Quantum Hierarchical State Machine - Port to C#

By Rainer Hessmer, Ph.D. (rainer@hessmer.org)

Revision History

Revision	Date	Author	Description
1.0	01/15/2003	Rainer Hessmer	Initial Release
1.1	12/11/2003	Rainer Hessmer	Added revision history and reference to download URL
1.2	12/29/2003	Rainer Hessmer	Modified to account for the change of the QState signature which now based on the IQEvent interface.

Table of Content

Introduction

This document describes my C# port of Samek's excellent Quantum Hierarchical State Machine as described in the book *Practical Statecharts in C/C++* (Miro Samek, Ph.D., CMPBooks, 2002; http://www.quantum-leaps.com/).

Note: The latest version of this document as well as the associated code can be downloaded from www.hessmer.org/dev/qhsm/index.html.

While Samek's implementation specifically targets embedded systems and hence is specifically designed to use minimal memory (both stack as well as heap based), I opted for a less stringent approach since C# is not currently used in embedded systems. One place where this difference becomes very apparent is the recording of the individual transition steps that are required to perform a static transition. Where Samek uses a fixed sized array, I temporarily use an ArrayList that can dynamically grow as needed and is later collapsed into a fixed size array. The usage of an ArrayList makes it possible to compile the QHsm into a utility dll that can be used for all kinds of hierarchical state machines without the need to readjust the array size in the base class for state machines that have deeper hierarchies than a given array size can deal with.

I decided to port the Quantum Hierarchical State Machine to C# even though C++ is one of the languages supported by .Net, because it is my experience that porting a framework forces you to really understand its structure. Apart from this I hope that the availability of a C# port might further increase the adoption of Samek's framework.

Apart from a mere port, the provided C# code offers a recipe for the optimized handling of transitions even in the case of multiple instances of derived state machines of a given reactive base class (see section Static Transitions and Derived State Machines). By applying the recipe, the fallback to expensive dynamic transitions (see section 6.3.1 *Static versus Dynamic State Transitions* in Samek's book) can be completely avoided.

Source Code

The code is provided as part of the VisualStudio.Net (2002) solution HierarchicalStateMachine.sln, which leverages version 1.0 of the Microsoft .Net Framework. The QHsm base class and the QEvent class are part of the project HierachicalStateMachine, which compiles into a dll. The solution also contains the project QHsmTest that acts a test harness and includes a state machine, a derived state machine, and a simple Console app that allows the user to dispatch signals to either state machine.

Finally the solution includes the project OptimizationBreaker which demonstrates how the normal usage of static transition fails in the case of derived state machines and shows how to fix this problem.

Porting the QHsm Implementation to C#

The following sections provide a detailed explanation of the C# port. Some of the differences between Samek's C++ code and the C# code are due to language intricacies; others are due to my decision to refactor the code that deals with the handling of transitions, which constitutes the most complicated part of the code. By refactoring the code I could more easily cope with the complexity. However, the breaking up of a single method into multiple methods and helper classes has certainly increased the memory footprint of the QHsm slightly and hence would typically be not acceptable for embedded systems.

Delegate Replaces Pointer to Member Function

In C#, strongly typed delegates replace the more loosely coupled pointer to member function constructs. As a result of this, QState is defined as a delegate. At the same time I introduced the interface IQEvent that a class representing a QEvent needs to implement:

```
/// <summary>
/// Delegate that all state handlers must be a type of
/// </summary>
public delegate QState QState(IQEvent qEvent);
```

Note that C# allows the definition of a delegate that returns its own type. Therefore there is no need for the definition of the QPseudoState.

QEvent and QSignal Values

Enums are strongly typed in .Net. This means we cannot use a signature like this for the constructor of QEvent:

```
public QEvent(QSignals qSignal)
```

where QSignals is the enumeration holding the reserved signals (Empty, Init, etc.). The problem with this signature is that the class that implements the actual custom state machine and inherits form QHsm has to define its own signals. Since it is not possible to inherit one enum from another, the inheriting class needs to create its own enumeration. However, then we have a casting problem since QEvent expects its own enumeration in the constructor.

