State machines and Statecharts

Bruce Powel Douglass, Ph.D.
Chief Evangelist
I-Logix
www.ilogix.com

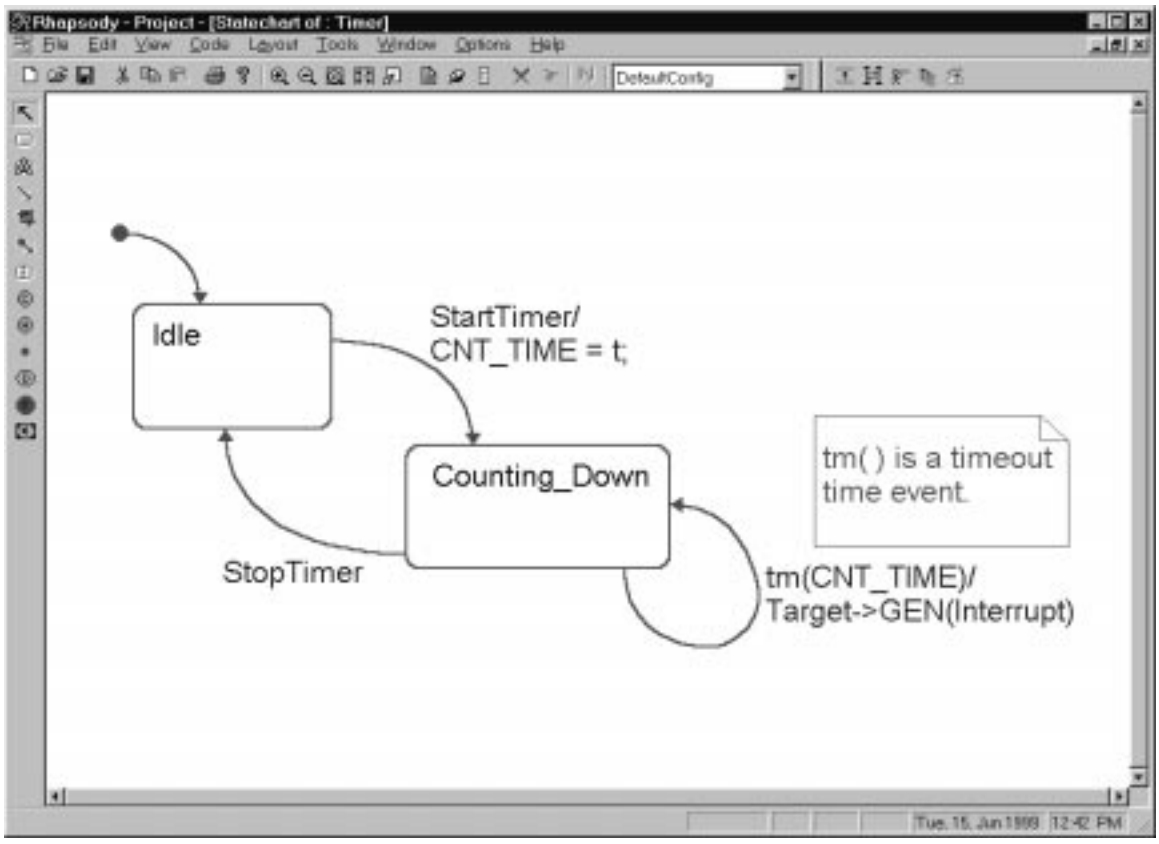
1. State Machines: Basic Concepts

A *finite state machine* (FSM) is a mathematical model of a system that attempts to reduce the model complexity by making simplifying assumptions. Specifically, it assumes

- The system being modeled can assume only a finite number of conditions, called *states.*
- The system behavior within a given state is essentially identical.
- The system resides in states for significant periods of time.
- The system may change these conditions only in a finite number of well-defined ways, called *transitions*.
- Transitions are the response of the system to *events*.
- Transitions take (approximately) zero time.

More precisely, *a state is a distinguishable ontological condition that that persists for a significant period of time*[1]*. Transitions are responses to events that move the system from state to state*. Consider a simple retriggerable one-shot timer. Such a timer is generally in one of two possible states: idle and counting down. When the timer has counted down, it issues a message (such as causing an interrupt leading to some system action), resets the timer, and returns to the state of counting down. This model is shown below:

[1] Some definitions used in this paper will differ from some other authors. If this doesn't bother you, then it doesn't bother me.



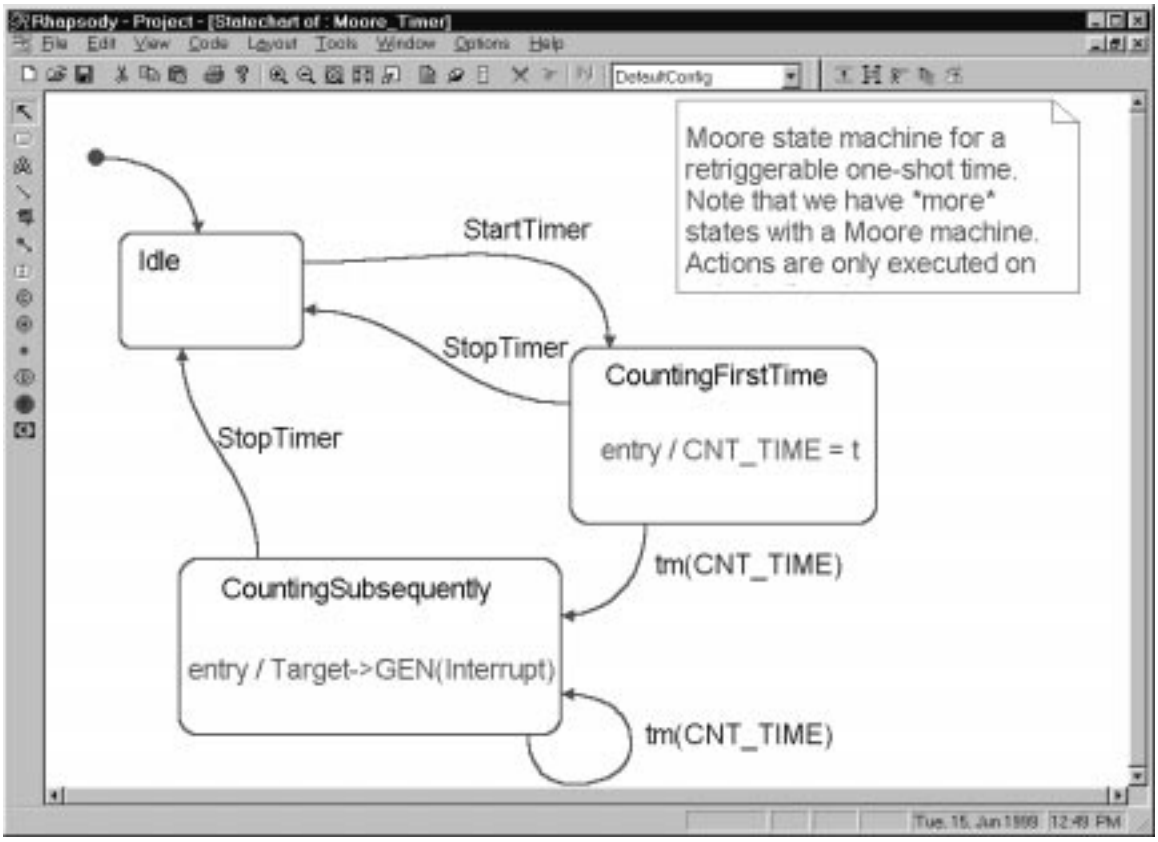
States here are shown as rounded rectangles. Transitions are directed lines beginning at the starting state for the transitions and finishing at the target state. Transitions here have names optionally followed by actions (i.e. functions or operations) executed when the transition is taken.

