Using Hierarchical State Machines in LabVIEW

Developing Complex Event-Driven Applications Using Active Objects

by Stanislav Rumega

A common LabVIEW™ programming architecture is the state machine. However, few LabVIEW developers are aware that there is more than one kind of state machine – finite state machines (FSMs) and hierarchical state machines (HSMs).

Typical LabVIEW programmers have little difficulty building finite state machines – National Instruments™ even provides templates for FSMs with LabVIEW. However, hierarchical state machines for complex systems can be a challenge to build from the ground up. Toward this end, I have created the LabHSM Toolkit.

Before I present the toolkit, it is important to have some background on the differences between "traditional" finite state machines and hierarchical state machines.

Overview

Most systems that application programmers deal with are event-driven. Some systems simply execute states (actions) in a predefined order, while others may perform their action and/or change their state as a result of internal or external events.

It's natural, then, when implementing such systems to break them down into subsystems that are themselves reactive. This leads us to using active objects as the most natural basic modules of a software application.

In addition to regular data and methods/actions of the traditional OOP objects, active objects possess a fundamentally different quality: they are endowed with their own thread of execution or process (sometimes referred to as being "alive"). Actions can be run and events can happen in an active object even in the absence of any communication with its environment. Thus, active objects can allow for more natural modeling of real life objects than traditional OOP objects.

However, what can be unclear is the question of what the internal mechanism driving active objects should be. How do we describe and code the object's behavior information?

Limitations of Traditional Finite State Machines

Often, complex behavior cannot easily be described by simple, "flat" state-transition diagrams (finite state machines, or FSMs). The FSM model works well for simple, state-driven systems, but doesn't scale up to larger systems. The lack of scalability in FSM stems from two fundamental problems: the flatness of the state model and its lack of support for concurrency.

Flat state machines do not provide the means to construct layers of abstraction. For example, consider a car. It is definitely a complex system, but on the highest level of abstraction, a car can be described as either "moving" or "stopped." On a much lower level of abstraction, the state of the car can be described as a combination of states of all the cylinders: whether each of them at the current moment of time is being filled with the gas/air mixture in, compressing it, exploding, or emptying.

The FSM model works well for simple, state-driven systems, but doesn't scale up to larger systems. The lack of scalability in the FSM model stems from two fundamental problems: the flatness of the state model and its lack of support for concurrency.

For some events, like "the driver has pushed the brake pedal," the reaction of the system (the car must stop) is the same regardless of the states of particular cylinders, while for other events the reaction might depend on the state of a particular cylinder, such as "valve 3 opened." If we need our abstraction model to provide the adequate reactions to both kinds of events, the flat model doesn't leave us a choice other than describing the car in terms of cylinder states. It means we need to explicitly assign a reaction to "the driver has pushed the brake pedal" event to every possible cylinder states combination separately! This is because traditional state machine design inflicts repetitions.

Such modeling can result in an unstructured, unrealistic, and chaotic state diagram, and classical FSMs can easily become unmanageable, even for moderately involved systems. The "moving" state in this example obviously contains all the states for all the cylinders. So, "moving" and "stopped" should not be considered at the same of abstraction/detail level as, say, "emptying cylinder 1."

Another potential limitation of the traditional state machine is its lack of support for concurrency. Consider modeling a traffic light. It can be thought of to be Green, Yellow, or Red. Imagine that you also need to take into account that it can run off a battery or from the city's power grid.

To describe this system with a traditional FSM, we will need the following states: Green-Battery, Yellow-Battery, Red-Battery, Green-Grid, Yellow-Grid, Red-Grid. The fact that the light is Green is obviously independent of whether or not it is running from battery or the grid. However, because traditional FSMs have no notion of independence, we must combine the independent states together.

Hierarchical State Machines (Statecharts)

Theoretical foundations on how to construct software for non-trivial event-driven systems have been around for more than 20 years. David Harel invented statecharts, or Hierarchical State Machines (HSMs), in 1983 as a powerful way of specifying complex reactive systems.

