Implementing Non-Trivial Event-Driven Applications in LabVIEW with Active Objects Based on a Universal Communicating Hierarchical State Machine Template.

By Stanislav Rumega

Active Object Computing

Most of the systems that application programmers deal with are reactive (event-driven). Such a system can perform some or no actions, and possibly change its state, as a reaction to events, external (received from the environment) or internal (generated by its own previous actions). This reaction can vary depending on the current state of the system and, possibly, even on state history. It can also send events/messages to the environment as a part of the reaction. It's natural then, while implementing such systems, to decompose them into subsystems that are reactive themselves. 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, i.e. they are "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 allow much better and more natural modeling of real-life objects than traditional OOP objects.

Consider a Husband / Wife "system" under the following circumstances: the Wife is cooking dinner and discovers that she is missing some ingredient—an event for Wife, but not for Husband, who is watching football and has no knowledge about this misfortune. She asks her Husband (sends a message) to go to a store and buy that stuff. Once received, the message becomes now an (unpleasant) event for Husband. What's going to happen next? The Husband might refuse to go, because he is busy (send the corresponding message to Wife). But, most likely, he will not dare to argue and will go to the store. Unless the Husband is totally retarded, the Wife will not have to explain him how many steps he should take and in which direction and what to do if he sees a red light when crossing a street (all that is internal behavior information of the Husband object), right? Moreover, the Wife can be doing something else while the Husband is out (concurrency).

However, what should be the internal mechanism driving active objects? How do we describe and code the object's behavior information?

Traditional Finite State Machine Model Limitations

As it turns out, 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 [Douglas 01].

Flat state machines do not provide the means for constructing layers of abstraction. Consider modeling a car, which is definitely a complex system, but on the highest level of abstraction can be described as either "moving" or "stopped". On a much lower level of abstraction and, hence, a much higher level of detail 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, compressing it, exploding, or emptying. Apparently, for some events, like "the driver has pushed the brake pedal", the reaction of the system (the car must stop) will be the same regardless of the states of particular cylinders, while for other events, like "valve 3 opened" the reaction may depend on the state of a particular cylinder. If we need our model to provide 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 "the driver has pushed the brake pedal" event in every possible cylinder states combination separately! Traditional state machine formalism inflicts repetitions. Such "modeling" results in an unstructured, unrealistic and chaotic state diagram. Classical FSMs can easily become unmanageable, even for moderately involved systems. The state "moving" in the car example above 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 serious problem with traditional state machines is the lack of support for concurrency. This leads to a combinatorial explosion in the number of states to model. 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 mains. To describe this system with a traditional FSM we will need the following states: Green-Battery, Yellow-Battery, Red-Battery, Green-Mains, Yellow- Mains, Red-Mains. The fact that the light is Green is obviously independent of whether it is running from battery or mains. However, since traditional FSMs have no notion of independence, we must combine the independent states together. This is called the "combinatorial state explosion" because the modeling of multiple concurrent state sets requires the multiplication of the number of states in each to model all conditions [Douglas 01].

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 [Harel 87].

HSMs allow nesting states within states. States that contain other states are called composite states; conversely, states without internal structure are called simple states. If a system is in the nested state (called substate), it is also (implicitly) in the surrounding state (the superstate). Structuring of the state space in this manner provides the ability to consider the system at different levels of abstraction which, as is widely known, is a powerful way for coping with complexity [Samek 02].

Composite states 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 (which is 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. Of course, state nesting is not limited to one level only, and the simple rule of event processing applies recursively to any level of nesting. In addition, the semantics of state nesting allows substates to define only 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, HSM relaxes the requirement of classical state machines to define explicitly 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 typical for traditional FSMs [Samek 02].

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 [Douglas 01].

The concept of orthogonal regions (AND states) [Harel 87] is another cornerstone of statecharts. Practical realization of this concept is rather heavy-weight though. Instead of AND states, concurrency within a system can be easily modeled by breaking it into concurrently running active objects/subsystems which themselves have no internal concurrency, i.e. have a hierarchy of OR states only and can be only in one state at a time.