This state machine consists of two states and three transitions. In the Idle state, the timer isn't counting down -- it sits patiently waiting for a Start Cmd. Once it receives a Start Cmd, it transitions to the Counting Down state. As a result of the transition, two actions are performed: The count value of the timer is set and the timer mechanism itself is started. The model assumes that these actions take zero time. Naturally, they don't take exactly zero time, but relative to the timeframe that the timer is in the states, the time required for these actions is essentially zero. The solid circle indicates the starting state when the system first begins.

Once in the Counting Down state, the timer can respond to two transitions: a Timeout and the receipt of a Stop Cmd. In the former case, the timer raises an interrupt, resets the timer, and resumes the Counting Down state from the reset value. In the latter case, the timer performs the *stop timer* action and enters the Idle state.

This kind of an FSM is called a *Mealy FSM*. A Mealy FSM associates actions with the transitions between states. A Moore FSM associates actions with the states themselves rather than the transitions. In general, Moore FSMs require more states to model the same system since a Mealy FSM can use different transitions to the same state and execute

different actions. A Moore FSM must use different states to represent conditions in which different actions are performed. A Moore FSM for the timer is shown below:



You can see that an additional state is needed since the actions performed during the Counting Down state differ from the actions performed on subsequent timer restarts.

Some theorists insist that the state of a system is defined by a snapshot of all attributes (data values) contained within the system at any point in time. By such a definition, a countdown timer, such as that shown here, which counts down using a 16-bit counter would have 65,537 states ($2^{16} + 1$) in the state model[2]. However theoretically pure such a definition might be, it isn't useful. The actions performed by the timer are essentially the same (i.e. decrementing the value) within the Counting Down state, and it receives the same transitions. The behavior of the counter can most profitably be decomposed into two sets of conditions, which are Counting Down and Idle. Therefore, we will opt for the more parsimonious description and use two or three states rather than tens of thousands.

Additionally, we will define 3 different kinds of system behavior -- stateless, continuous, and state-driven -- and apply FSM methods to only the latter of these. A stateless system is a simple system which never acts differently based on its past history. For example, $\cos\frac{\pi}{2}$ returns the value 0 regardless of what value the cosine function was called with

[2] Drawing this diagram is an exercise left to the reader.

previously. It does not remember its call history, and therefore has no state. Other systems, such as PID control loops and digital filters, have feedback loops which do remember previous values, but they do not form distinct states. Instead, they can assume an infinite number of values. We say such systems exhibit *continuous* behavior. Again, some authors will point out that even control loops using real numbers use finite floating point representations and are therefore FSMs. To answer this, we present two arguments. The first is the same as is presented above -- namely, it does not behave in a distinguishably different fashion, so it is not useful to consider an attribute having the value 0.1 to have a different state than the same attribute having the value 0.100000000002. Secondly, even if it were appropriate, the actual system being modeled (the "real world") is not limited by finite floating point representation and does exhibit truly continuous behavior. Since the point of a model is to represent the system, why represent it as an FSM when it is fundamentally not?

1.1 Problems with Classical FSMs

It is difficult to represent complex systems with FSM models. The methods work well for simple, state-driven systems, but don't "scale up" to larger systems. This is unfortunate because the more complex the system to be modeled, the greater our need for modelling tools. The scalability of FSM stems from two fundamental problems: the flatness of the state model and its lack of support for concurrency.

Almost all of our conceptual modelling techniques rest ultimately on the "divide and conquer" strategy. That is, to solve a difficult problem we break it up into a set of smaller, simpler problems. This is often achieved by constructing layers of abstraction. For example, consider computer programming. All programming can be fully and completely described by the states of all the transistors in the CPU. Electrical engineers building CPUs must consider them in exactly this way. Moving up a level in abstraction, we can consider the numerical op code being executed by the CPU as defining its state. Writing a program would then be nothing more than arranging the op codes in the proper sequence, a far, far simpler way to program than individually setting transistor states. Even better than op codes is to use mnemonic names for the op codes, called assembly language. Using instructions like,

```
LD      A, 15
STA     0x0FFFE
```

rather than

```
0xC9
0x0F
0xC3
0x0FF
0x0FE
```

programming is greatly simplified. Using a high level language simplifies this process even further. The example here might be coded in C as

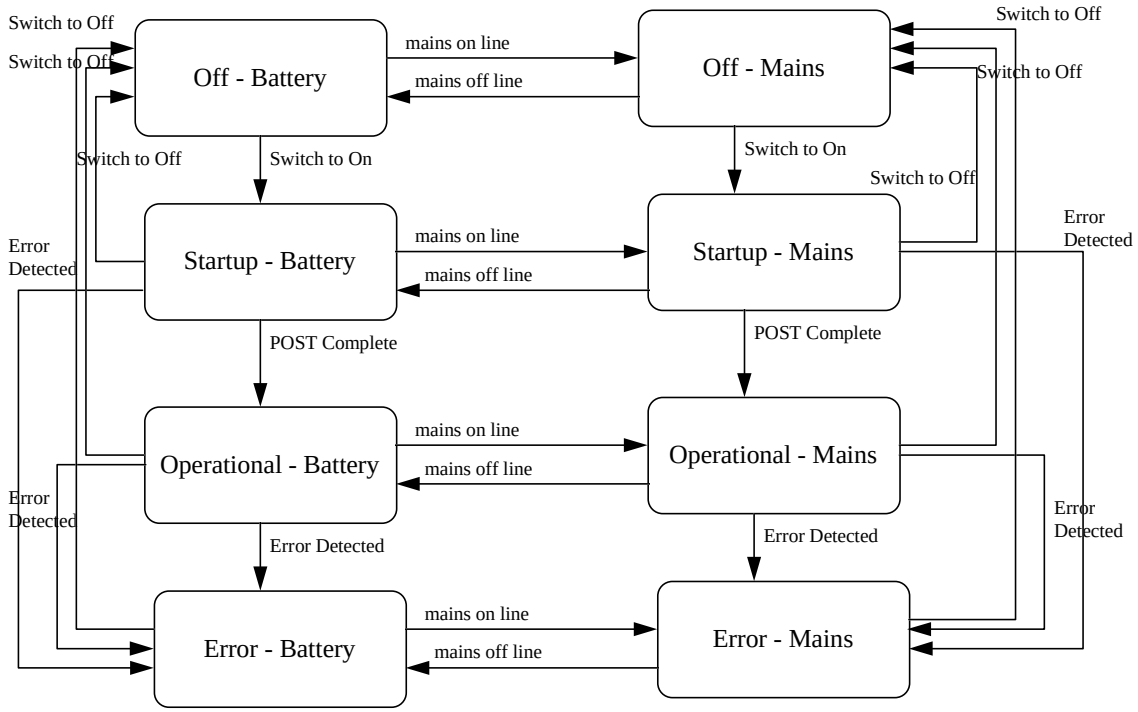
```
MyVar = 15;
```

What we have done is provided layers of abstraction to move the solution of the problem away from the domain of the implementation (transistors) to the domain of the problem (logic).

