



Brief Introduction to UML™ State Machines

Reactive Systems

Almost all embedded systems (and majority of general-purpose computers, for that matter) are **event-driven**, which means that they continuously wait for the occurrence of some external or internal **event** such as a time tick, an arrival of a data packet, a button press, or a mouse click. After recognizing the event, such systems **react** by performing the appropriate computation that may include manipulating the hardware or generating “soft” events that trigger other internal software components. (That’s why event-driven systems are alternatively called **reactive systems**.) Once the event handling is complete, the software goes back to a dormant state (e.g., a background-loop, an idle task, or a power-saving mode) in anticipation of the next event.

From the programming perspective, this scheme implies that upon the arrival of an event the CPU executes only a tiny fraction of the overall code. The main programming challenge is to quickly pick and execute the **right** code fragment. Indeed, you can trace the whole progress in understanding reactive systems just by studying the methods used to pick the correct code to execute.

The problem is not at all trivial and leads to control inversion compared to the traditional data-processing systems. A conventional data-processing program (such as a compiler) starts off with some initial data (e.g., a source file), which it transforms by a series of computations into the desired output data (an object file), maintaining full control of the processing sequence from the beginning to the end. In contrast, an event-driven program gains control only sporadically when handling events. The actions actually executed are determined by both events, and the current execution context (i.e., the sequence of past events in which the system was involved), so the path through the code is likely to differ every time such software is run.

Oversimplification of the Event-Action Paradigm

The currently dominating approach to structuring event-driven software is the venerable “event-action paradigm”, in which events are directly mapped to the code that is supposed to be executed in response. However, events alone do not fully determine the actions to be performed in response to these events. In all but the most trivial reactive systems, the response to events is determined also by the **execution context** (often called the mode) of the system. The prevalent event-action paradigm, however, neglects the latter dependency and leaves the handling of the context to largely ad-hoc techniques. With this approach, most reactive programs start out fairly simple, but as features are grafted on, more and more flags and variables are introduced to capture the relevant mode (event history). Then ifs and elses must be added to test the increasingly complex logical expressions built out of the various variables and flags, until no human being really has a good idea what parts of the code get executed in response to any given event. Such an approach is a fertile ground for bugs for at least three reasons:





1. It always leads to convoluted conditional logic in handling events ("spaghetti" code).
2. Each branching point in the logic requires evaluation of a complex expression.
3. Switching between different modes requires modifying many variables, which all can easily lead to inconsistencies.

The Essence of State Machines

And here is where **state machines** come in. When used correctly, state machines become powerful "spaghetti reducers" that drastically reduce the number of execution paths through the code, simplify the conditions tested at each branching point, and simplify transitions between different modes of execution.

All these benefits hinge on the concept of **state**. As it turns out, the behavior of most reactive systems can be divided into a relatively small number of chunks (states), where event responses within each individual chunk indeed depend only on the current event, but no longer on the sequence of past events. In other words, the event-action paradigm is still applied, but only **locally** within a state. In this model, change of behavior (that is, change in response to any event) corresponds to change of state (state transition).

A state captures the relevant aspects of the system's history very efficiently. For example, you can say that a computer keyboard is either in the "shifted" state, or the "default" state. The behavior of a keyboard depends only on certain aspects of event history, namely whether the Shift key has been depressed, but not, for example, on how many and which specific characters have been typed previously. A state can abstract away all possible (but irrelevant) event sequences and capture only the relevant ones.



Back to coding, this means that instead of recording the event history in a multitude of variables, you rely mainly on only **one** "state variable" that can take only a limited number of *a priori* known values. The value of the "state variable" crisply defines the state of the system at any given time. A state machine reduces the problem of identifying the execution context to testing just the "state variable" instead of many variables. Actually, in all but the most basic state machine implementations (such as the nested switch statement), even the explicit testing of the state variable disappears from the code, which reduces "spaghetti" further still (we will experience this effect later in this class). Moreover, switching between different states is vastly simplified as well, because you need to reassign just one state variable instead of changing multiple variables in a self-consistent manner.

State Diagrams

State machines have very compelling graphical representation in form of **state diagrams**. Such diagrams are directed graphs in which nodes denote states and connectors denote transitions. UML maintains this general form, as shown in the keyboard state diagram in Figure 1.





Note: All UML™ state diagrams in this section have been created with Visio™ Technical 5.0. The Visio stencil used in these diagrams is available for a free download from <http://www.quantum-leaps.com/resources/goodies.htm#Visio>. This stencil should be compatible with all newer versions of Microsoft Visio™.

States are represented as rounded rectangles labeled with state names. The transitions, represented as arrows, are labeled with the triggering events followed optionally by the list of event parameters (in parentheses) and triggered actions (following the slash '/'). Internal transitions, which don't lead to a change of state (e.g., transitions triggered by event ANY_KEY), are in UML different from self transitions (transitions that leave and re-enter a state) and have their own distinctive notation. The transition originating from the solid circle is called an initial transition, because it specifies the starting state when the system first begins. Every state diagram should have such a transition, which should not be labeled, since it is not triggered by an event.

