A crash course in UML state machines: Part 1

Miro Samek

3/9/2009 12:30 PM EDT

In this article series, Miro Samek introduces you to the basics of UML state machines basic notation, clarifying semantics, and provides some code examples to the role of UML state machines have in practical, everyday programming rather than mathematical abstractions. Up first: The over simplification of the event-action paradigm.

To read Part 2, go to "UML extensions to the traditional FSM formalism"

To read Part 3, go to "Designing a UML state machine"

Traditional, sequential programs can be structured as a single flow of control, using standard constructs such as loops and nested function calls. Such programs represent most of the execution context in the location of the program counter, in the procedure call tree, and in the temporary variables allocated on the stack.

Event-driven programs, in contrast, require a series of fine-granularity event-handler functions for handling events. These event-handler functions must execute quickly and always return to the main event-loop, so no context can be preserved in the call tree and the program counter. In addition, all stack variables disappear across calls to the separate event-handlers. Thus, event-driven programs rely heavily on static variables to preserve the execution context from one event-handler invocation to the next.

Consequently, one of the biggest challenges of event-driven programming lies in managing the execution context represented as data. The main problem here is that the context data must somehow feed back into the control flow of the event-handler code so that each event handler can execute only the actions appropriate in the current context. Traditionally, this dependence on the context very often leads to deeply nested if-else constructs that direct the flow of control based on the context data.

If you could eliminate even a fraction of these conditional branches (a.k.a. "spaghetti" code), the software would be much easier to understand, test, and maintain, and the sheer number of convoluted execution paths through the code would drop radically, typically by orders of magnitude. Techniques based on state machines are capable of achieving exactly this—a dramatic reduction of the different paths through the code and simplification of the conditions tested at each branching point.

In this article series, I briefly introduce UML state machines that represent the current state of the art in the long evolution of these techniques. My intention is not to give a complete, formal discussion of UML state machines, which the official OMG specification covers comprehensively and with formality. Instead, my goal in this article series is to lay a foundation by establishing basic terminology, introducing basic notation, clarifying semantics, and giving some code examples. This series is restricted to only a subset of those state machine features that are arguably most fundamental. The emphasis is on the role of UML state machines in practical, everyday programming rather than mathematical abstractions.

The oversimplification of the Event-Action Paradigm
The currently dominating approach to structuring event-driven software is the ubiquitous "event-action" paradigm, in which events are directly mapped to the code that is supposed to be executed in response. The event-action paradigm is an important stepping stone for understanding state machines, so in this section I briefly describe how it works in practice.

I will use an example from the graphical user interface (GUI) domain, given that GUIs make exemplary event-driven systems. In the book Constructing the User Interface with Statecharts, Ian Horrocks discusses a simple GUI calculator application distributed in millions of copies as a sample program with Microsoft Visual Basic, in which he found a number of serious problems. As Horrocks notes, the point of this analysis is not to criticize this particular program but to identify the shortcomings of the general principles used in its construction.

When you launch the Visual Basic calculator (available from the companion web site at www.state-machine.com/psicc2/, in the directory ), you will certainly find out that most of the time it correctly adds, subtracts, multiplies, and divides (see Figure 2.1a). What's not to like? However, play with the program for a while longer and you can discover many corner cases in which the calculator provides misleading results, freezes, or crashes altogether.

Figure 2.1: Visual Basic calculator before the crash (a) and after a crash with a runtime error (b).

For example, the Visual Basic calculator often has problems with the "-" event; just try the following sequence of operations: 2, --, --, --, 2, =. The application crashes with a runtime error (see Figure 2.1b). This is because the same button (–) is used to negate a number and to enter the subtraction operator. The correct interpretation of the "-" button-click event, therefore, depends on the context, or mode, in which it occurs. Likewise, the CE (Cancel Entry) button occasionally works erroneously--try 2, x, CE, 3, =, and observe that CE had no effect, even though it appears to cancel the 2 entry from the display. Again, CE should behave differently when canceling an operand than canceling an operator. As it turns out, the application handles the CE event always the same way, regardless of the context. At this point, you probably have noticed an emerging pattern. The application is especially vulnerable to events that require different handling depending on the context.