Flat state machines do not provide the means to construct layers of abstraction. All states are equally visible and are considered to be at the same level of abstraction. Consider a simple model of a car, which can be stopped, moving forward, or moving backwards. In a lower abstraction level, the pistons of the engine are compressing the gas in the firing chamber, expanding the gas in the firing chamber, filling the chamber with the gas/air mixture, or emptying it of exploded gas mixture. If we do not arrange the states in a hierarchical fashion, then we must consider "emptying gas chamber" at the same level of detail as "moving forward" which it clearly is not. The state of "moving forward" in principle contains all the states of the pistons.

Another serious problem with traditional state machines is its lack of support for concurrency. This leads to a combinatorial explosion in the number of states to model. Consider a simple system which can be thought of as in one of four states: Off, Startup, Operational, and Error. Additionally, it can be running from either batteries or from mains. The state model for this is shown below:

"Simple" Flat State Machine

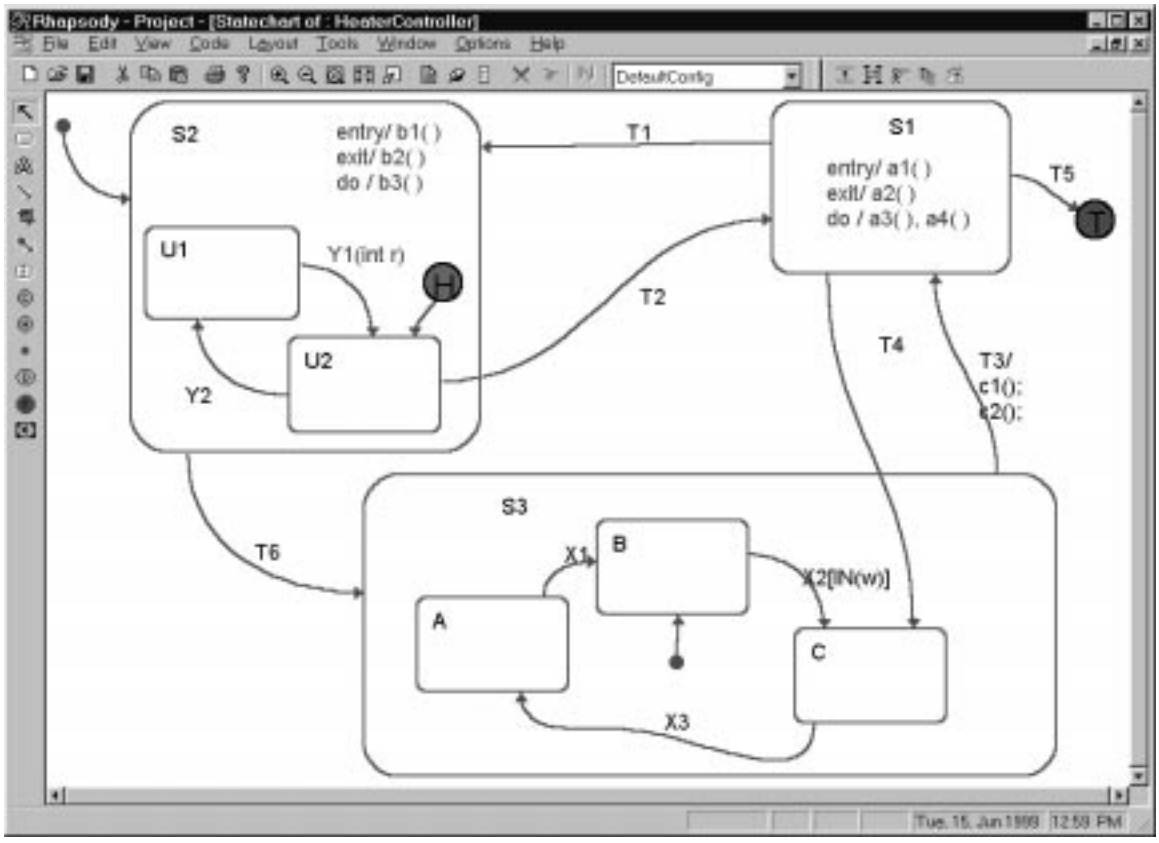


Really, the fact that the system is in Operational state is totally independent of whether or not it is running from battery or mains. However, since traditional FSMs have no notion of independence, we must combine the independent states together. This yields states like Operational-Battery and Operational-Mains. If we could model the FSM as two independent parts, the diagram would be much simplified. This is called the "combinatorial state explosion" because the modeling of multiple concurrent FSMs requires the multiplication of the number of states in each to model all conditions. This requires $O(x^n)$ states to model n state machines with an average of x states in each. Logically, it should be possible to model such a concurrent system in O(xn) states, a much simpler proposition.

2. Harel Statecharts

Harel statecharts[3] are an attempt to overcome the limitations of traditional FSMs while retaining their good features. Statecharts include both the notions of nested, hierarchical states and concurrency while extending the notion of actions.

The essential syntax of statecharts is very much like Mealy-Moore FSMs:



[3] Harel, David: *Statecharts: A Visual Formalism for Complex Systems* in "Science of Computer Programming", 8 (1987) 231-274.

Harel statecharts are represented as rounded rectangles[4]. Directed lines indicate transitions from state to state. Note that there are a number of differences between Harel statecharts and traditional state diagrams. The most noticeable is the appearance of states within states. The outer enclosing state is called a *superstate*. The inner states are called *substates*. For example, state S2 contains two substates, U1 and U2. While the system is in state S2, it must be in *exactly* one of the nested substates as well. The nested states may be shown physically within the superstate, or the superstate may be depicted on another diagram altogether. This allows CASE tools to provide hierarchical "zooming" capabilities to control user visibility of detail.

Transitions may be drawn to the specific substate, such as transition T4, or may be drawn to the containing superstate, such as transition T1. In this latter case, some rules must be applied to determine which substate is entered. When there is ambiguity, an initial state must be identified using the filled circle, just as in traditional FSMs. Additionally, a *history* annotation may be included, as in state S2. When this icon is present, it indicates that the default state is the last active substate for that superstate. If the last substate was U2, and transition T2 is taken, when a subsequent transition T1 is made, substate U2 will be reentered. When both an initial and history are indicated, then the initial state holds true only for the first time the superstate is entered. Thereafter, the last active state is used.