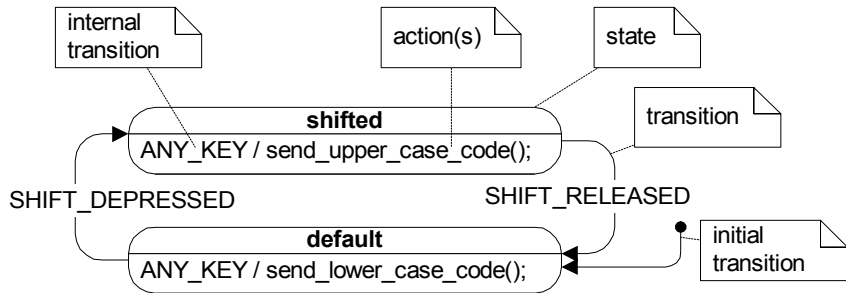


Figure 1 UML state diagram representing the keyboard state machine

The Essence of Statecharts

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. If you try to represent the behavior of just about any nontrivial system with a traditional FSM, you'll immediately notice that many events are handled identically in many states, so the behavior within states largely overlaps. 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 **reuse** it across many states.

The formalism of **statecharts**, invented by David Harel in the 1980s [Harel87] and subsequently adopted by most methodologies and modeling approaches (such as UML), 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) always results from the same pattern, which Charles Petzold calls the “Ultimate Hook” [Petzold96]. 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.

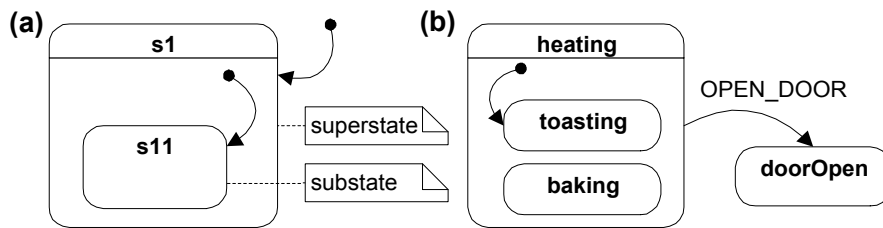


Figure 2 (a) UML notation for hierarchically nested states; (b) state model of a simple toaster-oven in which states toasting and baking share the common transition from state heating to doorOpen.

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**).

The semantics associated with state nesting (shown in Figure 2[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 HSM 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 exactly 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 (reuse) all aspects of behavior with their superstates.





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** [Samek00, 02, 03b]. 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 specialize a data type, behavioral inheritance allows substates to specialize behavior 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 superstate 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 ancestors. In both cases, overriding the inherited behavior leads to **polymorphism**.

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 that they provide means for guaranteed initialization and cleanup, very much like class constructors and destructors in OOP. For example, consider the `doorOpen` state from Figure 2(b), 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 `doorOpen` state (the user may open the door at any time during baking or toasting or when the oven is not in use). You also should not forget to extinguish the internal lamp with every transition leaving the `doorOpen` state. However, such a solution would cause the repetition of actions in many transitions. More importantly, such an approach is error-prone in view of changes to the state machine (e.g., a programmer working on a new feature, such as top-browning, might simply forget to disable the heater on transition to `doorOpen`).

Entry and exit actions allow you to implement the desired behavior in a safer, simpler, and more intuitive way. You could specify that the exit action from `heating` (see Figure 2[b]) disables the heater, the entry action to `doorOpen` lights up the oven lamp, and the exit action from `doorOpen` extinguishes the lamp. This solution is superior 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.

Obviously, you can also use entry and exit actions in the classical (nonhierarchical) FSMs. In fact, the lack of hierarchy in this case makes implementation of this feature trivial. However, entry and exit actions are particularly important and powerful in HSMs, because they determine the **identity** of hierarchical states. For example, the identity of the otherwise abstract 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 *toasting*, 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).

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

Summary

Many writings about HSMs cautiously use the term “inheritance” to describe sharing of behavior between substates and superstates (e.g., see the latest OMG specification of UML 2.0). 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).

Many embedded programmers seem intimidated by UML. However, the concerns are largely exaggerated because it takes just a few hours to get acquainted with the most basic UML diagrams (such as the sequence diagram, state diagram, or class diagram).

Another widely spread misconception is that UML statecharts mandate the use of automatic code generation tools. In the nutshell, the concepts of state, state nesting, and guaranteed initialization and cleanup, are fundamentally simple. Other “Recipes” available from Quantum Leaps describe how to implement statecharts in C, C++, Java, and other languages. In the end, HSM is a way of software design, not the use of a particular tool.

