A crash course in UML state machines: Part 2

Miro Samek

3/9/2009 4:50 PM EDT

Miro Samek's three-part series 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. Here's part two: UML extensions to the traditional FSM formalism.

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

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

Though the traditional 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 the Visual Basic calculator (introduced in Part 1 in this series) 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, 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 with event-driven systems) because it makes the state machine approach truly applicable to real-life problems.

UML state machines, known also as UML statecharts, are object-based variants of Harel statecharts and incorporate several concepts defined in ROOMcharts, a variant of the statechart defined in the Real-time Object-Oriented Modeling (ROOM) language. UML statecharts are extended state machines with characteristics of both Mealy and Moore automata. In statecharts, actions generally depend on both the state of the system and the triggering event, as in a Mealy automaton. Additionally, UML statecharts provide optional entry and exit actions, which are associated with states rather than transitions, as in a Moore automaton.

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 GUIs results from the same pattern, which the Windows guru Charles Petzold calls the "Ultimate Hook". 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 choose to react in any way it likes. 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.

**Hierarchically nested states**

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 statecharts are also called *hierarchical state machines*, or HSMs). The semantics associated with state nesting are as follows (see Figure 2.5a): If a system is in the nested state "s11" (called the substate), it also (implicitly) is in the surrounding state "s1" (called the superstate). This state machine will attempt to handle any event in the context of state "s11," which conceptually 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 the superstate "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.

Figure 2.5: UML notation for hierarchically nested states (a), and a state model of a toaster oven in which states "toasting" and "baking" share the common transition from state "heating" to "door_open" (b).

States that contain other states are called composite states; conversely, states without internal structure are called simple states. A nested state is called a direct substate when it is not contained by any other state; otherwise, it is referred to as a transitively nested substate.

Because the internal structure of a composite state can be arbitrarily complex, any hierarchical state machine can be viewed as an internal structure of some (higher-level) composite state. It is conceptually convenient to define one composite state as the ultimate root of state machine hierarchy. In the UML specification, every state machine has a top state (the abstract root of every state machine hierarchy), which contains all the other elements of the entire state machine. The graphical rendering of this all-enclosing top state is optional.

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 toaster oven shown in Figure 2.5b, states "toasting" and "baking" share a common transition to the "door_open" state, defined in their common superstate."
The aspect of state hierarchy emphasized most often is abstraction—an old and powerful technique for coping with complexity. Instead of facing all aspects of a complex system at the same time, it is often possible to ignore (abstract away) some parts of the system. Hierarchical states are an ideal mechanism for hiding internal details because the designer can easily zoom out or zoom in to hide or show nested states. Although abstraction by itself does not reduce overall system complexity, it is valuable because it reduces the amount of detail you need to deal with at one time. As Grady Booch notes:5

"... we are still constrained by the number of things that we can comprehend at one time, but through abstraction, we use chunks of information with increasingly greater semantic content."

However valuable abstraction in itself might be, you cannot cheat your way out of complexity simply by hiding it inside composite states. However, the composite states don't simply hide complexity, they also actively reduce it through the powerful mechanism of reuse (the "Ultimate Hook" pattern). Without such reuse, even a moderate increase in system complexity often leads to an explosive increase in the number of states and transitions. For example, if you transform the statechart from Figure 2.5b to a classical flat state machine,5 you must repeat one transition (from heating to "eating") in two places: as a transition from "eating" to "drinking" and from "drinking" to "sleeping." Avoiding such repetitions allows HSMs to grow proportionally to system complexity. As the modeled system grows, the opportunity for reuse also increases and thus counteracts the explosive increase in states and transitions typical for traditional FSMs.

**Behavioral inheritance**
Hierarchical states are more than merely the "grouping of [nested] state machines together without additional semantics".6 In fact, hierarchical states have simple but profound semantics. Nested states are also more than just "great diagrammatic simplification when a set of events applies to several substates".7 The savings in the number of states and transitions are real and go far beyond less cluttered diagrams. In other words, simpler diagrams are just a side effect of behavioral reuse enabled by state nesting.

The fundamental character of state nesting comes from the combination of abstraction and hierarchy, which is a traditional approach to reducing complexity and is otherwise known in software as *inheritance*. In OOP, the concept of class inheritance describes relations between classes of objects. Class inheritance describes the "is a ..." relationship among classes.