Transitions may be made to and from either a superstate or a substate. When a transition is indicated to a superstate, then the initial or last active substate is entered, depending on the annotations. When a transition is indicated from a superstate, it means that the transition applies to *all contained substates*. This is a great help in simplifying diagrams since a single transition from a superstate represents transitions from each of its contained substates.

In Harel statecharts, behavior may be more elaborate than in Mealy-Moore FSMs. Transitions can have actions, just as they do in Mealy FSMs. Additionally, states may have both *entry* and *exit actions* as well as *activities*. Entry actions are operations that are performed when the state is entered. Exit actions are performed when the state exits. This is indicated in the state by the word "Throughout:". Actions are still assumed to take an insignificant amount of time, while activities are performed as long as the state is active. This rich behavioral modeling allows efficient representation of a wide set of behaviors.

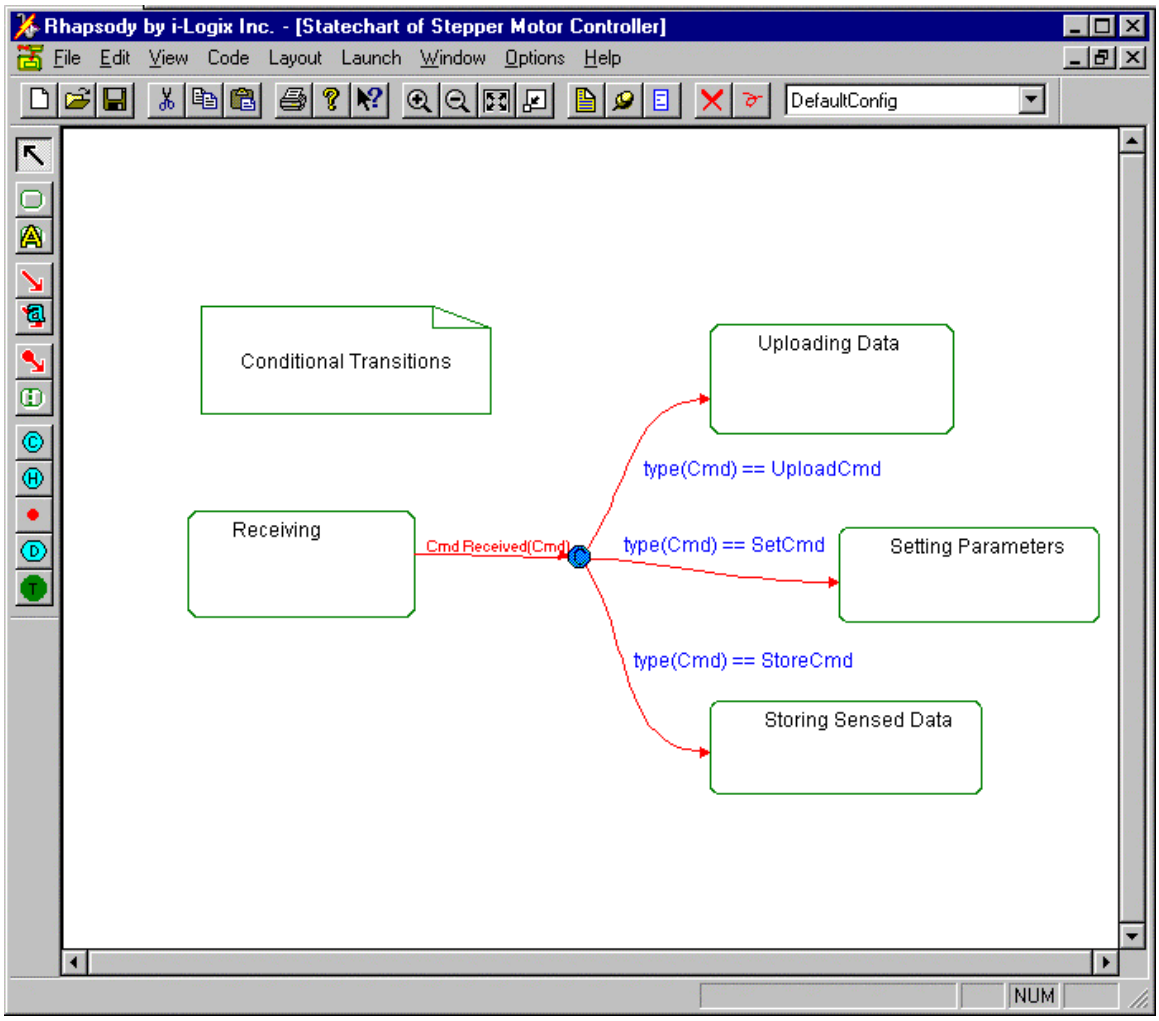
Because states can be nested, entry into a superstate will cause the execution of the superstate's entry actions as well as the entry actions of the substate it enters. Internal transitions within the superstate do not reexecute the superstate's entry actions, but do cause execution of the exit actions for the substate being left as well as the entry actions for the substate subsequently entered. When an exit is made from the superstate, the exit

[4] The Harel notation from the Unified Modeling Language (rev. 0.8) is used here. It differs slightly from that defined in Harel's original work. Interested readers are referred to Harel's original work for more information.

actions for both the terminating substate and the superstate are executed. The normal order of execution is that entry actions of the superstate are performed first, followed by the entry actions of the nested state. Exit actions are performed in reverse order -- the substate exit actions are executed first, followed by those of the superstate. States may be nested arbitrarily deeply and these rules apply recursively.

Transitions may have parameters and guards, as well as actions. Some authors model events (occurrences which give rise to transitions) as having no data. We define events as *occurrences which cause transitions in an FSM* and permit them to contain data. Events are modeled as a particular type of message that cause state transitions. Another kind of message is the data message, which does not cause state transitions. Either message type may contain an arbitrary amount of information. This data may be shown within parentheses exactly how is appears in a function parameter list.

It is possible to take different transitions from a given state based on the same event when the event contains data used to discriminate the path. This is called a conditional transition. The conditional icon is a diamond. Each transition emanating from the conditional icon is marked with the specific value(s) which cause that transition to be taken.

