Abstract

Process algebra [18, 2] proposes to address limitations in automata theory that, in perspective, are more a product of the historical development of automata theory and formal methods than substantive limitations. A detailed look at the basic argument in “Communication and Concurrency” and a deeper study of the state machine literature that preceded it, gives some insight into the nature of transition systems, state change, and concurrency.

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

Standard Automata Theory and Process Algebra

A Preprint

Victor Yodaiken

January 14, 2026

1 Process algebra’s critique of automata theory

1.1 An apology

The French idiomatic expression “staircase thoughts” refers to the response or counter argument that occurs to someone who has left the party, as they are descending the stairs on the way home. In this case, the argument happened more than 40 years ago when Computer Scientists, particularly in “formal methods”, came to a near consensus that classical automata theory was too limited for the task of mathematical representation of complex software. The argument was so successful that the ambitious scope of early automata theory (see appendix B) can come as a surprise. Milner’s “process algebra” provides a good example of that argument because because process algebra is so similar to state machines and because in *Communication and Concurrency*[18] and other works, Milner explicitly contrasts process algebra with automata to motivate the development of the former. So, here is the tiresome guest, trudging back up the staircase and knocking on the door to renew the argument: "But in defense of automata theory . . . ”.

1.2 Bisimulation and Covering

Chapter 4, of “Communication and Concurrency” introduces the “bisimulation” equivalence which embodies the idea that that two abstract processes or agents should be considered distinct only when *“the distinction can be detected by an external agent interacting with both of them.”*
Milner writes:

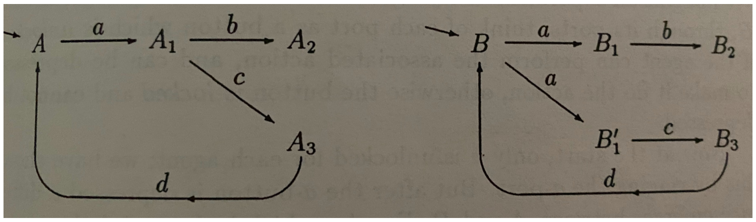
> *We begin ... by showing the need for a notion of equality stronger than that found standard automata theory* [...] *in standard automata theory an* automaton *is interpreted as a language.*([18] p. 85).

Baeten[2] 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.

A great deal of the automata theory literature, however, concerns *differences* between state machines that accept the same language. Rabin and Scott’s 1959 paper [14] describes state machine minimization (via a method of Nerode) to produce *distinct* state machines that recognize the same language. Rabin and Scott also provide an algorithm to convert non-deterministic finite automata (NDA) to distinct deterministic ones (DFA) that *accept the same language*. These machines are equivalent in one sense and not in another but are certainly not considered the same.

To illustrate his point about equivalence, Milner provides two state diagrams



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 example in a previous book [17] where he asks "*But are they equivalent in all senses?*"

In "standard automata theory" A and B are definitely not equivalent in all senses. One is an NFA and the other is a DFA, after all. The machines are also distinct in terms of algebraic automata theory [1, 10, 9]. A 1968 monograph by Ginzburg [6] reviewed the field and explained an equivalence relation called *covering*.

> 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. (p 97) - [6].

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 either.

In a 1971 paper that precedes his work on process algebra [19], Milner defines a "weak simulation" between programs in terms of two conditions and writes:

> 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 (7, p. 98.)

so he had seen this definition. Park[21], who is generally credited with bisimulation also references Ginzburg's definition of covering. Sangiorgi[24] has a detailed discussion of the automata theoretic origins of bisimulation - the following quote is most pertinent.

> [...] 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 1954; Moore 1956] (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.

Milner's familiarity with the equivalences used in algebraic automata theory seems incompatible with his claims about equality in automata theory, but development of the example in *Communication and Concurrency* helps resolve the mystery.

2 On the limitations of recognizers

The exposition in *Communication and Concurrency* continues:

> 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. – "Communication and Concurrency".

Baeten[2] 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.

The missing "notion of interaction" that Baeten mentions is due to both A and B being a type of state machine called a *recognizer*. Rabin and Scott came up with this form of state machine specifically to address decision problems. Here is their explanation from the previously cited paper.

