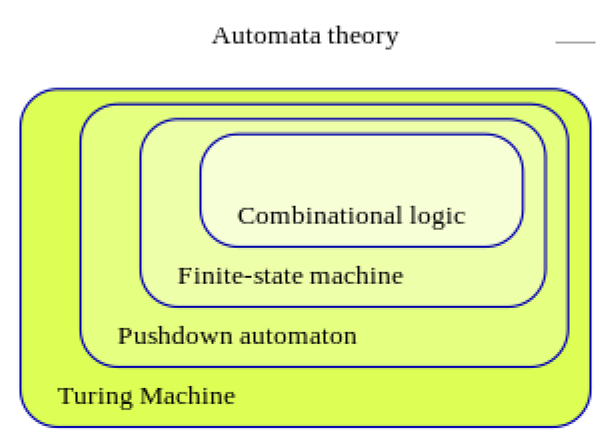
WIKIPEDIA
The Free Encyclopedia

Finite-state machine



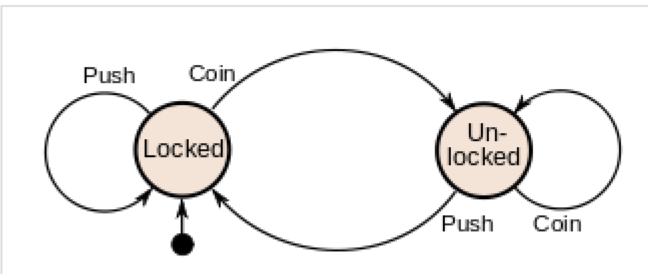
Classes of automata

A **finite-state machine** (**FSM**) or **finite-state automaton** (**FSA**, plural: *automata*), **finite automaton**, or simply a **state machine**, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of *states* at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a *transition*.[1] An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines.[2] For any non-deterministic finite-state machine, an equivalent deterministic one can be constructed.

The behavior of state machines can be observed in many devices in modern society that perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are: vending machines, which dispense products when the proper combination of coins is deposited; elevators, whose sequence of stops is determined by the floors requested by riders; traffic lights, which change sequence when cars are waiting; combination locks, which require the input of a sequence of numbers in the proper order.

The finite-state machine has less computational power than some other models of computation such as the Turing machine.[3] The computational power distinction means there are computational tasks that a Turing machine can do but an FSM cannot. This is because an FSM's memory is limited by the number of states it has. A finite-state machine has the same computational power as a Turing machine that is restricted such that its head may only perform "read" operations, and always has to move from left to right. FSMs are studied in the more general field of automata theory.

Example: coin-operated turnstile



State diagram for a turnstile

An example of a simple mechanism that can be modeled by a state machine is a turnstile.[4][5] A turnstile, used to control access to subways and amusement park rides, is a gate with three rotating arms at waist height, one across the entryway. Initially the arms are locked, blocking the entry, preventing patrons from passing through. Depositing a coin or token in a slot on the turnstile unlocks the arms, allowing a single customer to push through. After the customer passes through, the arms are locked again until another coin is inserted.

A turnstile

Considered as a state machine, the turnstile has two possible states: *Locked* and *Unlocked*.[4] There are two possible inputs that affect its state: putting a coin in the slot *coin* and pushing the arm *push*. In the locked state, pushing on the arm has no effect; no matter how many times the input *push* is given, it stays in the locked state. Putting a coin in – that is, giving the machine a *coin* input – shifts the state from *Locked* to *Unlocked*. In the unlocked state, putting additional coins in has no effect; that is, giving additional *coin* inputs does not change the state. A customer pushing through the arms gives a *push* input and resets the state to *Locked*.

The turnstile state machine can be represented by a state-transition table, showing for each possible state, the transitions between them (based upon the inputs given to the machine) and the outputs resulting from each input:

Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

The turnstile state machine can also be represented by a directed graph called a state diagram *(above)*. Each state is represented by a node (*circle*). Edges (*arrows*) show the transitions from one state to another. Each arrow is labeled with the input that triggers that transition. An input that doesn't cause a change of state (such as a *coin* input in the *Unlocked* state) is represented by a circular arrow returning to the original state. The arrow into the *Locked* node from the black dot indicates it is the initial state.

Concepts and terminology

A *state* is a description of the status of a system that is waiting to execute a *transition*. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received. For example, when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "next" stimulus results in moving to the next station. When the system is in the "CD" state, the "next" stimulus results in moving to the next track. Identical stimuli trigger different actions depending on the current state.