Simple HSM Example

Let's consider an abstract oven. At the highest level of abstraction (lowest level of detail) it can be considered to be "On" or "Off". However, it is a dual mode oven: when "On" it can be either "Baking" or "Grilling". 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. Well, every oven has a door. We will want to turn the light on whenever we open the door and turn it off whenever we close it. (The door doesn't have a window, so we don't need the light when it's closed.) If we open the door while the oven is "On" and then close the door we expect the oven to return to its business, whatever it had been before we opened the door, i.e. either "Baking" or "Grilling". 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 ubiquitous "Exit Requested", which is present in any application): "Stop button pushed", "Bake button pushed", "Grill 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.

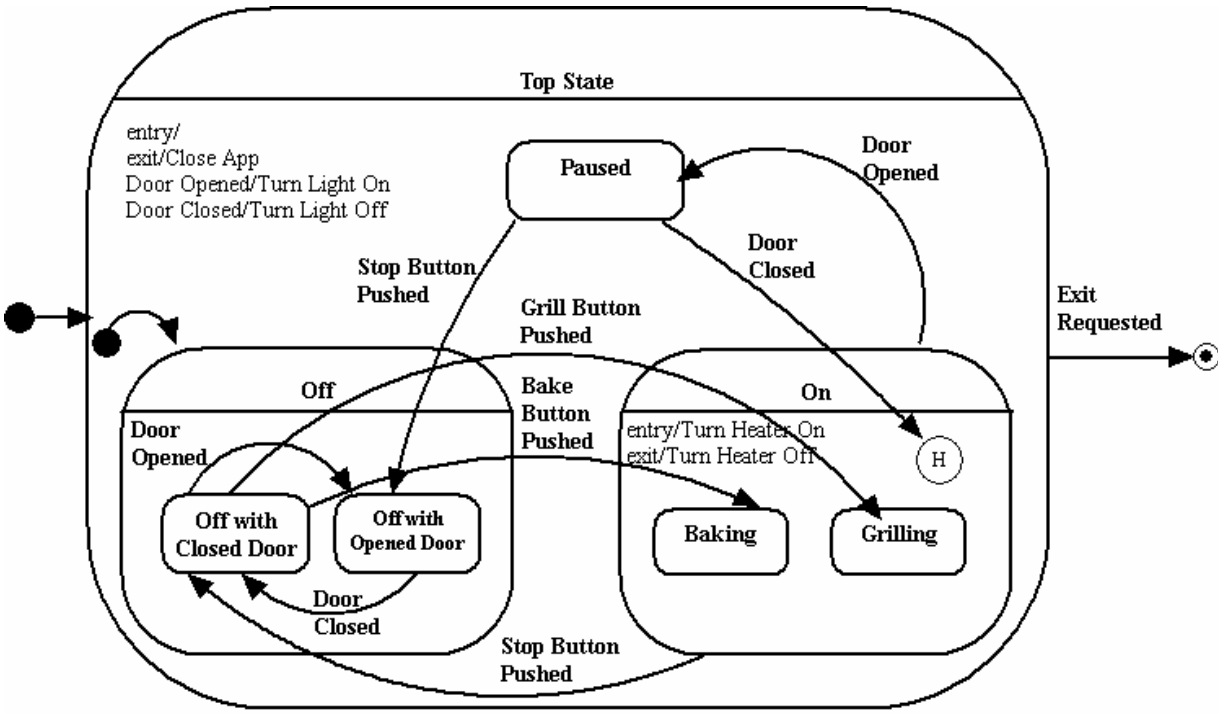

Figure 1. Abstract Oven Hierarchical State Machine Diagram (Statechart).

It's conceptually convenient to define one composite state, *top state*, as the root of the state machine hierarchy (Initial and Final pseudo states can be excluded, though). This simple example demonstrates in several ways how exactly HSM eliminates repetition.

1. If we don't define the exit transition from the top state but we want the system to exit whenever requested, we will need to explicitly create transitions to the final state from every other state.
2. Note that for the "Stop Button Pushed" and "Door Opened" events we need only define one transition each—from the more general "On" state rather than individually from "Baking" and "Grilling".
3. Thanks to 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 "Grilling" states.
4. HSM Semantics can be extended to include non-overriding behavior inheritance between states. It means that the substate can inherit the transition actions for some event from its superstate not only when it doesn't have them within itself, but also even when it already has a transition defined for that event. In the latter case, unless override is chosen, the actions assigned to an event in the superstates will be added to the list of actions assigned in the substate. The actual target state for the transition will be the one defined in the substate. The notation in 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 the actual state of the oven (it is always in the "Top State").

