# UML Tutorial: Complex Transitions

Robert C. Martin

Engineering Notebook Column

C++ Report, September 98

In my last column I talked about UML Finite State Machine diagrams. In this column we will be discussing multi-threaded state machines, and how to model them in UML.

Multithreaded state machines are not new. For years they have been modeled using petri-nets. UML has borrowed many of the petri-net concepts and mixed them with the notation for regular state machines. The result is a very complete and convenient language for expressing state machines of many different varieties, either single threaded or multi-threaded.

## *The Toaster*

Lets begin our discussion by looking at the behavior of a simple machine that requires a multithreaded FSM – a toaster. Our toaster has two slots. Each can hold a slice of bread to be toasted. If only one slice is to be toasted, it must be placed in slot 1. The user places the bread in the slots and then depresses the lever. The bread descends into the toaster. Heating filaments toast the bread. When the toast is complete, the carrier pops up, and the toast is made available to the user.

How do we know when the toast is done? There is a color selector knob on the toaster. A sensor in slot 1 measures the color of the toast. When the toast has acquired the color that the knob is set to, the toast is complete. Alternatively, a timer will terminate the toasting if the toast takes "too long" to reach the appropriate color (e.g. a white bathroom tile was placed in the slot). Alternatively, if the color changes too fast, then the toasting is aborted. This protects us from starting fires when someone puts a piece of paper in the slot.

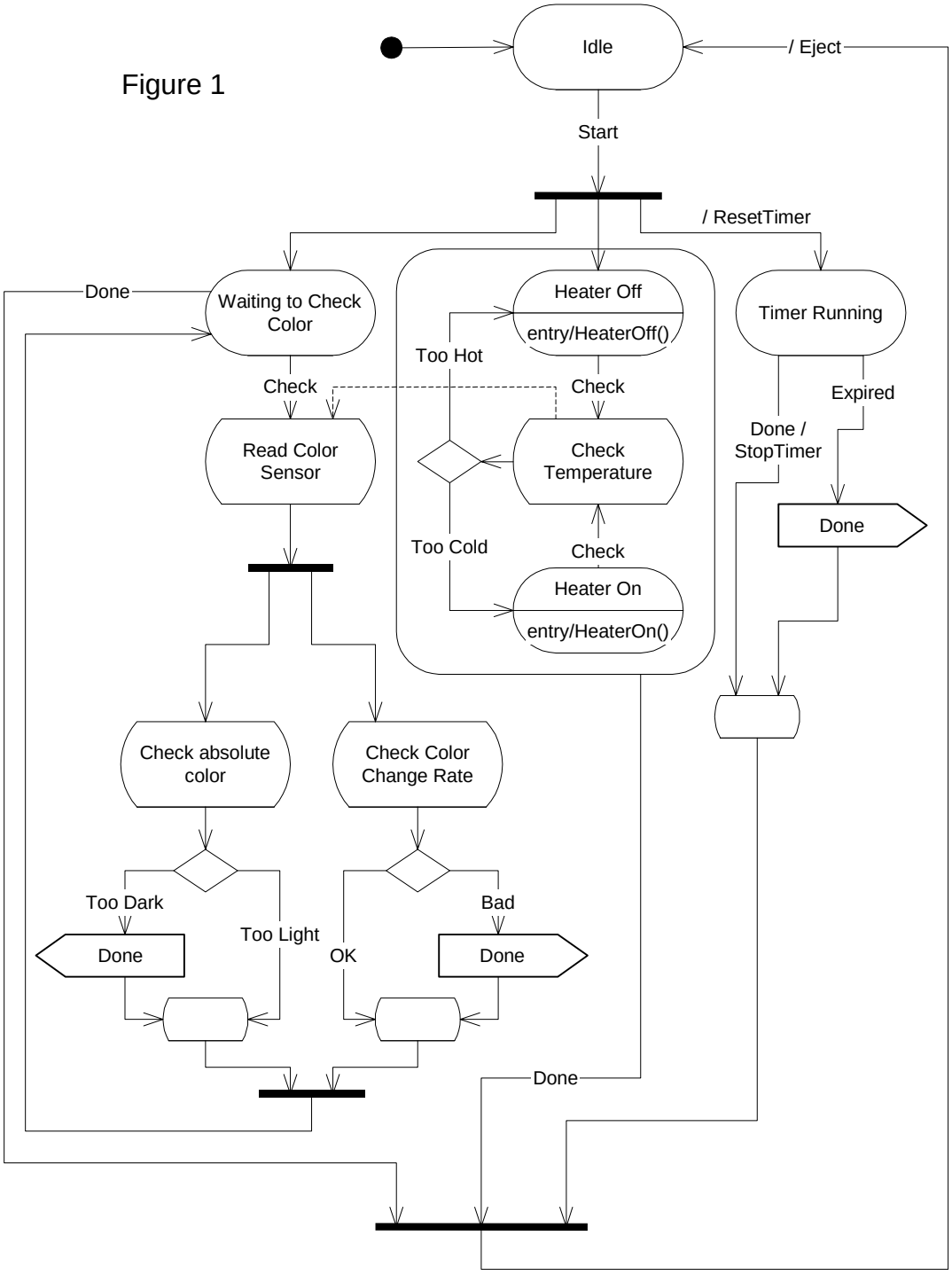
Toasting best occurs at a particular temperature. If too hot, the surface of the bread scorches but the inside remains too soft. If too cold, the toasting takes too long and the bread dries out too much. Thus, the temperature has to be carefully regulated. Moreover, as the color of the bread changes, the optimum toasting temperature rises due to decreased reflectivity of the bread. Thus, the temperature is a function of color.