In some finite-state machine representations, it is also possible to associate actions with a state:

- an entry action: performed *when entering* the state, and

- an exit action: performed *when exiting* the state.

Representations

State/Event table

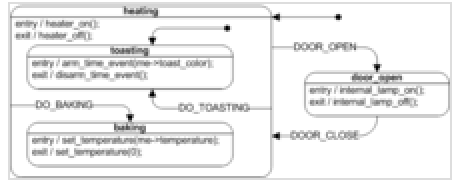


Fig. 1 UML state chart example (a toaster oven)

Several state-transition table types are used. The most common representation is shown below: the combination of current state (e.g. B) and input (e.g. Y) shows the next state (e.g. C). The complete action's information is not directly described in the table and can only be added using footnotes. An FSM definition including the full action's information is possible using state tables (see also virtual finite-state machine).

State-transition table

Current state / Input	State A	State B	State C
Input X	...	...	...
Input Y	...	State C	...
Input Z	...	...	...

Fig. 2 SDL state machine example

UML state machines

The Unified Modeling Language has a notation for describing state machines. UML state machines overcome the limitations of traditional finite-state machines while retaining their main benefits. UML state machines introduce the new concepts of hierarchically nested states and orthogonal regions, while extending the notion of actions. UML state machines have the characteristics of both Mealy machines and Moore machines. They support actions that depend on both the state of the system and the triggering event, as in Mealy machines, as well as entry and exit actions, which are associated with states rather than transitions, as in Moore machines.

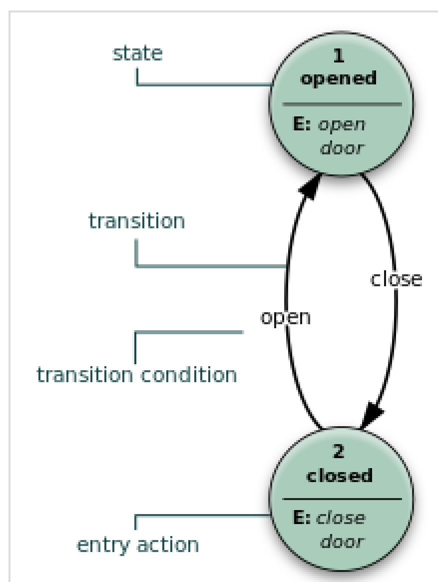


Fig. 3 Example of a simple finite-state machine

SDL state machines

The Specification and Description Language is a standard from ITU that includes graphical symbols to describe actions in the transition:

- send an event
- receive an event
- start a timer
- cancel a timer

- start another concurrent state machine
- decision

SDL embeds basic data types called "Abstract Data Types", an action language, and an execution semantic in order to make the finite-state machine executable.

Other state diagrams

There are a large number of variants to represent an FSM such as the one in figure 3.

Usage

In addition to their use in modeling reactive systems presented here, finite-state machines are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, video game programming, and logic. Finite-state machines are a class of automata studied in automata theory and the theory of computation. In computer science, finite-state machines are widely used in modeling of application behavior (control theory), design of hardware digital systems, software engineering, compilers, network protocols, and computational linguistics.

Classification

Finite-state machines can be subdivided into acceptors, classifiers, transducers and sequencers.[6]

Acceptors

Acceptors (also called *detectors* or **recognizers**) produce binary output, indicating whether or not the received input is accepted. Each state of an acceptor is either *accepting* or *non accepting*. Once all input has been received, if the current state is an accepting state, the input is accepted; otherwise it is rejected. As a rule, input is a sequence of symbols (characters); actions are not used. The start state can also be an accepting state, in which case the acceptor accepts the empty string. The example in figure 4 shows an acceptor that accepts the string "nice". In this acceptor, the only accepting state is state 7.

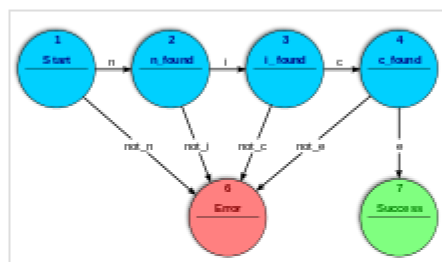


Fig. 4: Acceptor FSM: parsing the string "nice".

