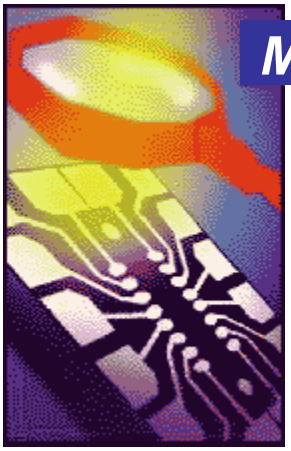

Miro Samek

C/C++ Users Journal
Advanced Solutions for Professional Developers

Déjà Vu

It's Déjà Vu all over again as Miro reveals the object-oriented nature of the behavioral abstractions that pervade embedded programs.

In the last installment of this column ("Who Moved my State?," *CUJ*, April 2003), I touched on the benefits of using state machines in programming reactive (event-driven) systems. However, while the traditional Finite State Machines (FSMs) are an excellent tool for tackling smaller problems, it's also generally known that they tend to become unmanageable even for moderately involved systems. Due to the phenomenon known as "state explosion," the complexity of a traditional FSM tends to grow much faster than the complexity of the reactive system it describes. This happens because the traditional state machine formalism inflicts repetitions. For example, if you try to represent the behavior of just about any nontrivial system (such as the calculator I described in the last column) with a traditional FSM, you'll immediately notice that many events (e.g., the Clear event) are handled identically in many states. A conventional FSM, however, has no means of capturing such a commonality and requires repeating the same actions and transitions in many states. What's missing in the traditional state machines is the mechanism for factoring out the common behavior in order to share it across many states.

The formalism of statecharts, invented by David Harel in the 1980s (see David Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 8, 1987, available online from <http://www.wisdom.weixmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf>), addresses exactly this shortcoming of the conventional FSMs. Statecharts provide a very efficient way of sharing behavior, so that the complexity of a statechart no longer explodes but tends to faithfully represent the complexity of the reactive system it describes. Obviously, formalism like this is a godsend to embedded systems programmers (or any programmers working on reactive systems), because it makes the state machine approach truly applicable to real-life problems.

Reuse of Behavior in Reactive Systems

All reactive systems seem to reuse behavior in a similar way. For example, the characteristic look-and-feel of all Graphical User Interfaces (GUIs) results from the same pattern, which Charles Petzold calls the "Ultimate Hook" (see Charles Petzold, *Programming Windows 95, The Definite Developer's Guide to the Windows 95 API*, Microsoft Press, 1996). The pattern is brilliantly simple. A GUI system dispatches every event first to the application (e.g., Windows calls a specific function inside the application, passing the event as an argument). If not handled by the application, the event flows back to the system. This establishes a hierarchical order of event processing. The application, which is conceptually at a lower level of the hierarchy, has the first shot at every event; thus, the application can customize every aspect of its behavior. At the same time, all unhandled events flow back to the higher level (i.e., to the GUI system), where they are processed according to the standard look-and-feel. This is an example of *programming-by-difference* because the application programmer needs to code only the differences from the standard system behavior.

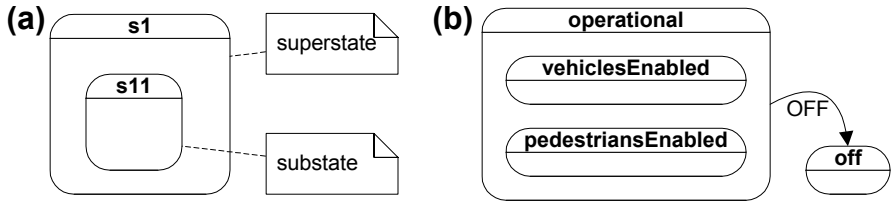

Figure 1 (a) UML notation for hierarchically nested states; (b) UML state diagram of a PELICAN (PEdestrian LIght CONtrolled) crossing, in which states `vehiclesEnabled` and `pedestriansEnabled` share the common transition `OFF` to the `off` state.