For example, class  might derive from class . If an object is a bird (instance of the class), it automatically is an animal, because all operations that apply to animals (e.g., eating, eliminating, reproducing) also apply to birds. But birds are more specialized, since they have operations that are not applicable to animals in general. For example,  applies to birds but not to fish.

The benefits of class inheritance are concisely summarized by Gamma and colleagues:8

"Inheritance lets you define a new kind of class rapidly in terms of an old one, by reusing functionality from parent classes. It allows new classes to be specified by difference rather than created from scratch each time. It lets you get new implementations almost for free, inheriting most of what is common from the ancestor classes."

As you saw in the previous section, all these basic characteristics of inheritance apply equally well to nested states (just replace the word class with state), which is not surprising because state nesting is based on the same fundamental "is a ..." classification as object-oriented class inheritance. For example, in a state model of a toaster oven, state  nests inside state .
If the toaster is in the "toast" state, it automatically is in the "bake" state because all behavior pertaining to "toast" applies also to "bake" (e.g., the heater must be turned on). But "heat" is more specialized because it has behaviors not applicable to "bake" in general. For example, setting toast color (light or dark) applies to "bake" but not to "toast."

In the case of nested states, the "is a ..." (is-a-kind-of) relationship merely needs to be replaced by the "is in ..." (is-in-a-state) relationship; otherwise, it is the same fundamental classification. State nesting allows a substate to inherit state behavior from its ancestors (superstates); therefore, it's called behavioral inheritance.

The concept of inheritance is fundamental in software construction. Class inheritance is essential for better software organization and for code reuse, which makes it a cornerstone of OOP. In the same way, behavioral inheritance is essential for efficient use of HSMs and for behavior reuse, which makes it a cornerstone of event-driven programming. The book this series is excerpted from—Practical UML Statecharts in C/C++, Second Edition—presents, a mini-catalog of state patterns shows ways to structure HSMs to solve recurring problems. Not surprisingly, behavioral inheritance plays the central role in all these patterns.

Liskov Substitution Principle for States
Identifying the relationship among substates and superstates as inheritance has many practical implications. Perhaps the most important is the Liskov Substitution Principle (LSP) applied to state hierarchy. In its traditional formulation for classes, LSP requires that a subclass can be freely substituted for its superclass. This means that every instance of the subclass should be compatible with the instance of the superclass and 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.

Because behavioral inheritance is just a special kind of inheritance, LSP can be applied to nested states as well as classes. LSP generalized for states means that the behavior of a substate should be consistent with the superstate. For example, all states nested inside the "on" state of the toaster oven (e.g., toasting or baking) should share the same basic characteristics of the "on" state. In particular, if being in the "on" state means that the heater is turned on, then none of the substates should turn the heater off (without transitioning out of the "on" state). Turning the heater off and staying in the "on" or "off" state would be inconsistent with being in the "on" state and would indicate poor design (violation of the LSP).

Compliance with the LSP allows you to build better (more 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 "on" state (thus abstracting away the specifics of "toasting" and "baking"). 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 the specifics of the substates will be incorrect.

Orthogonal regions
Hierarchical state decomposition can be viewed as exclusive-OR operation applied to states. For example, if a system is in the "on" superstate (Figure 2.5b), it means that it's either in "toasting" substate OR the "baking" substate. That is why the "on" superstate is called an OR-state.

UML statecharts also introduce the complementary AND-decomposition. Such decomposition means that a composite state can contain two or more orthogonal regions (orthogonal means independent in this context) and that being in such a composite state entails being in all its orthogonal regions simultaneously.

Orthogonal regions address the frequent problem of a combinatorial increase in the number of states when the
behavior of a system is fragmented into independent, concurrently active parts. For example, apart from the main keypad, a computer keyboard has an independent numeric keypad.

From the previous discussion, recall the two states of the main keypad already identified: "default" and "caps_locked" (Figure 2.2 in part 1 of this article). The numeric keypad also can be in two states--" " and " "--depending on whether Num Lock is active. The complete state space of the keyboard in the standard decomposition is the cross-product of the two components (main keypad and numeric keypad) and consists of four states: " " and " ".