The solution of course is to use a generic integer type as the signal identifier. So the signature of the QEvent constructor changes to

```
public QEvent(int qSignal)
```

Dependent on the number of signals that are required, int might be exchanged with any other integer data type. (Note that in order to allow VB.Net code to use the QHsm implementation, unsigned integer types like uint or ushort should not be used.)

It still makes sense to define the actual signal ids in an enumeration in order to make sure that each signal has a unique id. For the reserved signals the enumeration looks like this:

```
public enum QSignals : int
{
	Empty,
	Init,
	Entry,
	Exit,
	UserSig
};
```

Again, due to the fact that enums are strongly typed and not automatically downcast to the associated base type, it is necessary to explicitly downcast QSignal ids that are defined in an enumeration. Unfortunately this leads to rather unwieldy code. The following code snippet shows a typical example:

```
protected QState s0(IQEvent qEvent)
{
	switch (qEvent.QSignal)
	{
		case (int)QSignals.Entry: Console.Write("s0-ENTRY;"); return null;
		case (int)QSignals.Exit: Console.Write("s0-EXIT;");   return null;
		case (int)QSignals.Init: Console.Write("s0-INIT;");   InitializeState(m_s1);
			return null;
		case (int)MyQSignals.E_Sig: Console.Write("s0-E;");   TransitionTo(m_s211);
			return null;
	}
	return this.TopState;
}
```

Should QEvent be a class or a struct? I opted for making QEvent a class for two reasons: A class allows one to easily derive new QEvent types from the base class without incurring the overhead of boxing / unboxing. In addition it allows me to introduce the interface IQEvent and use it instead of the base class in the QState delegate. As a result the QHsm is independent from the actual implementation of QEvent.

Initialization of the State Machine:

The implementation of an initial pseudo state that is used in Samek's regular code is not feasible in C#. It would require that during the construction of the derived state machine class, you pass a delegate instance that points to the pseudo-state handler of your state machine back to the base class constructor like so:

```
public class MyQHsm : QHsm
{
	protected QState  m_Initial;

	public QHsmTest() : base(new QState(this.Initialize))
	{
	}

	protected QState Initialize(IQEvent qEvent)
	{
		…
```

```
        }
```

The problem is that MyQHsm is not constructed yet when we need to pass the delegate pointing to our handler of the initial state into the constructor of the base class. So, the part

```
public QHsmTest() : base(new QState(this.Initialize))
```

will fail to compile since the this pointer is not defined at this point yet. For this reason the C# code implements an alternative design that uses a polymorphic Initialize() method that is called by the QHsm base class. This alternative design is the topic of Exercise 4.2 in Samek's book.

Handling of Transitions

The C# code differs most from Samek's C++ code in the handling of transitions. The actual algorithm is the same in both cases. However, the C# code is factored differently. The class offers three methods TransitionTo():

```
protected void TransitionTo(QState targetState)
```

which performs a dynamic transition, and

```
protected void TransitionTo(
      QState targetState,
      ref TransitionChain transitionChain)
```

which performs a static transition. The third version

```
protected void TransitionTo(QState targetState, int chainIndex)
```

is discussed separately in the section Static Transitions and Derived State Machines and is not further covered here.

In the case of a static transition an instance of the helper class TransitionChain is used to store the individual transition steps. The class TransitionChain essentially is the same as the class Tran in Samek's code.

Both `TransitionTo()` methods first exit up to source state which is encoded in the function `ExitUpToSourceState()`. Then, in the case of a dynamic transition the method `TransitionFromSourceToTarget(targetState, null)` called by passing in a null instead of a instance of the helper class `TransitionChainRecorder`. The method `TransitionFromSourceToTarget()` is used for both a dynamic and a static transition and is used to dynamically discover the necessary transition steps to reach the target state. If no recorder is provided then the discovered steps are just executed but not persisted.

In the case of a static transition the associated version of the method `TransitionTo()` checks whether an instance of `TransitionChain` was passed in. If no instance is provided, then it creates an instance of `TransitionChainRecorder` and yields control to the method `TransitionFromSourceToTarget` by passing in the recorder. As a result the latter method not

only discovers the required transition steps and executes them but also records them into the recorder instance. The 'filled' recorder is passed back to the method `TransitionTo()` where the recorded `TransitionChain` is extracted and passed back to the caller. This allows the caller to pass in the filled `TransitionChain` the next time when the same static transition needs to be executed. In this case the method `TransitionTo()` can directly execute the recorded steps (see method `ExecuteTransitionChain()`).

