Extended State Diagrams and Reactive Systems

Designing systems for unpredictable inputs

Doron Drusinsky

As the cost of hardware continues its downward spiral, the application of embedded electronic control continues to accelerate into new domains. Many of these new applications have complex designs, however, and graphical tools are proving to be the most efficient way of specifying, designing, and documenting such systems. In particular, graphical tools are well suited for the design of systems based on state machines or data flow. Consequently, in this article I'll examine the use of extended state diagrams (also known as "Harel diagrams") for the design of reactive systems—those which endlessly react to a plurality of partially correlated entities in their environment. To illustrate extended state diagrams, I'll base my discussion on BetterState, a graphical state-machine design tool with a built-in code generator my company has developed.

Transformational systems are those invoked when the inputs are ready and the outputs are produced after a computation period; see Figure 1(a). Examples are voice-compression systems (software or hardware) or (sub)systems which calculate the square root of input. Top-down decompensation is a natural design methodology for transformational systems because it breaks down complex input/output (functional) relationships into simpler, more manageable ones. Similarly, conventional programming and system-level specification languages are transformationally oriented; they cater to top-down functional design.

Doron, who holds several patents in the areas of state-chart synthesis and finite state machine optimization, is the president of R-Active Concepts. Doron can be contacted at doron@infoserv.com.

Fundamentally different from transformational systems are reactive systems such as Figure 1(b), in which inputs are not ready at any given point in time. A typical reactive system is a traffic-light controller which never has all its inputs ready—the inputs arrive in endless and perhaps unexpected sequences. It is virtually impossible to write a transformational program that implements a controller such as this. In fact, most controllers are by definition reactive, not transformational, with application domains ranging from military, aerospace, and automotive applications to DSP, ASIC design, medical electronics, and similar embedded systems. Just about every system has a reactive component, because a system is seldom isolated from its environment. On the contrary, the reason the system exists is typically to collaborate or interact with some entity or entities in its environment. Such collaboration is done by sending, receiving, recognizing, and rejecting sequences of symbols—a reactive behavior.

Finite state machines (FSMs) and state diagrams (FSM's visual counterpart) have traditionally been used to specify and design reactive (sub)systems. They are well known, well accepted, highly visual, and intuitive. Their ability to describe finite and infinite sequences, combined with their visual appeal, made FSMs one of the most commonly accepted formalisms in the electronic industry. State diagrams are easier to design, comprehend, modify, and document than the corresponding textual approach. But FSMs and state diagrams haven't changed much over the past 30 years and suffer from limitations when applied to today's reactive applications:

- FSMs are flat. They do not cater to top-down design and information hiding.
- FSMs are purely sequential, whereas applications are not. Modern controllers need to react to signals to and from a plurality of entities in the environment. Consider an answering machine controller specified to cater to a "second-call waiting" situation in addition to the "first caller." A conventional FSM needs to account for all possible combinations of states catering to the first and second callers, which leads to the well-known state-blowup phenomenon.
- Text-based entry methods, which are by definition sequential, cannot effectively capture concurrent behavior. Therefore, drawing state diagrams on paper and entering them textually is no longer effective.
- Top-down design concepts require interactive software to enable the user to manipulate and browse through complex designs.

Because of such limitations, FSMs have been used sparingly in recent years. Compensating for these limitations are extended state diagrams (or "statecharts"), designed by David Harel and described in his paper "Statecharts: A Visual Approach to Complex Systems" published in the *Science of Computer Programming* (1987). (Harel, who was my PhD advisor, is also the author of *Algorithmics: The Spirit of Computing*, Addison-Wesley, 1987.) While addressing the hierarchy, concurrency, priorities, and synchronization within state diagrams, extended state diagrams retain the visual and intuitive appeal inherent to state diagrams.

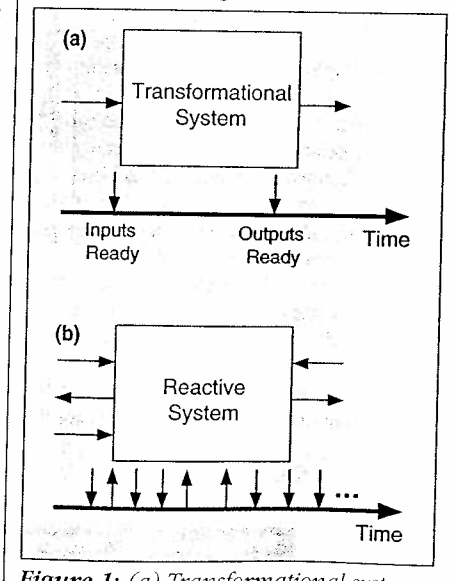

Figure 1: *(a) Transformational systems; (b) reactive systems.*

(continued from page 72)