HSMs allow nesting states within states. States that contain other states are called *composite states*. States without internal structure are called *simple states*. If a system is in the nested state (called *substate*), it also (implicitly) is in the surrounding state (the *superstate*). Structuring the state space in this manner provides the ability to consider the system at different levels of abstraction, which is a powerful way for coping with complexity.

HSMs allow nesting states within states. States that contain other states are called composite states. States without internal structure are called simple states.

Composite states can not only hide but also reduce complexity through the reuse of behavior. An HSM will attempt to handle any event in the context of substate (in the lower level of the hierarchy). However, if the substate does not prescribe how to handle the event, the event is automatically handled in the higher level context of the superstate. This is what is meant by the system being in substate as well as in superstate.

In addition, state nesting allows substates to define only the differences in behavior from the superstates – behavioral inheritance. If a reaction (transition) for some event is defined for a superstate, it is automatically defined for all its substates (unless it is explicitly overridden in a substate). Thus, HSMs relax the requirement of classical state machines to explicitly define the transitions for every possible state/event combination, economizing on the description of the system's behavior. Avoiding repetitions allows HSMs to grow proportionally to system complexity as opposed to the explosive increase in states and transitions that can occur with traditional FSMs.

Statecharts also introduce state entry and exit actions. 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. Since states may be nested arbitrarily deeply these rules apply recursively.

Simple HSM Example

Consider an abstraction of an oven. At the highest level of abstraction (lowest level of detail) it can be considered to be "On" or "Off." However, when "On" it can be either "Baking" or "Broiling." Naturally, we want the oven to turn the heater on whenever it goes into the "On" state and turn it off whenever the oven is not on.

Also, in this example we will want to turn the light on whenever we open the door and turn it off whenever we close it. If we open the door while the oven is "On" and then close the door we expect the oven to maintain the cooking state it had been in – either "Baking" or "Broiling" – before we opened the door. This functionality requires introduction of one more state, "Paused", which will be positioned on the same hierarchy level as "On" and "Off." We definitely don't want the oven to turn on if the door is opened, so we have to break the "Off" state into "Off with Closed Door" and "Off with Opened Door."

The events that need to be processed (in addition to the "Exit Requested", which is present in any application): "Stop button pushed", "Bake button pushed", "Broil button pushed", "Door opened", "Door closed". Actions: "Turn the heater on", "Turn the heater off", "Turn the light On", "Turn the light off", "Close Application." The resulting statechart/HSM is shown in ***Figure 1.***

This example demonstrates how the HSM eliminates repetition:

1. If we don't define the exit transition from the top state, we will need to explicitly create transitions to the final state for from every other state if we want the system to exit whenever the user (or system) requests.

