

Misunderstandings about state machines

By Ferdinand Wagner and Peter Wolstenholme

Introduction

The definition of a finite state machine (called also for simplicity a state machine) seems to be not quite obvious and requires discussion. The concept, although born 50 years ago, is still not well understood or interpreted in the software domain, despite its wide application in hardware design. Misunderstandings about state machines have produced several stories and half-truths: for instance the states explosion phenomenon which discourages the usage of this very useful concept. The concept of the state machine has been several times (unintentionally?) reinvented for software.

A state machine is the oldest known formal model for sequential behaviour i.e. behaviour that cannot be defined by the knowledge of inputs only, but depends on the history of the inputs. A state machine is not a heuristic model which could be interpreted in many ways, at the whim of the user, but rather has a good theoretical basis. This means that it is relatively easy to invent verification methods to prove the correctness of a system built as a system of state machines. Probably, the state machine is the only known model (of the many used in software development) that really gives a designer a chance to verify a control system and thus, it is the only way to produce reliable control software.

A state machine is intuitively understandable and therefore acceptable to many designers. There are some application domains where the usage of state machines seems natural; telecommunication and embedded systems are primary examples. Because control tasks are present in all software, state machine models could be used in most software developed nowadays. There are no reasons of principle to avoid the use of state machines. On the other hand, there are no reasonable justifications for believing that to code the behaviour of the control system without any formal model could produce better software than a system based on state machines.

The state machine concept is not the only common model. There are other models which have been invented, often being inspired by the ideas of states and flow charts: for instance, the models used in PLC languages. The central idea of all these control models is a state (sometimes renamed). Whether the derived method really implies an improved state machine can be questioned: the reasons for invention of a new method are not always of a technical nature. Anyway, it seems that the state machine remains the preferred model for describing the behaviour of control systems.

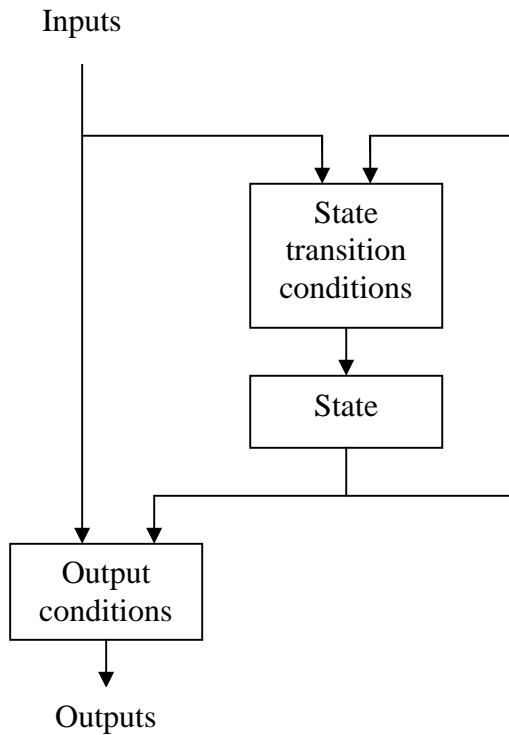
To quote some original sources I took some books [1] [2] which I had used to recommend to my students at the university several years ago and which are still on my book-shelf. The authors of these books did not invent the finite state machine but just cited the original books and papers. I copied a few examples of those references [3-7] from these books.

This paper tries to put the concept of a state machine into perspective and shows how the VFSM concept and its practical implementation StateWORKS realize the original, classic definition of a state machine. In addition, it shows how StateWORKS realizes a system of state machines.

State machine

Classic definition

A control system determines its outputs depending on its inputs. If the present input values are sufficient to determine the outputs the system is a combinatorial system. If the control system needs some additional information about the sequence of input changes to determine the outputs the system is a sequential system. The logical part of the system responsible for the system behavior is called a state machine. Sometimes, the combinatorial systems are treated separately or considered as a kind of degenerate state machine. To keep things simple, we call any logic that determines a system behavior a state machine. According to this definition, a state machine can be represented by the diagram.