5. Please take a look at the transition from the "Paused" state on the "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 while Baking" and "Paused while Grilling" to be able to return the oven to the state it was in before the door was opened.

So, how do we implement these asynchronously-communicating active objects, driven internally by Hierarchical State Machines (HSMs), in LabVIEW?

LabHSM Toolkit—an Easier Way to Develop Non-Trivial Event-Driven Applications in LabVIEW

As you know, the basic unit of program decomposition in LabVIEW is a VI. So, it is highly desirable to make an active object out of a VI in such a way that all the objects, regardless of their actual purpose, have the same code structure that provides a common look and feel but different and easily modifiable functionality. However, to implement complex hierarchical behavior in the code, multiple nested case and loop structures seem unavoidable. In fact, they are not! Information about desired behavior is just that—information. It basically consists of the state tree data (i.e. which state is a parent of which) and transition data (i.e. which actions need to be run and to which state the object should switch when a particular event happens in a particular state). Just like any other information the behavior can be stored in some data structure (or a set of structures) and kept in a file completely outside the VI. This allows creating a uniform, easily modified, diagram code structure. No more spaghetti or deeply nested code! There is also no need for coding, separate from design anymore. The design of the component (the behavior information) becomes code as soon as the external HSM file is saved. Design is code!

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 define the cases to be actions instead of states, any modifications of the diagram code are basically reduced to adding and removing actions (cases) and editing their actual code (case contents). This doesn't increase the complexity of the code no matter how many times you do it. If the change in behavior doesn't involve adding new or modifying existing actions, the block diagram need not even be touched at all. Moreover, the same actions can be run in different transitions in a mix and match fashion, providing code reuse within the same VI!

These basic ideas have been implemented in the LabHSM Toolkit offered by *H View Labs, Inc.* The LabHSM Toolkit basically follows the Quantum Programming paradigm [Samek 02] but adds 1) prioritization of the events queue mechanism, 2) a separate message queue, and, of course, 3) some LabVIEW specific code like capturing UI events.
The LabHSM Toolkit consists of the following:

- A universal active object template, which you just drop onto a blank new VI diagram
- An editor to edit the HSM behavior data files
- A library of supporting functions which 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 parts.

The initialization code includes reading the HSM behavior information from an external file.

The main part of the code in turn 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 actions' code is stored in the respective cases of the case structure inside the Processing loop. The special action Run State Machine prioritizes the Events queue, pulls the next event from it, and combines it with the behavior info, the current state, and the state history info. 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 enter. 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.

There are three possible sources of events: a user interaction, an action programmed to trigger an event, and a message received from another VI . The top loop provides for capturing user events and putting them into the Events queue. Additionally, events can be posted to self from within actions. 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.

To observe the encapsulation principle, only public events should be exposed. This is implemented by a separate message queue and the Message Receiver VI on the bottom of the diagram.

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

HSM behavior data file is created and modified with the LabHSM Editor. The main screen of the editor is shown in Figure 2.