As already mentioned in the introduction, the C# implementation uses an ArrayList instead of a fixed size array to hold the individual transition steps that are discovered when a dynamic transition is executed or a static transition is executed for the first time. As a result the C# version of the QHsm can be compiled into a utility dll that can handle arbitrarily complex / deep state hierarchies. (The ArrayList is defined in the method `TransitionFromSourceToTarget()`.)

Debugging & Tracing

The usage of delegates (or pointers to member functions) makes it difficult to find out what state a given state machine is currently in. E.g. during a debugging session, the value of the QHsm member variable `m_MyState` is displayed as {HierarchicalStateMachine.QState}, which certainly is not very helpful. In order to alleviate this problem, the C# version of QHsm adds the property `CurrentStateName`. The name is retrieved by using reflection to determine the name of the method that the delegate points to:

```
public string CurrentStateName
{
      get { return m_MyState.Method.Name; }
}
```

The exposed name property simplifies tasks like debugging, tracing, and logging.

Test Harness

The project QHsmTest constitutes a Console application (see Main) that allows the user to drive the state machine QHsmTest, which is an implementation of the Samek's QHsmTst statechart (see Figure 4.3 in his book).

Essentially, the code in the class QHsmTest is a replica of the code provided in Listing 4.5 of Samek's book. There are three main differences that are caused by the language differences between C# and C++:

- As discussed earlier the various signals must be cast to int
- Since C# does not (yet) support macros the call into TranistionTo needs to be spelled out instead of using a simpler macro (compare with Q_TRAN in Samek's code).
- C# does not support the concept of static variables in a method. Static variables must be defined in the class scope. Unfortunately, this means that the various static variable that hold TransitionChain objects must be manually defined outside of the associated method; e.g.:

```
private static TransitionChain s_Tran_s0_s211;
```

```
protected QState s0(IQEvent qEvent)
```

The project also includes sample code that demonstrates how to derive from an existing state machine (see class QHsmTestDerived). The derived class can also be driven from the console app.

Inheriting from a given state machine just requires that the actual state handler method (not the delegate declaration) is marked virtual.

This is demonstrated in the sample QHsmTest for the state s1:

```
protected virtual QState s1(IQEvent qEvent)
```

No other change is required in the base class. The code for the handler of state s1 in the inheriting class is available as part of the class QHsmTestDerived. Essentially it boils down to very few lines:

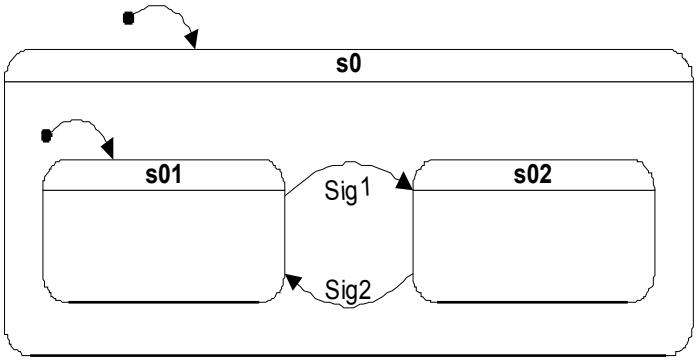
```
protected override QState s1(IQEvent qEvent)
{
   switch (qEvent.QSignal)
   {
      case (int)MyQSignals.B_Sig: Console.Write("s1-B-overriden;");
                                  return null;
   }
   // Everything else we pass on to the state handler of the base class
   return base.s1(qEvent);
}
```

Static Transitions and Derived State Machines

As discussed in section 6.3 of Samek's book, the optimized static transitions break if you use instances of different state machines that leverage the same base machine.

