensus that "classical" automata theory was inadequate for formalizing properties of complex software and systems. Robin Milner's "Communication and Concurrency"[23] provides one good example of what went into that consensus and this paper revisits Milner's critique of automata theory and takes a second look at the automata theory literature to evaluate that critique[1]. Section 1 examines Milner's critique of automata equivalences and section 2 shows how to construct automata models of concurrent systems and interaction. Section 3 provides some background on the development of the theory and a taxonomy of types of automata.

1 Automata theory and process algebra

1.1 Bisimulation and covering

In chapter 4 of "Communications and Concurrency" Robin Milner wrote:

> *We begin ... by showing the need for a notion of equality stronger than that found standard automata theory* –[23],p 84

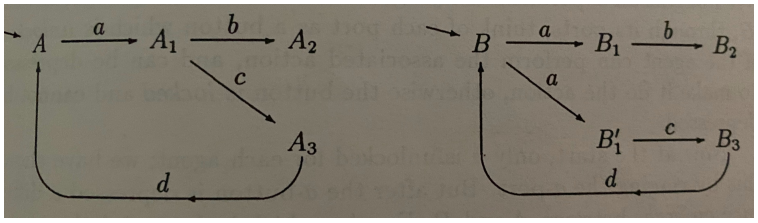
and then on the next page: "*in standard automata theory an automaton is interpreted as a language.*"(p. 85).

Baeten[4] in his 2005 "History of Process Algebra" puts it this way:

> *On automata, the basic notion of equivalence is language equivalence: a behaviour is characterized by the set of executions from the initial state to a final state.*

To illustrate, Milner provides two simple process algebra agents which "may be thought of as finite state automata"(p.85), then provides two state diagrams,

[1] This paper is about automata theory and has very little to say about process algebra itself.



and explains:

> *if we take A_2 and B_2 to be the accepting states of our two automata, we can argue ... that A and B denote the same language. ...*
> *But we now argue in favour of another interpretation in which A and B are different.*

Milner used the same examples in a previous book [22] where he asks "*But are they equivalent in all senses?*"

In automata theory A and B are definitely not equivalent in all senses even if both accept the same language. One is a nondeterministic finite automaton (NFA) and the other is deterministic (DFA), after all. They are also not equivalent in terms of algebraic automata theory where automata can be distinguished by behavior, structure, and even underlying semigroups[10, 2, 13, 12]. In a 1968 monograph, Ginzburg[10] defines an automata relation called "covering" and a covering equivalence.

> *The meaning of [B covers A] is that to every state s^A in S^A there corresponds at least one state $s^B \in S^B$, such that when started in s^B, B performs all the translations done by A. [..]*
> *If for some A and B, B covers A and A covers B, these automata are said to be equivalent. - [10], p. 97.*

Milner's example B does not cover A (there is no B state to map A_1 to) but A does cover B so the state machines are *not* equivalent in this sense. There is no mention of any of the equivalences of algebraic automata theory in "Communication and Concurrency" or "Calculus of Communicating Systems" but Milner was certainly aware of them.

In a 1971 paper that preceded his work on process algebra, [24] Milner cited the same definition from Ginzburg's book:

> *Condition (ii) simply states that R is a weak homomorphism between the algebraic structures (D,F),(D',F'). This concept is used in automata theory to define the notion of covering - see for example Ginzburg ([10], p. 98.)*