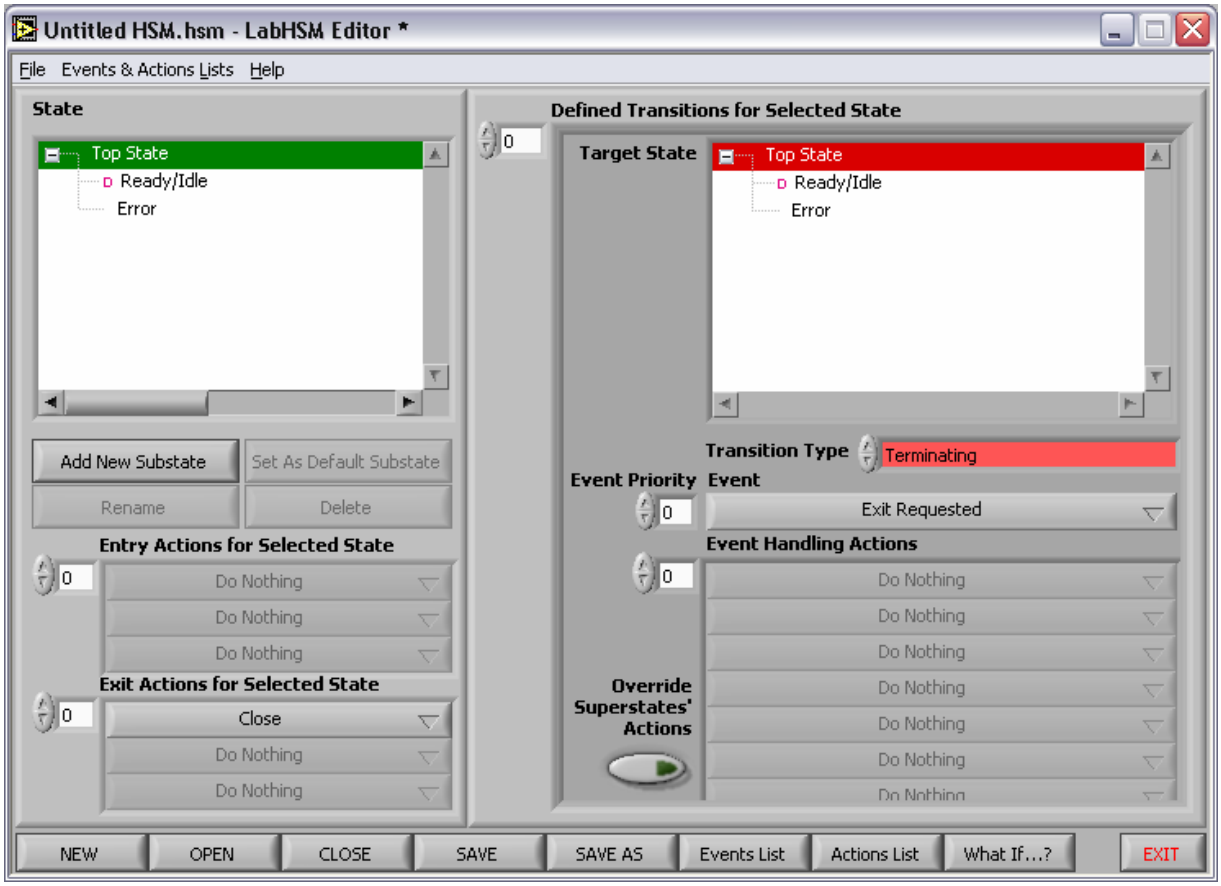

Figure 2. LabHSM Editor Main Screen.

Although statecharts were developed as a graphical formalism, the state structure of an HSM is just a tree, so it can be suitably depicted with just a tree control. The State control allows selecting a current state. The currently selected state can then be dragged within the state control to another position, deleted, renamed, set as default, or modified to include a substate. 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, the Transition Type, the Event that triggers the transition, the Event's priority, a list of Event Handling Actions, and an Override Flag.

Behavior inheritance makes it trivial to encode or decode: To (Deep) History transitions, entry and exit actions for states positioned at different levels of hierarchy, the actual target state, and the full list of the actions that must run in a particular transition. Everything is handled automatically, partially by the LabHSM editor when saving HSM file, partially by the Run State Machine action at run time. However, the developer sometimes will want to make sure at design time that the HSM will run exactly the expected actions and end up in the expected state. For this purpose, the LabHSM Editor is equipped with the What If…? Screen (Figure 3).

