StateProto – Executing Multiple StateMachines

statedriven@users.sourceforge.net

12 July 2006

Table of Contents

Illustration Index

Introduction

In the first article on stateProto I introduced a statemachine visual modelling application that generates code for a modified qf4net library. I showed a basic watch as a demonstration of executable code. This sample was really simple as it showed only how to run one state machine modelled class in an application. How can one run multiple instances of the statemachine based class? What was that *EventManager* and *Runner* classes anyway?

This article is an attempt to answer these questions. The eventManager and runner design is central to the mechanism of multiple instance execution and later, inter-statemachine communication.

The Sample

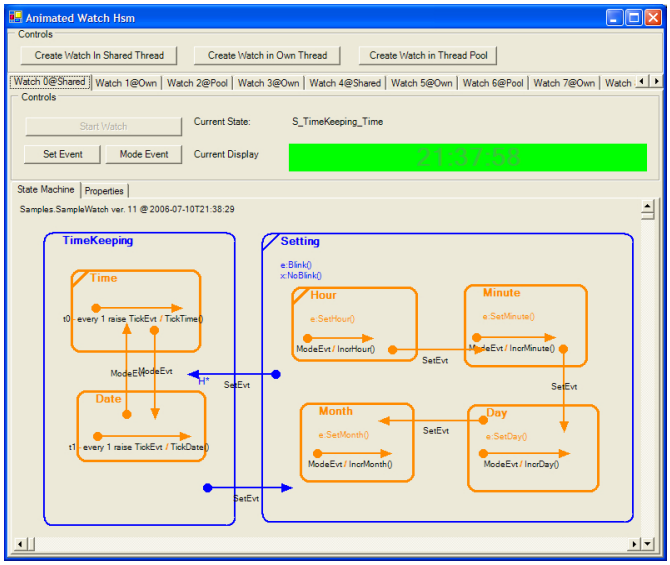

Illustration 1: The Sample - Animated Multiple Hsms

On downloading the sample and executing SampleWatch.exe – notice that the new sample watch application provides a main form with a couple of “Create ...” buttons at the top of the screen. There is a large blank area beneath the "Create..." controls that will fill with tabs when these create buttons are clicked. Each tab represents a unique watch state machine instance, with its visual representation, a property grid and signal buttons. Note that this application is for demonstration purposes only. It is unnecessary and inefficient to run a StateMachine viewer and property grid against a statemachine running in a production environment.

Each state machine instance can run in one of three different thread models.

These thread models are:

1. **Shared Thread**: Create a watch instance and run it in a shared thread context. The shared thread can be reused by many state machines as long as none of them block. This is generally okay since a state machine should not be designed to block.
2. **Own Thread**: This is also called the *Thread-Per-Hsm* model. Here every watch instance created will be given its own thread.
3. **Thread Pool**: This is similar to "*Shared Thread*" excepting that instead of just one thread – you have more than one thread. A state machine when running within the context of a thread pool can be executed by any thread within the pool as long as the run-to-completion semantics of qf4net is maintained. The current model is simplistic in that a state machine once registered with the pool will be affined to one specific thread within the pool. Even though this model is simplistic it can still be a better solution than creating a new thread for every hsm instance.

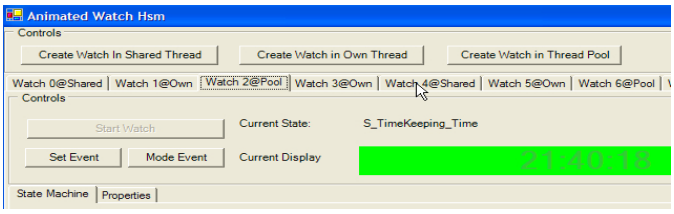

Illustration 2: Controls of Demo

The Log

```
2006-07-10 07:01:37.391 INFO [ThrShared] SampleWatch.ConsoleStateEventHandler
```

Illustration 3: Console - First Part

The console output is a concatenation of the "First Part" and "Second Part" samples. The output is the same as most default Log4Net implementations – noting that the logging facility is wrapped in an *ILogger* interface that is not log4net. This default implementation is a default Console implementation of this *ILogger* interface. A wrapper for Log4Net will be made available later.