Figure 1 shows the multi-threaded state machine that expresses the control logic of the toaster. Like all state machines in UML, this one begins at the initial pseudo-state, (the black ball) and immediately transitions into the Idle state.

When the user presses the lever to start the toaster, the machine receives the 'Start' event. This event triggers a *complex transition*. A complex transition is a transition that either splits or joins threads. The notation used is the heavy bar that the 'Start' arrow points to. This bar is called a *synchronization bar*.

A synchronization bar can have many incoming arrows, and many outgoing arrows. Its semantics are relatively simple. The arrows that terminate on the bar must come from states. These state are called *source states*. The arrows that leave the bar must terminate on states. These states are called the *destination states*. A complex transition fires if, and only if, all of its source states are currently occupied. Once the complex transition fires, all the source states become unoccupied, and all the destination states become occupied.

Figure 1



Astute readers will notice that the above definition implies that the state machine in Figure 1 will not wait for the 'Start' event. After all, as soon as the Idle state is occupied, the complex transition will be triggered, and its three destination states will become occupied. Actually, I should have drawn the complex transition in Figure 1 as shown in Figure 2. Notice the that the 'Start' transition causes an unnamed state to be occupied; which then triggers the complex transition. However, I consider this to be a nuisance. So whenever I draw an arrow with an event into a synchronization bar, I presume it means that the event will cause a transition to an unnamed source state of the complex transition.

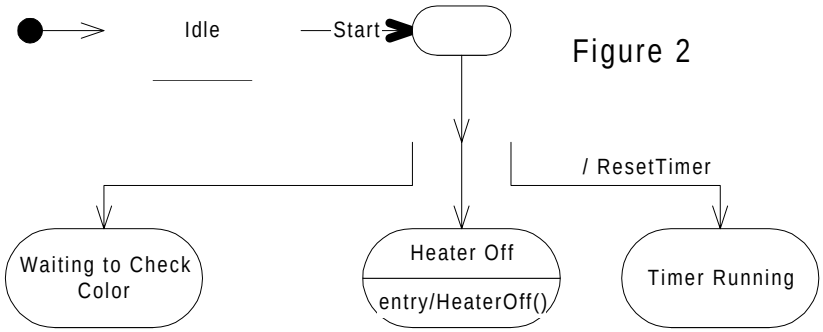



Figure 2

Once the start event occurs, and the complex transition fires, three new threads are begun. The first, and simplest of these is the timer thread. Notice the "/ ResetTimer" action attached to the arrow that connects the synchronization bar to the 'Timer Running' state. This means that the ResetTimer function will be called when the complex transition fires. The ResetTimer function restarts the timer, and sets it to expire after too much time has elapsed.

When the timer expires, the 'Expired' event takes place. Notice the arrow labeled with this event terminates on a pentagon named 'Done'. This pentagon is a 'Signal'. It means that the 'Done' signal will be sent. If you look around on Figure 1, you will see a number of transitions that are labeled with the 'Done' event.