However, this is unnatural because the behavior of the numeric keypad does not depend on the state of the main keypad and vice versa. Orthogonal regions allow you to avoid mixing the independent behaviors as a cross-product and, instead, to keep them separate, as shown in Figure 2.6.

Figure 2.6: Two orthogonal regions (main keypad and numeric keypad) of a computer keyboard.

Note that if the orthogonal regions are fully independent of each other, their combined complexity is simply additive, which means that the number of independent states needed to model the system is simply the sum \( k + l + m + \ldots \), where \( k, l, m, \ldots \) denote numbers of OR-states in each orthogonal region. The general case of mutual dependency, on the other hand, results in multiplicative complexity, so in general, the number of states needed is the product \( k \cdot l \cdot m \cdot \ldots \).

In most real-life situations, however, orthogonal regions are only approximately orthogonal (i.e., they are not independent). Therefore, UML statecharts provide a number of ways for orthogonal regions to communicate and synchronize their behaviors. From these rich sets of (sometimes complex) mechanisms, perhaps the most important is that orthogonal regions can coordinate their behaviors by sending event instances to each other.

Even though orthogonal regions imply independence of execution (i.e., some kind of concurrency), the UML specification does not require that a separate thread of execution be assigned to each orthogonal region (although it can be implemented that way). In fact, most commonly, orthogonal regions execute within the same thread. The UML specification only requires that the designer not rely on any particular order in which an event instance will be dispatched to the involved orthogonal regions.

**Entry and exit actions**

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 its entry and exit actions will be executed. Because of this characteristic, statecharts behave like Moore automata. The UML notation for state entry and exit actions is to place the reserved word "entry" (or "exit") in the state right below the name compartment, followed by the forward slash and the list of arbitrary actions (see Figure 2.7).

Figure 2.7: Toaster oven state machine with entry and exit actions.
The value of entry and exit actions is that they provide means for guaranteed initialization and cleanup, very much like class constructors and destructors in OOP. For example, consider the "heating" state from Figure 2.6b, which corresponds to the toaster oven behavior while the door is open. This state has a very important safety-critical requirement: Always disable the heater when the door is open. Additionally, while the door is open, the internal lamp illuminating the oven should light up.

Of course, you could model such behavior by adding appropriate actions (disabling the heater and turning on the light) to every transition path leading to the "heating" state (the user may open the door at any time during "heating" or "toasting" or when the oven is not used at all). You also should not forget to extinguish the internal lamp with every transition leaving the "heating" state.

However, such a solution would cause the repetition of actions in many transitions. More important, such an approach is error-prone in view of changes to the state machine (e.g., the next programmer working on a new feature, such as top-browning, might simply forget to disable the heater on transition to "heating").

Entry and exit actions allow you to implement the desired behavior in a much safer, simpler, and more intuitive way. As shown in Figure 2.7, you could specify that the exit action from "heating" disables the heater, the entry action to "heating" lights up the oven lamp, and the exit action from "heating" extinguishes the lamp. The use of entry and exit action is superior to placing actions on transitions because it avoids repetitions of those actions on transitions and eliminates the basic safety hazard of leaving the heater on while the door is open. The semantics of exit actions guarantees that, regardless of the transition path, the heater will be disabled when the toaster is not in the "heating" state.

Because entry actions are executed automatically whenever an associated state is entered, they often determine the conditions of operation or the identity of the state, very much as a class constructor determines the identity of the object being constructed. For example, the identity of the "heating" state is determined by the fact that the heater is turned on.

This condition must be established before entering any substate of "heating" because entry actions to a substate of "heating," like "heating," rely on proper initialization of the "heating" 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 very root 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).

**Internal transitions**

Very commonly, an event causes only some internal actions to execute but does not lead to a change of state (state transition). In this case, all actions executed comprise the internal transition. For example, when you...
type on your keyboard, it responds by generating different character codes. However, unless you hit the Caps
Lock key, the state of the keyboard does not change (no state transition occurs). In UML, this situation
should be modeled with internal transitions, as shown in Figure 2.8. The UML notation for internal
transitions follows the general syntax used for exit (or entry) actions, except instead of the word entry (or
exit) the internal transition is labeled with the triggering event (e.g., see the internal transition triggered by the
CAPS_LOCK event in Figure 2.8).