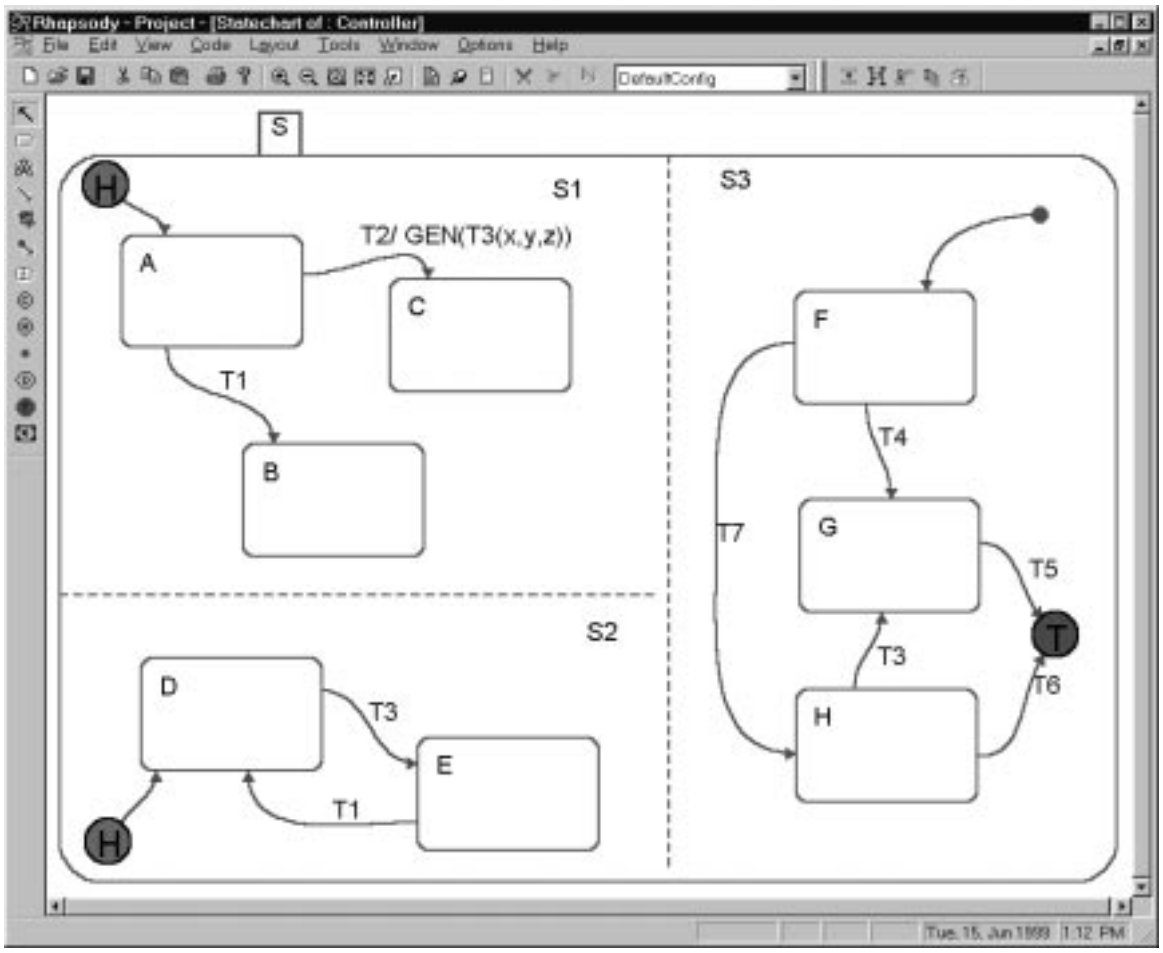


Guards are conditions which must be met for the transition to be taken even when the event causing the transition has occurred. Guards are shown in square brackets to distinguish them from event data (which is shown in parentheses). A common guard is that some other concurrent state machine must be in a certain state. This is normally shown as [in(G)] indicating that the other state machine must be in state G for the transition to be taken.

Actions may be either operations executed or other transitions initiated. Transitions caused as a result of another transition occurring are called *propagated transitions*. This allows transitions in one concurrent state machine to influence others. In UML 1.3, sending an event is a kind of action and is included in the action list.

Transitions may not only be *propagated* from one state machine to another, but may also be *broadcast* to all state machines simultaneously. Broadcast transitions are shown expediently by using the same name in multiple concurrent state machines. This implies that transitions names must be unique throughout the entire system. This can be achieved

by either forcing a flat namespace or by using a relative distinguished naming (RDN) strategy. An RDN strategy uses the state hierarchy to uniquely identify the nesting of states by separating nesting levels with a period. For example, in the Basic Statechart Syntax diagram above, transition Y2 could be referred to using its RDN as S2::Y2, since it is nested within S2.



Concurrency in statecharts is shown using dashed lines, as shown above. The overall state being modeled is S, and is broken up into three distinct, concurrent state machines, S1, S2, and S3. Each of these is an independent state machine with its own initial state, history, and behavior. In such a concurrent system it is important to note that *while in S, the system is in one state each from S1, S2, and S3 at the same time.*

In this figure we see examples of both broadcast and propagated transitions. Transition T1 appears in both S1 and S2, just as the transition T3 appears in both S2 and S3. Both of these are broadcast to the concurrent state machines. Transition T3 is propagated from transition T2 in state machine S1. When T2 occurs in S1, it causes T3 to appear in both S2 and S3.

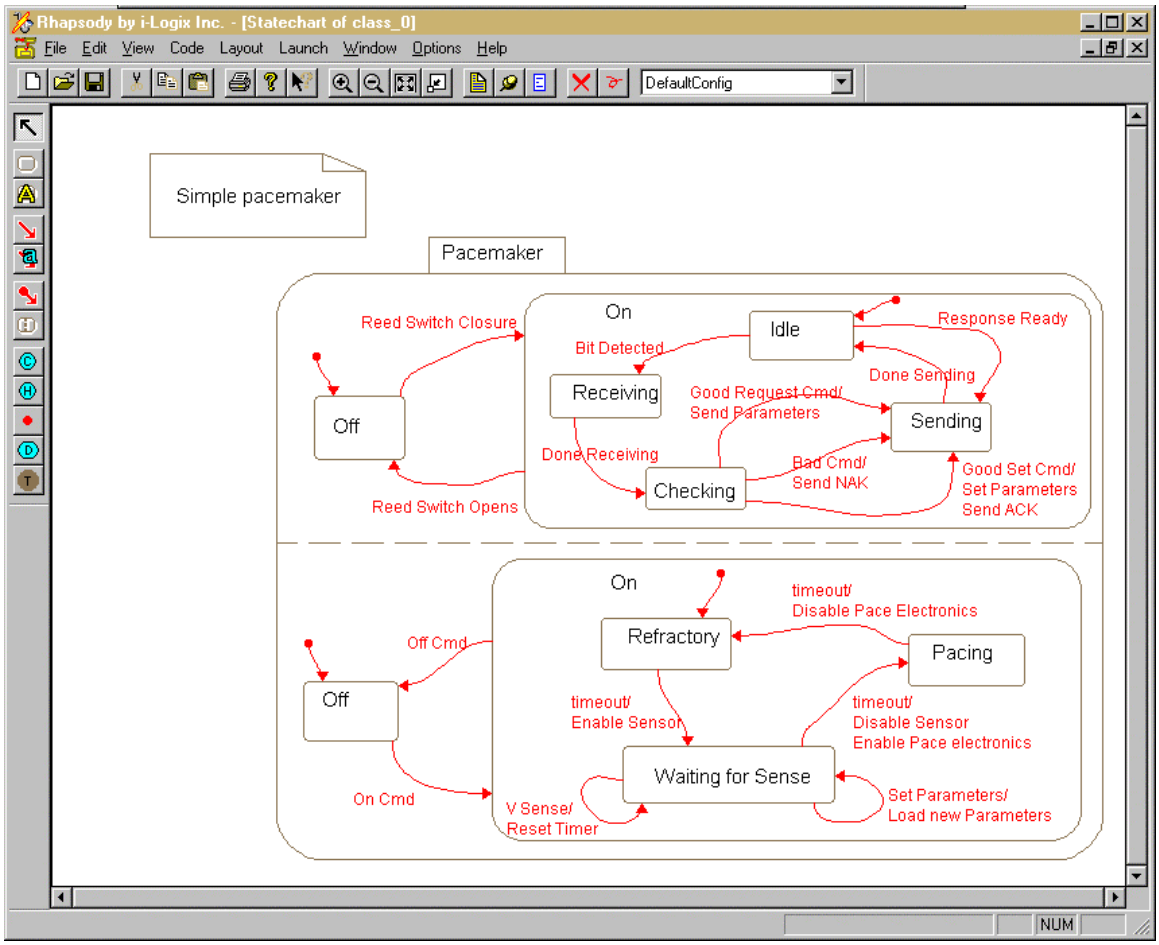
A simple pacemaker provides an example of a Harel statechart. This pacemaker can receive commands from a programming device to