Demonstration of the Problem

The project OptimizationBreaker demonstrates the problem. When the project is executed, the user is asked to select the pair of classes that should be tested. This section uses the first pair of classes, so a 1 should be entered. In `TestStateMachines(…)` an instance of the state machine QHsmBase1 is created, which has the following simple state hierarchy:

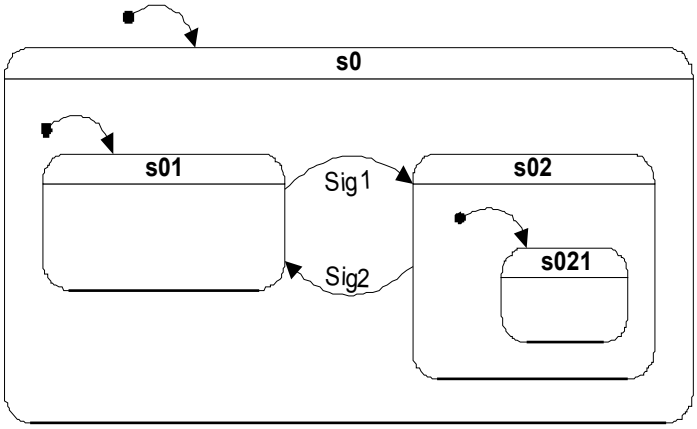


By simply sending the signals Sig1 and Sig2 after the state machine was initialized, we cover all possible static transitions. We send those signals twice; the first time the transition chain for the static transitions is cached; the second time just proves that the cached chains are correctly replayed.

The generated output looks like this:

```
Creating instance of QHsmBase1 and sending signals to it:
top-INIT;s0-ENTRY;s0-INIT;s01-ENTRY;
s01-Sig1;s01-EXIT;s02-ENTRY;
s02-Sig2;s02-EXIT;s01-ENTRY;
s01-Sig1;s01-EXIT;s02-ENTRY;
s02-Sig2;s02-EXIT;s01-ENTRY;
```

There is nothing special so far. Now we create an instance of the class QHsmDerived1. It derives from QHsmBase1and adds the sub state s021. Its state chart looks like this:



In contrast to the state machine QHsmBase1, QHsmDerived1 should automatically transition into state s021, when it receives the signal Sig1 in state s01. However, when we send signal Sig1 to state s01, it ends up in state s02 as before. Here is the generated output:

```
Creating instance of QHsmDerived1 and sending signals to it:
top-INIT;s0-ENTRY;s0-INIT;s01-ENTRY;
s01-Sig1;s01-EXIT;s02-ENTRY;
```

The reason, of course, is that it used the transition chain that was recorded when the earlier created instance of QHsmBase1first handled signal Sig1 in state s01.

A Naive Solution

One solution would be to change the code so that instances of state machine classes store transition chains in member variables instead of static variables as demonstrated in the following code snippet for a modified version of QHsmBase1 (the relevant parts are highlighted in yellow):

```
protected QState s01(IQEvent qEvent)
{
      TransitionsChain m_Tran_s01_s02 = null;
      switch (qEvent.QSignal)
      {
            case (ushort)QSignals.Entry:  Console.Write("s01-ENTRY;"); return null;
            case (ushort)QSignals.Exit:   Console.Write("s01-EXIT;");  return null;
            case (ushort)MyQSignals.Sig1: Console.Write("s01-Sig1;");
                                          TransitionTo(m_s02, ref m_Tran_s01_s02);
                                          return null;
      }
      return m_s0;
}

protected virtual QState s02(IQEvent qEvent)
{
      TransitionsChain m_Tran_s02_s01 = null;
      switch (qEvent.QSignal)
      {
            case (ushort)QSignals.Entry:   Console.Write("s02-ENTRY;"); return null;
            case (ushort)QSignals.Exit:    Console.Write("s02-EXIT;");  return null;
            case (ushort)MyQSignals.Sig2:  Console.Write("s02-Sig2;");
                                           TransitionTo(m_s01, ref m_Tran_s02_s01);
                                           return null;
      }
      return m_s0;
}
```

Changing the base class in this way and running the application again, yields the correct behavior:

```
Creating instance of QHsmDerived1 and sending signals to it:
top-INIT;s0-ENTRY;s0-INIT;s01-ENTRY;
s01-Sig1;s01-EXIT;s02-ENTRY;s02-INIT;s021-ENTRY;
s02-Sig2;s021-EXIT;s02-EXIT;s01-ENTRY;
```

Unfortunately this solution is rather draconian since now each instance carries the memory burden of all the TransitionChain objects that are required to cache the various transition chains. While this might be acceptable for relatively low numbers of state machine instances, it becomes prohibitively expensive, when many instances are required.

