A crash course in UML state machines: Part 3

Miro Samek

3/11/2009 12:00 AM EDT

In this three-part series, Miro Samek introduces you to the basics of UML state machines basic notation, clarifying semantics, and provides some code examples and explains the advantages of UML state machines have in practical, everyday programming rather than mathematical abstractions. Part 3: Designing a UML State Machine.

To read Part 1, go to "The over simplification of the event-action paradigm"

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

Designing a UML state machine, as any design, is not a strict science. Typically it is an iterative and incremental process: You design a little, code a little, test a little, and so on. In that manner you may converge at a correct design in many different ways, and typically also, more than one correct HSM design satisfies a given problem specification.

To focus the discussion, here I walk you through a design of an UML state machine that implements correctly the behavior of a simple calculator similar to the Visual Basic calculator used at the beginning of this series. Obviously, the presented solution is just one of the many possible.

Figure 2.13: A simple electronic calculator used as a model for the statechart example.

Problem specification
The calculator (see Figure 2.13) operates broadly as follows: a user enters an operand, then an operator, then another operand, and finally clicks the equals button to get a result. From the programming perspective, this means that the calculator needs to parse numerical expressions, defined formally by the following BNF grammar:
The problem is not only to correctly parse numerical expressions, but also to do it interactively ("on the fly"). The user can provide any symbol at any time, not necessarily only the symbols allowed by the grammar in the current context. It is up to the application to ignore such symbols. (This particular application ignores invalid inputs.

Often an even better approach is to actively prevent generation of the invalid inputs in the first place by disabling invalid options, for example.) In addition, the application must handle inputs not related to parsing expressions, for example Cancel (C) or Cancel Entry (CE). All this adds up to a nontrivial problem, which is difficult to tackle with the traditional event-action paradigm (see part 1 of this article series) or even with the traditional (nonhierarchical) FSM.

**High-level design**

Figure 2.14 shows first steps in elaborating the calculator statechart. In the very first step (panel (a)), the state machine attempts to realize the primary function of the system (the primary use case), which is to compute expressions: \textit{operand1 operator operand2 equals...} The state machine starts in the "operand1" state, whose function is to ensure that the user can only enter a valid operand. This state obviously needs some internal submachine to accomplish this goal, but we ignore it for now.

The criterion for transitioning out of "operand1" is entering an operator (+, –, *, or /). The statechart then enters the "opEntered" state, in which the calculator waits for the second operand. When the user clicks a digit (0 .. 9) or a decimal point, the state machine transitions to the "operand2" state, which is similar to "operand1." Finally, the user clicks '=' at which point the calculator computes and displays the result. It then transitions back to the "operand1" state to get ready for another computation.

Figure 2.14: The first two steps in elaborating the calculator statechart

![Figure 2.14](http://www.eetimes.com/General/DisplayPrintViewContent?contentItemI...)

The simple state model from Figure 2.14a has a major problem, however. When the user clicks '=' in the last step, the state machine cannot transition directly to "operand1" because this would erase the result from the display (to get ready for the first operand). We need another state "result" in which the calculator pauses to display the result (Figure 2.14b). Three things can happen in the "result" state: (1) the user may click an operator button to use the result as the first operand of a new computation (see the recursive production in line 2 of the calculator grammar), (2) the user may click Cancel (C) to start a completely new computation, or (3) the user may enter a number or a decimal point to start entering the first operand.

**TIP:** Figure 2.14b illustrates a trick worth remembering: the consolidation of signals PLUS, MINUS, MULTIPLY, and DIVIDE into a higher-level signal OPER (operand). This transformation avoids repetition of the same group of triggers on two transitions (from "operand1" to "opEntered" and from "result" to "opEntered"). Although most events are generated externally to the statechart, in many situations it is still possible to perform simple transformations before dispatching them (e.g., a transformation of raw button presses into the calculator events). Such transformations often simplify designs more than the trickiest state and transition topologies.
Scavenging for reuse
The state machine from Figure 2.14b accepts the C (Cancel) command only in the result state. However, the user expects to be able to cancel and start over at any time. Similarly, the user expects to be able to turn the calculator off at any time. Statechart in Figure 2.15a adds these features in a naïve way. A better solution is to factor out the common transition into a higher-level state named "on" and let all substates reuse the C (Cancel) and OFF transitions through behavioral inheritance, as shown in Figure 2.15b.