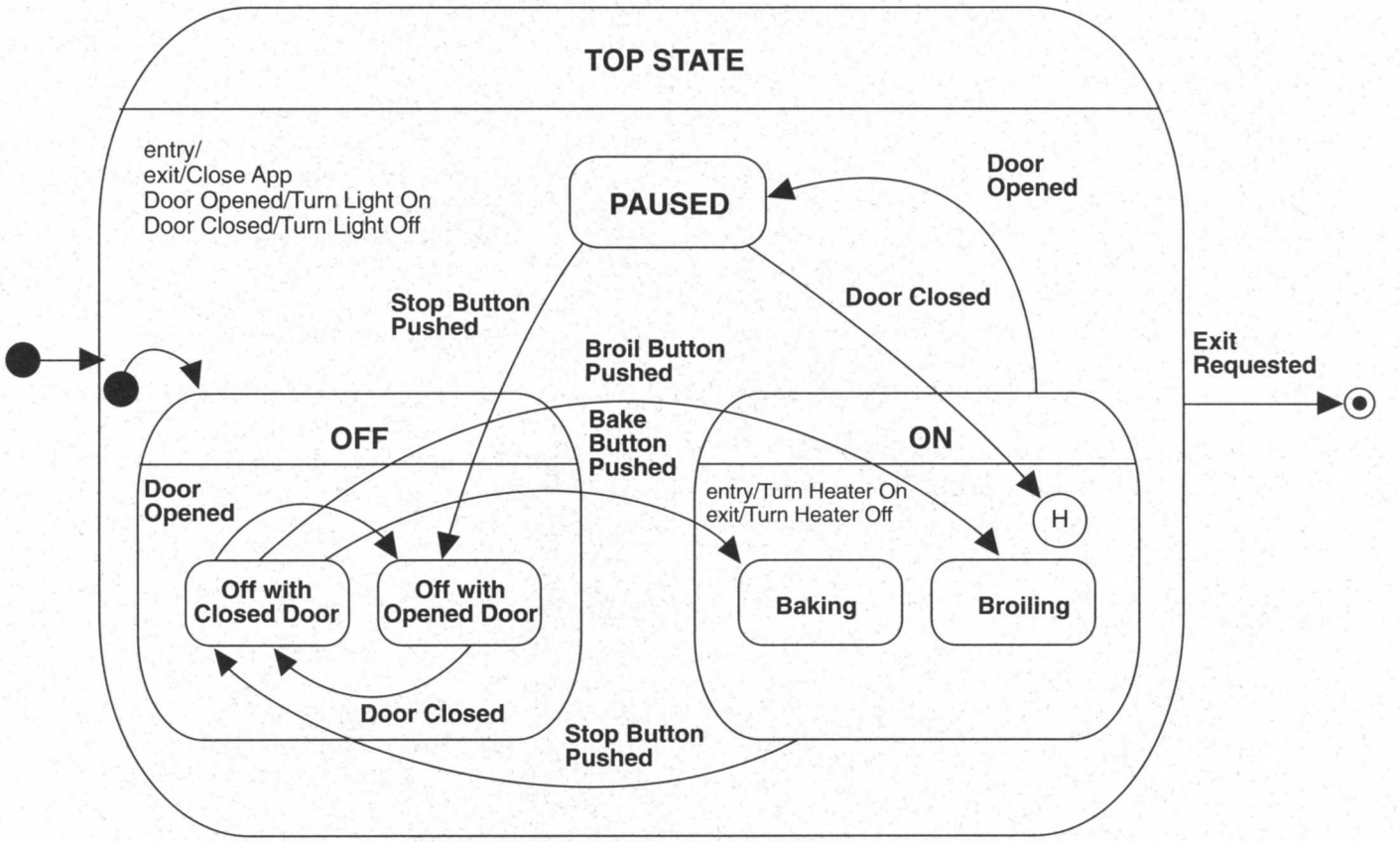

Figure 1: Example HSM Diagram (Statechart) of an Oven

2. For the "Stop Button Pushed" and "Door Opened" events, we now need only one transition defined – from the more general "On" state rather than individually from "Baking" and "Broiling."

3. By assigning "Turn Heater On" and Turn Heater Off" as entry and exit actions, respectively, for the "On" state, we avoid the need to include them into every transition into and from the "Baking" and "Broiling" states.

4. HSM semantics can be extended to include non-overriding behavior inheritance between states. This means that not only can the substate inherit the transition actions from its superstate when it doesn't have them in itself for some event, but that it can also be done even when the substate already has a transition defined for that event. In the latter case, unless an override is chosen, the actions assigned to an event in the superstates will be added to the list of actions assigned in the substate and the actual target state for the transition will be the one from the transition defined in the substate. The notation in the top left corner of the "Top State" indicates that the "Turn Light On" action must be executed whenever the event "Door Opened" happens and the "Turn Light Off" action must be executed whenever the event "Door Closed" happens, regardless of an actual state of the oven (it is always in the "Top State").

5. Notice the transition from the "Paused" state on "Door Closed" Event. It ends not at the "On" state, but at a circle with letter "H" in it. This is a "To History" type of transition. The circle with the letter "H" denotes a shallow-history pseudostate. It means that the actual target state of the transition will be the most recent active direct substate of the state containing this pseudostate. If it were not for this HSM feature, we would have to define states like "Paused after Baking" and "Paused after Broiling" to be able to return the oven to the state it was in before opening the door.