- set pulse width, pulse amplitude, and pacing rate.
- transmit its current parameter settings.
- Set pacing parameters.
- Enable or disable pacing.

If the pacemaker is commanded to set parameters, the command is checked, and if valid, the parameters are set for the next pacing cycle and an ACK is returned to the programmer. If the pacemaker receives a command to transmit its pacing parameters, then they are transmitted without an additional ACK. Any command which is illegal or invalid results in the pacemaker transmitting a NAK.

At the same time, it paces the heart using a VVI pacing mode. In such a pacing mode, the electrical activity in the ventricle is sensed. If an intrinsic heart beat is detected before the timeout (which depends directly on the pacing rate) occurs, then the pacemaker resets the timer and waits for the next beat. If a timeout occurs, then the sense hardware is turned off (to protect it against current inrush), and an electrical pace is delivered to the ventricle. After waiting a suitable period of time to allow the current to dissipate, known as the *refractory time*, the sensor electronics are reenabled and the pacemaker again waits for a heart beat or timeout event.

A Harel statechart for this pacemaker is shown below:



Note that the communications system is an FSM which operates concurrently with the pacing engine. It would be inappropriate to disable pacing while communication occurs. The two FSMs are synchronized by the Set Parameters transition. It is initiated by the Good Set Cmd received by the communications subsystem, and results in a propagated transition Set Parameters in the pacing FSM.

2.1 State Transition Encyclopedia

As with both structured and object-oriented methods, not all information must be put on a statechart. Although statecharts provide a means for putting entry and exit actions and activities in a state icon, in complex systems this can make the diagram unreadable. The same is true of transitions with elaborate parameters, guards, propagated transitions, and actions. Further, most software processes for highly reliable systems must trace requirements into code, and this is more difficult when the requirement is on a diagram than when it is in text. For these reasons, the details of the states and transitions are often captured in a *state encyclopedia* and the diagram itself is used to help navigate the textual descriptions of the states and transitions. A typical state encyclopedia will have each state and transition defined in a short specification as shown below:

State Name: <name>
Applies to which objects: <object names>
Description: <text description>
Entry Actions: <comma separated action list>
Exit Actions: <comma separated action list>
Activities: <comma separated activity list>

Transition: <name>
Applies to which objects: <object names>
Description: <text description>
Source of event: <name of event>
Parameter list: <comma separated list of data parameters>
Guard conditions: <comma separated list of guards>
Propagated transitions: <comma separated list of transitions>
Actions: <comma separated action list>

3. State Tables

Another popular way to represent FSMs is with a state table. This table is usually organized with the starting states along the left edge and the transitions along the top. The contents of the cells are the target states, along with any unique actions taken with that instance of the transition. Occasionally, one sees a state table with the initial state along the left edge and the target state along the top with the contents of the cells being the transition.

State tables are used instead of or in conjunction with state diagrams or statecharts. They provide a space-efficient means for representing a large number of states and transitions. More importantly, they provide a different view of the FSM. With a statechart, the overall structure of the state space is clear and it is relatively simple to navigate a sequence of state transitions. However, it is not equally clear when transitions are missing. In a state table, the structure of the state space is more obscure, but missing transitions are much more obvious. For this reason, some authors, such as Shlaer and Mellor, recommend that both diagrams and tables are done.

Concurrent FSMs are typically represented using different tables, so that a given table is within the same thread of execution. Just as with statecharts, transition names common to multiple state tables indicate synchronization, either via broadcast or propagated transitions. Because structure is less visible with state tables, it is even more important to have an encyclopedia defining the states and transitions in detail.

An example state table is shown below for the pacemaker example:

Table 1: State Table for Pacemaker Communications

	Reed switch closure	Reed switch opens	Bit detected	Done Receiving	Response Ready	Done Sending	Good Request Cmd	Bad Cmd	Good Set Cmd

Off	Idle								
Idle		Off	Receiving		Sending				
Receiving		Off		Checking					
Checking		Off					Sending	Sending	Sending
Sending		Off				Idle			

Table 2: State Table for Pacing Engine

	Pace Start Cmd	Pace Stop Cmd	Timeout	Ventricular Sense	Set Parameters	Done
Off	Waiting					
Refractory		Off	Waiting			
Waiting		Off	Pacing	Waiting	Waiting	
Pacing		Off				Refractory

4. Scenarios and State Models

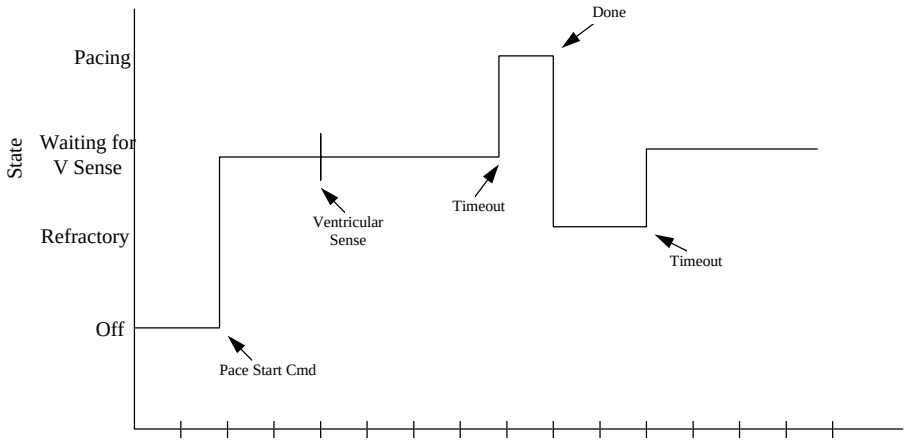
A state diagram provides a static view of the entire state space of a system. What it does not show are typical paths through the state space as the system is used. These typical paths are called *scenarios*. Scenarios may not visit all states in the system nor activate all transitions, but they provide an order-dependent view of how the system is expected to behave when actually used. This paper will provide two methods for showing scenarios and illustrate how they tie in to the static state machine defined by state diagrams and tables. The first is the timing diagram, which is best used when strict timing must be shown. The other is the message Sequence Chart, which shows order but not strict timing.