Actually, I'm taking a bit of license here. The 'Done' signal should be sent to an object. Indeed, there should be a dashed line from the signal icon to the object which recieves it. See Figure 3.

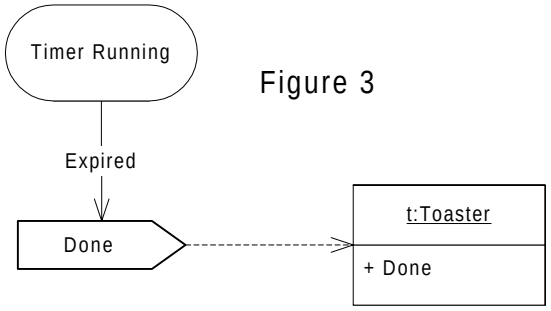



Figure 3

Alternatively, I could have used a more compact syntax that does not involve the Signal icon. The Expired event could have been written: "Expired / ^t.Done()". The 'caret' symbol (^) begins a *send clause*. A send clause simply causes a message to be sent to an object.

Returning to Figure 1. Once the 'Done' signal has been sent, An unnamed state becomes occupied. This state is a source state to the synchronization bar at the bottom of Figure 1. Thus, this thread will wait here until the other three source states of that synchronization bar are occupied.

Why the unnamed state? Why not just draw an arrow from the Signal icon to the synchronization bar? Because there is another way out of the TimerRunning state. If the Done event is received, then a transition occurs from the TimerRunning state to the unnamed state . Without the unnamed state, I would have had

two distinct arrows from this thread terminating on the synchronization bar. This is legal, since both arrows come from the same source state, namely: 'TimerRunning'; but it could be confusing since it would make it appear as though the synchronization bar would have four incoming arrows but only three source states. Thus, to keep the number arrows incoming to the synchronization bar equal to the number of source states to the complex transition, I use the unnamed states to merge the transitions. It's simply a matter of style.

The second thread that gets started by the 'Start' event is the thread that controls the heater. The heater begins in the 'Heater Off' state. Periodically, (perhaps twenty times per second) the 'Check' even occurs. Notice that the arrow labeled 'Check' points to an icon of a somewhat different shape. This is an *activity*. Activities are special kinds of states. They a single transition leading away from them. When they are entered, they execute an action, which is typically the name of the activity. When the action is complete the single exit transition is triggered. Thus, activities are very much like processes on a flowchart.

The CheckTemperature activity causes the temperature of the inside of the toaster to be compared with the appropriate set point. This set point depends upon the color of the toast, which is why there is a dashed arrow from the CheckTemperature activity to the ReadColorSensor activity. More on that later.

Once the temperature has been checked, the exit transition leads us to a small diamond. This is a *decision*, similar to the ones found on flowcharts. The decision splits the flow into two directions. If the temperature was too cold, then the transition leads to the 'Heater On' state. Otherwise it leads to the 'Heater Off' state.

The entire thread is held in an unnamed superstate that has a single transition leading out of it. This transition is triggered by the 'Done' event, and leads to the bottommost synchronization bar.

The final thread is the color checking thread. It begins in the 'Waiting to Check Color' state. When a 'Check' event occurs, the 'Read Color Sensor' activity is begun. We presume that this activity squirrels away the color reading for the 'Check Temperature' activity to use later. Once the color has been read, two more threads are started. One checks the absolute color of the toast, and the other checks the rate of color change. Both threads can signal the 'Done' event if either the toast has reached its target color, or if the toast does not seem to be changing color in the proper manner.

The two threads lead to a synchronization bar which waits until both are complete, and then causes the 'Waiting to check Color' state to be reentered. Thus, this thread loops nicely, once per 'Check' event, and then waits in the 'Waiting to Check Color' state.

The 'Waiting to Check Color' state can be exited by a 'Done' event. The resulting transition terminates on the bottommost synchronization bar. Thus, the only way that the complex transition mediated by that synchronization bar will every fire, is when all three threads finish processing the 'Done' event. When this occurs, there is a final transition back to the 'Idle' state that is accompanied by an 'Eject' action.

## *Conclusion*

The combination of complex transitions, activities, and decisions makes the UML state machine notation extremely rich. In one stroke, it combines normal state machines, petri-nets, and flowcharts into a single, easy to use notation.

The next column in this series will deal with use cases, use case diagrams, and their relationship to the rest of UML.