A Traffic-Light Controller Example

To illustrate how you design systems around extended state diagrams, I'll use a typical traffic-light controller as an example. The specification for this traffic-light controller (TLC) is as follows:

- There are two directions, *Main* and *Sec*, with alternating lights.
- Lights alternate based on a *Timeout* signal, which can be read from the *Timeout* variable.
- Initially, all lights flash yellow. Upon reset going low (0), the on-going operation can start. When reset goes high (1), the system must reset into this initial state.
- The priority order is: *Reset*, *Timeout*, all other signals.
- A *Camera*, positioned in the *Main* direction, operates only when the *Main* direction has the red light. It should take shots of cars going through a red light in the *Main* direction, unless a policeman is present (signal *Manual_on* is 1).
- When the *Main* direction has the red light, and four or more cars or a truck that follows one or more cars are waiting in that direction, *Main* gets the green light.
- When the *Main* direction has the red light and three cars are waiting in that direction, *Camera* should shoot.

The extended state diagram in Figure 2 realizes the highest level of the TLC's behavior. It captures the most high-level events and state transitions. State *Red2Main*, however, has "hidden" information that can be accessed by double clicking on *Red2Main*. Such information hiding makes the diagram more readable and manageable. Note how the transition labeled *Reset* takes effect no matter what the present state is within *on_going*. Such high-level transitions are a powerful tool for managing work between designers. Any change made to *Red2Main* by one designer is automatically captured by the high-level transition designed by another.

Figure 3 shows two concurrent threads of control, one capturing the state sequence for the *Camera*'s (sub)state machine, and the other capturing the state sequence for the *Counter*'s machine. Concurrency here has little to do with real time—we are not specifying how fast the design will work. Concurrency, in this case, is related to independent activities. The *Counter* and *Camera* are independent most of the time, but are always active in the *Red2Main* state. The semantics of extended state diagrams implement the desired behavior, without the designer explicitly implementing suspend-and-resume behavior. Despite their independence, the specification dictates some correlation between *Camera* and *Counter*. When the *Counter* counts two cars waiting in *Main*, it tells the *Camera* to shoot. The transition from *c_2* to *Shoot* effects that behavior precisely, without any message passing.

Code Generation

Code generation for conventional FSMs is straightforward: A case statement (C *switch* statement) over all possible states is a common representation. Similarly, code generation for concurrent FSMs is no more than a set of code blocks, one per each FSM. However, code generation for extended state diagrams is more complicated because:

- An extended state machine has flexible concurrency. When the TLC example is in the *Red2Main* state, there are two active threads of control (*Camera* and *Counter*); in the *YellowAll* state there is only one active thread.
- Hierarchy is an additional source of potentially concurrent transitions.
- Due to concurrency, a case statement is inappropriate: More than one state might be active at any given time.

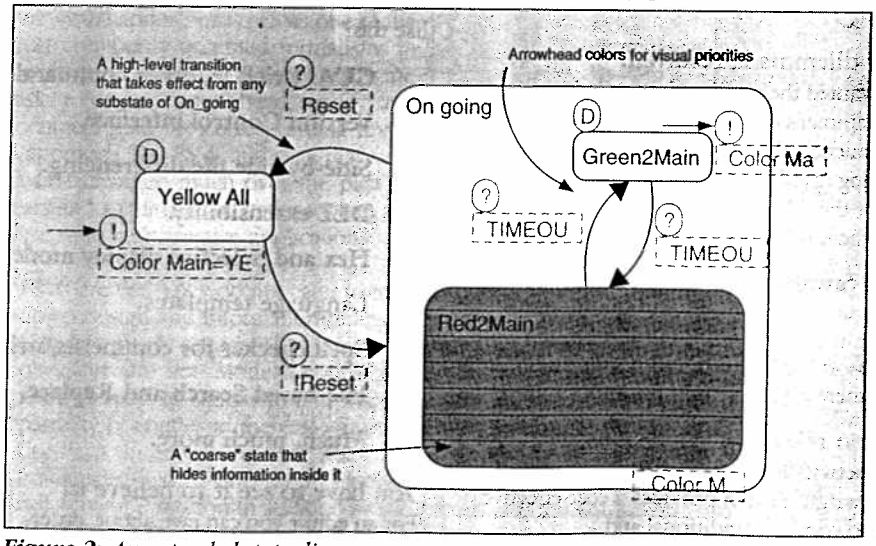

Figure 2: *An extended state diagram.*

(continued from page 74)

- Support for visual synchronization and visual priorities is nontrivial.