The history of input changes required for clear determination of the state machine behavior is stored in an internal variable **State**.

Two basic models of state machine have been defined: Moore and Mealy types. In the Mealy type the outputs are functions of the state and inputs. The Moore model means that outputs are functions of the state only. Mixed models are also used.

The history of input changes required for clear determination of the state machine behavior is stored in an internal variable **State**.

Example:

Let us design a state machine that has 3 digital (Boolean) inputs: a , b , c and a digital (Boolean) output Y . The output is to be *true* if all inputs are *true*.

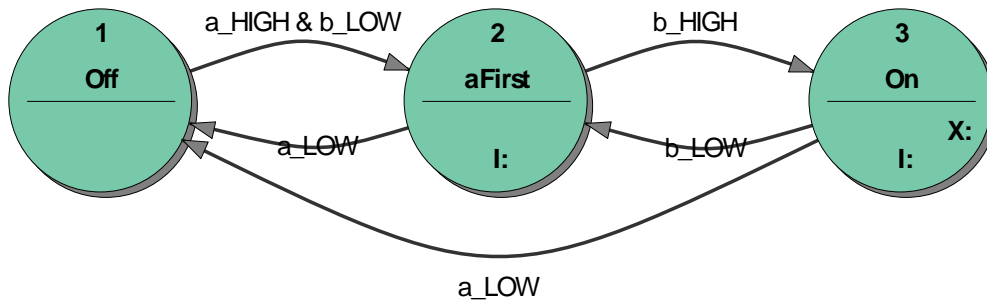
The solution for this task is rather obvious and can be expressed by a logical equation:

$$Y = a \text{ AND } b \text{ AND } c$$

Let us now complicate the problem a little: the output is to be *true* if all inputs are *true* and the input a has been *true* before b .

It is obvious that in this case the inputs do not directly determine the output. The output depends also on the sequence of input changes. The missing information (a being *true* before b) can be supplied by the variable *State*.

Using the typical state machine presentation we can describe the state machine which is a solution for the problem by means of the following state transition diagram:



The *State* can have values: *Off*, *aFirst* and *On*. The transition conditions are indicated directly in the state transition diagram. The combinatorial part of the state machine which determines the output *Y* is a function of an input (*c*) and the *State* and reads:

$$Y = \text{On AND } c$$

The details of this solution can be seen in the StateWORKS project *Example3*¹.

The state machine specified for the example is one of a number of possible solutions, corresponding to the Mealy model. We could define other state machines, for instance corresponding to the Moore model.

Software systems

The above definition of a state machine was introduced for hardware design and resulted in better organized and understandable hardware projects (in the days when there was no software).

Programming started some time later as a more or less ad-hoc activity based rather on human mental abilities to solve puzzles, than on any methods. The problems in software (as programming came to be called) were growing very fast. In addition, the limited number of brilliant programmers required software to be treated as a normal engineering activity with some organization, methods, supervisions, budgets, etc. Among others, the state machine concept started to be used in software design. The integration of a state machine model into software resulted in some new ideas or reinventions.

Some extensions and changes in the state machine terminology have taken place. Especially, the outputs are in software state machines rather called actions and several types of actions have been defined: entry, exit, input and transition ones. These changes are welcome and useful extensions reflecting additional possibilities offered by software implementation. They do not have any relevant influence on the state machine model. They have increased the difficulty of graphical representation of state machines.