> "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)

Milner's experimenter cannot detect the difference between A and B because A and B are designed to be black-boxes that hide the difference and just reveal "yes" or "no" about whether a sequence belongs to a language.

The combination of treating recognizers as the full extent of "standard automata theory" and then requiring that *"the distinction can be detected by an external agent interacting with both of them"*, which rules out using covering equivalence or other methods that open up the black box, explains the weak notion of equivalence that Milner identified as a problem. Recognizers are, however, not the only types of automata in standard automata theory.

If the experimenter needs to be able to see whether a button is locked without looking at the internal structure, the experimenter could use state machines with more general output. Moore's 1954 paper that defined what are now called *Moore machines* is primarily concerned with strikingly similar experiments.

> 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 [20]

Converting Milner's A and B automata to Moore machines by adding outputs to indicate available transitions from the current state fixes the problem Milner identified and allows the experimenter to distinguish between the machines. In a Moore machine version of A, state A_1 outputs $\{b, c\}$ and A_3 outputs $\{d\}$ *etc.*. A Mealy machine [16] 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, don't push that" (see appendix A for an automata taxonomy).

In a 1962 paper Hartmanis [8] even provided a Moore machine *product* that allows arbitrary interconnection (this paper is referenced by Ginzburg!).

> 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, Loop-free structure[8], p 117)

Those connected Moore machines provide a mathematically coherent model of composition, information hiding[22], and concurrent systems with true parallelism and arbitrary interconnection (without interleaving or some primitive message passing method).

Standard automata theory can accommodate the kind of equivalence Milner requested and also can represent concurrent computation and composition of systems.

3 The persistence of memory

There are no mentions of coverings and machine homomorphisms, algebraic automata theory, automata with output, or automata products in ”Communication and Concurrency”. An obvious question is how Milner could have passed over Moore machines and Moore machine products which seem so relevant to his topic (and which are discussed in Ginzburg’s text). In historical context, however, the ommission is not all that surprising because much of the automata literature became relatively obscure in Computer Science in the 1970s.

By 1979 Hopcroft and Ullman’s standard automata theory text for computer scientists[11] devoted just three out of more than four hundred pages to “State machines with output”. There is no mention of algebraic automata theory or products of automata in that text either. Some of this was undoubtedly due to the attention grabbing successful use of recognizers in both parsing and searching software (e.g. [26]) and for characterizing formal languages. Some is due to Computer Science developing as a separate discipline from Electrical Engineering where state machines with output have been routinely employed in digital circuit design since the 1950s [12, 27, 28]. Also, by 1970, algebraic automata theory had turned more towards the algebra of semigroups and the Krohn-Rhodes theorem [23, 10] and away from computer engineering concerns that motivated earlier work such as in Hartmanis and Stearns book[9]. Although the paper in which Hartmanis defines the concurrent product was influential, Hartmanis only used the concurrent product there to set the stage for the more restricted “loop-free” product and seems to have never come back to it. See Gecseg [5] for later work along similar lines.

Another issue is that automata theory began in the era of what now seem like very small digital circuits and methods for defining automata did not scale easily to the enormous state spaces of concurrent software systems. Although products of automata were a central concern of algebraic automata theory, the motivating engineering issue was simplifying relatively small state machines. Moore’s explanation of one use of his experiments assumes the whole state system can be set out explicitly (my bold added).

> Another application might occur during the course of the design of actual automata. Suppose an engineer has gone far enough in the design of some machine intended as a part of a digital computer, telephone central office, automatic elevator control, etc., **to have described his machine in terms of the list of states and transitions between them, as used in this paper**. He may then wish to perform some gedanken-experiments on his intended machine. If he can find, for instance, that there is no experimental way of distinguishing his design from some machine with fewer states, he might as well build the simpler machine.

Harel’s paper[7] includes an excellent summary of why it’s hard to specify complex systems with state machines using state diagrams and similar. One can argue that this limitation is methodological and not fundamental to automata theory, but that was certainly not the consensus in the 1980s. See [29] for a method of working with large scale state machines and products within automata theory (there is a very brief summary in appendix A).

4 A last note

In the context of the contemporaneous computer science literature, Milner’s familiarity with covering and machine homomorphisms is more remarkable than his neglect of state machines with output and automata products. The second just reflected a widespread belief in the field that state machines were not adequate for the purpose of modeling large scale systems or concurrency (a belief that merits some skepticism at this date).