For these reasons, handcoding for extended state diagrams can become a confusing and time-consuming task.

I have invented two code-generation methods for extended state diagrams. The first method, coinvented with David Harel, is more hierarchical in nature; the code generated preserves the hierarchy in the original diagram. The second method, currently in use by BetterState, flattens the diagram in an attempt to generate simpler code. For this reason, the BetterState code generates is simple in structure; it is no more than a large set of If statements. The process of generating this code however, is entirely nontrivial.

The BetterState code generator that builds the extended state diagram is based on the following code-generation methodology:

- Additional code is not required to run the design on a particular device. For example, the C code generated doesn't require a real-time operating system to implement concurrency and hierarchy, and is compatible with any processor or microcontroller equipped with a C compiler.
- There is a one-to-one mapping between the transitions in the diagram and the C code generated. Blowups won't occur.
- Concurrency, hierarchy, visual priorities, and synchronization features are implemented by the code generator. For generated C/C++ code, concurrency is implemented as a "fair" interleaving of statements, one per transition in the diagram. The code generator makes sure that all concurrent transitions that can fire at a given point in time will fire one after the other and that no others will fire. This is a compile-time schedule.

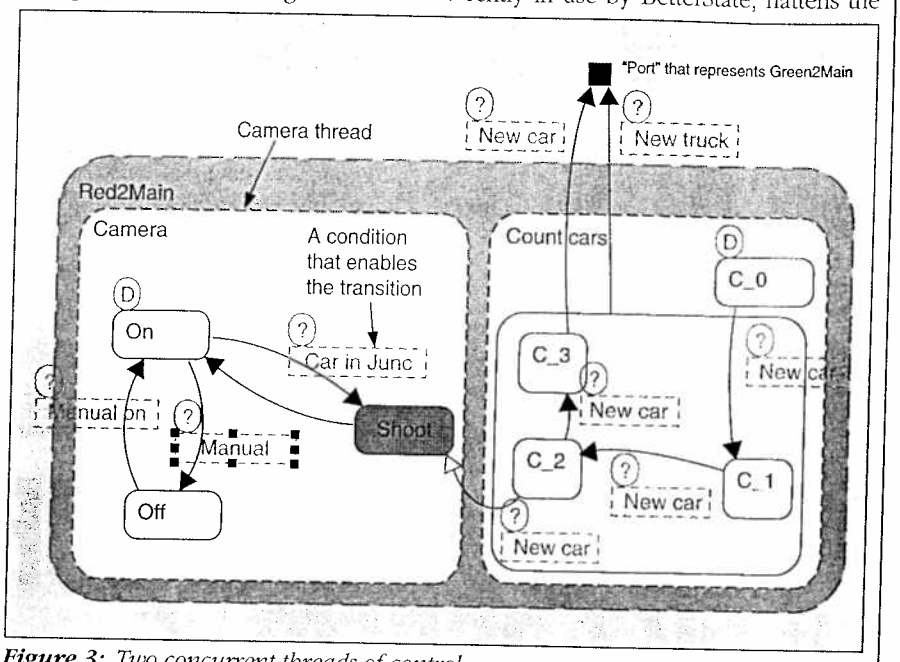

Figure 3: *Two concurrent threads of control.*

(continued from page 74)

- Support for visual synchronization and visual priorities is nontrivial.

For these reasons, handcoding for extended state diagrams can become a confusing and time-consuming task.

I have invented two code-generation methods for extended state diagrams. The first method, coinvented with David Harel, is more hierarchical in nature; the code generated preserves the hierarchy in the original diagram. The second method, currently in use by BetterState, flattens the diagram in an attempt to generate simpler code. For this reason, the BetterState code generates is simple in structure; it is no more than a large set of If statements. The process of generating this code however, is entirely nontrivial.

The BetterState code generator that builds the extended state diagram is based on the following code-generation methodology:

- Additional code is not required to run the design on a particular device. For example, the C code generated doesn't require a real-time operating system to implement concurrency and hierarchy, and is compatible with any processor or microcontroller equipped with a C compiler.
- There is a one-to-one mapping between the transitions in the diagram and the C code generated. Blowups won't occur.
- Concurrency, hierarchy, visual priorities, and synchronization features are implemented by the code generator. For generated C/C++ code, concurrency is implemented as a "fair" interleaving of statements, one per transition in the diagram. The code generator makes sure that all concurrent transitions that can fire at a given point in time will fire one after the other and that no others will fire. This is a compile-time schedule.