A (possibly infinite) set of symbol sequences, called a formal language, is a regular language if there is some acceptor that accepts *exactly* that set. For example, the set of binary strings with an even number of zeroes is a regular language (cf. Fig. 5), while the set of all strings whose length is a prime number is not.[7]:18,71

An acceptor could also be described as defining a language that would contain every string accepted by the acceptor but none of the rejected ones; that language is *accepted* by the acceptor. By definition, the languages accepted by acceptors are the regular languages.

The problem of determining the language accepted by a given acceptor is an instance of the algebraic path problem—itself a generalization of the shortest path problem to graphs with edges weighted by the elements of an (arbitrary) semiring.[8][9]

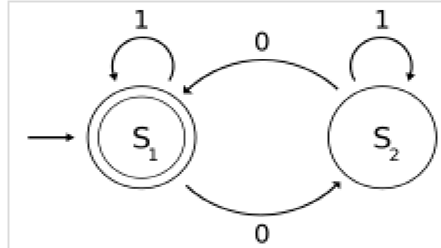


Fig. 5: Representation of an acceptor; this example shows one that determines whether a binary number has an even number of 0s, where S_1 is an *accepting state* and S_2 is a *non accepting state*.

An example of an accepting state appears in Fig. 5: a deterministic finite automaton (DFA) that detects whether the binary input string contains an even number of 0s.

S_1 (which is also the start state) indicates the state at which an even number of 0s has been input. S_1 is therefore an accepting state. This acceptor will finish in an accept state, if the binary string contains an even number of 0s (including any binary string containing no 0s). Examples of strings accepted by this acceptor are ε (the empty string), 1, 11, 11..., 00, 010, 1010, 10110, etc.

Classifiers

Classifiers are a generalization of acceptors that produce *n*-ary output where *n* is strictly greater than two.[10]

Transducers

Transducers produce output based on a given input and/or a state using actions. They are used for control applications and in the field of computational linguistics.

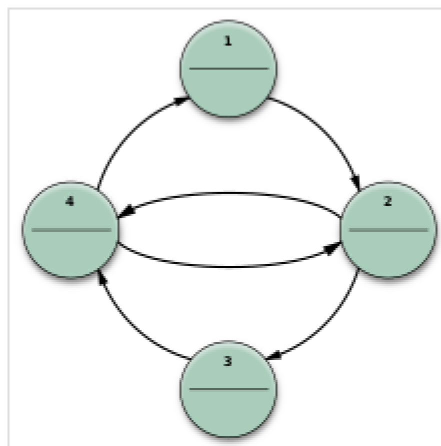


Fig. 6 Transducer FSM: Moore model example

In control applications, two types are distinguished:

Moore machine
: The FSM uses only entry actions, i.e., output depends only on state. The advantage of the Moore model is a simplification of the behaviour. Consider an elevator door. The state machine recognizes two commands: "command_open" and "command_close", which trigger state changes. The entry action (E:) in state "Opening" starts a motor opening the door, the entry action in state "Closing" starts a motor in the other direction closing the door. States "Opened" and "Closed" stop the motor when fully opened or closed. They signal to the outside world (e.g., to other state machines) the situation: "door is open" or "door is closed".

Fig. 7 Transducer FSM: Mealy model example

Mealy machine
: The FSM also uses input actions, i.e., output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states. The example in figure 7 shows a Mealy FSM implementing the same behaviour as in the Moore example (the behaviour depends on the implemented FSM execution model and will work, e.g., for virtual FSM but not for event-driven FSM). There are two input actions (I:): "start motor to close the door if command_close arrives"

and "start motor in the other direction to open the door if command_open arrives". The "opening" and "closing" intermediate states are not shown.

Sequencers

Sequencers (also called *generators*) are a subclass of acceptors and transducers that have a single-letter input alphabet. They produce only one sequence, which can be seen as an output sequence of acceptor or transducer outputs.[6]

Determinism

A further distinction is between *deterministic* (DFA) and *non-deterministic* (NFA, GNFA) automata. In a deterministic automaton, every state has exactly one transition for each possible input. In a non-deterministic automaton, an input can lead to one, more than one, or no transition for a given state. The powerset construction algorithm can transform any nondeterministic automaton into a (usually more complex) deterministic automaton with identical functionality.