On the other hand, Milner’s early reference to covering *was* unusual – one of a very few mentions of algebraic automata theory in the formal methods literature. A clue in Ginzburg’s book may be the explanation. Ginzburg wrote his monograph while he was a visiting faculty member at Carnegie Mellon and the acknowledgement section thanks an exceptionally varied and accomplished group of people: Albert Meyer for writing the chapter on Krohn-Rhodes theory, and Abraham Lempel, David Parnas, Carol Thompson, and Zohar Manna for reading and commenting on the manuscript. Manna doesn’t seem to have ever referred to algebraic automata theory in his work – he is best known for temporal logic[15] – but after he left Carnegie Mellon, Manna went to Stanford where he worked in John McCarthy’s lab with, among other people, Robin Milner. Milner's 1971 paper references multiple early papers by Manna as well as Ginzburg's text.

A Appendix State machine taxonomy

Following Ginzburg's taxonomy, mostly.

- A state machine or subautomaton consists of a tuple (Σ, S, s_0) where Σ is the event (input) alphabet, S is a set of states with $s_0 \in S$ the "start state",$\delta : S \times \Sigma \to S$ is the transition function.
- A recognizer[14] 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[20] $(\Sigma, S, s_0, X, \lambda)$ replaces the set of accept states with a set X of outputs and a map $\lambda : S \to X$.
- A Mealy machine[16] $(\Sigma, S, s_0, X, \gamma)$ replaces the Moore machine output map λ with $\gamma : S \times \Sigma \to X$.
- A *sequential machine* is a generic term, usually used more in circuit design which usually has some output see Huffman [12] and Burks and Wang[4]
- A *sequence function* [29] is a map $f : \Sigma^* \to X$ where Σ is the set of inputs, Σ^* is the set of finite sequences over Σ and X is the set of outputs. The intuition is that for $w \in \Sigma^*$, $f(w)$ is the output of the state machine in the state reached by following w from the initial state. Let ϵ be the empty (zero length) sequence and $w \cdot a$ be the sequence obtained by appending event a to finite sequence w on the right. A primitive recursive sequence function is defined with the schema

$$\begin{array}{l} f(w) = \lambda(f'(w)) \\ \text{where } f'(\epsilon) = c, \text{ and } f'(w \cdot a) = \gamma(f'(w), a) \\ \text{for some constant } c, \text{ and function } \gamma \end{array}$$

  Note the similarity to *Moore machines*. As with the arithmetic primitive recursive functions, there are many additional schemas that are conservative in the sense that they also define primitive recursive sequence functions. In particular there is a schema for definining concurrent products like those of Hartmanis.

Much of the work in automata theory was, and continues to be, in the area of digital circuit design particularly 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 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.

See also [13] for an account of the origins of switching theory.

Turing machines are also often considered automata as are linear bounded automata [11] and Buchi automata among others. Those unbounded automata are not discussed here.

B Appendix: The initial modest scope of 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 [4]

For background, Burks was one of the primary engineers who built ENIAC[3] and Wang wrote the first automated theorem prover, invented Wang Tiles, and was the Ph.D. advisor to Stephen Cook, among many other accomplishments.

From Rabin and Scott:

> 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.

The table of contents for a volume edited by Claude Shannon and John McCarthy[25] gives some sense of the field.

CONTENTS

References

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

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

[3] 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.

[4] 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`.

[5] 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`.

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

[7] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. Tech. rep. Weizmann Institute, 1984.

[8] 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.

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

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

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

[12] 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`.

[13] 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`.

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

[15] Z. Manna and A. Pnueli. "The modal logic of programs." In: *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming*. Vol. 71. Lecture Notes in Computer Science. New York: Springer-Verlag, 1979, pp. 385–408.

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

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

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

[19] 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.

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

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

[22] 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`.

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

[24] 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`.

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

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

[27] John F. Wakerly. *Digital Design: Principles and Practices*. 3rd. USA: Prentice-Hall, Inc., 2000. ISBN: 0130555207.

[28] John F. Wakerly. *Digital Design: Principles and Practices (5th Edition)*. 5th. Pearson, 2017. ISBN: 013446009X.

[29] 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`.