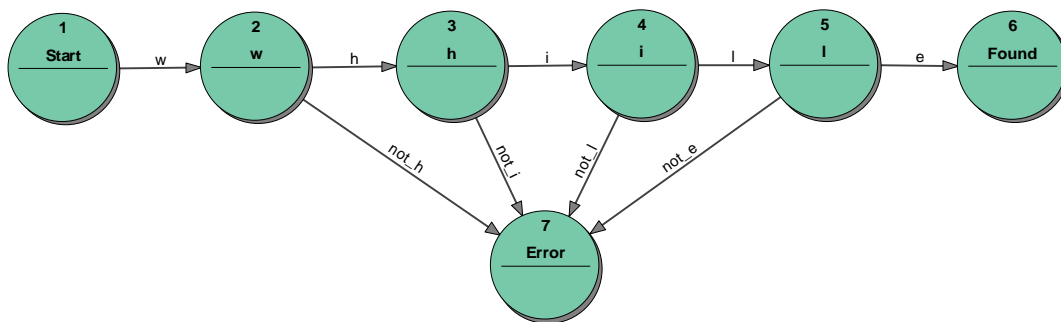
Not all ideas have been justified and they may have been caused by missing knowledge. Whatever were the reasons, we should not accept things that do not make sense and often are indeed against common sense. Two typical software issues had a strong influence on state machine misinterpretations and implementations: the "Event Driven" concept and the "Parser" problem.

Software systems are often event driven. This concept means that the software, in principle, does nothing, while waiting for an event. If the event occurs the software reacts to it and returns to the waiting state. The concept is in opposition to polling systems that continuously check inputs.

¹ The examples are available in StateWORKS Studio. They can be tested using StateWORKS Studio tools. Alternatively, the xml representation can be displayed using any browser.

Conceptually, from the state machine point of view the way of delivering inputs (events or polling) is irrelevant. If the software control system uses a state machine model to describe the system behavior its model should not be determined by the input acquisition system but rather by the application requirements. Unfortunately, we find in some software implementations of state machines a purely event-driven state machine model which implies that the state machine has to store not only the history of state changes but the actual present value of inputs.

This extreme interpretation of a state machine is at least partly explained by the background of some programmers who encountered state machines as a model for Parser behavior. In its basic type a Parser² has one input of characters and detects words in the incoming string. If the Parser has to detect for instance a word “**while**” its specification using a state machine leads to a state machine with few states shown in the following state transition diagram.



Here, the *State* represents really the history of (including the present value of) the (single) input. This is correct but such a concept is in practice limited to the parser application. There is no concept of time in this application, and the state machine does not need to know about inputs other than the character stream it is processing. Transferring this idea to other applications does not make sense as there is no reason to store the present input values in the *State* because they are anyway explicitly available. The only “reason” would be to make things more complex than they are.

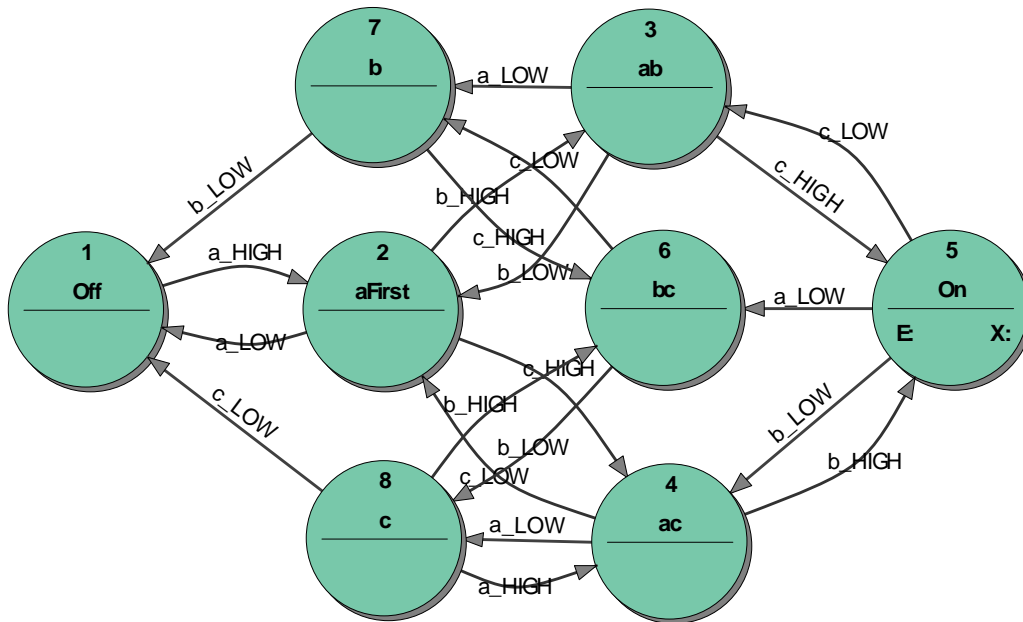
Here we are coming to the next misunderstanding found in discussions about state machines: the *state explosion* phenomenon. This phenomenon is directly derived from the above discussed problem of pure event driven models of state machines. Of course, if we start to store in the state the present value of inputs we get the state explosion problem. In this model the number of states increases enormously as each truly required state path must be repeated for all possible input values.