A finite-state machine with only one state is called a "combinatorial FSM". It only allows actions upon transition *into* a state. This concept is useful in cases where a number of finite-state machines are required to work together, and when it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools.[11]

Alternative semantics

There are other sets of semantics available to represent state machines. For example, there are tools for modeling and designing logic for embedded controllers.[12] They combine hierarchical state machines (which usually have more than one current state), flow graphs, and truth tables into one language, resulting in a different formalism and set of semantics.[13] These charts, like Harel's original state machines,[14] support hierarchically nested states, orthogonal regions, state actions, and transition actions.[15]

Mathematical model

In accordance with the general classification, the following formal definitions are found.

A *deterministic finite-state machine* or *deterministic finite-state acceptor* is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the input alphabet (a finite non-empty set of symbols);
- S is a finite non-empty set of states;
- s_0 is an initial state, an element of S;
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$ (in a nondeterministic finite automaton it would be $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, i.e. δ would return a set of states);
- F is the set of final states, a (possibly empty) subset of S.

For both deterministic and non-deterministic FSMs, it is conventional to allow δ to be a partial function, i.e. $\delta(s, x)$ does not have to be defined for every combination of $s \in S$ and $x \in \Sigma$. If an FSM M is in a state s, the next symbol is x and $\delta(s, x)$ is not defined, then M can announce an error (i.e. reject the input). This is useful in definitions of general state machines, but less useful when transforming the machine. Some algorithms in their default form may require total functions.

A finite-state machine has the same computational power as a Turing machine that is restricted such that its head may only perform "read" operations, and always has to move from left to right. That is, each formal language accepted by a finite-state machine is accepted by such a kind of restricted Turing machine, and vice versa.[16]

A *finite-state transducer* is a sextuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$, where:

- Σ is the input alphabet (a finite non-empty set of symbols);
- Γ is the output alphabet (a finite non-empty set of symbols);
- S is a finite non-empty set of states;
- s_0 is the initial state, an element of S;
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$;
- ω is the output function.

If the output function depends on the state and input symbol ($\omega : S \times \Sigma \rightarrow \Gamma$) that definition corresponds to the *Mealy model*, and can be modelled as a Mealy machine. If the output function depends only on the state ($\omega : S \rightarrow \Gamma$) that definition corresponds to the *Moore model*, and can be modelled as a Moore machine. A finite-state machine with no output function at all is known as a semiautomaton or transition system.

If we disregard the first output symbol of a Moore machine, $\omega(s_0)$, then it can be readily converted to an output-equivalent Mealy machine by setting the output function of every Mealy transition (i.e. labeling every edge) with the output symbol given of the destination Moore state. The converse transformation is less straightforward because a Mealy machine state may have different output labels on its incoming transitions (edges). Every such state needs to be split in multiple Moore machine states, one for every incident output symbol.[17]

Optimization

Optimizing an FSM means finding a machine with the minimum number of states that performs the same function. The fastest known algorithm doing this is the Hopcroft minimization algorithm.[18][19] Other techniques include using an implication table, or the Moore reduction procedure.[20] Additionally, acyclic FSAs can be minimized in linear time.[21]

Implementation

Hardware applications

In a digital circuit, an FSM may be built using a programmable logic device, a programmable logic controller, logic gates and flip flops or relays. More specifically, a hardware implementation requires a register to store state variables, a block of combinational logic that determines the state transition, and a second block of combinational logic that determines the output of an FSM. One of the classic hardware implementations is the Richards controller.

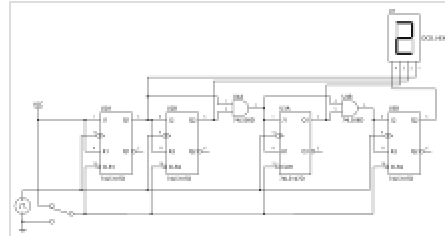


Fig. 9 The circuit diagram for a 4-bit TTL counter, a type of state machine

In a *Medvedev machine*, the output is directly connected to the state flip-flops minimizing the time delay between flip-flops and output.[22][23]

Through state encoding for low power state machines may be optimized to minimize power consumption.