4.1 Timing Diagrams

Electrical engineers have used timing diagrams for a long time. A timing diagram is a simple representation with time along the horizontal axis and state along the vertical. Of course, electrical engineers usually only concern themselves with two states: On and Off. Software engineers can use timing diagrams just as easily on more elaborate state machines to show the changes of state over time.

Software timing diagrams (or just “timing diagrams” for short) depict state as a horizontal band across the diagram. When the system is in that state, a line is drawn in that band for the duration of time the system is in the state. The time axis is linear, although special notations are sometimes used to indicate long uninteresting periods of time. The simple form of a timing diagram is shown below:

Simple Timing Diagram

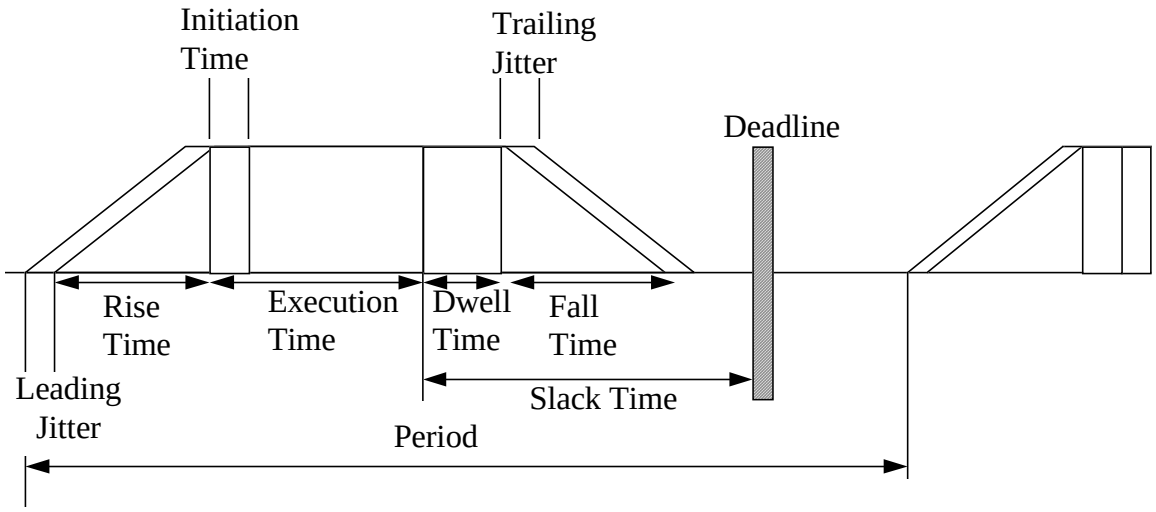


This timing diagram shows a particular path through the pacing engine state machine. The pacing engine begins in the Off state and remains there until it receives a command to enter begin pacing. At this point, it jumps to the Waiting for V Sense state. The vertical lines connecting states show that the timing for the transition is zero relative to the scale of the timing diagram. Later, a ventricular sense is detected (as shown by the transition annotation on the diagram) and the engine returns to the Waiting for V Sense state. Sometime later, the timeout occurs and the engine enters the pacing state. In this state, the engine is actively putting an electrical charge through the heart muscle. When the pacing pulse width is complete, the system transitions to the Refractory state. Once this times out, the system again enters the Waiting for V Sense state.

In this simple form, only a single object (or system) is represented. It is possible to show multiple objects on the same diagram. By separating these with dashed lines, the different (and possibly concurrent) objects can be clearly delineated. Propagated transitions can be clearly marked with directed lines showing event dependency.

Other extensions to timing diagrams can be shown as well. The figure below shows the complex syntax available for timing diagrams. Although it is only shown for a two-state system, it applies to more elaborate state machines as well.

Complex Timing Diagrams



For state processes that reoccur periodically, a number of state characteristics may be shown. These include:

- period — The time between initiations for the same state.
- deadline — The time by which the state must be exited and a new state entered.
- Initiation time — The time required to completely enter the state (i.e. execute state entry actions)
- execute time — The time required to execute the entry and exit actions and the required activities.
- dwell time — The time the object remains in the state after the execute time before the state is exited. Includes time for exit actions.
- slack time — The time between the end of actions and activities and the deadline.
- rise time — The time required for the transition into the state to complete.
- fall time — The time required for the transitions out of the state to complete.
- jitter — Variations in the start time for a periodic transition or event.

When the transition time is not zero, it is shown using a slanted line indicating the time that it takes to complete the transition. This is the rise and fall times for the transitions.

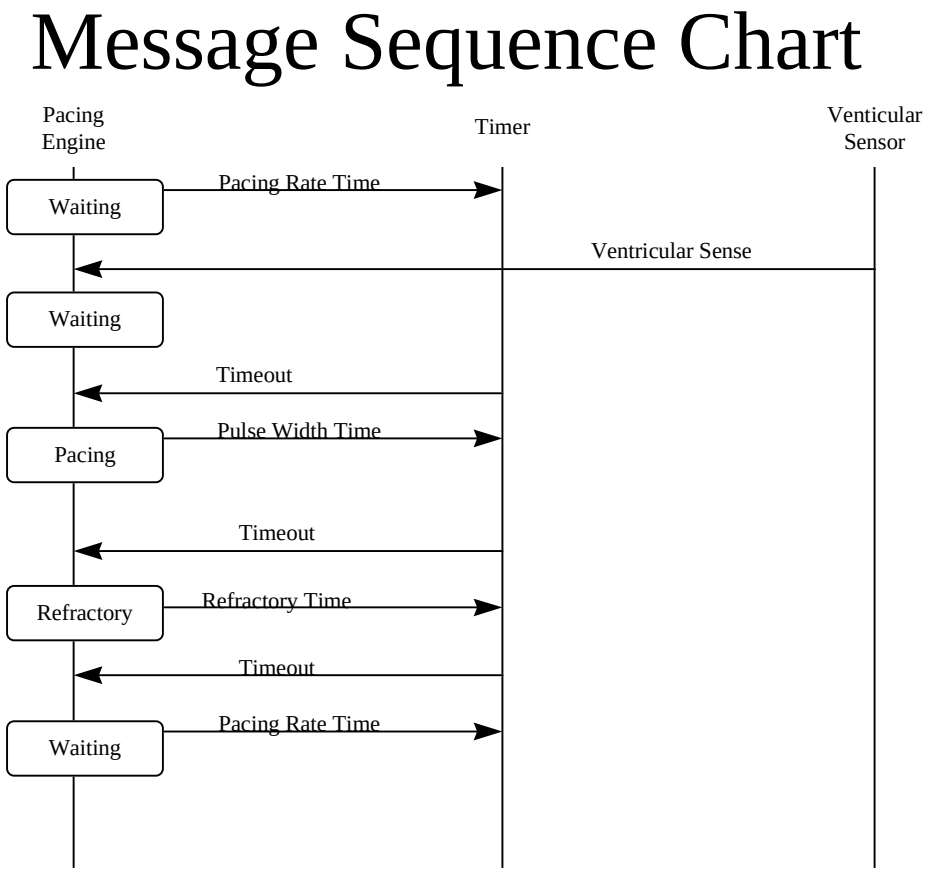
4.2 Message Sequence Chart

Message Sequence Charts (MSCs) are a more common way to show scenarios. MSCs use vertical lines to represent the objects participating in the scenario and horizontal directed lines to represent the messages sent from one object to another. Time flows from top to bottom: that is, messages shown lower on the page take place later that ones above it.