Harel statecharts bring the "Ultimate Hook" pattern to the logical conclusion by combining it with the state machine formalism. The most important innovation of statecharts over the classical FSMs is the introduction of hierarchically nested states (that's why state-charts are also called hierarchical state machines). The semantics associated with state nesting (shown in Figure 1(a)) are as follows: If a system is in the nested state `s11` (called substate), it also (implicitly) is in the surrounding state `s1` (the superstate). This state machine will attempt to handle any event in the context of state `s11` (which is at the lower level of the hierarchy). However, if state `s11` does not prescribe how to handle the event, the event is not quietly discarded (as in a traditional "flat" state machine); rather, it is automatically handled at the higher level context of state `s1`. This is what is meant by the system being in state `s11` as well as `s1`. Of course, state nesting is not limited to one level only, and the simple rule of event processing applies recursively to any level of nesting.

As you can see, the semantics of hierarchical state decomposition are designed to facilitate sharing of behavior through the direct support for the "Ultimate Hook" pattern. The substates (nested states) need only define the differences from the superstates (surrounding states). A substate can easily reuse the common behavior from its superstate(s) by simply ignoring commonly handled events, which are then automatically handled by higher level states. In this manner, the substates can share all aspects of behavior with their superstates. For example, in a state model of a PELICAN (PEdestrian LIght CONtrolled) crossing shown in Figure 1(b), states `vehiclesEnabled` and `pedestriansEnabled` share a common transition OFF to the `off` state, defined in their common superstate `operational`.

Behavioral Inheritance

The fundamental character of state nesting in Hierarchical State Machines (HSMs) comes from combining hierarchy with programming-by-difference, which is otherwise known in software as *inheritance*. In Object-Oriented Programming (OOP), the concept of class inheritance lets you define a new kind of class rapidly in terms of an old one by specifying only how the new class differs from the old class. State nesting introduces another fundamental type of inheritance, called *behavioral inheritance* (see M. Samek and Paul Y. Montgomery "State-Oriented Programming," *Embedded Systems Programming*, August 2000, available online from <http://www.embedded.com/2000/0008/0008feat1.htm>). Behavioral inheritance lets you define a new state as a specific kind of another state, by specifying only the differences from the existing state rather than defining the whole new state from scratch.

As class inheritance allows subclasses to *adapt* to new environments, behavioral inheritance allows substates to *mutate* by adding new behavior or by overriding existing behavior. Nested states can introduce new behavior by adding new state transitions or reactions (also known as internal transitions) for events that are not recognized by superstates. This corresponds to adding new methods to a subclass. Alternatively, a substate may also process the same events as the superstates but will do it in a different way. In this manner, the substate can override the inherited behavior, which corresponds to a subclass overriding a (virtual in C++) method defined by its parents. In both cases, overriding the inherited behavior leads to polymorphism.

Liskov Substitution Principle for States

Because behavioral inheritance is just a specific kind of inheritance, the universal law of generalization — the Liskov Substitution Principle (LSP) — should be applicable to state hierarchies as well as class taxonomies. In its traditional formulation for classes, LSP requires that every instance of a subclass must continue to act as though it were also an instance of the superclass. From the programming standpoint, LSP means that any code designed to work with the instance of the superclass should continue to work correctly if an instance of the subclass is used instead.

The LSP extends naturally for hierarchical states and requires in this case that every substate continue to behave as though it were also the superstate. For example, all substates nested inside the `vehiclesEnabled` state of the PELICAN crossing (such as `vehiclesGreen` or `vehiclesYellow` , shown in Figure 2) should share the same basic characteristics of the `vehiclesEnabled` state. In particular, being in the `vehiclesEnabled` state means that the vehicles are allowed (by a green or yellow light) and simultaneously the pedestrians are not allowed (by the DON'T WALK signal). To be compliant with the LSP, none of the substates of `vehiclesEnabled` should disable the vehicles or enable the pedestrians. In particular, disabling vehicles (by switching the red light), or enabling the pedestrians (by displaying the WALK signal) in any of the nested states `vehiclesGreen` or `vehiclesYellow` would be inconsistent with being in the superstate `vehiclesEnabled` and would be a violation of the LSP (it will also be a safety hazard in this case).