For instance, the trivial *Example3* shown before with 3 inputs would require 8 states if we redesign to use the output function and will express the output by a state only. The details of this solution can be seen in the StateWORKS project *Example8*³.

This solution could be acceptable as we do not consider 8 states as a state explosion yet (but why use a solution which is obviously more complex than the 3-states automaton?). Imagine the consequences for a more realistic state machine with 6 or more inputs. Any relatively simple problem explodes immediately. And if you want to kill completely the state machine usage make some calculation with 100 or more inputs “proving” that state machines are useless for any practical application.

² Modern parsers use more sophisticated tools and methods for string analyzing, especially regular expressions allowing whole words to be filtered from strings. We do not discuss here parsers and the basic idea of character parsing is taken from text books.

³ See footnote ¹) on the previous page.



The state explosion topic is an example of an artificially created problem which works against usage of state machines.

Let's now present the last difficulty: the size of a state machine. Sometimes we read remarks like: a computer is a state machine but the specification of the computer state machine would require millions (maybe billions) of states. Therefore, it would be useless to try to realize a computer using a state machine model. This argument has approximately the same value as a statement like: to write an operating system requires millions of lines of code. Because a programmer cannot see and conquer so many lines it is impossible to program an operating system.

The method of solving complex problems is known: partitioning into smaller units with a good interface among these units. Very often, there is nobody who knows well all parts of a large system which were written by several individuals but the system is in some way comprehensible and conquerable. This principle applied to state machine specification means that a complex control problem should be partitioned among several state machines, the entire system of state machines being an equivalent to a large single state machine. Note, that a pure theoretical calculation shows that for instance 10 state machines each having 100 states represents $100 \times 100 \times \dots \times 100 = 100^{10}$ states. In addition, building of complex control systems does not mean that we start with a complex state machine and then try to partition it. We rather specify several state machines for specific tasks and then link them together.

Hence, we arrive at the second topic relating to systems of state machines – the interface among the state machines. Very often we see systems of state machines which are built as separate units with some communications channel between them which allow any state machine to exchange messages with any other state machine. This is a naïve concept based on a programmers' favorite structure, to have a system where each part can send a message to another part. The astonishing fact is that we have known since at least 30 years that this kind of solution leads to deadlocks and similar timing problems. We have even invented nice models like Petri nets to illustrate these difficulties. Anyway, sometimes we forget the basic rules. To make another comparison: building a system of communicating state machines without any restriction on the communication reminds of programming with "go to". Sending unconstrained messages to another state machine is like a jump to another program part (Dijkstra showed us 30 years ago the consequences of such "programming style"). Exactly, the same problem is generated by unconstrained systems of state machines.