MSCs show only sequence, not absolute time. Since the time axis is not linear, most methodologists permit the inclusion of textual timing annotations when absolute time is particularly important.

We have extended MSCs in a few fundamental ways. First, we add small filled circles at the originator object for broadcast messages. This allows clear distinction of which object initiated a message that goes to multiple targets. Second, only some of the messages drawn will cause a change in state (event messages). The target object is assumed to remain in the same state until another state box appears on the line. We optionally add state boxes on the object lines to show when the object undergoes a state change as a result of receiving a message.

The same scenario is shown below using an MSC. Note that the presence of the other objects (Timer and Ventricular Sensor) encourages us to add more messages which naturally arise from looking at the problem somewhat differently.



5. Implementing State machines

There are a number of ways to implement a state machine in software. The most common is to provide a single scalar variable called a state variable and use this as the

discriminator in a switch statement. Each case clause in the switch statement can implement the various actions and activities. For example, consider the simple retriggerable one-shot timer state machine presented earlier. It might result in C++ source code something like this:

```
void FSM(int &timer_state, message msg) { // C programs would use int *timer_state
        switch (timer_state) {
                case IDLE_STATE:
                        switch (msg.msg_type) {
                                case START_CMD:
                                        timer.countValue = msg.cmd
                                        timer.start();
                                        timer_state = COUNTING_STATE;
                                        break;
                                default:
                                        // do nothing
                                        break;
                                }; // end switch msg
                        break;
                case COUNTING_STATE:
                        switch (msg.msg_type) {
                                case TIMEOUT:
                                        sw_interrupt(xx);
                                        timer.start();
                                        break;
                                case STOP_CMD:
                                        timer.stop();
                                        timer_state = IDLE_STATE;
                                        break;
                                default:
                                        // do nothing
                                        break;
                                }; // end switch msg
                        break;
                default:
                        cout << “Illegal state value “ << endl;
                        break;
        }; // end switch
}; // end FSM function
```

Another approach is to write a function that accepts a state table and executes a transition on it. This solution is somewhat more work initially, but more flexible and capable for larger state machines. The secret to this approach is the structuring of the state table. It must not only contain a potentially sparse state x transition array containing the new target state, but also entry and exit actions for the states, activities for the states, and actions and guards for the transitions. The exact structuring of the data will depend on the problem being solved. Generally, the code looks something like this when implemented in C++:

```
#include <stddef.h>

// crude, but it's simple
#define MAX_ACTIONS 10
#define MAX_STATES 10
#define MAX_TRANSITIONS 10

typedef int TstateID;
typedef int TtransitionID;
typedef void (*f_ptr)();  // pointer to a function taking no arguments
                                    // and returning no values

void DoNada(void) {
        // do nothing
};
```

```
class transition {
public:
        TtransitionID ID;
        TstateID targetState; // where we end up
        f_ptr action[MAX_ACTIONS]; // up to 11 actions
        transition(void) {
                for (int j=0; j<MAX_ACTIONS; j++)
                        action[j] = &DoNada; // point to null func
        };
};


class state {
public:
        TstateID ID;
        f_ptr entry_action[MAX_ACTIONS];
        f_ptr exit_action[MAX_ACTIONS];
        f_ptr  activity[MAX_ACTIONS];
        state(void) {
                for (int j=0; j<MAX_ACTIONS; j++) {
                        entry_action[j] = &DoNada;
                        exit_action[j] = &DoNada;
                        activity[j] = &DoNada;
                };
        };
};

// In class stateTable, the first template value index is the number
// of states. The second is the number of transitions.
// The contents of each cell in the state table is a transition
// The currentState variable holds the ID of the current state
// which coincides with its index.

class stateTable {
public:
        stateTable(int startState = 0): currentState(startState) { };
        TstateID currentState; // current active state
        TstateID cell[MAX_STATES][MAX_TRANSITIONS]; // contains ID of target state
        state stateList[MAX_STATES];
        transition transitionList[MAX_TRANSITIONS];
};

class FSM {
public:
        void processFSM(stateTable s, int transitionID) {
                int j;
                if ((transitionID >=0) && (transitionID <MAX_TRANSITIONS)) {
                        if s.cell[s.currentState][transitionID] > 0 { // valid transition
                                // now, execute exit actions, transitions actions and
                                // entry actions
                                for (j = 0; j<MAX_ACTIONS; j++)
                                        s.stateList[s.currentState].exit_action[j]();

                                // change current state
                                s.currentState = s.cell[s.currentState][transitionID];

                                // execute transition actions
                                for (j = 0; j<MAX_ACTIONS; j++)
                                        s.transitionList[transitionID].action[j]();

                                // new state entry actions
                                for (j = 0; j<MAX_ACTIONS; j++)
                                        s.stateList[s.currentState].entry_action[j]();
                        }; // end inner if
                }; // end outer if
        }; // end method processFSM
}; // end

int main(void) {
        state a; // creates a state with up to 10 entry and exit
                         // actions and 10 activities
        transition b; // creates a transition with up to 5 actions

        stateTable s; // creates a 5 state x 10 transition table
```

```
        return 0;
};
```

The FSM class may be then serve as a base class from which subclasses are derived. Derived classes are then a specialized type of state machine which know how to process state transitions. Alternatively, classes wanting to model state behavior may use the facilities of a single centralized FSM object and pass state tables off to it for processing.

This particular set of classes takes the easy way out in several cases so as not to obscure the code. Fixed sized actions lists are used, but several alternative approaches could be used instead. Templates could make custom-sized classes. Probably best is to use a linked list for the action lists because they only need to be sequentially accessed. Also, guards are not checked, but they should be.

The processFSM method accepts a state table and a transition. It applies the state exit actions, changes to the new states, executes the transitions actions, and then executes the new state entry actions. It assumes that the function (action) pointers may be called freely with no ill effects. The code also assumes that transitions all have unique IDs.

6. References

[1] Douglass, Bruce Powel *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns* Addison-Wesley, 1999

[2] Douglass, Bruce Powel *Real-Time UML: Developing Efficient Objects for Embedded Systems* Addison-Wesley, 1998