Compliance with the LSP allows you to build better (correct) state hierarchies and make efficient use of abstraction. For example, in an LSP-compliant state hierarchy, you can safely "zoom out" and work at the higher level of the `vehiclesEnabled` state (thus abstracting away the specifics of `vehiclesGreen` and `vehiclesYellow`). As long as all the substates are consistent with their superstate, such abstraction is meaningful. On the other hand, if the substates violate basic assumptions of being in the superstate, zooming out and ignoring specifics of the substates will be incorrect.

Guaranteed Initialization and Cleanup

Every state in a UML statechart can have optional entry actions, which are executed upon entry to a state, as well as optional exit actions, which are executed upon exit from a state. Entry and exit actions are associated with states, not transitions. Regardless of how a state is entered or exited, all of its entry and exit actions will be executed.

The value of entry and exit actions is often underestimated. However, entry and exit actions are as important in HSMs as class constructors and destructors are in OOP because they provide for guaranteed initialization and cleanup. For example, consider the `vehiclesEnabled` state from Figure 1(b), which corresponds to the traffic lights configuration that enables vehicles and disables pedestrians. This state has a very important safety-critical requirement: always disable the pedestrians (by turning on the DON'T WALK signal) when vehicles are enabled. Of course, you could arrange for such a behavior by adding an appropriate action (switching the DON'T WALK signal) to every transition path leading into the `vehiclesEnabled` state. However, such a solution would potentially cause the repetition of this action on many transitions. More importantly, such an approach is error-prone in view of changes to the state machine. For instance, a programmer upgrading a PELICAN crossing to a PUFFIN (Pedestrian User Friendly INtelligent) crossing might simply forget to turn on the DON'T WALK signal on all transitions into the `vehiclesEnabled` state or any of its substates. Entry and exit actions allow you to implement the desired behavior in a safer, simpler, and more intuitive way. You could specify turning on the DON'T WALK signal upon the entry to `vehiclesEnabled` . This solution is superior because it avoids potential repetitions of this action on transitions and eliminates the basic safety hazard of leaving the WALK signal turned on while vehicles may be allowed into the crossing. The semantics of entry actions guarantees that, regardless of the transition path, the DON'T WALK signal will be turned on when the traffic controller is in the `vehiclesEnabled` state.

NOTE: An equally correct alternative design is to switch the DON'T WALK signal in the exit action from `pedestriansEnabled` and to switch the red light for vehicles in the exit action from `vehiclesEnabled`.

Obviously, you can also use entry and exit actions in the classical (nonhierarchical) FSMs. In fact, the lack of hierarchy in this case makes the implementation of this feature almost trivial. For instance, one way of implementing entry and exit actions is to dispatch reserved signals (e.g., `ENTRY_SIG` and `EXIT_SIG`) to the state machine. The `FsmTran_()` method (see the April installment of this column) could dispatch the `EXIT_SIG` signal to the current state (transition source) and then dispatch the `ENTRY_SIG` signal to the tar-get. Such an implementation of the `FsmTran_()` method might look as follows (the complete code in C and C++ is available from the code archive at <www.cuj.com/code>):

```
static Event const entryEvt = { ENTRY_SIG };
static Event const exitEvt = { EXIT_SIG };

void FsmTran_(Fsm *me, State target){
   FsmDispatch(me, &exitEvt);     /* exit the source */
   me->state__ = target;     /* change current state */
   FsmDispatch(me, &entryEvt);   /* enter the target */
}
```