**Figure 2.8: UML state diagram of the keyboard state machine with internal transitions.**

In the absence of entry and exit actions, internal transitions would be identical to self-transitions (transitions
in which the target state is the same as the source state). In fact, in a classical Mealy automaton, actions are
associated exclusively with state transitions, so the only way to execute actions without changing state is
through a self-transition (depicted as a directed loop in Figure 2.2 from the first part of this article). However,
in the presence of entry and exit actions, as in UML statecharts, a self-transition involves the execution of
exit and entry actions and therefore it is distinctively different from an internal transition.

In contrast to a self-transition, no entry or exit actions are ever executed as a result of an internal transition,
even if the internal transition is inherited from a higher level of the hierarchy than the currently active state.
Internal transitions inherited from superstates at any level of nesting act as if they were defined directly in the
currently active state.

**Transition execution sequence**
State nesting combined with entry and exit actions significantly complicates the state transition semantics in
HSMs compared to the traditional FSMs. When dealing with hierarchically nested states and orthogonal
regions, the simple term current state can be quite confusing. In an HSM, more than one state can be active at
once. If the state machine is in a leaf state that is contained in a composite state (which is possibly contained
in a higher-level composite state, and so on), all the composite states that either directly or transitively contain
the leaf state are also active. Furthermore, because some of the composite states in this hierarchy might have
orthogonal regions, the current active state is actually represented by a tree of states starting with the single
top state at the root down to individual simple states at the leaves. The UML specification refers to such a
state tree as state configuration. ²

**Figure 2.9: State roles in a state transition.**
In UML, a state transition can directly connect any two states. These two states, which may be composite, are designated as the main source and the main target of a transition. Figure 2.9 shows a simple transition example and explains the state roles in that transition. The UML specification prescribes that taking a state transition involves executing the following actions in the following sequence (see [8, Section 15.3.14]):

1. Evaluate the guard condition associated with the transition and perform the following steps only if the guard evaluates to TRUE.

2. Exit the source state configuration.

3. Execute the actions associated with the transition.

4. Enter the target state configuration.

The transition sequence is easy to interpret in the simple case of both the main source and the main target nesting at the same level. For example, transition T1 shown in Figure 2.9 causes the evaluation of the guard followed by the sequence of actions: and , assuming that the guard g() evaluates to . However, in the general case of source and target states nested at different levels of the state hierarchy, it might not be immediately obvious how many levels of nesting need to be exited. The UML specification prescribes that a transition involves exiting all nested states from the current active state (which might be a direct or transitive substate of the main source state) up to, but not including, the least common ancestor (LCA) state of the main source and main target states.

As the name indicates, the LCA is the lowest composite state that is simultaneously a superstate (ancestor) of both the source and the target states. As described before, the order of execution of exit actions is always from the most deeply nested state (the current active state) up the hierarchy to the LCA but without exiting the LCA. For instance, the LCA(s1,s2) of states "s1" and "s2"shown in Figure 2.9 is state "s."

Entering the target state configuration commences from the level where the exit actions left off (i.e., from inside the LCA). As described before, entry actions must be executed starting from the highest-level state down the state hierarchy to the main target state. If the main target state is composite, the UML semantics prescribes to "drill" into its submachine recursively using the local initial transitions. The target state configuration is completely entered only after encountering a leaf state that has no initial transitions.

**NOTE**

The HSM implementation described in the book this series is excerpted from--Practical UML Statecharts in C/C++---preserves the essential order of exiting the source configuration followed by entering the target state configuration, but executes the actions associated with the transition entirely in the context of the source state, that is, before exiting the source state configuration. Specifically, the implemented transition sequence is as follows:
1. Evaluate the guard condition associated with the transition and perform the following steps only if the guard evaluates to TRUE.

2. Execute the actions associated with the transition.

3. Atomically exit the source state configuration and enter the target state configuration.

For example, the transition T1 shown in Figure 2.9 will cause the evaluation of the guard \( \text{guard} \) followed by the sequence of actions: \( \text{action1} \) and \( \text{action2} \), assuming that the guard \( \text{guard} \) evaluates to TRUE.

One big problem with the UML transition sequence is that it requires executing actions associated with the transition after destroying the source state configuration but before creating the target state configuration. In the analogy between exit actions in state machines and destructors in OOP, this situation corresponds to executing a class method after partially destroying an object. Of course, such action is illegal in OOP. As it turns out, it is also particularly awkward to implement for state machines.