This is not to say that the Visual Basic calculator does not attempt to handle the context. Quite the contrary, if you look at the calculator code (available from the companion web site at www.state-machine.com/psicc2/, in the directory ), you'll notice that managing the context is in fact the main concern of this application. The code is littered with a multitude of global variables and flags that serve only one purpose: handling the context. For example, indicates a decimal point has been entered, represents a pending operation, indicates the type of the last button press event, denotes the number of operands, and so on. With this representation, the context of the computation is represented ambiguously, so it is difficult to tell precisely in which mode the application is at any given time. Actually, the application has no notion of any single mode of operation but rather a bunch of tightly
coupled and overlapping conditions of operation determined by values of the global variables and flags.

**Listing 2.1** shows the conditional logic in which the event-handler procedure for the operator events (+, -, *, and /) attempts to determine whether the – (minus) button-click should be treated as negation or subtraction.

### Listing 1:

```
Listing 2.1
```

View image of author's code

The approach exemplified in **Listing 2.1** is a fertile ground for the "corner case" behavior (a.k.a. bugs) for at least three reasons:

- It always leads to convoluted conditional logic (a.k.a. "spaghetti" code).
- Each branching point requires evaluation of a complex expression.
- Switching between different modes requires modifying many variables, which can easily lead to inconsistencies.

Convoluted conditional expressions like the one shown in **Listing 2.1**, scattered throughout the code, are unnecessarily complex and expensive to evaluate at runtime. They are also notoriously difficult to get right, even by experienced programmers, as the bugs still lurking in the Visual Basic calculator attest. This approach is insidious because it appears to work fine initially, but doesn't scale up as the problem grows in complexity. Apparently, the calculator application (overall only seven event handlers and some 140 lines of Visual Basic code including comments) is just complex enough to be difficult to get right with this approach.

The faults just outlined are rooted in the oversimplification of the event-action paradigm. The Visual Basic calculator example makes it clear, I hope, that an event alone does not determine the actions to be executed in response to that event. The current context is at least equally important. The prevalent event-action paradigm, however, recognizes only the dependency on the event-type and leaves the handling of the context to largely ad hoc techniques that all too easily degenerate into spaghetti code.

### Basic state machine concepts

The event-action paradigm can be extended to explicitly include the dependency on the execution context. As it turns out, the behavior of most event-driven systems can be divided into a relatively small number of
chunks, where event responses within each individual chunk indeed depend only on the current event-type but no longer on the sequence of past events (the context). In other words, the event-action paradigm is still applied, but only locally within each individual chunk.

A common and straightforward way of modeling behavior based on this idea is through a finite state machine (FSM). In this formalism, "chunks of behavior" are called states, and change of behavior (i.e., change in response to any event) corresponds to change of state and is called a state transition. An FSM is an efficient way to specify constraints of the overall behavior of a system. Being in a state means that the system responds only to a subset of all allowed events, produces only a subset of possible responses, and changes state directly to only a subset of all possible states.

The concept of an FSM is important in programming because it makes the event handling explicitly dependent on both the event-type and on the execution context (state) of the system. When used correctly, a state machine becomes a powerful "spaghetti reducer" that drastically cuts down the number of execution paths through the code, simplifies the conditions tested at each branching point, and simplifies the transitions between different modes of execution.

**States**

A state captures the relevant aspects of the system's history very efficiently. For example, when you strike a key on a keyboard, the character code generated will be either an uppercase or a lowercase character, depending on whether the Caps Lock is active. Therefore, the keyboard's behavior can be divided into two chunks (states): the "default" state and the "caps_locked" state. (Most keyboards actually have an LED that indicates that the keyboard is in the "caps_locked" state.) The behavior of a keyboard depends only on certain aspects of its history, namely whether the Caps Lock key has been pressed, but not, for example, on how many and exactly which other keys have been pressed previously. A state can abstract away all possible (but irrelevant) event sequences and capture only the relevant ones.

To relate this concept to programming, this means that instead of recording the event history in a multitude of variables, flags, and convoluted logic, you rely mainly on just one state variable that can assume only a limited number of a priori determined values (e.g., two values in case of the keyboard). The value of the state variable crisply defines the current state of the system at any given time. The concept of state reduces the problem of identifying the execution context in the code to testing just the state variable instead of many variables, thus eliminating a lot of conditional logic. Actually, in all but the most basic state machine implementation techniques, such as the "nested-switch statement" technique, even the explicit testing of the state variable disappears from the code, which reduces the "spaghetti" further still. Moreover, switching between different states is vastly simplified as well, because you need to reassign just one state variable instead of changing multiple variables in a self-consistent manner.

**State diagrams**

FSMs have an expressive graphical representation in the form of state diagrams. These diagrams are directed graphs in which nodes denote states and connectors denote state transitions.