However, entry and exit actions are particularly important and powerful in HSMs because they often determine the identity of hierarchical states. For example, the identity of the `vehiclesEnabled` state is determined by the fact that the vehicles are enabled and pedestrians disabled. These conditions must be established before entering any substate of `vehiclesEnabled` because entry actions to a substate, such as `vehiclesGreen` (see Figure 2), rely on proper initialization of the `vehiclesEnabled` superstate and perform only the differences from this initialization. Consequently, the order of execution of entry actions must always proceed from the outermost state to the innermost state. Not surprisingly, this order is analogous to the order in which class constructors are invoked. Construction of a class always starts at the top of the class hierarchy and follows through all inheritance levels down to the class being instantiated. The execution of exit actions, which corresponds to destructor invocation, proceeds in the exact reverse order, starting from the innermost state (corresponding to the most derived class).

HINT: Try to make your state machine as much a Moore automaton as possible. (Moore automata associate actions with states rather than transitions.) That way you achieve a safer design (in view of future modifications) and your states will have better-defined identity.

Zooming in...

Figure 2 shows a complete UML state diagram of the PELICAN crossing (a "zoomed in" version of the diagram from Figure 1(b)). I believe that this HSM represents quite faithfully the behavior of the pedestrian crossing in front of the midtown shopping center on Middlefield Road in Palo Alto. I have tested the crossing several times (the drivers sure loved me for it) and have determined that it operates as follows. Nominally, vehicles are enabled and pedestrians disabled. To activate the traffic light switch, a pedestrian must push a button (let's call this event `PEDESTRIAN_WAITING`). In response, the vehicles get the yellow light. After a few seconds, vehicles get a red light and pedestrians get the WALK signal, which shortly thereafter changes to a flashing DON'T WALK signal. When the DON'T WALK signal stops flashing, vehicles get the green light. After this cycle, the traffic lights don't respond to the `PEDESTRIAN_WAITING` event immediately, although the button "remembers" that it has been pushed. The traffic light controller always gives the vehicles a minimum of several seconds of green light before repeating the cycle.

Perhaps the most interesting element of the state model from Figure 2 (except the aforementioned use of entry actions to avoid basic safety hazards) is the way it guarantees the minimal green light time for the vehicles. The HSM models it with two states, `vehiclesGreen` and `vehiclesGreenInt`,as well as with the `isPedes-`

trianWaiting flag. The state vehiclesGreen corresponds to the uninterruptible green light for the vehicles. The occurrence of the PEDESTRIAN_WAITING event in this state doesn't trigger any transition but merely sets the isPedestrianWaiting flag to "remember" that the button was pressed. The only criterion for exiting vehiclesGreen is the occurrence of the TIMEOUT event, which triggers a transition either to vehiclesYellow (if the isPedestrianWaiting flag is set) or to the vehiclesGreenInt state, the latter corresponding to the interruptible green light for vehicles.