Software applications

The following concepts are commonly used to build software applications with finite-state machines:

- Automata-based programming
- Event-driven finite-state machine
- Virtual finite-state machine
- State design pattern

Finite-state machines and compilers

Finite automata are often used in the frontend of programming language compilers. Such a frontend may comprise several finite-state machines that implement a lexical analyzer and a parser. Starting from a sequence of characters, the lexical analyzer builds a sequence of language tokens (such as reserved words, literals, and identifiers) from which the parser builds a syntax tree. The lexical analyzer and the parser handle the regular and context-free parts of the programming language's grammar.[24]

See also

- Abstract state machines
- Alternating finite automaton
- Communicating finite-state machine
- Control system
- Control table
- Decision tables
- DEVS
- Hidden Markov model
- Petri net
- Pushdown automaton
- Quantum finite automaton
- SCXML
- Semiautomaton
- Semigroup action
- Sequential logic
- State diagram
- Synchronizing word
- Transformation semigroup
- Transition system
- Tree automaton
- Turing machine
- UML state machine

References

1. Wang, Jiacun (2019). *Formal Methods in Computer Science*. CRC Press. p. 34. ISBN 978-1-4987-7532-8.

2. "Finite State Machines – Brilliant Math & Science Wiki" (https://brilliant.org/wiki/finite-state-machines/). *brilliant.org*. Retrieved 2018-04-14.
3. Belzer, Jack; Holzman, Albert George; Kent, Allen (1975). *Encyclopedia of Computer Science and Technology* (https://books.google.com/books?id=W2YLBIdeLIEC). Vol. 25. USA: CRC Press. p. 73. ISBN 978-0-8247-2275-3.
4. Koshy, Thomas (2004). *Discrete Mathematics With Applications* (https://books.google.com/books?id=90KApidK5NwC&pg=PA762). Academic Press. p. 762. ISBN 978-0-12-421180-3.
5. Wright, David R. (2005). "Finite State Machines" (https://web.archive.org/web/20140327131120/http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf) (PDF). *CSC215 Class Notes*. David R. Wright website, N. Carolina State Univ. Archived from the original (http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf) (PDF) on 2014-03-27. Retrieved 2012-07-14.
6. Keller, Robert M. (2001). "Classifiers, Acceptors, Transducers, and Sequencers" (http://www.cs.hmc.edu/~keller/cs60book/12%20Finite-State%20Machines.pdf) (PDF). *Computer Science: Abstraction to Implementation* (http://www.cs.hmc.edu/~keller/cs60book/%20%20%20All.pdf) (PDF). Harvey Mudd College. p. 480.
7. John E. Hopcroft and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation* (https://archive.org/details/introductiontoau00hopc). Reading/MA: Addison-Wesley. ISBN 978-0-201-02988-8.
8. Pouly, Marc; Kohlas, Jürg (2011). *Generic Inference: A Unifying Theory for Automated Reasoning*. John Wiley & Sons. Chapter 6. Valuation Algebras for Path Problems, p. 223 in particular. ISBN 978-1-118-01086-0.
9. Jacek Jonczy (Jun 2008). "Algebraic path problems" (https://web.archive.org/web/20140821054702/http://www.iam.unibe.ch/~run/talks/2008-06-05-Bern-Jonczy.pdf) (PDF). Archived from the original (http://www.iam.unibe.ch/~run/talks/2008-06-05-Bern-Jonczy.pdf) (PDF) on 2014-08-21. Retrieved 2014-08-20., p. 34
10. Felkin, M. (2007). Guillet, Fabrice; Hamilton, Howard J. (eds.). *Quality Measures in Data Mining - Studies in Computational Intelligence*. Vol. 43. Springer, Berlin, Heidelberg. pp. 277–278. doi:10.1007/978-3-540-44918-8_12 (https://doi.org/10.1007%2F978-3-540-44918-8_12). ISBN 978-3-540-44911-9.
11. Brutscheck, M., Berger, S., Franke, M., Schwarzbacher, A., Becker, S.: Structural Division Procedure for Efficient IC Analysis. IET Irish Signals and Systems Conference, (ISSC 2008), pp.18–23. Galway, Ireland, 18–19 June 2008. [1] (http://arrow.dit.ie/engschececon/2/)
12. "Tiwari, A. (2002). Formal Semantics and Analysis Methods for Simulink Stateflow Models" (http://www.csl.sri.com/users/tiwari/papers/stateflow.pdf) (PDF). *sri.com*. Retrieved 2018-04-14.
13. Hamon, G. (2005). *A Denotational Semantics for Stateflow*. International Conference on Embedded Software. Jersey City, NJ: ACM. pp. 164–172. CiteSeerX 10.1.1.89.8817 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.8817).
14. "Harel, D. (1987). A Visual Formalism for Complex Systems. Science of Computer Programming, 231–274" (https://web.archive.org/web/20110715110405/http://www.fceia.unr.edu.ar/asist/harel01.pdf) (PDF). Archived from the original (http://www.fceia.unr.edu.ar/asist/harel01.pdf) (PDF) on 2011-07-15. Retrieved 2011-06-07.
15. "Alur, R., Kanade, A., Ramesh, S., & Shashidhar, K. C. (2008). Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. International Conference on Embedded Software (pp. 89–98). Atlanta, GA: ACM" (https://web.archive.org/web/20110715110405/http://drona.csa.iisc.ernet.in/~kanade/publications/symbolic_analysis_for_improving_simulation_coverage_of_simulink_stateflow_models.pdf) (PDF). Archived from the original (http://drona.csa.iisc.ernet.in/~kanade/publications/symbolic_analysis_for_improving_simulation_coverage_of_simulink_stateflow_models.pdf) (PDF) on 2011-07-15.