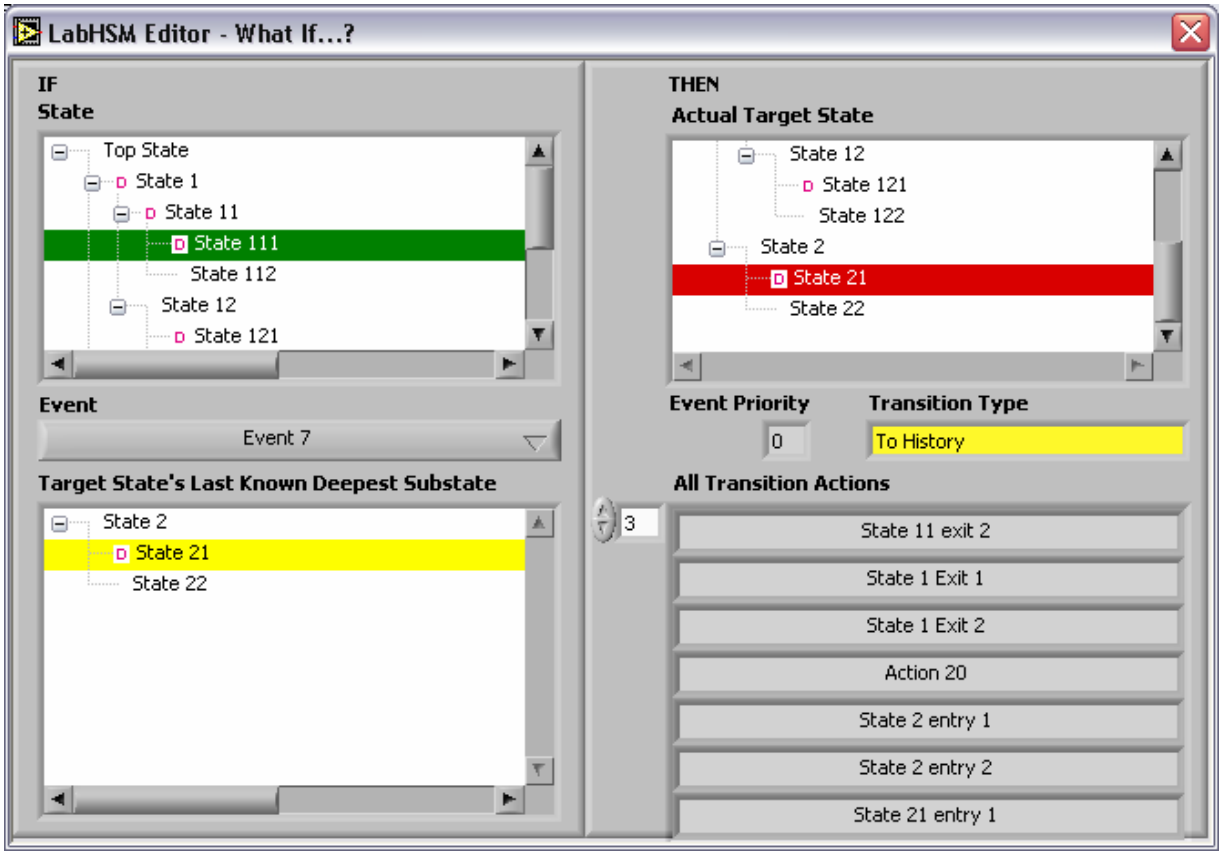

Figure 3. LabHSM Editor What If…? Screen.

It allows entering a source state, an event, and the latest active substate of the target state (if applicable). Then the editor (based also on what was entered in the main screen, of course) will show 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 investigation of a “To History” transition from some State 111 (which is a substate 1 of State 11, which, in turn is a substate 1 of State 1, 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…? Screen 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). Hence, the actual target state for this scenario will be State 21. Although the transition has Action 20 assigned as its one and only event-handling action, the actual actions list turns out to be quite long (longer than the 7 slots available) because it includes the exit and entry actions of all the states involved. Please note the order in which exit and entry actions are scheduled to run with respect to the event handling action. First, all the exit actions will be executed starting from the source state all the way up to (but not including) the Least Common Ancestor (LCA) of the source and the target states. In this case LCA is the top state. Second, the event handling action will be executed. And, finally, all the entry actions from the LCA to the actual target state will run.

The LabHSM toolkit comes with a set of examples, which clearly 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 instantiate multiple active objects from the same class template. Recent download packages also include a nice set of plain English tutorials written by Mr. Paul F. Sullivan (SULLutions.com). The tutorials can also be checked out at the LabHSM site.

Summary

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

The toolkit contains pure LabVIEW code only. It is now available for Mac and Linux platforms as well as Windows with LabVIEW version 7.0 and higher.

It is available for a free ULIMITED PERIOD TRIAL at *labhsm.com*.

Acknowledgements

In conclusion I would like to thank Mr. Paul F. Sullivan for helping to port LabHSM to Mac, writing the excellent tutorials and a lot of good ideas. Also I would like to thank Mr. German Schumacher for the Linux versions. Thanks also to J.R. Allen for keeping a link to LabHSM site on the front page of the LabVIEW Zone for a while. Of course, thanks to everybody who has downloaded LabHSM and provided constructive feedback about it.

References

[Douglass 01] Douglass, Bruce Powel. 2001. Class 505/525: State machines and Statecharts. *Proceedings of Embedded Systems Conference*, Fall. San Francisco.

[Harel 87] Harel, David. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* (no. 8), 231-274.

[Samek 02] 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 the R&D/Lab Automation Category. He can be reached at styrum@yahoo.com.