For example, **Figure 2.2** shows a UML state transition diagram corresponding to the computer keyboard state machine. In UML, states are represented as rounded rectangles labeled with state names. The transitions, represented as arrows, are labeled with the triggering events followed optionally by the list of executed actions. The initial transition originates from the solid circle and specifies the starting state when the system first begins. Every state diagram should have such a transition, which should not be labeled, since it is not triggered by an event. The initial transition can have associated actions.

**Figure 2.2 UML state diagram representing the computer keyboard state machine.**
**State diagrams versus flowcharts**

Newcomers to the state machine formalism often confuse state diagrams with flowcharts. For a long time, the UML specification wasn't helping in this respect because it used to lump activity graphs in the state machine package (the new UML 2 [1] has finally separated activity diagrams from state machines). Activity diagrams are essentially elaborate flowcharts.

**Figure 2.3** shows a comparison of a state diagram with a flowchart. A state machine (panel (a)) performs actions in response to explicit triggers. In contrast, the flowchart (panel (b)) does not need explicit triggers but rather transitions from node to node in its graph automatically upon completion of activities.

**Figure 2.3: Comparison of (a) state machine (statechart) with (b) activity diagram (flowchart).**

Graphically, compared to state diagrams, flowcharts reverse the sense of vertices and arcs. In a state diagram, the processing is associated with the arcs (transitions), whereas in a flowchart, it is associated with the vertices. A state machine is idle when it sits in a state waiting for an event to occur. A flowchart is busy executing activities when it sits in a node. **Figure 2.3** attempts to show that reversal of roles by aligning the arcs of the statecharts with the processing stages of the flowchart.

You can compare a flowchart to an assembly line in manufacturing because the flowchart describes the progression of some task from beginning to end (e.g., transforming source code input into object code output by a compiler). A state machine generally has no notion of such a progression. A computer keyboard, for example, is not in a more advanced stage when it is in the "caps_locked" state, compared to being in the "default" state; it simply reacts differently to events. A state in a state machine is an efficient way of specifying a particular behavior, rather than a stage of processing.

The distinction between state machines and flowcharts is especially important because these two concepts represent two diametrically opposed programming paradigms: event-driven programming (state machines) and transformational programming (flowcharts). You cannot devise effective state machines without constantly thinking about the available events. In contrast, events are only a secondary concern (if at all) for flowcharts.

**Extended state machines**
One possible interpretation of state for software systems is that each state represents one distinct set of valid values of the whole program memory. Even for simple programs with only a few elementary variables, this interpretation leads to an astronomical number of states. For example, a single 32-bit integer could contribute to over 4 billion different states. Clearly, this interpretation is not practical, so program variables are commonly dissociated from states. Rather, the complete condition of the system (called the extended state) is the combination of a qualitative aspect (the state) and the quantitative aspects (the extended state variables). In this interpretation, a change of variable does not always imply a change of the qualitative aspects of the system behavior and therefore does not lead to a change of state.\(^3\)

State machines supplemented with variables are called extended state machines. Extended state machines can apply the underlying formalism to much more complex problems than is practical without including extended state variables. For instance, suppose the behavior of the keyboard depends on the number of characters typed on it so far and that after, say, 1,000 keystrokes, the keyboard breaks down and enters the final state. To model this behavior in a state machine without memory, you would need to introduce 1,000 states (e.g., pressing a key in state stroke123 would lead to state stroke124, and so on), which is clearly an impractical proposition. Alternatively, you could construct an extended state machine with a key_count down-counter variable. The counter would be initialized to 1,000 and decremented by every keystroke without changing state. When the counter reached zero, the state machine would enter the final state.

**Figure 2.4:** Extended state machine of "cheap keyboard" with extended state variable key_count and various guard conditions.

The state diagram from **Figure 2.4** is an example of an extended state machine, in which the complete condition of the system (called the extended state) is the combination of a qualitative aspect--the "state"--and the quantitative aspects--the extended state variables (such as the down-counter \(\text{key\_count}\)). In extended state machines, a change of a variable does not always imply a change of the qualitative aspects of the system behavior and therefore does not always lead to a change of state.

The obvious advantage of extended state machines is flexibility. For example, extending the lifespan of the "cheap keyboard" from 1,000 to 10,000 keystrokes would not complicate the extended state machine at all. The only modification required would be changing the initialization value of the down-counter in the initial transition.