Figure 2.15: Applying state nesting to factorize out the common Cancel transition (C).

Elaborating composite states
The states "operand1" and "operand2" need submachines to parse floating-point numbers. Figure 2.16 refers to both these states simultaneously as "operandX" state.

Figure 2.16: Internal submachine of states "operand1" and "operand2."

These submachines consist of three substates. The "zero" substate is entered when the user clicks 0. Its function is to ignore additional zeros that the user may try to enter (so that the calculator displays only one 0). Note my notation for explicitly ignoring an event. I use the internal transition (DIGIT_0 in this case) followed by an explicitly empty list of functions (a semicolon in C).

The function of the "int" substate is to parse integer part of a number. This state is entered either from outside or from the "zero" peer substate (when the user clicks 1 through 9). Finally, the substate "frac" parses the fractional part of the number. It is entered from either outside or both peer substates when the user clicks a decimal point ('). Again, note that the "frac" substate explicitly ignores the decimal point POINT event, so that the user cannot enter multiple decimal points in the fractional part of a number.

Refining the behavior
The last step brings the calculator statechart to the point at which it can actually compute expressions. However, it can handle only positive numbers. In the next step, I will add handling of negative numbers. This turns out to be perhaps the toughest problem in this design because the same button, '-' (minus), represents in some contexts the binary operator of subtraction and sometimes the unary operator of negation.
There are only two possible contexts in which ‘−’ can unambiguously represent the negation rather than the subtraction: (1) in the "opEntered" state (as in the expression: $2 * 2 =\), and (2) at the beginning of a new computation (as in the expression: $2*2=\).

**Figure 2.17: Two cases of handling negative numbers.**

The solution to the first case (shown in **Figure 2.17a**) is simpler. We need one more state "negated2," which is entered when the operator is MINUS (note the use of the guard). Upon entry, this state sets up the display to show ‘−0’ and subsequently does not clear the display when transitioning to the "operand2" state. This is a different behavior from "opEntered" because in this latter state the display must be cleared to prepare for entering of the second operand.

The second case in which ‘−’ represents the negation is trickier because the specification "beginning of new computation" is much more subtle. Here it indicates the situation just after launching the application or after the user clicks Cancel but not when the calculator displays the result from the previous computation. **Figure 2.17b** shows the solution. A new state "begin" is created to capture the behavior specific to the "beginning of new computation" (note the initial transition pointing now to "begin" rather than to "operand1"). The rest of the solution is analogous as in the first case, except now the state "begin" plays the role of "opEntered."

**Final touches**
The calculator is almost ready now. The final touches (which I leave as an exercise) include adding Cancel-Entry transitions in appropriate contexts and adding an "error" state to capture overflows and division by zero. **Figure 2.18** shows the final calculator state diagram.

**Figure 2.18: The final calculator statechart.**
Executing the example code

The actual C and C++ implementations of the calculator hierarchical state machine from Figure 2.18 are available from the Website accompanying the book on which this series is based - "Practical UML Statecharts in C/C++" at www.state-machine.com/psicc2/. The calculator example in C is found in the directory <qp>\qpc\examples\80x86\dos\tcpp101\l\calc\ and the C++ version is found in the directory <qp>\qcpp \examples\80x86\dos\tcpp101\l\calc\. The source code consists of the following files:

- `calc.h` contains the declaration of signals, events, and the Calc state machine structure.
- `calc.cpp` (in the C++ version) contains the implementation of the Calc state machine.
- `bsp.h` contains the board support package interface.
- `bsp.cpp` (in the C++ version) contains the implementation of the board-specific functions.
- `loop.h` (in the C++ version) contains the `loop()` function and the event loop.
- `main.cpp` is the Turbo C++ project file for building the application.

As always, the code I provide is executable and I encourage you to try it out. You can run the example on any Windows PC by either double-clicking on the executable located in the directory <qp>\qpc\examples\80x86 \dos\tcpp101\l\calc\dbg\CALC.EXE.
The calculator example is interactive and you can perform computations with it. You use the keyboard to send keypress events to the application; the state of the calculator display is shown at the command prompt. The calculator recognizes keys: 0, 1, . . ., 9, ., +, −, *, /, =, C, and E (cancel entry, CE). The Esc key terminates the application. All other keys are ignored.