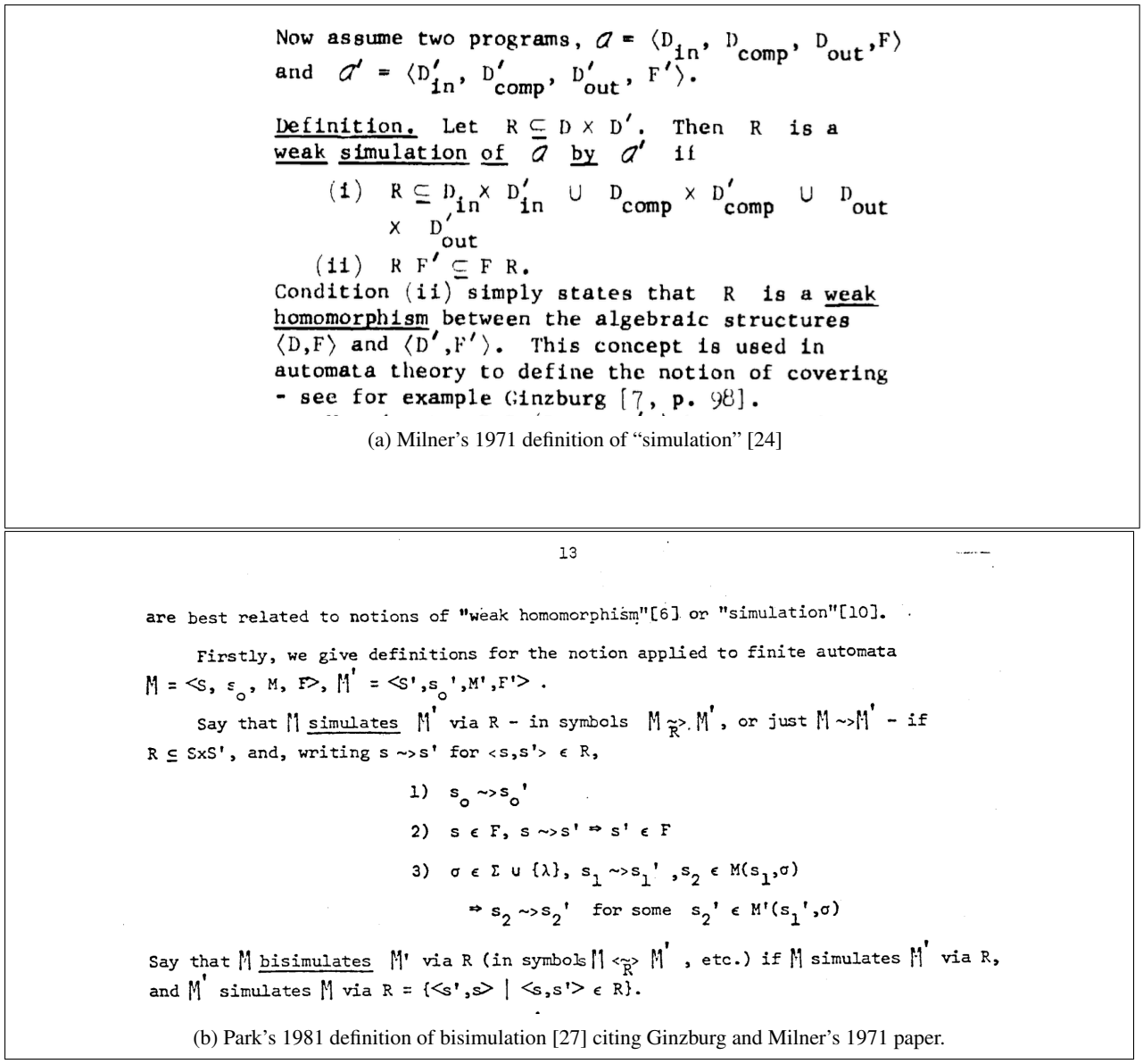
See figure 1a for details. A weak homomorphism is a minor variation on covering ([10], p 99) with a relation in place of the map between states (the correspondence). Park[27] who is generally credited with introducing bisimulation to process algebra substitutes the terms "simulates" and "bisimulates" for Ginzburg's "covers" and "is equivalent" and cites both Ginzburg and Milner's 1971 paper.

> *The sort of rule to be discussed can be seen to develop from the known decision procedures for problems concerning these automata- But in the form given here, they are best related to notions of weak homomorphism[10] or "simulation" [24].* [...] *Say that M "bisimulates" M' via R [...] if M simulates M' via R, and M' simulates M via R–* Park

See figure 1b for details.

Relations of the kind Ginzburg called "cover" are sometimes called "simulations" in the 1960s automata theory literature (e.g. [1] p. 39). The reader familiar with process algebra will see the similarity between covering equivalence and process algebra bisimulation. Sangiorgi[31] (p. 132) points out that not only is bisimulation similar to automata homomorphisms and covering but it is related to an even older automata theoretic concept.

> *[...] in the 1960s weak homomorphism is well-known in automata theory and ... this notion is not that far from simulation. Another emblematic example, again from automata theory, is given by the algorithm for minimisation of deterministic automata, already known in the 1950s [Huffman [15]; Moore 1956 [25]] (also related to this is the Myhill-Nerode theorem [Nerode 1958]).*
> [...] *The algorithm strongly reminds us of the Paige-Tarjans partition refinement algorithm [Paige and Tarjan 1987], the best known algorithm for computing bisimilarity and for minimisation modulo bisimilarity.*



Now assume two programs, $\mathcal{A} = \langle D_{in}, D_{comp}, D_{out}, F \rangle$ and $\mathcal{A}' = \langle D'_{in}, D'_{comp}, D'_{out}, F' \rangle$.