**Guard conditions**
This flexibility of extended state machines comes with a price, however, because of the complex coupling between the "qualitative" and the "quantitative" aspects of the extended state. The coupling occurs through the guard conditions attached to transitions, as shown in **Figure 2.4**.

Guard conditions (or simply guards) are Boolean expressions evaluated dynamically based on the value of extended state variables and event parameters (see the discussion of events and event parameters in the next section). Guard conditions affect the behavior of a state machine by enabling actions or transitions only when
they evaluate to TRUE and disabling them when they evaluate to FALSE. In the UML notation, guard conditions are shown in square brackets (e.g., \[\])

The need for guards is the immediate consequence of adding memory (extended state variables) to the state machine formalism. Used sparingly, extended state variables and guards make up an incredibly powerful mechanism that can immensely simplify designs. But don't let the fancy name ("guard") and the concise UML notation fool you. When you actually code an extended state machine, the guards become the same IFs and ELSEs that you wanted to eliminate by using the state machine in the first place. Too many of them, and you'll find yourself back in square one ("spaghetti"), where the guards effectively take over handling of all the relevant conditions in the system.

Indeed, abuse of extended state variables and guards is the primary mechanism of architectural decay in designs based on state machines. Usually, in the day-to-day battle, it seems very tempting, especially to programmers new to state machine formalism, to add yet another extended state variable and yet another guard condition (another if or an else) rather than to factor out the related behavior into a new qualitative aspect of the system—the state. From my experience in the trenches, the likelihood of such an architectural decay is directly proportional to the overhead (actual or perceived) involved in adding or removing states. (That's why I don't particularly like the popular state-table technique of implementing state machines, because adding a new state requires adding and initializing a whole new column in the table.)

One of the main challenges in becoming an effective state machine designer is to develop a sense for which parts of the behavior should be captured as the "qualitative" aspects (the "state") and which elements are better left as the "quantitative" aspects (extended state variables). In general, you should actively look for opportunities to capture the event history (what happened) as the "state" of the system, instead of storing this information in extended state variables.

For example, the Visual Basic calculator uses an extended state variable \[entering_the_fractional_part_of_a_number,\] to remember that the user entered the decimal point to avoid entering multiple decimal points in the same number. However, a better solution is to observe that entering a decimal point really leads to a distinct state \["entering_the_fractional_part_of_a_number,"\] in which the calculator ignores decimal points.

This solution is superior for a number of reasons. The lesser reason is that it eliminates one extended state variable and the need to initialize and test it. The more important reason is that the state-based solution is more robust because the context information is used very locally (only in this particular state) and is discarded as soon as it becomes irrelevant. Once the number is correctly entered, it doesn't really matter for the subsequent operation of the calculator whether that number had a decimal point.

The state machine moves on to another state and automatically "forgets" the previous context. The extended state variable, on the other hand, "lays around" well past the time the information becomes irrelevant (and perhaps outdated!). Worse, you must not forget to reset before entering another number or the flag will incorrectly indicate that indeed the user once entered the decimal point, but perhaps this happened in the context of the previous number.

Capturing behavior as the quantitative "state" has its disadvantages and limitations, too. First, the state and transition topology in a state machine must be static and fixed at compile time, which can be too limiting and inflexible. Sure, you can easily devise "state machines" that would modify themselves at runtime (this is what often actually happens when you try to recode "spaghetti" as a state machine). However, this is like writing self-modifying code, which indeed was done in the early days of programming but was quickly dismissed as a generally bad idea. Consequently, "state" can capture only static aspects of the behavior that are known a priori and are unlikely to change in the future.
For example, it's fine to capture the entry of a decimal point in the calculator as a separate state "entering_the_fractional_part_of_a_number," because a number can have only one fractional part, which is both known a priori and is not likely to change in the future. However, implementing the "cheap keyboard" without extended state variables and guard conditions would be practically impossible. This example points to the main weakness of the quantitative "state," which simply cannot store too much information (such as the wide range of keystroke counts). Extended state variables and guards are thus a mechanism for adding extra runtime flexibility to state machines.

**Events**

In the most general terms, an event is an occurrence in time and space that has significance to the system. Strictly speaking, in the UML specification the term event refers to the type of occurrence rather than to any concrete instance of that occurrence [OMG 07]. For example, Keystroke is an event for the keyboard, but each press of a key is not an event but a concrete instance of the Keystroke event. Another event of interest for the keyboard might be Power-on, but turning the power on tomorrow at 10:05:36 will be just an instance of the Power-on event.