Inventions

After the critical analysis of some incorrect ideas about the state machine concept in software control systems let's mention some positive developments. The most interesting, true invention has been the

introduction of the Statechart concept. Statecharts have extended the state machine model by elements which make it a completely different modeling tool. The most important element is the view of sub-states through a state. That means that, per definition, a design begins with a single state which can be expanded into several sub-states, each of them being again expanded into further sub-states. Hence, instead of dealing with several state machines by a complex system we always stay in one state machine expanding the sub-states. To implement this idea Statecharts introduces several additional concepts, some of them quite sophisticated, like: several state entries (enter-by-history H, conditional C, selection S), several state exits (split, merging by condition, independent), activities and actions, special actions (clear history, internal actions), ORing states, ANDing states and others. Thus, Statecharts differ from state machines, not only by using rounded rectangles for states instead of circles, but as a truly different way of specifying the behavior of sequential systems.

The StateCharts concept originated in attempts to design a complex avionics system, and define its behaviour as a finite state machine. In the course of this work, the state machine transition diagram became remarkably complex, covering a complete room wall, and those involved decided that they had "proved" that the classical transition diagram was impossible to work with! Tragically, they took the wrong direction, having missed the possibility of building systems out of sub-systems, or components, so that each was in itself small enough to be manageable. This was perhaps the result of an attempt to extend top-down system design practices, commendable in themselves, to the implementation phase. It seems strange that the lessons of "structured programming" of the previous two decades were forgotten in this instance.

We do not want to discuss Statecharts any further here. We mention it only as an example of an interesting concept which introduces a new model for specifying a behavior of a control system. Unfortunately, Statecharts is sometimes considered as a replacement for the state machine concept. These assumptions have either a commercial aspect (UML apostles try to kill anything that does not fit into their world) or just ignorance. There is no direct translation between a system of state machines and a Statecharts representation of a control system. Statecharts is a new, interesting model of control systems which is used in UML for behavior specification and it not too helpful for creating the software.

We should also like to mention also some reinventions of state machines that are true "wheel reinventions" caused probably by their author's ignorance. The authors of this paper have seen at least three proposals of this kind, the last where the state machine is called a "transition network". Other elements of a state machine: states and actions have been not renamed in that reinvention.

The VFMS solution and StateWORKS implementation

State machine concepts are defined for a Boolean environment: the transition and input action conditions are Boolean equations. In most hardware designs the environment has been by definition a Boolean one: the inputs are pure digital inputs. Software systems are designed very often for a less homogenous environment: the inputs may be of any sort, numerical (analog) and messages whose content determines the behavior being the best examples. The software solutions provide the answer for this problem: the Boolean conditions can be filtered out from the inputs by writing a program. The coding solution is an obvious way of implementing state machines.

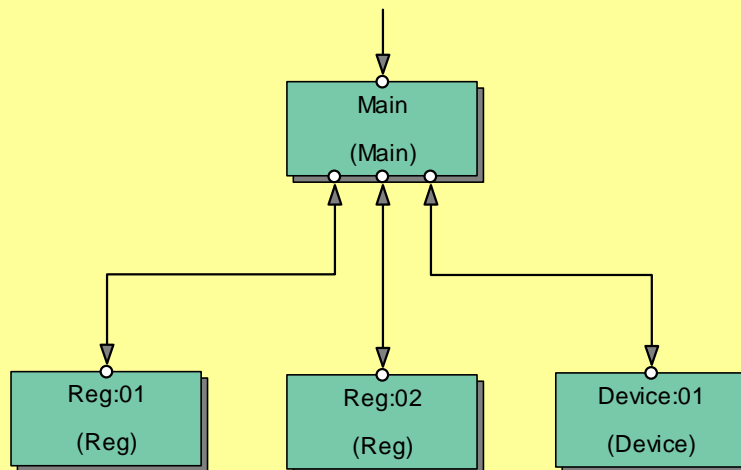
VFMS

A virtual finite state machine (VFMS) is a state machine which operates in a virtual environment. This virtual environment (VI) is created defining all control-relevant information about inputs and outputs of a control system. The information is in the form of name lists, for instance: Door_Open and Door_Closed or Temperature_Ok, Temperature_Low and Temperature_High. The VFMS behaviour is then specified using only this information and states. In establishing specifications of the transition and input action conditions only AND and OR logical operators can be used (the multi-valued control information does not allowed the usage of the NOT operator).