This is the layout of the log messages:

Output	***Explanation***
2006-07-10	Current Date
07:01:37.391	Current Time
INFO	Information Item (vs DEBUG, WARNing, ERROR).
[ThrShared]	The name of the thread from which the log line was written.
SampleWatch.ConsoleStateEventHandler	The class from which the logging was done.

What follows (the "Second Part") is an unformatted log output written by the developer for each different log line. In this application when logging the state change events in SampleWatch.ConsoleStateEventHandler the format can be defined as follows:

Output	***Explanation***
StateChange:	The event occurring
[Watch 11	The watch hsm instance's id
Samples.SampleWatch[303]	The ToString() from the Hsm which by default is the Hsm plus its HashCode.
Exit S_TimeKeeping_Time	The action taking place. In this case an Exit on the TimeKeeping::Time nested state.

```
StateChange: [Watch 11 - Samples.SampleWatch[303]] Exit S_TimeKeeping_Time
```

Illustration 4: Console - Second Part

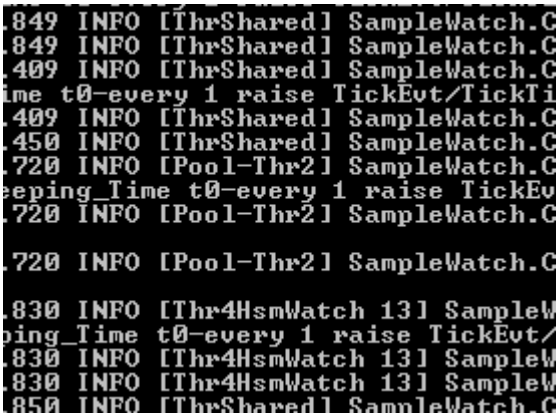

Illustration 5: Multiple Threads

Just to make the point clear – notice that there are many threads executing in this sample.

EventManager

What is it?

QHsm works by receiving QEvents (signals) that it responds to. The event dispatching mechanism is via a call to a method called Dispatch(IQEvent). This dispatch is synchronous and can cause timing issues when a dispatch is called during a transition. To overcome this all state machines must run as "Active Objects". This means that all event dispatching is done asynchronously. This can be achieved by calling the AsyncDispatch(IQEvent) method.

AsyncDispatch logically places the event on a queue. Events are then pulled off the queue in FIFO order and executed – generally from a separate thread. In the LQHsm modified state machine this logical queue is provided in the form of a QEventManager interface.

The *QEventManager* is then stepped through via a separate *QEventManagerRunner* interface which essentially activates and runs the *QEventManager*. Two types of runner is currently provided – a gui thread timer based runner (a windows message loop runner will be added at some future date) and a threaded runner (a thread-pool based runner will be added if the necessity arises).

A single *QEventManager* can hold more than one state machine instance and will correctly route an event to the statemachine that it was sent to based on the state machines id parameter.

The Run-To-Completion model

After the QEventManager dispatches an event to a state machine – the manager will not dispatch another event to it until the Hsm completes its handling of the preceding event. The eventManager essentially will not preempt the Hsm in any way until its processing is complete and it reaches a stable state. This "Run-To-Completion" model allows the state machine designer to design every state machine as if they were devoid of multithreading concerns. There will be no need for synchronisation primitives (locking, etc) for as long as there is no shared memory between different parts of the system.

This model of execution means that individual responses to events should be as short as possible.

Constructing the various models

To facilitate constructing the various thread models for this sample application I defined an IHsmExecutionModel interface as follows:

```
using System;
using qf4net;

namespace SampleWatch
{
    /// <summary>
    /// IHsmExecutionModel.
    /// </summary>
    public interface IHsmExecutionModel
    {
        Samples.SampleWatch CreateHsm (string id);
    }
}
```

Illustration 6: IHsmExecutionModel used in Sample

The three models available then implement the CreateHsm(id) method which then allows the rest of the sample to be very similar to the previous sample (bar the new gui additions).