A Better Solution

The naïve solution, which uses member variables to store the transition chains went too far. We don't need TransitionChain objects per instance. We need one TransitionChain instance for each transition per **class** or **type**. I.e., when we directly create an instance of QHsmBase1 then we want to use transition chains that are statically assigned in the class QHsmBase1. However, an instance of QHsmDerived1 should use transition chains that are statically assigned to the class QHsmDerived1, even for transitions that are fully defined in the base class. Essentially we need to 'virtualize' the static transition chains in use by QHsmBase1.

The pair of classes QHsmBase**2** and QHsmDerived**2** show a solution. As before, the code can be tested by executing the project OptimizationBreaker. This time a 2 must be entered to exercise the second pair of classes.

Instead of directly using a static TransitionChain variable that is defined for a given transition (like `s_Tran_s01_s02` in QHsmBase1), we now use an index into an array of TransitionChain objects. The array is accessed through a getter function that is defined as virtual in the base class QHsmBase2 so that the deriving class can override it to provide access to its own static array of TransitionStore objects. By means of the polymorphic getter function, the correct store is accessed even though the same index is used within the TransitionTo(…) method:

```
TransitionTo(m_s02, ref TransitionChains[(int)TransIdx.s01_s02]);
```

In order to make sure that the right index is used, the sample code uses an enumeration:

```
protected enum TransIdx
{
      s01_s02,
      s02_s01,
      End
}
```

The last entry of the enumeration (`End`) is used to identify the highest required index in the static array of TransitionsStore objects:

```
private static TransitionChain[] s_TransitionChains =
              new TransitionChain[(int)TransIdx.End];
```

The last missing piece is the virtual getter method that wraps the access to the static array in a polymorphic way:

```
protected virtual TransitionChain[] TransitionChains
{
      get { return s_TransitionChains; }
}
```

As demonstrated in the class QHsmDerived2, the deriving class creates its own static array of TransitionsChain objects and overrides the getter function to return its own static array instead of the array of the base class. The array of the deriving class must be at least big enough to hold all entries required by the base class. Since the enumeration in the base class is marked

protected, the deriving class can size its own array using the entry ‘end’ in the enumeration again:

```
private static TransitionChain[] s_TransitionChains =
               new TransitionChain[(int)QHsmBase2.TransIdx.End];
```

If the derived class introduces additional transitions (not shown in the sample code) then it should create its own enumeration whose first entry has the same value as the entry ‘end’ in the base class enumeration like so:

```
protected new enum TransIdx
{
      s021_s02 = (int)QHsmBase2.TransIdx.End,
      End
}

private static TransitionChain[] s_TransitionChains =
               new TransitionChain[(int)TransIdx.End];
```

For symmetry reasons I use the same enumeration name TransIdx in the derived class. Since this name clashes with the same name that is used in the base class the keyword new is required. The start index is wired up with the entry ‘end’ in the enumeration of the base class. This solution avoids the waste of memory introduced by the naïve solution, and at the same time still makes sure that expensive dynamic transitions can be completely avoided in a generic way. It is worth noting that all concepts introduced in this solution are available in C+(+) and hence can be used in conjunction with Samek’s code, too.

The price for the suggested solution is the additional code: Each class must define a static array of TransitionChain objects of the correct size and must implement the getter function that provides access to the array. Both aspects constitute boilerplate code and therefore pose no real obstacle. However, unfortunately the solution requires that every *potential* transition be identified, and an enum entry provided, in case the transition is actually taken at some time. It is then subject to the error-prone requirement that the correct enumeration be supplied when the transition is taken.

The Final(?) Solution

Apart from the need to define unique indices and use them correctly, the previous solution has all the required ingredients. The solution presented in this section removes the error prone requirement to define unique indices by leveraging the reflection capabilities of the .Net platform. A similar approach should be possible in JAVA but I don’t know whether it could be ported to C/C++.

The solution is demonstrated by the pair of classes QHsmBase**3** and QHsmDerived**3**., which can be executed by entering a 3 in the associated console application.