Definition. Let $R \subseteq D \times D'$. Then R is a weak simulation of $\mathcal{A}$ by $\mathcal{A}'$ if

(i) $R \subseteq D_{in} \times D'_{in} \cup D_{comp} \times D'_{comp} \cup D_{out} \times D'_{out}$

(ii) $R F' \subseteq F R$.

Condition (ii) simply states that R is a weak homomorphism between the algebraic structures $\langle D, F \rangle$ and $\langle D', F' \rangle$. This concept is used in automata theory to define the notion of covering - see for example Ginzburg [7, p. 98].

(a) Milner's 1971 definition of "simulation" [24]

13

are best related to notions of "weak homomorphism"[6] or "simulation"[10].

Firstly, we give definitions for the notion applied to finite automata $M = \langle S, s_o, M, F \rangle$, $M' = \langle S', s_o', M', F' \rangle$.

Say that M simulates M' via R - in symbols $M \underset{R}{\sim>} M'$, or just $M \sim> M'$ - if $R \subseteq S \times S'$, and, writing $s \sim> s'$ for $\langle s, s' \rangle \in R$,

1) $s_o \sim> s_o'$

2) $s \in F$, $s \sim> s' \Rightarrow s' \in F$

3) $\sigma \in \Sigma \cup \{\lambda\}$, $s_1 \sim> s_1'$, $s_2 \in M(s_1, \sigma)$

$\Rightarrow s_2 \sim> s_2'$ for some $s_2' \in M'(s_1', \sigma)$

Say that M bisimulates M' via R (in symbols $M \underset{R}{<\sim>} M'$, etc.) if M simulates M' via R, and M' simulates M via $R = \{\langle s', s \rangle \mid \langle s, s' \rangle \in R\}$.

(b) Park's 1981 definition of bisimulation [27] citing Ginzburg and Milner's 1971 paper.

Figure 1: Origins of Bisimulation

For readers who are interested, Sangiorgi provides an extensive discussion of what he sees as the differences between automata homomorphisms and bisimulation (p. 125-129), but here our concern is automata theory and understanding what Milner is getting at with this argument.

To illustrate, Milner describes an "experiment".

> *According to our earlier treatment of examples, A and B are agents which may interact with their environment through the ports a, b, c, and d. We imagine an experimenter trying to interact with the agent A or with B through its ports; think of each port as a button which is* unlocked *if the agent can perform the associated action and can be depressed to make it do the action, otherwise the button is* locked *and cannot be depressed.*

> [...] *after the a-button is depressed [in the initial state] a difference emerges between A and B. For A – which is deterministic – b and c will be unlocked, while for B – which is non-deterministic – sometimes only c will be unlocked.* – p. 86.

The experiment works for the process algebra agents because their defining equations are transparent in terms of what events can happen in any state. The experimenter cannot distinguish A and B *as state machines* because they do not have buttons or ports as output. State B' does not output any information which would tell the experimenter that the c event is the only enabled event. In fact, the outputs of A and B are limited because they are both a particular type of state machine called a recognizer or an acceptor.

Rabin and Scott's 1959 paper [19] explains the distinction between acceptors/recognizers and automata with more general output.

> *"A neat form of the definition of automata has been used by Burks and Wang' and by E. F. Moore, and our point of view is closer to theirs than it is to the formalism of nerve-nets. However, we have adopted an even simpler form of the definition by doing away with a complicated output function and having our machines simply give "yes" or "no" answers."*– (page 64)
>
> [...]
>
> *An automaton will be considered as a black box of which questions can be asked and from which a "'yes" or "no" answer is obtained.* (page 66)

The same argument, with the same examples, in Milner's earlier book has the more illuminating title *Traditional equivalence of finite state* ***acceptors*** ([22] p.9) (with my bold added) and the term "standard automata theory" only appears in the later book. Apparently, "standard automata theory", for Milner, only includes these black box state machines and not machines with more general output, like Mealy and Moore machines[14], which one might normally consider to be part of standard automata theory (see section 3.2 for a taxonomy of automata).

Milner has also required that the structure of the automata not be considered.

> *we only wish to distinguish between two agents P and Q if the distinction can be detected by an external agent interacting with each of them.* – ([23],p 84)

In his first book, he writes:

> *"two systems are indistinguishable if we cannot tell them apart without pulling them apart."* ([22], p. 2).