16. Black, Paul E (12 May 2008). "Finite State Machine" (https://web.archive.org/web/20181013023517/https://xlinux.nist.gov/dads/HTML/finiteStateMachine.html). *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology. Archived from the original (https://xlinux.nist.gov/dads/HTML/finiteStateMachine.html) on 13 October 2018. Retrieved 2 November 2016.
17. Anderson, James Andrew; Head, Thomas J. (2006). *Automata theory with modern applications* (https://books.google.com/books?id=ikS8BLdLDxIC&pg=PA105). Cambridge University Press. pp. 105–108. ISBN 978-0-521-84887-9.
18. Hopcroft, John E. (1971). An *n* log *n* algorithm for minimizing states in a finite automaton (ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf) (PDF) (Technical Report). Vol. CS-TR-71-190. Stanford Univ.
19. Almeida, Marco; Moreira, Nelma; Reis, Rogerio (2007). On the performance of automata minimization algorithms (https://web.archive.org/web/20090117201637/http://www.dcc.fc.up.pt/dcc/Pubs/TReports/TR07/dcc-2007-03.pdf) (PDF) (Technical Report). Vol. DCC-2007-03. Porto Univ. Archived from the original (http://www.dcc.fc.up.pt/dcc/Pubs/TReports/TR07/dcc-2007-03.pdf) (PDF) on 17 January 2009. Retrieved 25 June 2008.
20. Edward F. Moore (1956). C.E. Shannon and J. McCarthy (ed.). "Gedanken-Experiments on Sequential Machines". *Annals of Mathematics Studies*. Princeton University Press. **34**: 129–153. Here: Theorem 4, p.142.
21. Revuz, D. (1992). "Minimization of Acyclic automata in Linear Time". *Theoretical Computer Science*. **92**: 181–189. doi:10.1016/0304-3975(92)90142-3 (https://doi.org/10.1016%2F0304-3975%2892%2990142-3).
22. Kaeslin, Hubert (2008). "Mealy, Moore, Medvedev-type and combinatorial output bits" (https://books.google.com/books?id=gdRStcYgf2oC&q=medvedev+fsm&pg=PA787). *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press. p. 787. ISBN 978-0-521-88267-5.
23. Slides (http://users.etech.haw-hamburg.de/users/Schwarz/En/Lecture/Ds/Notes/DigSys1.pdf) Archived (https://web.archive.org/web/20170118123034/http://users.etech.haw-hamburg.de/users/Schwarz/En/Lecture/Ds/Notes/DigSys1.pdf) 18 January 2017 at the Wayback Machine, *Synchronous Finite State Machines; Design and Behaviour*, University of Applied Sciences Hamburg, p.18
24. Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986). *Compilers: Principles, Techniques, and Tools* (1st ed.). Addison-Wesley. ISBN 978-0-201-10088-4.

Further reading

General

- Sakarovitch, Jacques (2009). *Elements of automata theory*. Translated from the French by Reuben Thomas. Cambridge University Press. ISBN 978-0-521-84425-3. Zbl 1188.68177 (https://zbmath.org/?format=complete&q=an:1188.68177).
- Wagner, F., "Modeling Software with Finite State Machines: A Practical Approach", Auerbach Publications, 2006, ISBN 0-8493-8086-3.
- ITU-T, *Recommendation Z.100 Specification and Description Language (SDL)* (http://www.itu.int/rec/T-REC-Z.100-200711-I/en)
- Samek, M., *Practical Statecharts in C/C++* (http://www.state-machine.com/psicc/index.php), CMP Books, 2002, ISBN 1-57820-110-1.