At the center of the solution stands the class `TransitionChainStore`, which is defined as a nested class in `QHsm`. It encapsulates an array of `TransitionChain` objects and provides a means to get a unique index. A state machine assigns a new instance of the class

`TransitionChainStore` to its static protected variable `s_TransitionChainStore` right after the class declaration line:

```
public class QHsmBase3 : QHsm
{
   protected static new TransitionChainStore s_TransitionChainStore =
     new TransitionChainStore(
       System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
```

Since this line is the first static variable assignment in the class, the `TransitionChainStore` instance is created first during the static construction of the state machine class. Note that the actual class (type) information is passed into the constructor of the `TransitionChainStore` via the call into the reflection subsystem of .Net:

```
System.Reflection.MethodBase.GetCurrentMethod().DeclaringType
```

In its constructor, the `TransitionChainStore` first retrieves the base type of the passed in type and retrieves its static TransitionChainStore (again using reflection). Access to the static aspects of the base class is always possible since the static construction of a base class is always completed before the static constructor of the deriving class is invoked.

The access to the TranstionChainStore of the base class allows the constructing TranstionChainStore to reserve the TransitionChain slots that are in use by the base class. Once the TranstionChainStore is constructed all other static variable assignments that are defined outside of the actual static constructor are executed.

These are the static integer variables that hold the slot numbers of the TransitionChain for the associated transition:

```
private static int s_TranIdx_s01_s02 = s_TransitionChainStore.GetOpenSlot();
```

Instead of hard coding the indices via the enumerations that we used in the previous solution, now the slot numbers are retrieved dynamically by calling into the static TransitionChainStore that is defined in the class.

Once all the static transition indices have been assigned, the explicit static constructor code is executed:

```
static QHsmBase3()
{
      s_TransitionChainStore.ShrinkToActualSize();
}
```

It is used to shrink the array of TransitionChain objects in the TransitionChainStore to its actual size in order to conserve memory. (Before, the array size was increased in steps whenever the array was too small to accommodate a new requested slot.)

This completes the static construction of the state machine. At this time all indices are determined. In order to simplify the code for the execution of an actual transition, I added a

new version of the method TransitionTo() to QHsm. It takes the target state and the index of the associated TransitionChain as an argument.

```
protected void TransitionTo(QState targetState, int chainIndex)
```

The class QHsmBase3 in the project OptimizationBreaker demonstrates its usage. Now if only C# would allow the usage of static variables inside a method (as C++ does) then even the same variable names for transition indices could be used. Without this we are forced to define static variables with unique names holding the chain indices (as before with TranisitionChain objects). However, this limitation is not introduced by the handling of derived state machines but is an inherent part of the C# port that also exists in the class QHsmTest.

This discussion of the solution only provides a high level overview. The comments in the code hopefully explain the details. It is important to note that the solution leverages the knowledge about how the C# compiler orders the execution of the individual static construction steps. Other compilers like the VB.Net compiler might result in a different sequence and hence might break the solution. I did not test whether a state machine coded in VB.Net against the QHsm would work in the same way.

I think for C# the code provided in this section provides a generic and simple solution that makes it no longer necessary to deal with non-optimized dynamic transitions when implementing inherited state machines. Therefore I recommend that the solution described in this section should be always applied even if there are no plans to derive from the developed state machine yet. The generic solution only requires a few lines of generic boilerplate code to be added to the state machine. It could be argued that due to the usage of relatively expensive reflection methods the generic solution might impose too much overhead. However, reflection is only used during the static construction of a state machine and hence constitutes only a one time overhead per state machine type.

Summary

The provided sample code provides a complete port of the Quantum Hierarchical State Machine to C#. The discussion of the code explains the main differences between C++ and C# as they affect the port. The provided implementation of the QHsm is embedded in a dll assembly that can be used for implementing hierarchical state machines of arbitrary complexity and depth without the need to change and recompile the base class.

Apart from the actual port the sample code also introduces a recipe for enabling static transitions in derived state machines. Two solutions are provided:

The first relies on the definition of unique indices via enumerations. It can be easily ported back to C/C++ but is relatively cumbersome and error prone since the implementer has to make sure that unique indices are correctly defined and used.

The final provided solution leverages the powerful reflection capabilities of the .Net platform to automatically ensure unique indices. I think that applying this solution is no more complex

than developing a hierarchical state machine that is not prepared for static transitions in derived classes but offers the big advantage that expensive dynamic transitions are no longer required.