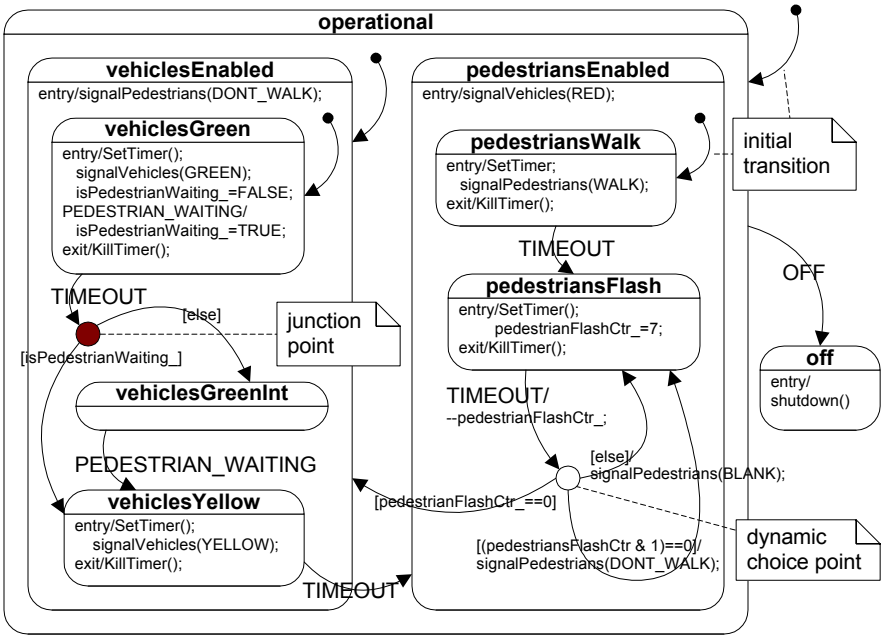

Figure 2 Complete UML state diagram of the PELICAN crossing.

Another interesting aspect of this state model is the generation of the TIMEOUT events by means of only one timer object. (A timer is a facility that dispatches an event to the state machine after a preprogrammed time interval.) Typically, timers are scarce system resources that need to be allocated (here indicated by the SetTimer() action) and recycled (by the KillTimer() action). Note how the state machine enables using only one TIMEOUT event (different states handle the same event differently), and how entry and exit actions help to initialize and clean up the timer. In particular, please note that the timer is never leaked, even in the case of the always enabled OFF transition inherited from the operational superstate.

A Pie-in-the-Sky?

If I left this quick introduction to HSMs only at the level of the state diagram from Figure 2, you would probably walk away with the impression that the whole approach is just a pie-in-the-sky. The diagram in Figure 2 might look clever, perhaps, but how does it lead to better code?

Contrary to a widespread misconception, you don't need sophisticated CASE tools to manually translate UML state diagrams to efficient and highly maintainable C or C++. To prove the point, I have placed the full implementation of the PELICAN crossing both in C and C++ into the CUJ code archive at <www.cuj.com/-code>. I have embedded the HSM into a Windows application (see Figure 3), because I needed a timer to dispatch TIMEOUT events to the state machine. Note, however, that the underlying coding technique is not at all Windows- or GUI-specific. In particular, the HSM implementation can be used in embedded systems in conjunction with any infrastructure to execute state machines.

EXERCISE: test the GUI application (either C or C++ version) and correlate its behavior with the state diagram from Figure 2 (you might find the current state display handy). Subsequently, set breakpoints at all exit actions from states and convince yourself that the application never leaks the Windows timer.

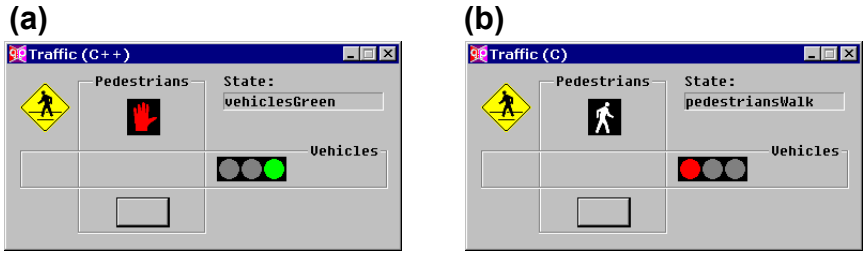

Figure 3 A GUI application representing the PELICAN crossing. The traffic signals are activated by pedestrians pushing the control button. Pane (a) shows the GUI in the `vehiclesEnabled` state, while pane (b) in the `pedestriansEnabled` state