**Figure 2.19: The calculator HSM running in a Windows console.**

![Calculator HSM running in a Windows console](image)

**Figure 2.19** shows a screen shot in which you can see how the calculator handles the expression 2, −, −, −, 2, = that has crashed the Visual Basic calculator discussed in part 1 of this article series. I'd like to challenge you to crash the state machine-based calculator. The calculator starts with displaying zero aligned at the right edge of the display [0]. To the right of the display, you can see the key sent to the calculator. For example, the first key is 2. The key event is followed by the sequence of actions that the calculator HSM performs in response to the key event. I recommend that you correlate this output with the calculator state diagram from **Figure 2.18**.

**Summary**

The main challenge in programming event-driven systems is to identify the appropriate actions to execute in response to a given event. In general, the actions are determined by two factors: by the nature of the event and by the current context (i.e., by the sequence of past events in which the system was involved). The traditional techniques, such as the event-action paradigm, neglect the context and result in code riddled with a disproportionate amount of convoluted conditional logic that programmers call "spaghetti" code.

Techniques based on state machines are capable of achieving a dramatic reduction of the different paths through the code and simplification of the conditions tested at each branching point. A state machine makes the event handling explicitly dependent on both the nature of the event and on the context (state) of the system. States are "chunks of behavior," whereas the event-action paradigm is applied locally within each state. The concept of state is a very useful abstraction of system history, capable of capturing only relevant sequences of stimuli (and ignoring all irrelevant ones). In extended state machines (state machines with "memory"), state corresponds to qualitative aspects of system behavior, whereas extended state variables (program memory) correspond to the quantitative aspects.

An event is a type of instantaneous occurrence that can cause a state machine to perform actions. Events can have parameters, which convey the quantitative information regarding that occurrence. Upon reception of an event instance, a state machine responds by performing actions (executing code). The response might include changing state, which is called a state transition. Classical FSMs have two complementary interpretations of actions and transitions. In Mealy automata, actions are associated with transitions, whereas in Moore automata, actions are associated with states.

State machine formalisms universally assume the run-to-completion (RTC) execution model. In this model, all
actions triggered by an event instance must complete before the next event instance can be dispatched to the state machine, meaning that the state machine executes uninterruptible steps (RTC steps) and starts processing each event in a stable state configuration.

UML state machines are an advanced formalism for specifying state machines, which extends the traditional automata theory in several ways. The UML state machine formalism is a variant of extended state machines with characteristics of both Mealy and Moore automata. UML state machines include notations of nested hierarchical states and orthogonal regions as well as extending the notation of actions.

The most important innovation of UML state machines over classical FSMs is the introduction of hierarchically nested states. The value of state nesting lies in avoiding repetitions, which are inevitable in the traditional "flat" FSM formalism. The semantics of state nesting allow substates to define only the differences in behavior from the superstates, thus promoting sharing and reuse of behavior. The relation between a substate and its superstate has all the characteristics of inheritance and is called behavioral inheritance in this book. Behavioral inheritance is as fundamental as class inheritance and allows building whole hierarchies of states, which correspond to class taxonomies in OOP. Properly designed state hierarchies comply with the Liskov Substitution Principle (LSP) extended for states.

UML state machines support state entry and exit actions for states, which provide the means for guaranteed initialization and cleanup, very much as constructors and destructors do for classes. Entry actions are always executed starting with the outermost state, which is analogous to class constructors executed from the most general class. The exit actions, similar to destructors, are always executed in exact reverse order.

Entry and exit actions combined with state nesting complicate transition sequence. The precise semantics of state transitions can be confusing. The exhaustive example discussed in this chapter can precisely "answer" virtually all your questions regarding the semantics of state machine execution.

Statecharts were first invented as a visual formalism; therefore, they are heavily biased toward graphical representation. However, it is important to distinguish the underlying concept of the HSM from the graphical notation. It is also important to distinguish between statecharts and flowcharts.

Designing effective UML state machines is not trivial and, as with most designs, typically requires incremental, iterative process. Reuse does not come automatically, but you must actively look for it.

To read Part 1, go to "The over simplification of the event-action paradigm"

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

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, March30 – April 2, 2009 in San Jose, Ca.

ESC course ESC-477.

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

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.)