That requirement rules out the use of cover equivalence, machine homomorphisms, minimization, DFA/NFA, differences and other parts of automata theory one might ordinarily consider standard. We are only permitted to examine the output of automata to distinguish them.

Given this combination of requirements, Milner's opening argument could be more precisely expressed by:

> *We begin ... by showing the need for a notion of equality stronger than that can be supported by only examining the outputs of recognizers in automata theory.*

But that brings up the question of why the agents were associated with recognizers and not automata with more general output. Rabin and Scott limit the automata output, because they are using automata to decide language membership, not to model devices or software. But modeling digital systems and software is what formal methods is about.

Moore's 1954 paper[25] that defined what are now called Moore machines is concerned with "thought experiments" that are strikingly similar to Milner's experiment.

> *The experimenter will choose which finite sequence of input symbols to put into the machine, either a fixed sequence, or one in which each symbol depends on the previous output symbols. This sequence of input symbols, together with the sequence of output symbols, will be called the outcome of the experiment.* [...] *the experimenter may be thought of as a human being who is trying to learn the answer to some question about the nature of the machine or its initial state. This is not the only kind of experimenter we might imagine in application of this theory; in particular the experimenter might be another machine.* – E.F. Moore, Gedanken-Experiments on Sequential Machines [25]

Converting Milner's A and B automata to Moore machines by adding outputs to indicate enabled transitions from the current state allows the experimenter to distinguish between the machines purely on outputs. In a Moore machine

version of A, state A_1 outputs $\{b, c\}$ and A_3 outputs $\{d\}$ *etc.*. A Mealy machine [21] where the output is a function of the current state and the input that drove the system to its current state could be defined so that every input is enabled on every step, and buttons that are "locked" cause a transition back to the current state and output a message "Locked".

Moore [25] and Mealy [21] machines were introduced in the 1950s, are still widely used in digital circuits and software, and were central to the algebraic automata theory literature. State machines with output are extensively discussed in Ginzburg's text - the definitions of coverings and automata homomorphism cited by Milner and Park are for Mealy machines. Recognizers became very important in computer science for definition of formal languages, parsing, and searching software (e.g. [33]) and in the 1970s computer science separated more from electrical engineering where Moore and Mealy machines were used in digital circuit design. By 1979 Hopcroft and Ullman's well-known automata theory text for computer scientists[14] devoted just three out of more than four hundred pages to "State machines with output", but they did mention and define them. It might not have been out of the question for a computer science researcher in the 1970s and 1980s to wrongly consider "standard" automata theory to be limited to recognizers. Edsger Dijkstra appears to have done just that in his widely cited one line comment:

> *traditional automata theory tends to make us insensitive to the role interfaces could and should play in coping with complex designs.* [8]

But it's peculiar that such a prominent part of computer science should have become invisible in part of the formal methods literature. To specify devices, interfaces, interaction, and even concurrency with automata, Moore and Mealy machines are more appropriate than recognizers. That is the topic of the next section.

2 Interaction, concurrency, and composition

> *But meanwhile I got somehow interested, and I don't know how, in concurrency. I remember that, without linking it to verification particularly, I wondered about interacting automata. I had an idea that automata theory was inadequate, because it hadn't said what it was for two automata to interact with each other. Except for the Krohn-Rhodes Representation Theorem, which said something about feeding the output of one automata into another. But there wasn't interaction between the automata.*
> – Robin Milner, interview [5]

Baeten[4] amplifies.

> *Basically, what is missing [in automata theory] is the notion of interaction: during the execution from initial state to final state, a system may interact with another system. This is needed in order to describe parallel or distributed systems, or so-called reactive systems.*

2.1 Moore machines

The usual definition of a Moore machine tuple is $M = (A, Y, S, s_0, \delta, \lambda)$ where A is a set of events, Y is a set of outputs, S is the state set, $\delta : S \times A \to S$ is the transition map and $\lambda : S \to Y$ is the output map. Moore machine state sets are usually finite, in which case the automaton is called a finite state Moore machine.

A Moore machine accepts data in the form of inputs and responds with output. If the Moore machine is finite and in minimal form, the ratio $|S|/|Y|$ is rough indication of modularity – how much information from the state set is visible. The distinction between internal state and visible state (output) provides a rigorous definition of *interface* which is specified by the output map and the set of input events together.