Then to recap there are a couple of steps to constructing a state machine:

1. Create an *EventManager* (this can be done any time prior to constructing the statemachine instance).
2. Create an EventManager runner.
3. For the thread pool mechanism also create a *LifeCycleManager*.

```
IQEventManager eventManager = new QMultiHsmEventManager(new QSystemTimer());
IQEventManagerRunner runner = new QThreadedEventManagerRunner ("Thr4Hsm" + id, eventManager);
runner.Start ();
```

Illustration 7: Creating an Event Manager

4. And then just to make this entirely clear: steps 1 to 3 need only be done once in an application.
5. Create the Hsm – there are a number of constructors. It is recommended to always supply a unique id and a groupid. For now groupid's are only necessary for the thread pooled model – but will become much more important later for state models that for interaction groups.

```
Samples.SampleWatch sampleWatch
    = new Samples.SampleWatch (id, id, _LifeCycleManager);
```

Illustration 8: Creating an Hsm with a LifeCycleManager

Finally – pass through either an eventManager or a lifeCycleManager.

```
Samples.SampleWatch sampleWatch
    = new Samples.SampleWatch (id, eventManager);
```

Illustration 9: Creating an Hsm with an EventManager

6. Call the hsm's init method().

What else can be done?

It is possible to instrument the state machine by hooking to its stateChange event and UnhandledTransition handlers. This is what *ConsoleStateEventHandler* and *StateProtoViewAnimator* does.

Thread Per Hsm

Thread Per Hsm basically means creating a new *QEventManager* for every new state machine instance. The advantage of this model is that no statemachine will block while any other is busy executing. The disadvantage is that not too many threads an be created because os level threads are quite resource intensive (especially memory).

Shared Thread

This is achieved by creating one *QEventManager* upfront and giving it to every state machine instance. The disadvantage of this approach is that while any single hsm is busy all other hsm's are made to wait.

Thread Pool

The current thread pool mechanism is basically a multiple "Shared Thread" mechanism where state machines are linked to one of a set of *QEventManagers* that all run under the context of a new entity called a *QLifeCycleManager* (one of its roles is to be a container of multiple *QEventManagers*).

```
IQEventManager[] eventManagers = new IQEventManager[]
    {
        new QMultiHsmEventManager(new QSystemTimer()),
        new QMultiHsmEventManager(new QSystemTimer()),
        new QMultiHsmEventManager(new QSystemTimer()),
        new QMultiHsmEventManager(new QSystemTimer()),
        new QMultiHsmEventManager(new QSystemTimer())
    };
QHsmLifeCycleManagerWithHsmEventsBaseAndMultipleEventManagers lifeCycleManager
    = new QHsmLifeCycleManagerWithHsmEventsBaseAndMultipleEventManagers (eventManagers);

int counter = 0;
foreach (IQEventManager eventManager in eventManagers)
{
    IQEventManagerRunner runner = new QThreadedEventManagerRunner (
        "Pool-Thr" + counter.ToString(), eventManager);
    runner.Start ();
    counter++;
}
_LifeCycleManager = lifeCycleManager;
```

Illustration 10: Create Thread Pool using a LifeCycleManager

A Little Bit Extra

EventManager

This section elaborates a bit more on using the *EventManager* and *EventManagerRunner* classes for creating executable state machine instances. The *EventManager* is passed to the constructor of the state machine and cannot be changed once assigned. However, if you look at the generated code you will find that the state machine has a number of different constructors. Closer inspection will reveal that id and groupId are there – as are EventManager's. However, as introduced earlier there is also the option to pass in a *LifeCycleManager* and, something that has not been discussed yet, an *ExecutionContext*.

LifeCycleManager

As it turns out – if you pass through a *lifeCycleManager* – the expectation is that the state machine registers with the provided lifeCycleManager (done in the base LQHsm class) which in turn provides the state machine with an *EventManager*. The lifeCycleManager also provides notifications of state machine additions and removals – a feature that will be tapped into at a later stage.

ExecutionContext

The last item is an *ExecutionContext*. This, for a start, provides the *LifeCycleManager* as a property. It also provides a service access point query method (GetService(...)) which can be queried within the "LocateServicesUsingExecutionContext()" method of the state machine and that every state machine implementor has an option to override. This allows the state machine to get hold of resources and services it might not have had access to without using a global accessor class (meaning some static class to act as a global accessor). *ExecutionContext* has an Administrative interface that can be used to add a set of services that would be of use to the state machines accessing it.

Summary

This sample showed three big ideas:

1. How to run multiple state machines.
2. How to run multiple state machines under multiple threads.
3. That executing state machines can be visualised. This can be quite useful (though not essential) during testing.

Coming Up

1. State Machine interaction via ports.
2. Saving the State Machine for later rehydration.