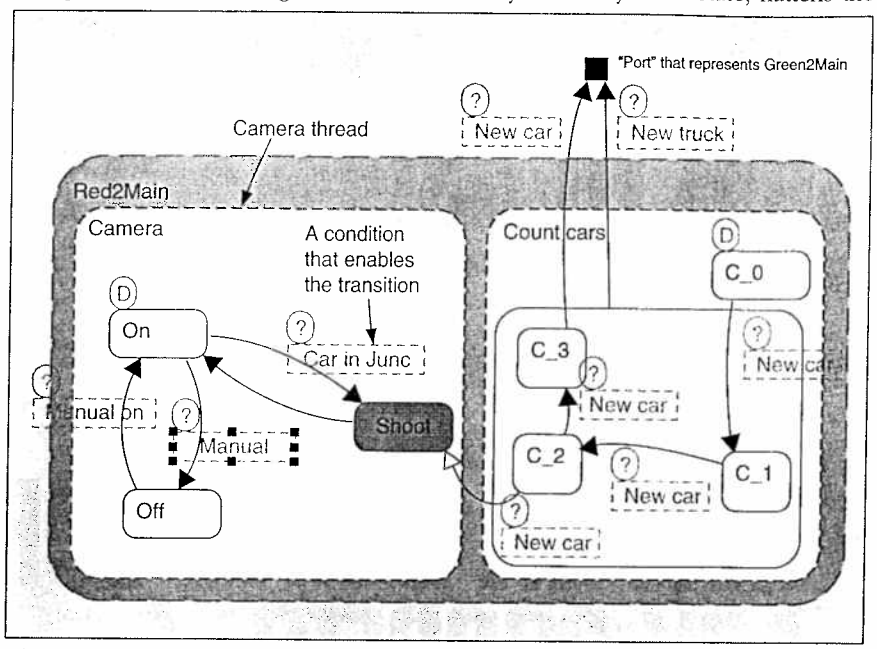

Figure 3: *Two concurrent threads of control.*

- In all languages supported by the code generator (C, C++, VHDL, and Verilog HDL), the generated code fits into the system-level design as a "component," without forcing a system-level design methodology. In C, the controller is a function called by the system-level C program (the main program) whenever it wants. Each invocation fires all concurrent transitions enabled at that time, then returns control to the calling program. This way the controller can be scheduled in any way that C permits. In VHDL, the code generated is an "architecture" for a controller entity. The designer provides the entity as well as the entire system-level VHDL code. The controller's generated code fits in as an architecture for that entity. In Verilog HDL, the code is a "task" that can be invoked by the system-level Verilog module. Each invocation fires all concurrent transitions enabled at that time, then returns control to the calling program. Listing One (page 106) is the C code generated for the traffic-light controller example.

Visual Synchronization in the Traffic-Light Controller

Once concurrency is provided for, visual synchronization—a means for visually specifying dependencies and relationships between concurrent threads of control—is needed. In the traffic-light controller, when three cars have been counted by the counter, it causes a transition to state *Green2Main*, thereby aborting everything else inside *Red2Main* state (the *Camera*). The programmer simply draws the transition from state *c_3* to state *Green2Main*; everything else is automatically derived from the diagram's semantics.

Another example is in the same diagram, where the *Counter* thread synchronizes the *Camera* thread into the *Shoot* state when the *Counter* is in state *c_2*; a behavior specified by the transition from state *c_2* to state *Shoot*. Another instance of visual synchronization includes compound transitions with multiple sources and/or targets, which act as a rendezvous (a meeting place between the threads). Visual priorities in BetterState are visually programmed using arrowhead colors. This is superior to hierarchical prioritization, because event and condition priorities are not necessarily associated with states. For example, a transition based on a *new_car* condition inside the counter might be more important than the *Timeout* condition.

Scheduling the Traffic-Light Controller

As discussed earlier, the code generated for the controller is a component in the overall system-level design (in C, this

(continued from page 78)

component is a function; in C++, it's a class; and so on). In each language, this component needs to be invoked by the system-level code. This is done in C/C++ by a function call, where each call to the controller's function realizes one pass over all transitions in a diagram, firing one or more concurrent transitions, and then returning control to the calling program. In Verilog, the invocation is done using an *always* statement that invokes the task, typically based on a clock event. In VHDL, the invocation is done by a CLOCK input signal to the entity for this architecture. This simple way of scheduling lets you invoke the controller in a flexible way. Listing Two (page 108) shows some possibilities.