Moore machine tuples can be viewed as specifying a map from finite sequences of inputs to outputs or a map from finite sequences of inputs to finite sequences of outputs (a transducer). See section 3.3 for more detail.

2.2 Products

In 1962, Juris Hartmanis published an influential paper called "Loop free structure of sequential machines"[11] (this paper is referenced by Ginzburg!). The "loop-free" products are the kind associated with the Krohn-Rhodes theorem and, as Milner noted, model communication in a cascade, with "no loops" and, because of that, only one way communication between components. Loop-free, also called "cascade", factoring of automata was a central concern of algebraic automata theory in the 1960s because the successful work by Hartmanis and his colleagues and because of the mathematical significance of the relationship between loop-free products and semigroup theory [18] or see [13]. This relationship is briefly discussed in section 3.4. Hartmanis' paper, however begins with the definition of more general "concurrent automata product" that does permit arbitrary communication and interaction.

DEFINITION 1. Let $M_1, M_2, ., M_n$, be a set of Moore type machines in which the outputs of any machine $M_i, i = 1, 2, ..., n$, may be used as inputs to other machines. We shall say that this set of interconnected machines is concurrently operating if the next state (state at time t + 1) of each machine M_i depends on the present state of M_i, the present outputs (which depend only on the present states) of the machines to which it is connected, and the present external input. The ordered n-tuple (or "configuration") of the present states of the machines $M_1, M_2, ., M_n$ will be referred to as the state of the interconnected machine. – Hartmanis[11], p 117.

At each step, each component M_i gets input that is a function of the external input (the input to the system) and the outputs of some or all of the components. Hartmanis and Stearns defined a slightly different version of this product they called a *network* in their book[12] (figure 2). Gecseg[9] has another version and cites other work dating back to the 1970s and earlier. These products seem to have been considered obvious in automata theory. See for example a passing mention in Assmus and Florentin [3] (p. 30).

Both Hartmanis and Gecseg show how by limiting feedback, the interconnection can be varied. The connector map which produces input for the factors from the system input and the feedback from the outputs of the factors determines the connection graph. In the cascade product the factor automata are ordered so that inputs generated for M_i depend only on the outputs of $M_1, \ldots M_i$ and ignore the outputs of the remaining factors. For interacting programs or devices there will be cycles.

DEFINITION 3.9. An *abstract network* $\mathcal{N}$ of machines consists of:
1. $\{M_i = (S_i, I_i, \delta_i)\}$, $1 \leqslant i \leqslant n$, a set of state machines referred to as *component machines*;
2. I—a nonvoid finite set of external inputs;
3. O—a nonvoid finite set of external outputs;
4. $f_i: (\times S_j) \times I \longrightarrow I_i$, $1 \leqslant i, j \leqslant n$, machine connecting rules;
5. $g: (\times S_i) \times I \longrightarrow O$, the output function.

Figure 2: Hartmanis and Stearns, 1966 [12]

The feeback or concurrent product, especially in Hartmanis' version, provides an interesting model of composition and concurrency.

- Moore machine tuples are closed under this product.
- If all the factors are finite state the product automaton is finite state.
- The product models composite systems with true concurrency, not interleaving.
- The product does not incorporate any particular model of communication, but shared variables, messages, shared memory, wires, or other mechanisms can be modeled by varying the interconnection.
- There is no need for a renaming operation to accomplish composition and events with the same names in multiple event alphabets don't have any special properties.
- Another property of this particular product is that it respects modularity[28], each component can see only the outputs of connected components, not their states.

The product is a model inspired by computing devices, rather than being inspired by concurrent programming on uniprocessors.

To make the product precise, I'm going to use a version[34] which specifically permits components to change state at different rates. Suppose for each $0 < i \leq n$ $M_i = (A_i, Y_i, S_i, s_{0,i}, \delta_i, \lambda_i)$. To connect these machines what is needed is a new input set A and a *connector map*:

$$\kappa : Y \times A \to Y_1^* \cdots \times Y_n^*$$

where $Y = Y_1 \times \ldots Y_n$ is over the tuples of outputs from the components. Write $\kappa_i(y, a)$ for the i^{th} element of the tuple $\kappa(y, a)$. The product machine

$$M = (A, Y, S, s_0, \delta, \lambda)$$

has $S = S_1 \cdots \times S_n$ and for $s = (s_1, \ldots s_n)$, $\lambda(s) = (\lambda_1(s_1), \ldots \lambda_n(s_n))$ so $\lambda(s) \in Y$. The initial state s_0 is the tuple of initial states $(s_{0,1}, \ldots s_{0,n})$ The transition map is given by

$$\delta(s, x) = (\delta_1^*(s_1, \kappa_1(\lambda(s), x)) \ldots \delta_n^*(s_n, \kappa_n(\lambda(s), x))$$

where

$$\delta_i^*(s_i, \epsilon) = s_i \text{ and } \delta_i^*(s_i, z \cdot a) = \delta_i(\delta_i^*(s_i, z), a)$$

and $w \cdot a$ is the sequence obtained by appending event a to finite sequence w on the right.