Implementing HSMs In LabVIEW

Because the basic unit of program decomposition in LabVIEW is a VI, it is highly desirable to make an active object out of a VI in such a way that all the objects have a singular code structure (regardless of their actual purpose) that provides a common look and feel and also provides different and easily modifiable functionality. However, it appears that to implement complex hierarchical behavior in the code, one has to use multiple nested case and loop structures. In fact, this complexity is not required.

Information about a behavior is just that – information. It consists of the state data in a tree structure that determines which state is a parent of which, and also contains any transition data. The behavior can be stored in some data structure (or a set of them) and kept in a file outside the VI. This approach allows you to create a uniform, easily modified diagram code structure.

Another fundamental step to reach our goal of easily modifiable code for an HSM active object is to change the meaning of the contents of the Case structure in the state machine loop. If we make the cases to be *actions* instead of *states*, any modifications of the diagram code can be reduced to adding and removing actions and editing the contents of the main Case structure.

If the change in behavior doesn't involve adding new or modifying existing actions, the block diagram need not be touched at all. A solution for creating LabVIEW HSM code is the LabHSM Toolkit.

Using the LabHSM Toolkit

The LabHSM Toolkit consists of the following:

- A Universal active object template, which you drop onto a blankVI's block diagram
- An editor to edit the HSM behavior data files
- A library of supporting functions that provide all the "plumbing" work to enable creation of a LabVIEW application as a collection of communicating active objects.

The Universal Communicating Hierarchical State Machine Template

The Universal Communicating Hierarchical State Machine Template includes initialization, main, and clean-up states. The initialization code includes reading the HSM behavior info from an external file. The main part of the code breaks down into three major subparts: the Processing loop, the UI Events Capturing loop (optional), and a call of a reentrant Message Receiver VI (optional).

The code for these actions is stored in the respective cases of the Case structure inside the Processing loop. The special Run State Machine action takes the behavior information, the current state, the state history information, and the next event pulled from the Events queue after prioritizing it. The Run State Machine action determines the next (target) state of the system and enqueues all the actions that need to run in the transition. The actions then execute in the order they were enqueued. At the beginning of each iteration, the next action is pulled from the Actions queue, defining which case the program will go into. Whenever the Actions queue becomes empty, Run State Machine gets executed again. If there are no events in the Events Queue, the VI just waits for the next event without consuming CPU cycles.

The LabHSM toolkit makes it possible to create and maintain complex event-driven applications in LabVIEW as a collection of Hierarchical State Machine-driven active object VIs using a higher level of abstraction and agile software development methodologies.

To preserve the encapsulation principle, only public events are exposed. This is implemented by a separate message queue and Message Receiver VI.

There are three possible sources of events: a user interaction event, an event resulted from some action, and a message received from another VI. The User Events Loop provides the code for capturing user events and puts them into the Events queue. Additionally, events can be posted from within action states. The Message Receiver translates external messages into the respective events and puts them into the same Events queue. Thus, uniform handling of all the events is provided regardless of their source.

Modifying the template code to provide the desired functionality of a particular active object is reduced basically to adding, removing and editing the code of the actions. If UI events need to be captured, the Event structure in the optional UI Capturing loop is easily edited as well. Usually, calling the Post Event subVI with a correct event name will be the only code there.

HSM Editor

The HSM behavior data file is created and modified with the LabHSM Editor. The main screen of the editor is shown in *Figure 2*. The state structure of an HSM is depicted with a tree control. The State control allows you to select a current state. The currently selected state can then be dragged within the state control to another position, deleted, renamed, set as default, or a substate can be added to it. Entry and Exit actions can be assigned to the state in the respective controls. The right part of the screen provides access to an array of transitions for the selected state. Transition information consists of the Target State, Transition Type, Event that triggers the transition, its priority, a list of Event Handling Actions, and an Override Flag.

In the presence of behavior inheritance, the concept of To (Deep) History transitions, possible entry and exit actions for states positioned at different levels of hierarchy, determining the actual target state, and the full list of the actions that must run in a particular transition are handled automatically. However, the developer sometimes will want to make sure at design time that the HSM will run exactly the actions he expects it to run and will end up in the state he expects it to. For this purpose, the LabHSM Editor is equipped with the **What If?** window, as shown in *Figure 3*.

