StateProto White Paper

statedriven@users.sourceforge.net

01 July 2006

draft

Table of Contents

Illustration Index

About StateProto

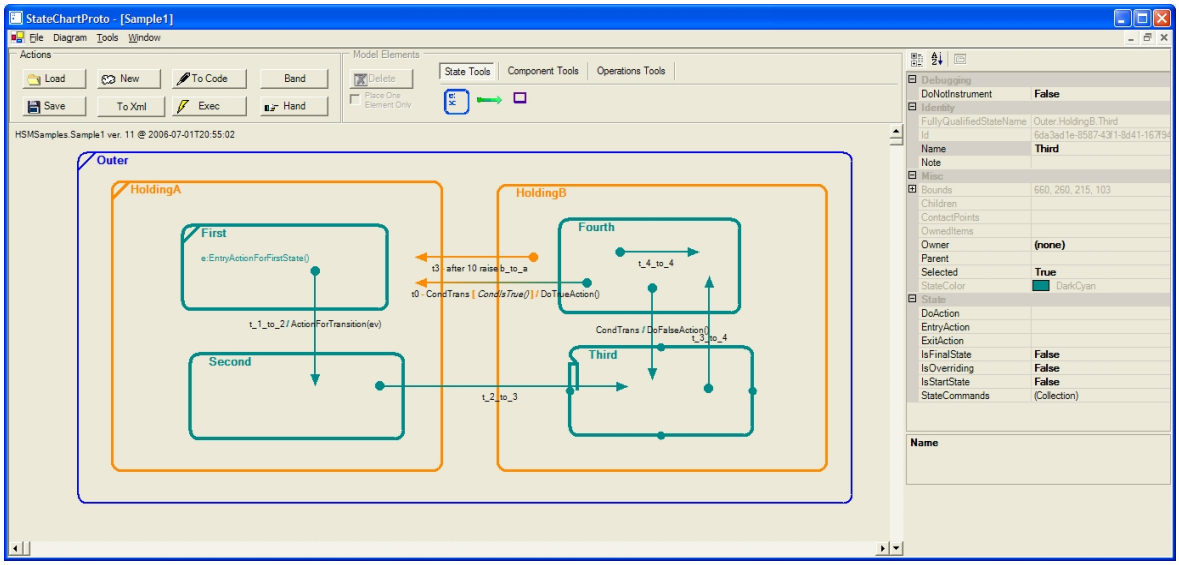

Illustration 1: State Chart Proto Application

I wrote StateProto because I needed something that would generate the boiler plate code needed for a stereotypical state machine based class design. I am not going to go into why use state machines in this document. The focus will instead be that of using StateProto and the underlying QF4Net library to implement a class whose behaviour is defined by a set of explicit states and the transitions from one state to the next.

This software and the design decisions taken is the result of a combination of two things:

a. my experience thus far with state driven behaviour concepts, and

b. the easiest coding that would get me there.

In terms of status - for now StateProto gets the job done for the design scenarios I have encountered. While I am not altogether proud of the code in StateProto - I do know that it can generate the boiler plate C# code I need to implement a state machine design.

The State Machine Engine

While StateProto is used for drawing state machines – it is not the State Machine Execution Engine. I use a modified version of a framework called qf4net written by Rainer Hessmer, Ph.D. and modified by David Shields. Qf4net is itself a C# port of the Quantum Framework (C and C++) by Miro Samek, Ph.D.

Miro Samek, Ph.D. http://www.quantum-leaps.com.

Rainer Hessmer, Ph.D. http://www.hessmer.org/dev/qhsm/index.html.

David Shields david@shields.net.

The Quantum Framework and Qf4net are all much more flexible implementations of the StateMethod style of state machine implementation. It should be noted that many of the negatives stated in the StateMethod definition is not evident in Samek's design nor Hessmer's translation.

I currently only support the qf4net library and can only generate C# code. With a bit of redesign this can change to include language plugins – but this work has not yet been done.

Please spend time reading the excellent articles written by Samek as well as the explanation by Hessmer on how qf4net was implemented[1].

StateProto as a State Machine Drawing Tool

StateProto has its own variations on drawing a Hierarchical State Machine.

