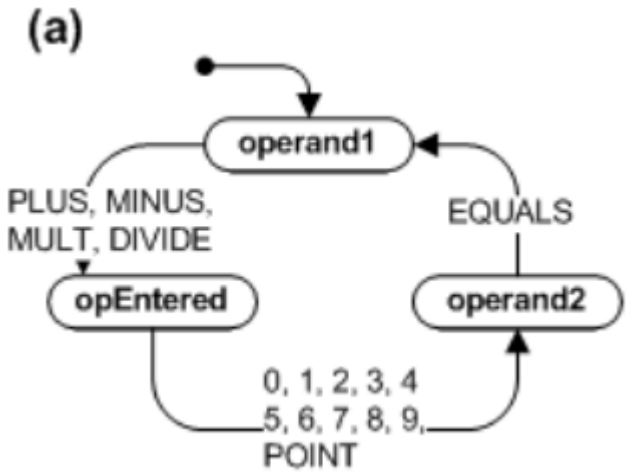
State machines exercise answers

1. **Design a UML state machine for a simple calculator**

Start by designing a simple state machine which handles the sequence of a calculation consisting of an operand, followed by an operator, followed by an operand.

Figure 3-2 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: 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.

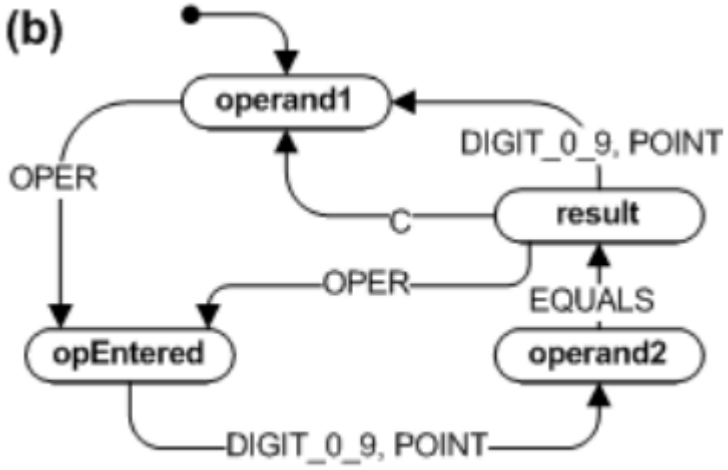


2. **Modify your state machine so that once a result has been found the user can either:**

Click an operator button to use the result as the first operand of a new calculation

Click Cancel © to start a new calculation

Enter a number to start a new calculation



The simple state model from Figure 3-2(a) 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 3-2(b)). 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 3-2(b) 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.

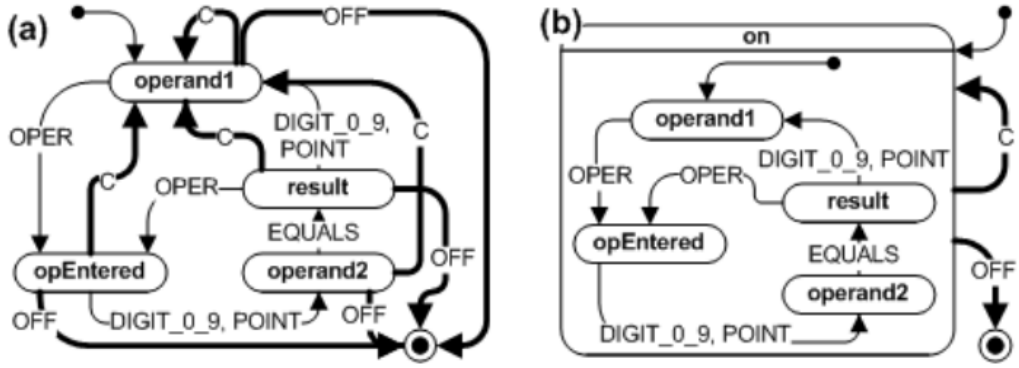
3. Expand your state machine to allow for:

The user can cancel and start over at any time ©

The user can turn the calculator off at any time

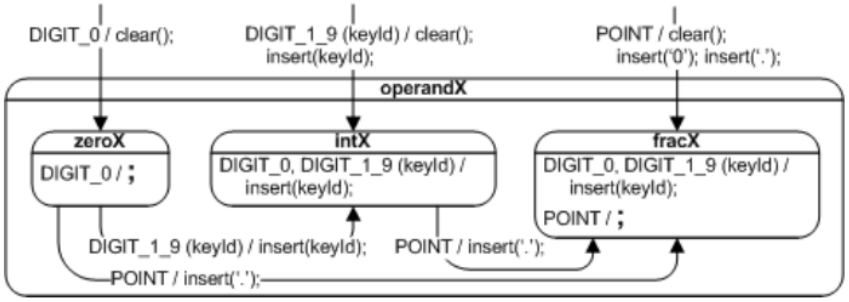
The state machine from Figure 3-2(b) accepts the C (Cancel) command only in the result state. However, the user expects to be able to cancel and start over at any time. Similarly, the user expects to be able to turn the calculator off at any time. Statechart in Figure 3-3(a) adds these features in a 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 3-3(b).

Figure 3-3: Applying state nesting to factorize out the common Cancel transition (C).



4. Create a sub-machine for the operands to handle floating point numbers

Figure 3-4: 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.