The **What If?** window allows you to enter a source state, choose an event, and select the latest active substate of the target state (if applicable). The editor then (based also on what was entered in the main screen) shows which state will become the actual target state and the full list of the transition actions.

The snapshot of the **What If?** Screen shown in *Figure 3* illustrates an investigation of the "To History" transition from some State 111 (which is substate 1 of State 11, which in turn is a substate 1 of State 1, which is a substate of the top state) on some Event 7. The target state defined for this transition in the main screen is State 2. Therefore, the **What If?** window requests the user to choose one of the scenarios possible at run-time for this source state/event combination by selecting the last known deepest substate of State 2. In the depicted example the user has chosen the State 21 (substate 1 of State 2, which is also a substate of the top state, just like State 1). Therefore, the actual target state for this scenario will be State 21.

The LabHSM Toolkit includes a set of examples that demonstrate how to launch one active object from another, work with external timers, communicate with other objects on the same or another machine, and also how to programmatically

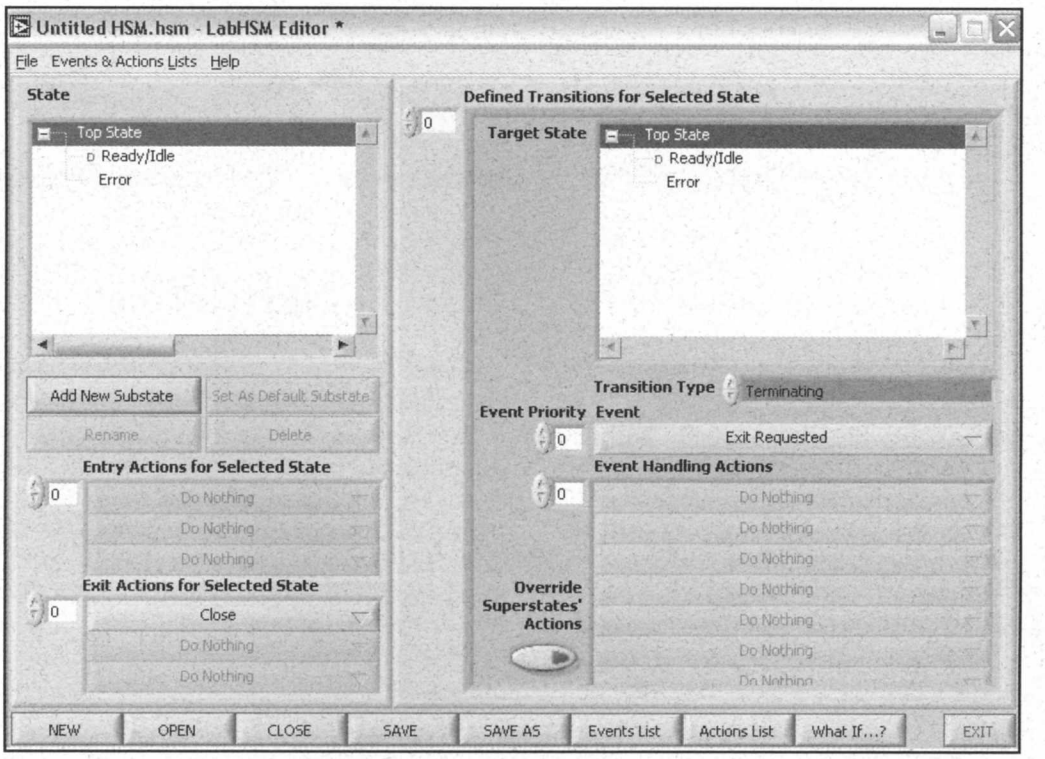

Figure 2: LabHSM Editor Main Window

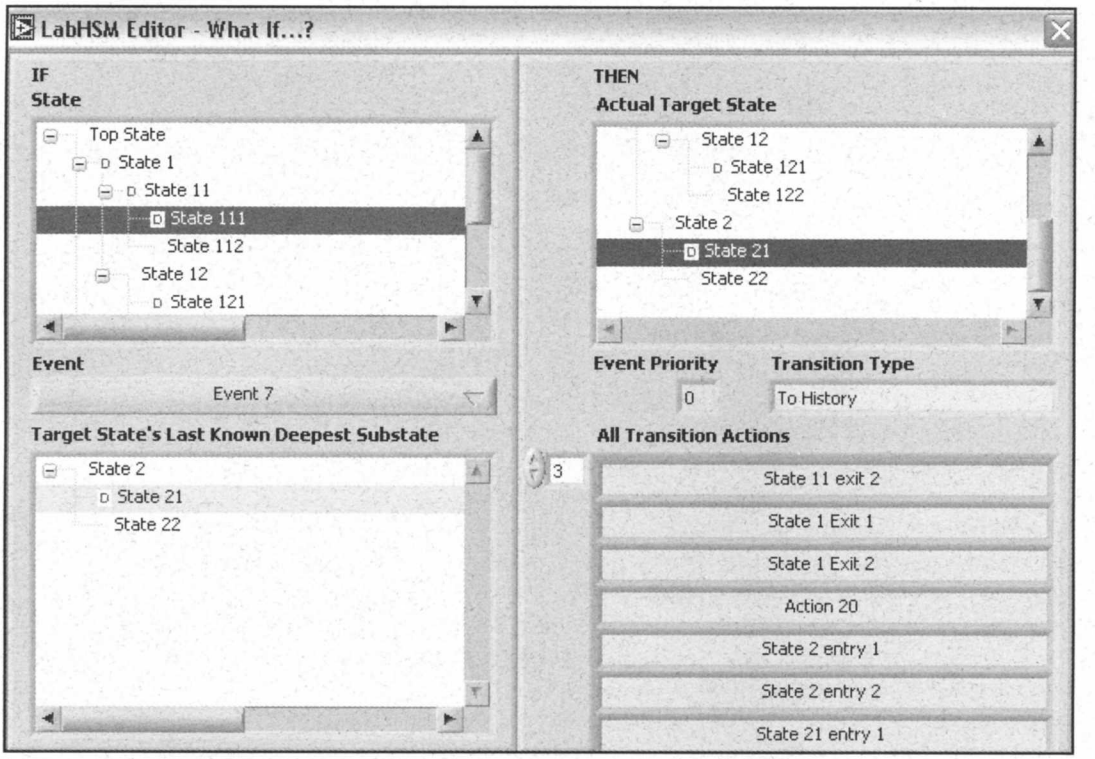

Figure 3: LabHSM Editor What If...? Window