3 Background on state machines

3.1 Scope of automata theory

Burks and Wang presented an ambitious scope for automata theory.

> *To begin with we will consider any object or system (e.g., a physical body, a machine, an animal, or a solar system) that changes its state in time; it may or may not change its size in time, and it may or may not interact with its environment. When we describe the state of the object at any arbitrary time, we have in general to take account of: the time under consideration, the past history of the object, the laws governing the inner action of the object or system, the state of the environment (which itself is a system of objects), and the laws governing the interaction of the object and its environment. If we choose to, we may refer to all such objects and systems as automata. The main concern of this paper is with a special class of these automata: viz., digital computers and nerve nets.* – Burks and Wang, 1957 [7]

Burks was one of the primary engineers who built ENIAC[6] and Wang wrote the first automated theorem prover, invented Wang Tiles, and was the Ph.D. advisor to Stephen Cook, among many other accomplishments.

Von Neumann [26] wrote:

> *It has been pointed out by A.M.Turing in 1937 and by W.S. McCulloch ad W. Pitts in 1943 that effectively constructive logics, that is, intuitionistic logics, can be best studied in terms of automata.* (p. 44).

Rabin and Scott also had strong claims.

> *Turing machines are widely considered to be the abstract prototype of digital computers; workers in the field, however, have felt more and more that the notion of a Turing machine is too general to serve as an accurate model of actual computers. It is well known that even for simple calculations it is impossible to give an a priori upper bound on the amount of tape a Turing machine will need for any given computation. It is precisely this feature that renders Turing's concept unrealistic. In the last few years the idea of a finite automaton has appeared in the literature. These are machines having only a finite number of internal states that can be used for memory and computation. The restriction of finiteness appears to give a better approximation to the idea of a physical machine.*

Both Rabin and Scott and Von Neumann [26] trace automata theory back to nerve nets[17] but a case could be made that the origin was mostly in switching theory.

Much of the work in automata theory was in the area of digital circuit design originally for design of telephone switches, and it was common to see articles on relays in the same conferences and journals that presented articles on topics like formal languages, logic, and semigroups. Huffman's paper [15] seems to have been the first to develop the notion of a sequential circuit (a state machine with output).

> *We may generalize from these two simple examples: In a circuit having no secondary relays there can be no "memory"; the states of operation of the primary relays uniquely determine the output transmissions. Such a circuit is called a combinational circuit. In a circuit having secondary relays, the possibility of a "memory" exists since the states of operation may not uniquely determine the output transmissions. A circuit having secondary relays will be called a sequential circuit.*–Huffman.

See also [16] for an account of the origins of switching theory.

An amazing volume edited by Claude Shannon and John McCarthy[32] gives some sense of the field in the mid 1950s.

3.2 Types of automata

Mostly following Ginzburg's taxonomy the types of state machines are as follows.

- A state machine or subautomaton consists of a tuple (A, S, s_0) where A is the event (input) alphabet, S is a set of states with $s_0 \in S$ the "start state",$\delta : S \times A \to S$ is the transition function.
- A recognizer[19] adds a set $F \subset S$, the set of accept states. This machine accepts or recognizes a finite sequence of inputs if following it via δ from the start state through the end of the sequence terminates at a state in F.
- A Moore machine tuple[25] (A, S, s_0, X, λ) replaces the set of accept states with a set X of outputs and a map $\lambda : S \to X$. Moore machines can act as acceptors when the output map is suitable.
- A Mealy machine[21] (A, S, s_0, X, γ) replaces the Moore machine output map λ with some $\gamma : S \times A \to X$.
- *Sequential machine* is a generic term, usually used more in circuit design which usually has some output see Huffman [15] and Burks and Wang[7]

Turing machines are also often considered automata as are linear bounded automata [14] and Buchi automata among others. Those are not discussed here.