Executing actions associated with a transition is much more natural in the context of the source state—the same context in which the guard condition is evaluated. Only after the guard and the transition actions execute, the source state configuration is exited and the target state configuration is entered atomically. That way the state machine is observable only in a stable state configuration, either before or after the transition, but not in the middle.

**Local versus external transitions**

Before UML 2, the only transition semantics in use was the *external transition*, in which the main source of the transition is always exited and the main target of the transition is always entered. UML 2 preserved the "external transition" semantics for backward compatibility, but also introduced also a new kind of transition called local transition [8, Section 15.3.15].

For many transition topologies, external and local transitions are actually identical. However, a local transition doesn't cause exit from the main source state if the main target state is a substate of the main source. In addition, local state transition doesn't cause exit and reentry to the target state if the main target is a superstate of the main source state.

**Figure 2.10** contrasts local (a) and external (b) transitions. In the top row, you see the case of the main source containing the target. The local transition does not cause exit from the source, while the external transition causes exit and re-entry to the source. In the bottom row of **Figure 2.10**, you see the case of the target containing the source. The local transition does not cause entry to the target, whereas the external transition causes exit and reentry to the target.

**Figure 2.10: Local (a) versus external transitions (b). QP implements only the local transitions.**
The HSM implementation described in the book *Practical UML Statecharts in C/C++* supports exclusively the local state transition semantics.

**Event deferral**

Sometimes an event arrives at a particularly inconvenient time, when a state machine is in a state that cannot handle the event. In many cases, the nature of the event is such that it can be postponed (within limits) until the system enters another state, in which it is much better prepared to handle the original event.

UML state machines provide a special mechanism for deferring events in states. In every state, you can include a clause 'deferred / [event list]'. If an event in the current state's deferred event list occurs, the event will be saved (deferred) for future processing until a state is entered that does not list the event in its deferred event list. Upon entry to such state, the UML state machine will automatically recall any saved event(s) that are no longer deferred and process them as if they have just arrived.

NOTE: The HSM implementation described in the book this series is excerpted from--*Practical UML Statecharts in C/C++*--does not directly support the UML-style event deferral. However, the "Deferred Event" state pattern presented shows how to approximate this feature in a much less expensive way by explicitly deferring and recalling events.

**UML statecharts and automatic code synthesis**

UML statecharts provide sufficiently well-defined semantics for building executable state models. Indeed, several design automation tools on the market support various versions of statecharts (see the sidebar "Design Automation Tools Supporting Statecharts"). The commercially available design automation tools typically not only automatically generate code from statecharts but also enable debugging and testing of the state models at the graphical level.7

But what does automatic code generation really mean? And more important, what kind of code is actually generated by such statechart-based tools?

Many people understand automatic code synthesis as the generation of a program to solve a problem from a statement of the problem specification. Statechart-based tools cannot provide this because a statechart is just a higher-level (mostly visual) solution rather than the statement of the problem.

As far as the automatically generated code is concerned, the statechart-based tools can autonomously generate only so-called "housekeeping code".7 The modeler explicitly must provide all the application-specific code, such as action and guard expressions, to the tool. The role of housekeeping code is to "glue" the various action and guard expressions together to ensure proper state machine execution in accordance with the statechart semantics. For example, synthesized code typically handles event queuing, event dispatching, guard evaluation, or transition chain execution (including exit and entry of appropriate states). Almost universally, the tools also encompass some kind of real-time framework that integrates tightly with the underlying operating system.
Design automation tools supporting statecharts

Some of the computer-aided software-engineering (CASE) tools with support for statecharts currently available on the market are:

Telelogic Statemate, [www.telelogic.com](http://www.telelogic.com) (the tool originally developed by I-Logix, Inc. acquired by Telelogic in 2006, which in turn is in the process of being acquired by IBM).

Telelogic Rhapsody, [www.telelogic.com](http://www.telelogic.com).


ARTiSAN Studio, ARTiSAN Software Tools, Ltd., [www.artisansw.com](http://www.artisansw.com).

