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."