3.3 Maps

Arbib [2] (1968) writes

> *"we may say automata theory is the study of partial functions* $F : A^* \to Y^*$*"* – (p. 6).

Similarly, Pin[30] (p. 100) defines a sequential function $\sigma : A^* \to Y^*$ as "a function whose behavior is defined by a machine called a 'sequential transducer' ". The set names on both of these are changed for consistency here.

For any set X, let X^* be the set of finite sequences over X including the empty sequence ϵ, closed under right append so $w \cdot a \in X^*$ if $w \in X^*$ and $a \in X$.

Each Moore machine tuple M determines maps $M^* : A^* \to X$ and $M^{**} : A^* \to X^*$ by primitive recursion on finite sequences[34, 29] with $M^*(w) = \lambda(\delta^*(w))$ where

$$\delta^*(\epsilon) = s_0 \text{ and } \delta^*(w \cdot a) = \delta(\delta^*(w), a)$$

and

$$M^{**}(\epsilon) = \epsilon \text{ and } M^{**}(w \cdot a) = M^{**}(w) \cdot M^*(w)$$

Variations of these maps can be found all over the automata theory literature (e.g. [3, 10, 13, 30]). In [3] (p. 17) they are called input-output functions[2].

In a prior paper[34], I show how to define Moore machines directly from primitive recursive maps on finite sequences, without needing the Moore machine tuple representation and how to construct automata products using composition of these functions. This approach has advantages for scaling Moore machines to large state sets and complex systems.

3.4 Monoids

The relationship between automata and automata products and factoring semigroups and monoids is extensively discussed in algebraic automata theory. See [13] for an introduction.

There are several similar ways to construct a monoid from a state machine. One way is to use the input/output maps of the previous section and define an equivalence.

$$\text{For } w, u \in A^*, w \sim u \bmod M \text{ iff } (\forall z_1, z_2 \in A^*) M^*(z_1 \texttt{ concat } w \texttt{ concat } z_2) = M^*(z_1 \texttt{ concat } u \texttt{ concat } z_2)$$

Then the elements of the monoid are the equivalence classes $[w]_M = \{u \in A^* : u \sim w \bmod M\}$. The monoid operation is $[w]_M \circ [u]_M = [w \texttt{ concat } u]_M$ and the identity element is $[\epsilon]_M$.

If a monoid has a cancelation property, it is a group. The Krohn-Rhodes theorem [18] surprisingly ties factoring of finite automata using loop-free (cascade) products to the Jordan-Holder theorem of group theory[20] (p. 430-432). As summarized by Zeiger[35], one consequence of Krohn-Rhodes is:

[2]It doesn't seem as if automata theory researchers recognized these maps as primitive recursive or, if they did, they apparently didn't consider it important enough to mention.

> *Each finite state automaton can be built as a cascade of two state automata and simple-group automata* (p. 77)

The engineering motivation for automata factoring research was simplification what now seem like tiny circuits, but were challenging in the 1960s. By factoring the state machine, one could construct an equivalent one as a connected sequence of simpler machines.

References

[1] Michael Arbib. "Automaton Decompositions and Semigroup Extensions". In: *Algebraic theory of machines, languages, and semigroups*. Ed. by Michael Arbib. Vol. 34. Annals of Mathematics Studies. 1968, pp. 37–54.

[2] Michael A Arbib. *Theories of Abstract Automata (Prentice-Hall Series in Automatic Computation)*. USA: Prentice-Hall, Inc., 1969. ISBN: 0139133682.

[3] E.F. Assmus and J. J. Florentin. "Algebraic machine theory and logical design". In: *Algebraic theory of machines, languages, and semigroups*. Ed. by Michael Arbib. Vol. 34. Annals of Mathematics Studies. 1968, pp. 15–35.

[4] Jos CM Baeten. "A brief history of process algebra". In: *Theoretical Computer Science* 335.2-3 (2005), pp. 131–146.

[5] Martin Berger. *An interview with Robin Milner*. Sept. 2003. URL: %5Curl%7Bhttps://users.sussex.ac.uk/~mfb21/interviews/milner/%7D.

[6] Arthur W Burks, Herman H Goldstine, and J von Neumann. *Preliminary Discussion of the Logical Design of an Electronic Computing*. Tech. rep. Institute for Advanced Study, June 1946.

[7] Arthur W. Burks and Hao Wang. "The Logic of AutomataPart I". In: *J. ACM* 4.2 (Apr. 1957), pp. 193–218. ISSN: 0004-5411. DOI: 10.1145/320868.320880. URL: https://doi.org/10.1145/320868.320880.