- Samek, M., *Practical UML Statecharts in C/C++, 2nd Edition* (http://www.state-machine.com/psicc2/index.php), Newnes, 2008, ISBN 0-7506-8706-1.
- Gardner, T., *Advanced State Management* (http://www.troyworks.com/cogs/) Archived (https://web.archive.org/web/20081119071252/http://www.troyworks.com/cogs/) 2008-11-19 at the Wayback Machine, 2007
- Cassandras, C., Lafortune, S., "Introduction to Discrete Event Systems". Kluwer, 1999, ISBN 0-7923-8609-4.
- Timothy Kam, *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Publishers, Boston 1997, ISBN 0-7923-9842-4
- Tiziano Villa, *Synthesis of Finite State Machines: Logic Optimization*. Kluwer Academic Publishers, Boston 1997, ISBN 0-7923-9892-0
- Carroll, J., Long, D., *Theory of Finite Automata with an Introduction to Formal Languages (https://philpapers.org/archive/CARTOF.pdf)*. Prentice Hall, Englewood Cliffs, 1989.
- Kohavi, Z., *Switching and Finite Automata Theory*. McGraw-Hill, 1978.
- Gill, A., *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.
- Ginsburg, S., *An Introduction to Mathematical Machine Theory*. Addison-Wesley, 1962.

Finite-state machines (automata theory) in theoretical computer science

- Arbib, Michael A. (1969). *Theories of Abstract Automata* (1st ed.). Englewood Cliffs, N.J.: Prentice-Hall, Inc. ISBN 978-0-13-913368-8.
- Bobrow, Leonard S.; Arbib, Michael A. (1974). *Discrete Mathematics: Applied Algebra for Computer and Information Science* (https://archive.org/details/discretemathemat0000bobr) (1st ed.). Philadelphia: W. B. Saunders Company, Inc. ISBN 978-0-7216-1768-8.
- Booth, Taylor L. (1967). *Sequential Machines and Automata Theory* (1st ed.). New York: John Wiley and Sons, Inc. Library of Congress Card Catalog Number 67-25924.
- Boolos, George; Jeffrey, Richard (1999) [1989]. *Computability and Logic* (https://archive.org/details/computabilitylog0000bool_r8y9) (3rd ed.). Cambridge, England: Cambridge University Press. ISBN 978-0-521-20402-6.
- Brookshear, J. Glenn (1989). *Theory of Computation: Formal Languages, Automata, and Complexity*. Redwood City, California: Benjamin/Cummings Publish Company, Inc. ISBN 978-0-8053-0143-4.
- Davis, Martin; Sigal, Ron; Weyuker, Elaine J. (1994). *Computability, Complexity, and Languages and Logic: Fundamentals of Theoretical Computer Science* (2nd ed.). San Diego: Academic Press, Harcourt, Brace & Company. ISBN 978-0-12-206382-4.
- Hopcroft, John; Ullman, Jeffrey (1979). *Introduction to Automata Theory, Languages, and Computation* (1st ed.). Reading Mass: Addison-Wesley. ISBN 978-0-201-02988-8.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2001). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Reading Mass: Addison-Wesley. ISBN 978-0-201-44124-6.
- Hopkin, David; Moss, Barbara (1976). *Automata*. New York: Elsevier North-Holland. ISBN 978-0-444-00249-5.
- Kozen, Dexter C. (1997). *Automata and Computability* (1st ed.). New York: Springer-Verlag. ISBN 978-0-387-94907-9.
- Lewis, Harry R.; Papadimitriou, Christos H. (1998). *Elements of the Theory of Computation* (2nd ed.). Upper Saddle River, New Jersey: Prentice-Hall. ISBN 978-0-13-262478-7.
- Linz, Peter (2006). *Formal Languages and Automata* (4th ed.). Sudbury, MA: Jones and Bartlett. ISBN 978-0-7637-3798-6.