Often, the controller will be invoked in some infinite loop, based on a clock, a certain input, or some other event. Sometimes, the design needs to abort this infinite invocation when the controller reaches a certain state; a terminal state supports this property. When it reaches a terminal state, it returns a value indicating that a terminal state has been reached, thereby allowing the C invocation in Listing Three (page 108) to break out of the infinite loop.

Similarly, in VHDL, when the controller reaches a terminal state, it will suspend itself, generate a suspended signal, and resume only when the entity receives a resume signal from the system-level design.

Animated Playback and Graphing

Often, you need to view the behavioral execution of a reactive component to verify the design or analyze the actual behavior in the field under the real stimuli. A playback mechanism allows the execution of the generated code to be recorded in a database, then played back in an animated fashion onto the original extended state diagram graphics. A state box flashing on/off might indicate, for instance, that a state is being "visited." The execution might use simulated stimuli (using a C or C++ debugger, for example) or the real stimuli from the field.

Graphing is a vehicle for visually displaying visitation information from the recorded database. This gives both you and system users insight into the actual behavior in the field. For example, a controller for automatic-door handling might recognize the fact that certain doors are open more than others during certain time periods.

DDJ

(Listings begin on page 106.)

Listing One *(Text begins on page 72.)*

```
/* C Code-Generation Output File #1 of 1: Generated for Chart Traffic_light
by the Better-State Code Generator. Proprietary of R-Active Concepts
Cupertino CA, 95014, (408)252-2808  */

/*----- State ID dictionary
Use this dictionary to symbolically reference your states (for those States
that you gave names in the chart editor). You can also examine the state
register, using the mapping provided as a remark.
-------*/
#define DUMMY -7
#define DONT_CARE 0
#define St_c_3_P3 6  /* mapped to PS[0] */
#define St_Yellow_All_P2 14  /* mapped to PS[0] */
#define St_Green2Main_P2 24  /* mapped to PS[0] */
#define St_c_0_P3 86  /* mapped to PS[0] */
#define St_c_1_P3 89  /* mapped to PS[0] */
#define St_c_2_P3 90  /* mapped to PS[0] */
#define St_On_P3 73  /* mapped to PS[1] */
#define St_Off_P3 76  /* mapped to PS[1] */
#define St_Shoot_P3 78  /* mapped to PS[1] */
#define St_Red2Main_P2 33  /* mapped to PS[3] */
#define St_On_going_P2 23  /* mapped to PS[4] */
int CHRT_Traffic_light(int BS_Reset)
{
/* RESETing: calling CHRT_Traffic_light with BS_Reset>0 will reset your
controller to it's default composite state, and will execute all on_entry
actions for that state. */
 static int PS[5]= {St_Yellow_All_P2, DUMMY, DUMMY, DUMMY, DUMMY};
 int NS[5]= {0, 0, 0, 0, 0};
 int BS_i;
 if (BS_Reset>0)
  {
   /* Reset state assignments */
   NS[0]=St_Yellow_All_P2;
   NS[1]=DUMMY;
   NS[2]=DUMMY;
   NS[3]=DUMMY;
   NS[4]=DUMMY;
   /* On_entry actions for reset states */

 Color_Main=YELLOW;Color_Sec=YELLOW;
  }
 else
 {
  /*-------*/
  if (PS[4]==St_On_going_P2)
      if (   (NS[0] == DONT_CARE)
          && (NS[1] == DONT_CARE)
          && (NS[2] == DONT_CARE)
          && (NS[3] == DONT_CARE)
          && (NS[4] == DONT_CARE)
         )
    if (Reset)
     {
      NS[0] = St_Yellow_All_P2 ;
      NS[1] = DUMMY ;
      NS[2] = DUMMY ;
      NS[3] = DUMMY ;