Stateflow, The Mathworks, [www.mathworks.com](http://www.mathworks.com)

VisualSTATE, IAR Systems, [www.iar.com](http://www.iar.com)

The limitations of the UML state diagrams

Statecharts have been invented as "a visual formalism for complex systems,"¹ so from their inception, they have been inseparably associated with graphical representation in the form of state diagrams. However, it is important to understand that the concept of HSMs transcends any particular notation, graphical or textual. The UML specification² makes this distinction apparent by clearly separating state machine semantics from the notation.

However, the notation of UML statecharts is not purely visual. Any nontrivial state machine requires a large amount of textual information (e.g., the specification of actions and guards). The exact syntax of action and guard expressions isn't defined in the UML specification, so many people use either structured English or, more formally, expressions in an implementation language such as C, C++, or Java.¹⁰ In practice, this means that UML statechart notation depends heavily on the specific programming language.

Nevertheless, most of the statecharts semantics are heavily biased toward graphical notation. For example, state diagrams poorly represent the sequence of processing, be it order of evaluation of guards or order of dispatching events to orthogonal regions. The UML specification sidesteps these problems by putting the burden on the designer not to rely on any particular sequencing. But, when you actually implement UML state machines, you will always have full control over the order of execution, so the restrictions imposed by UML semantics will be unnecessarily restrictive.

Similarly, statechart diagrams require a lot of plumbing gear (pseudostates, like joins, forks, junctions, choicepoints, etc.) to represent the flow of control graphically. These elements are essentially the old flowchart in disguise, which structured programming techniques proved far less significant a long time ago. In other words, these elements of the graphical notation do not add much value in representing flow of control as compared to plain structured code.

This is not to criticize the graphical notation of statecharts. In fact, it is remarkably expressive and can scarcely be improved. Rather, I want merely to point out some shortcomings and limitations of the pen-and-paper diagrams.
The UML notation and semantics are really geared toward computerized design automation tools. A UML state machine, as represented in a tool, is not just the state diagram, but rather a mixture of graphical and textual representation that precisely captures both the state topology and the actions. The users of the tool can get several complementary views of the same state machine, both visual and textual, whereas the generated code is just one of the many available views.

**UML state machine semantics: an exhaustive example**

The very rich UML state machine semantics might be quite confusing to newcomers, and even to fairly experienced designers. Wouldn't it be great if you could generate the exact sequence of actions for every possible transition so that you know for sure what actions get executed and in which order?

In this section, I present an executable example of a hierarchical state machine shown in Figure 2.11 that contains all possible transition topologies up to four levels of state nesting. The state machine contains six states: "s," "s1," "s11," "s2," "s21," and "s211." The state machine recognizes nine events A through I, which you can generate by typing either uppercase or lowercase letters on your keyboard.

All the actions of this state machine consist only of statements that report the status of the state machine to the screen. This complete example with source code is available from the companion Website for the book "Practical UML Statecharts in C/C++, Second Edition" at www.state-machine.com/psicc2. The executable console application for Windows is located in the directory <qp>\qpc\examples\80x86\dos\cpp101\\qsmtst\dbg\. The name of the application is QHSMTST.EXE.

**Figure 2.11: Hypothetical state machine that contains all possible state transition topologies up to four levels of state nesting**

Figure 2.12 shows an example run of the QHSMTST.EXE application. Note the line numbers in parentheses at the left edge of the window, added for reference. Line (1) shows the effect of the topmost initial transition. Note the sequence of entry actions and initial transitions ending at the "s211-ENTRY" printout. Injecting events into the state machine begins in line (2).

Every generated event (shown on a gray background) is followed by the sequence of exit actions from the source state configuration followed by entry actions and initial transitions entering the target state configuration. From these printouts you can always determine the order of transition processing as well as the active state, which is the last state entered. For instance, the active state before injecting event G in line (2) is "s211" because this is the last state entered in the previous line.

**Figure 2.12: QHSMTST.EXE example application running in the command window. The line numbers**
Per the semantics of UML state machines, the event G injected in line (2) is handled in the following way. First, the active state ("s211") attempts to handle the event. However, as you can see in the state diagram in Figure 2.11, state "s211" does not prescribe how to handle event G. Therefore the event is passed on to the next higher-level state in the hierarchy, which is state "s21." The superstate "s21" prescribes how to handle event G because it has a state transition triggered by G. State "s21" executes the actions associated with the transition (printout "s21-G;" in line (2) of Figure 2.12).