- Minsky, Marvin (1967). *Computation: Finite and Infinite Machines* (https://archive.org/details/computationfinit0000mins) (1st ed.). New Jersey: Prentice-Hall.
- Papadimitriou, Christos (1993). *Computational Complexity* (1st ed.). Addison Wesley. ISBN 978-0-201-53082-7.
- Pippenger, Nicholas (1997). *Theories of Computability* (1st ed.). Cambridge, England: Cambridge University Press. ISBN 978-0-521-55380-3.
- Rodger, Susan; Finley, Thomas (2006). *JFLAP: An Interactive Formal Languages and Automata Package* (1st ed.). Sudbury, MA: Jones and Bartlett. ISBN 978-0-7637-3834-1.
- Sipser, Michael (2006). *Introduction to the Theory of Computation* (2nd ed.). Boston Mass: Thomson Course Technology. ISBN 978-0-534-95097-2.
- Wood, Derick (1987). *Theory of Computation* (1st ed.). New York: Harper & Row, Publishers, Inc. ISBN 978-0-06-047208-5.

Abstract state machines in theoretical computer science

- Gurevich, Yuri (July 2000). "Sequential Abstract State Machines Capture Sequential Algorithms" (http://research.microsoft.com/~gurevich/Opera/141.pdf) (PDF). *ACM Transactions on Computational Logic*. **1** (1): 77–111. CiteSeerX 10.1.1.146.3017 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.146.3017). doi:10.1145/343369.343384 (https://doi.org/10.1145%2F343369.343384). S2CID 2031696 (https://api.semanticscholar.org/CorpusID:2031696).

Machine learning using finite-state algorithms

- Mitchell, Tom M. (1997). *Machine Learning* (1st ed.). New York: WCB/McGraw-Hill Corporation. ISBN 978-0-07-042807-2.

Hardware engineering: state minimization and synthesis of sequential circuits

- Booth, Taylor L. (1967). *Sequential Machines and Automata Theory* (1st ed.). New York: John Wiley and Sons, Inc. Library of Congress Card Catalog Number 67-25924.
- Booth, Taylor L. (1971). *Digital Networks and Computer Systems* (https://archive.org/details/digitalnetworksc00boot) (1st ed.). New York: John Wiley and Sons, Inc. ISBN 978-0-471-08840-0.
- McCluskey, E. J. (1965). *Introduction to the Theory of Switching Circuits* (1st ed.). New York: McGraw-Hill Book Company, Inc. Library of Congress Card Catalog Number 65-17394.
- Hill, Fredrick J.; Peterson, Gerald R. (1965). *Introduction to the Theory of Switching Circuits* (1st ed.). New York: McGraw-Hill Book Company. Library of Congress Card Catalog Number 65-17394.

Finite Markov chain processes

> "We may think of a Markov chain as a process that moves successively through a set of states s_1, s_2, …, s_r. … if it is in state s_i it moves on to the next stop to state s_j with probability p_{ij}. These probabilities can be exhibited in the form of a transition matrix" (Kemeny (1959), p. 384)

Finite Markov-chain processes are also known as subshifts of finite type.

- Booth, Taylor L. (1967). *Sequential Machines and Automata Theory* (1st ed.). New York: John Wiley and Sons, Inc. Library of Congress Card Catalog Number 67-25924.
- Kemeny, John G.; Mirkil, Hazleton; Snell, J. Laurie; Thompson, Gerald L. (1959). *Finite Mathematical Structures* (https://archive.org/details/finitemathematic0000keme_h5g0) (1st ed.). Englewood Cliffs, N.J.: Prentice-Hall, Inc. Library of Congress Card Catalog Number 59-12841. Chapter 6 "Finite Markov Chains".

External links

- Finite State Automata (https://curlie.org/Computers/Computer_Science/Theoretical/Automata_Theory/Finite_State_Automata/) at Curlie
- *Modeling a Simple AI behavior using a Finite State Machine* (https://archive.today/20121202054532/http://blog.manuvra.com/modeling-a-simple-ai-behavior-using-a-finite-state-machine/) Example of usage in Video Games
- Free On-Line Dictionary of Computing (https://web.archive.org/web/20171211180457/http://foldoc.org/finite+state+machine) description of Finite-State Machines
- NIST Dictionary of Algorithms and Data Structures (https://web.archive.org/web/20181013023517/https://xlinux.nist.gov/dads/HTML/finiteStateMachine.html) description of Finite-State Machines
- A brief overview of state machine types (https://blogs.itemis.com/en/a-brief-overview-of-state-machine-types), comparing theoretical aspects of Mealy, Moore, Harel & UML state machines.