instantiate multiple active objects from the same class template. There are tutorials on the toolkit included with the download or available from the LabHSM site.

Summary

Hierarchical State Machines, while non-trivial, can be implemented using various techniques in LabVIEW. The LabHSM toolkit makes it possible to create and maintain complex event-driven applications in LabVIEW as a collection of Hierarchical State Machine-driven active object VIs using a higher level of abstraction and agile software development methodologies.

The toolkit contains pure LabVIEW code, and is available on Windows, Mac OS X, and Linux platforms for LabVIEW version 7.0 and higher. It is available for a free unlimited time period trial at *www.labhsm.com.*

Additional Resources

- Douglass, Bruce Powel. 2001. *Class 505/525: State machines and Statecharts.* Proceedings of Embedded Systems Conference, Fall. San Francisco.
- Harel, David. 1987. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming (no. 8), 231-274.
- Samek, Miro. 2002. *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems.* CMP Books.

About the author:

Stanislav Rumega is a programmer at Target Labs, Inc. He is a Certified LabVIEW Developer and NI Week 2003 Best Application Contest Winner in R&D/Lab Automation Category. He can be reached at styrum@yahoo.com.

Additional Contributors: The author would like to thank Mr. Paul F. Sullivan (www.SULLutions.com) for helping to port LabHSM to Mac OS X and for writing the tutorials. He also would like to thank Mr. German Schumacher for the Linux versions.