[8] Edsger W. Dijkstra. "Some questions". circulated privately. Nov. 1974. URL: http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD463.PDF.

[9] Ferenc Gécseg. *Products of Automata*. Vol. 7. EATCS Monographs on Theoretical Computer Science. Springer, 1986. ISBN: 978-3-642-64884-7. DOI: 10.1007/978-3-642-61611-2. URL: https://doi.org/10.1007/978-3-642-61611-2.

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

[11] J. Hartmanis. "Loop-free structure of sequential machines". In: *Sequential Machines: Selected Papers*. Ed. by E.F. Moore. Reading MA: Addison-Welsey, 1964, pp. 115–156.

[12] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.

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

[14] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading MA: Addison-Welsey, 1979.

[15] D.A. Huffman. "The synthesis of sequential switching circuits". In: *Journal of the Franklin Institute* 257.3 (1954), pp. 161–190. ISSN: 0016-0032. DOI: https://doi.org/10.1016/0016-0032(54)90574-8. URL: https://www.sciencedirect.com/science/article/pii/0016003254905748.

[16] Toma KAWANISHI. "Prehistory of Switching Theory in Japan:Akira Nakashima and His Relay-circuit Theory". In: *Historia Scientiarum. Second Series: International Journal of the History of Science Society of Japan* 29.1 (2019), pp. 136–162. DOI: 10.34336/historiascientiarum.29.1_136.

[17] S. C. Kleene. "Representation of Events in Nerve Nets and Finite Automata". In: *Automata Studies*. Ed. by C.E Shannon and J McCarthy. Vol. 34. Annals of Mathematics Studies. 1956, pp. 129–153.

[18] K. B. Krohn and J. L. Rhodes. "Algebraic theory of machines". In: *Transactions of the American Mathematical Society* 116 (1965), pp. 450–464.

[19] Rabin M.O. and Dana Scott. "Finite automata and their decision problems". In: *IBM Journal of Research and Development* 2 (Apr. 1959).

[20] S.M. MacLane and G. Birkhoff. *Algebra*. AMS Chelsea Publishing Series. Chelsea Publishing Company, 1999. ISBN: 9780821816462. URL: https://books.google.com/books?id=L6FENd8GHIUC.

[21] G. Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* (Sept. 1955).

[22] R. Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer Verlag, 1979.

[23] R. Milner. *Communication and Concurrency*. USA: Prentice-Hall, Inc., 1989. ISBN: 0131150073.

[24] Robin Milner. "An Algebraic Definition of Simulation between Programs". In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. IJCAI'71. London, England: Morgan Kaufmann Publishers Inc., 1971, pp. 481–489.

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

[26] J. Von Neumann. "Probabalistic Logics". In: *Automata Studies. (AM-34) (Annals of Mathematics Studies)*. Ed. by C. E. Shannon and J. McCarthy. USA: Princeton University Press, 1956. ISBN: 0691079161.

[27] David Michael Ritchie Park. "Concurrency and Automata on Infinite Sequences". In: *Theoretical Computer Science*. 1981.

[28] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: http://doi.acm.org/10.1145/361598.361623.

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

[30] J.E. Pin. *Varieties of Formal Languages*. New York: Plenum Press, 1986.

[31] Davide Sangiorgi. "On the Origins of Bisimulation and Coinduction". In: *ACM Trans. Program. Lang. Syst.* 31.4 (May 2009). ISSN: 0164-0925. DOI: 10.1145/1516507.1516510. URL: https://doi.org/10.1145/1516507.1516510.

[32] C. E. Shannon and J. McCarthy. *Automata Studies. (AM-34) (Annals of Mathematics Studies)*. USA: Princeton University Press, 1956. ISBN: 0691079161.

[33] Ken Thompson. "Programming techniques: Regular expression search algorithm". In: *Communications of the ACM* 11.6 (1968), pp. 419–422.

[34] Victor Yodaiken. "State Machines for Large Scale Computer Software and Systems". In: *Form. Asp. Comput.* 36.2 (June 2024). ISSN: 0934-5043. DOI: 10.1145/3633786. URL: https://doi.org/10.1145/3633786.

[35] H. Paul Zeiger. "Automaton Decompositions and Semigroup Extensions". In: *Algebraic theory of machines, languages, and semigroups*. Ed. by Michael Arbib. Vol. 34. Annals of Mathematics Studies. 1968, pp. 55–80.