Next the state executes the transition chain that exits the source state configuration and enters the target state configuration. The transition chain starts with execution of exit actions from the active state through higher and higher levels of hierarchy. Next the target state configuration is entered in exact opposite order, that is, starting from highest levels of state hierarchy down to the lowest. The transition G from state "s21" terminates at state "s1." However, the transition chain does not end at the direct target of the transition but continues via the initial transition defined in the direct target state "s21." Finally, a new active state is established by entering state "s11" that has no initial transition.

In line (3) of Figure 2.12, you see how the statechart handles an internal transition. Event I is injected while state machine is in state "s11." Again, the active state does not prescribe how to handle event I, so it is passed on to the next level of state hierarchy, that is, to state "s1." State "s1" has an internal transition triggered by I defined in its internal transition compartment; therefore "s1" handles the event (printout "s1-I;" in line (3)). And at this point the processing ends. No change of state ever occurs in the internal transition, even if such transition is inherited from higher levels of state hierarchy.

In the UML state machines internal transitions are different from self-transitions. Line (4) of Figure 2.12 demonstrates the difference. The state machine is in state "s11" when event A is injected. As in the case of the internal transition, the active state "s11" does not prescribe how to handle event A, so the event is passed on to the superstate "s1." The superstate has a self-transition triggered by A and so it executes actions associated with the transition (printout "s1-A;" in line (4)). This time, however, a regular transition is taken, which requires exiting the source state configuration and entering the target state configuration.

UML statecharts are extended state machines, meaning that in general, the actions executed by the state machine can depend also on the value of the extended state variables. Consider for example event D injected in line (5). The active state "s11" has transition D, but this transition has a guard $\text{is}\neq 0$. The variable $\text{is}$ is an extended state variable of the state machine from Figure 2.11. You can see that $\text{is}$ is initialized to 0 on the topmost initial transition.

Therefore the guard $\text{is}\neq 0$, which is a test of $\text{is}$ against 0, evaluates to false. The guard condition temporarily disables the transition D in state "s11," which is handled as though state "s11" did not define the transition in the first place. Therefore, the event D is passed on to the next higher level, that is, to...
state "s1." State "s1" has transition D with a complementary guard. This time, the guard evaluates to false and the transition D in state "s1" is taken.

As indicated in the diagram, the transition action changes the value of the extended state variable to 1. Therefore when another event D is injected again in line (6), the guard condition on transition D in state "s11" evaluates to false and this transition is taken, as indicated in line (6) of Figure 2.12.

Line (7) of Figure 2.12 demonstrates that all exit and entry actions are always executed, regardless of the exit and entry path. The main target of transition C from "s1" is "s2." The initial transition in the main target state goes "over" the substate "s21" all the way to the substate "s211." However, the entry actions don't skip the entry to "s21." This example demonstrates the powerful concept of guaranteed cleanup of the source state configuration and guaranteed initialization of the target state configuration, regardless of the complexity of the exit or entry path.

Interestingly, in hierarchical state machines, the same transition can cause different sequences of exit actions, depending on which state inherits the transition. For example, in lines (8) and (9) of Figure 2.12, event E triggers the exact same state transition defined in the superstate "s." However, the responses in lines (8) and (9) are different because transition E fires from different state configurations--once when "s211" is active (line (8)) and next when "s11" is active (line (9)).

Finally, lines (11) and (12) of Figure 2.12 demonstrate that guard conditions can also be used on internal transitions. States "s2" and d both define internal transition I with complementary guard conditions. In line (11), the guard enables internal transition in state "s2." In line (12) the same guard disables the internal transition in "s2," and therefore the internal transition defined in the superstate d is executed.

You can learn much more about the semantics of UML state machines by injecting various events to the QHSMTST.EXE application and studying its output. Because the state machine from Figure 2.11 has been specifically designed to contain all possible state transition configurations up to level 4 of nesting, this example can "answer" virtually all your questions regarding the semantics of statecharts. Moreover, you can use the source code that actually implements the state machine (located in <qp>\qpc\examples\80x86\dos\cpp\qhsmtst\qhsmtst.c) as a template for implementing your own statecharts. Simply look up applicable fragments in the diagram from Figure 2.11 and check how they have been implemented in qhsmtst.c.

Next in Part 3: **Designing a UML State Machine**

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

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:**


8. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.


---

**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.)