An event can have associated parameters, allowing the event instance to convey not only the occurrence of some interesting incident but also quantitative information regarding that occurrence. For example, the Keystroke event generated by pressing a key on a computer keyboard has associated parameters that convey the character scan code as well as the status of the Shift, Ctrl, and Alt keys.

An event instance outlives the instantaneous occurrence that generated it and might convey this occurrence to one or more state machines. Once generated, the event instance goes through a processing life cycle that can consist of up to three stages. First, the event instance is received when it is accepted and waiting for processing (e.g., it is placed on the event queue). Later, the event instance is dispatched to the state machine, at which point it becomes the current event. Finally, it is consumed when the state machine finishes processing the event instance. A consumed event instance is no longer available for processing.

**Actions and transitions**

When an event instance is dispatched, the state machine responds by performing actions, such as changing a variable, performing I/O, invoking a function, generating another event instance, or changing to another state. Any parameter values associated with the current event are available to all actions directly caused by that event.

Switching from one state to another is called state transition, and the event that causes it is called the triggering event, or simply the trigger. In the keyboard example, if the keyboard is in the "default" state when the CapsLock key is pressed, the keyboard will enter the "caps_locked" state. However, if the keyboard is already in the "caps_locked" state, pressing CapsLock will cause a different transition--from the "caps_locked" to the "default" state. In both cases, pressing CapsLock is the triggering event.

In extended state machines, a transition can have a guard, which means that the transition can "fire" only if the guard evaluates to TRUE. A state can have many transitions in response to the same trigger, as long as they have nonoverlapping guards; however, this situation could create problems in the sequence of evaluation of the guards when the common trigger occurs. The UML specification intentionally does not stipulate any particular order; rather, UML puts the burden on the designer to devise guards in such a way that the order of their evaluation does not matter. Practically, this means that guard expressions should have no side effects, at least none that would alter evaluation of other guards having the same trigger.

**Run-to-completion execution model**

All state machine formalisms, including UML statecharts, universally assume that a state machine completes processing of each event before it can start processing the next event. This model of execution is called run to
completion, or RTC.

In the RTC model, the system processes events in discrete, indivisible RTC steps. New incoming events cannot interrupt the processing of the current event and must be stored (typically in an event queue) until the state machine becomes idle again. These semantics completely avoid any internal concurrency issues within a single state machine. The RTC model also gets around the conceptual problem of processing actions associated with transitions, where the state machine is not in a well-defined state (is between two states) for the duration of the action. During event processing, the system is unresponsive (unobservable), so the ill-defined state during that time has no practical significance.

Note, however, that RTC does not mean that a state machine has to monopolize the CPU until the RTC step is complete. The preemption restriction only applies to the task context of the state machine that is already busy processing events. In a multitasking environment, other tasks (not related to the task context of the busy state machine) can be running, possibly preempting the currently executing state machine. As long as other state machines do not share variables or other resources with each other, there are no concurrency hazards.

The key advantage of RTC processing is simplicity. Its biggest disadvantage is that the responsiveness of a state machine is determined by its longest RTC step. Achieving short RTC steps can often significantly complicate real-time designs.

**Next in Part 2: UML Extensions to the traditional FSM Formalism**

To read Part 2, go to "UML extensions to the traditional FSM formalism"

To read Part 3, go to "Designing a UML state machine"

**Miro Samek, Ph.D.,** is president of Quantum Leaps, LLC. He can be contacted at miro@quantum-leaps.com. With Michael Barr, he is teaching the course "Event Driven Programming (ESC-447) at the Embedded Systems Conference, Silicon Valley, March 30 – April 2, 2009 in San Jose, Ca.

ESC course ESC-477.

Endnotes:


**Editor's Note:** This series of articles is based on material used with permission from *Practical UML State Charts In C/C++, Second Edition*, by Miro Samek, which was published by Elsevier/Newnes Publishing, which holds all copyrights. It can be purchased online at [www.elsevier.com](http://www.elsevier.com)

Many embedded software developers believe that the customary superloop (main+ISRs) and the traditional preemptive RTOS are the only choices for the embedded software architecture. This book offers an alternative based on event-driven programming and modern hierarchical state machines (UML statecharts). The book bridges the gap between high-level abstract concepts of the Unified Modeling Language (UML) and the actual programming aspects of UML statecharts. The book describes the design and implementation of a lightweight, open source, event-driven framework, called QP that enables direct manual coding of UML statecharts and concurrent event-driven applications in C or C++ without big tools.)