      NS[4] = DUMMY ;
      Color_Main=YELLOW;Color_Sec=YELLOW;
     }
  /*-------*/
  if (PS[0]==St_Yellow_All_P2)
      if (   (NS[0] == DONT_CARE)
          && (NS[1] == DONT_CARE)
          && (NS[2] == DONT_CARE)
          && (NS[3] == DONT_CARE)
          && (NS[4] == DONT_CARE)
         )
    if (!Reset)
     {
      NS[0] = St_Green2Main_P2 ;
      NS[1] = DUMMY ;
      NS[2] = DUMMY ;
      NS[3] = DUMMY ;
      NS[4] = St_On_going_P2 ;
      Color_Main=GREEN; Color_Sec=RED;
     }
  /*-------*/
  if (PS[0]==St_Green2Main_P2)
      if (   (NS[0] == DONT_CARE)
          && (NS[1] == DONT_CARE)
          && (NS[3] == DONT_CARE)
         )
    if (TIMEOUT)
     {
      NS[0] = St_c_0_P3 ;
      NS[1] = St_On_P3 ;
      NS[3] = St_Red2Main_P2 ;
      Color_Main=RED; Color_Sec=GREEN;
     }
  /*-------*/
  if (PS[3]==St_Red2Main_P2)
      if (   (NS[0] == DONT_CARE)
          && (NS[1] == DONT_CARE)
          && (NS[2] == DONT_CARE)
          && (NS[3] == DONT_CARE)
         )
    if (TIMEOUT)
     {
      NS[0] = St_Green2Main_P2 ;
      NS[1] = DUMMY ;
      NS[2] = DUMMY ;
      NS[3] = DUMMY ;
      Color_Main=GREEN; Color_Sec=RED;
     }
  /*-------*/
  if (PS[1]==St_On_P3)
      if (  (NS[1] == DONT_CARE)
         )
```

```
      if (Car_in_Junct)
       {
        NS[1] = St_Shoot_P3 ;
       }
     /*-------*/
     if (PS[1]==St_Shoot_P3)
        if (  (NS[1] == DONT_CARE)
        )
       {
        NS[1] = St_On_P3 ;
       }
     /*-------*/
     if (PS[1]==St_On_P3)
        if (  (NS[1] == DONT_CARE)
        )
       if (Manual_on)
        {
         NS[1] = St_Off_P3 ;
        }
     /*-------*/
     if (PS[1]==St_Off_P3)
        if (  (NS[1] == DONT_CARE)
        )
       if (!Manual_on)
        {
         NS[1] = St_On_P3 ;
        }
     /*-------*/
     if (PS[0]==St_c_0_P3)
        if (    (NS[0] == DONT_CARE)
             && (NS[2] == DONT_CARE)
        )
       if (New_car_waiting)
        {
         NS[0] = St_c_1_P3 ;
         NS[2] = 122 ;
        }
     /*-------*/
     if (PS[0]==St_c_1_P3)
        if (NS[0] == DONT_CARE)
       if (New_car_waiting)
        {
         NS[0] = St_c_2_P3 ;
        }
     /*-------*/
     if (PS[0]==St_c_2_P3)
        if (NS[0] == DONT_CARE)
       {
        NS[0] = St_c_3_P3 ;
       }
     /*-------*/
     if (PS[0]==St_c_2_P3)
        if (  (NS[1] == DONT_CARE)
        )
       {
        NS[1] = St_Shoot_P3 ;
       }
   } /* if BS_Reset */
   /* Assigning next state to present-state */
   for (BS_i=0;BS_i < 5;BS_i++)
       if (NS[BS_i] != DONT_CARE)
           {PS[BS_i]=NS[BS_i]; NS[BS_i]=DONT_CARE;}
     return 1;
   }
   /* end of BS controller */
```

Listing Two

```
/* calling the two controllers in a multi-rate scheme: the TLC runs twice
   for each cycle of the TM */

for (i=0;i<100; i++)
  {for (j=0;j<2;j++) CHRT_TLC(0);
   CHRT_TM(0);
  }
/* calling the charts stochastically */
for (i=0;i<100;i++)
      {x1=rand();
       if (x1/RAND_MAX  0.5) CHRT_TLC(0);
       x2=rand();
       if (x2/RAND_MAX  0.3) CHRT_TM(0);
      }
/* A Round-robin scheduler in an  endless execution (infinite-loop) */
while(1)
  {CHRT_chart1(0);
   CHRT_chart2(0);
   CHRT_chart3(0);
   ...
  }
```

Listing Three

```
/* Jumping out of an infinite-loop execution of the controller using Terminal
states. When the controller reaches a Terminal state (designated as such using
the State C-Code Dialog while drawing the state), it returns a 0 value. */

while (1)
  {
  if (!CHRT_TLC(0)) break;  /* will break if Terminal state has been reached */
  }
```

End Listings