The VFMS concept is based on an assumption that any input contains some information relevant for the control. This information is multi-valued and therefore cannot be handled directly as a Boolean value. The definition of a virtual environment and positive logic algebra VFMS concept allows conditions to be expressed like Boolean values, which means that a table based implementation of transition and input action conditions is possible. Details of the VFMS concept can be found in [8].

To keep state machines to a manageable size the behavior of a complex system should be specified using several, or perhaps very many, state machines. To get a manageable system of state machines one has to have a well defined and understandable structure. StateWORKS [9] defines a “command / state” interface among state machines and recommends a hierarchical structure for realization of complex control systems. Details of the StateWORKS implementation can be found in [10][11].

Constructing a hierarchical system of VFMSs



The recommended way of designing a large system is to use a quite large number of state machines, each with a defined task of sufficiently restricted complexity so that it remains understandable. In most cases, these ought to be arranged in a hierarchy, and interconnections need to be carefully designed, so as to ensure that the entire system is coherent. It is sufficient to allow for each state machine's state to be made available to any layers above it, and to arrange for commands to be issued by upper-layer state machines to machines in the next lower level. A mechanism for cancelling (or acknowledging) these commands when they have been acted upon is also needed. This strategy produces large systems with deterministic and reliable behaviour. It is usually best to implement such systems from the lower levels upwards, after an initial top-down planning phase which gives a rough idea of the anticipated structure.

Conclusions

We discussed some misunderstandings and misinterpretations which have come into being while applying the concept of a state machine to software problems: the pure event driven state machine model resulting from the “parser” solution, the states explosion phenomenon and unconstrained systems of state machines. The problems have emerged while some people are trying to “reinvent the wheel” defining anew but incorrectly and unnecessarily the well known concept of state machines.

A proper interpretation of the classic state machine idea is seen in the VFSM concept and its practical implementation StateWORKS. StateWORKS is a development system and a package of run-time systems which gives designers a chance to build reliable software control systems. Systems built with the help of StateWORKS do not have states explosion or other disastrous timing and synchronization problems encountered in other systems that use self-invented state machine models.

Although one of the objectives of StateWORKS is to eliminate large amounts of complex coding, by use of formal, “Platform-Independent Models” directly in the run-time systems, so as to avoid even automatic code generation, the second aspect of major significance is its ability to create large-scale systems of correctly-managed finite state machines working in a well-organized fashion.

References

1. Hopcroft, J.E., Ullman, J.D.: "Introduction to Automata Theory, Languages and Computation", Addison-Wesley, Reading, Mass. 1979.
2. Kohavi Z.: "Switching and Finite Automata Theory", McGraw-Hill, New York, 1970.
3. Gill, A.: "Introduction to the Theory of Finite-state Machines", McGraw-Hill, New York, 1962.
4. Huffman, D.A.: The Synthesis of Sequential Switching Circuits, J. Franklin Inst., vol. 257, pp. 161-190, 1954; pp. 275-303, April, 1954. Reprinted in Moore [7].
5. Mealy, G.H.: A Method of Synthesizing Sequential Circuits, Bell System Tech. J., vol 34, pp. 1054-1079, September, 1955.
6. Moore, E.F. (ed.): "Gedanken-experiments on Sequential Circuits", pp.129-153, Automata Studies, Annals of Mathematical Studies, no. 34, Princeton University Press, Princeton, N.J., 1956.
7. Moore, E.F. (ed.): "Sequential Machines: Selected Papers", Addison Wesley, Reading, Mass., 1964.
8. StateWORKS Technical Note: The Virtual Environment and Positive-Logic Algebra.
9. StateWORKS Technical Note: What is StateWORKS?
10. StateWORKS Technical Note: Hierarchical system of state machines.
11. StateWORKS Technical Note: Commands.

Note: all StateWORKS Technical Notes are available on the www.stateworks.com web site.