While I don't have here room for a detailed discussion of the HSM implementation that you'll find online, I'd only like to mention the it is a straightforward extension of the FSM technique I presented in the April column. As before, `State` is defined as a pointer-to-function in C (and pointer-to-member-function in C++), however the signature of state handler methods has changed. In an HSM, the state handler additionally returns the superstate, thus providing information about the nesting of a given state. More precisely, a state handler method must return a pointer to the superstate handler, if it doesn't handle the event, or `NULL` if it does. To guarantee that all states have superstates, the `QHsm` class (base class for derivation of HSMs, analogous to the `Fsm` class from the April column) provides the top state as the ultimate root of the state hierarchy. The top state handler cannot be overridden and always returns `NULL` . These conventions make the implementation of the `QHsmDispatch()` method simple:

```
void QHsmDispatch(QHsm *me, QEvent const *e) {
   me->source__ = me->state__;
   do {
      me->source__ = (QState)(*me->source__)(me, e);
   } while (me->source__ != 0);
}
```

As you can see, every event dispatched to an HSM passes through a chain of state handlers until a state handler returns `NULL` , which indicates that either the event has been handled or the top state has been reached. As a client programmer, you don't need to know the internal details of the `QHsm` class. The main point is that the resulting HSM implementation provides a very straightforward mapping between the diagram and the code, and it is easy to modify at any stage of the development process. I encourage you to undertake the following two exercises:

EXERCISE: Modify the state machine to implement the alternative design of switching the DON'T WALK signal in the exit action from `pedestriansEnabled` and switching the red light for vehicles in the exit action from `vehiclesEnabled`. Recompile and test.

EXERCISE: Modify the initial transition in the `vehiclesEnabled` state to enter the `vehiclesGreenInt` substate. In addition, change the target of the `TIMEOUT` transition for state `pedestriansFlash` from `vehiclesEnabled` to `vehiclesGreen` .

End Around Check

Many writings about HSMs cautiously use the term "inheritance" to describe sharing of behavior between substates and superstates (e.g., see the OMG specification of UML v1.4 at <cgi.omg.org/docs/formal/01-09-67.pdf>). Please note, however, that the term *behavioral inheritance* is not part of the UML vocabulary and should not be confused with the traditional (class) inheritance applied to entire state machines (classes that internally embed state machines). A few readers of my last column were disappointed that after criticizing the Visual Basic calculator I didn't present a better state machine-based designs. Actually, I do have a bullet-proof calculator based on an HSM, which you can find at <www.quantum-leaps.com/cookbook/recipes.htm-#HSM_Design>.

I have also received a few letters from embedded programmers who seemed intimidated by UML. I believe that the concerns are largely exaggerated. It takes just a few hours to get acquainted with the most basic UML diagrams. (I get the most mileage from the sequence diagram, state diagram, and class diagram.) I think that the ability to read basic UML diagrams belongs to every embedded programmer's skill set in the same way as the ability to read schematics. If I were to recommend a book, I'd probably start with Martin Fowler's *UML Distilled: A Brief Guide to the Standard Object Modeling Language (2nd Edition)*, Addison-Wesley, 1999.

If you want to know the difference between Zebra, Pelican, Puffin, Toucan, and Pegasus pedestrian crossings, go to <www.2pass.co.uk/crossing.htm>.

Finally, I hope you experienced a strange feeling of *déjà vu* when you read about programming-by-difference, inheritance, guaranteed initialization and cleanup, and the LSP in the context of state machines. Such a close analogy between the fundamental concepts of OOP and HSMs is truly remarkable. Indeed, the analogy adds another dimension to the OOP. The traditional OO method seems to stop short at the boundary of a class, leaving the *internal* implementation of individual class methods to mostly procedural techniques. The concept of behavioral inheritance goes beyond that frontier. Behavioral inheritance allows applying the OO concepts *inside* reactive classes. ❑

Miro Samek has been developing embedded real-time software for almost 12 years. He is the author of *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems* (CMP Books, 2002). Dr. Samek is the lead software architect at IntegriNautics Corporation (Menlo Park, CA) and a consultant to industry. He previously worked at GE Medical Systems, where he has developed safety-critical, real-time software for diagnostic imaging x-ray machines. He earned his Ph. D. in nuclear physics at GSI (Darmstadt, Germany) where he conducted heavy-ion experiments. Miro welcomes feedback and can be reached at `miro@quantum-leaps.com`.