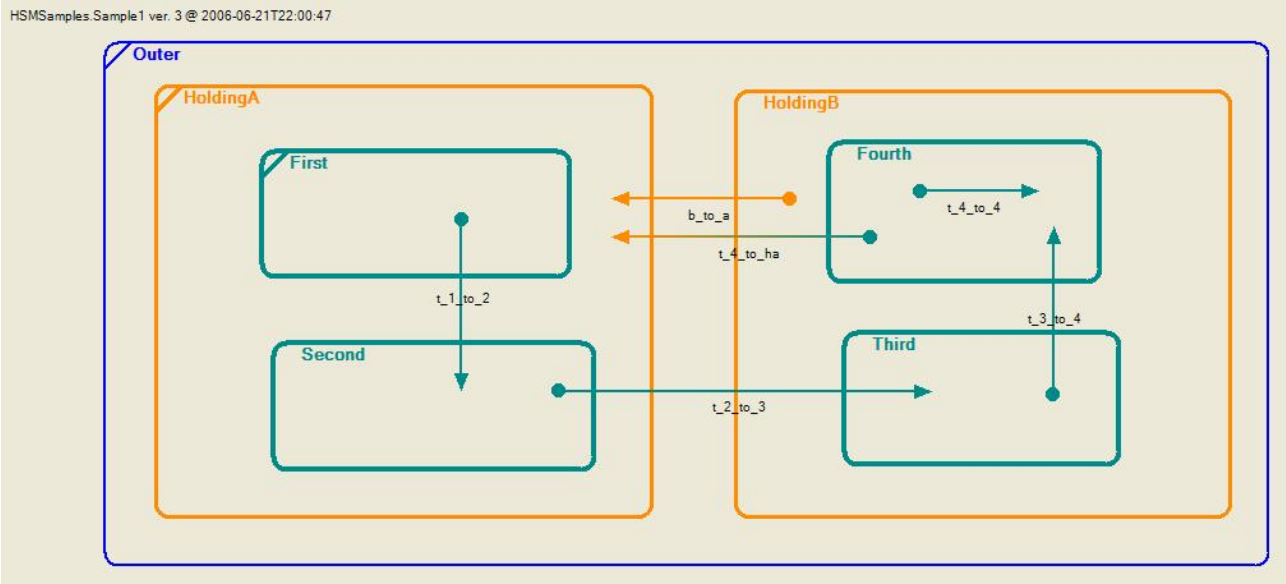

Illustration 2: Sample Meaningless State Machine

Features:

- All states at the same nesting level have the same colour. This makes a number of things easier:
    1. It is easy to see peer states, and
    2. If one state is nested within another it takes on the next level colour, and
    3. When a transition goes from one state to another then it transitions in colour from the source state to that of the target state.
- The border of a state becomes slightly thicker at each nesting level. This helps, once again, with immediate differentiation. A thicker border means the state is more deeply nested.
- The Default or Start State at any level is indicated with a horizontal line in the top left corner of the state. There must be one and only one start state at the outermost level. At any other nesting level there can be no more than one start state within an immediate parent state.
- A Final State is indicated by a diagonal line in the bottom right corner of a state. Although the software does not constrain the design if a state is defined as a final state.
- The currently selected state is marked with resize dots. To move a state just grab inside it and drag. To size a state grab on one of its resize handles and drag.

Note that in the sample state diagram above the state First will be the initial real starting state as Outer will be entered which will immediately handover to HoldingA which in turn will handover to state First. State First can then take a transition to state Second on receiving an event t_1_to_2.

1 For now this document expects that you already understand how to use the Qf4net framework.

What is different from other State Machine Drawing Tools?

1. The default start state is usually indicated via a transition from a dot – StateProto just draws a top-left diagonal in the state glyph. Similarly for final states StateProto just draws a diagonal in the bottom-right corner of the state glyph. I do not directly support actions for entering the default state[2].
2. Transitions to self are usually drawn as a curved transition line leaving the state and then re-entering it. StateProto just draws it as a transition that is a line inside the state (*see transition t_4_to_4*).
3. Transitions do not start cleanly on a states boundary and link to another states boundary. This can be visually appealing but I found that just drawing the transition from anywhere within the source state to anywhere within the target state was (a) easier to do and (b) has the result that by not giving me other options it forced me to just accept the look and not spend endless time fiddling trying to make the diagram look neat[3].

StateProto as a C# Code Generator

Currently, StateProto only generates C# code for a modified version of the standard Qf4net port from Rainer Hessmer. Remember that this is a Method-Per-State type of Hierarchical State Machine (hsm) implementation. I will discuss some of the modifications later. For now I will show some of the main bits of the generated code.

The state machine in Illustration 2: Sample Meaningless State Machine above was given a name of Sample1 and lives in a C# namespace of HSMSamples.

```
34      [TransitionEvent ("t_3_to_4")]
35      [TransitionEvent ("t_4_to_4")]
36      [TransitionEvent ("t_4_to_ha")]
37      public class Sample1 : LQHsm, ISigSample1 {
38
39          //-------------------------------------------------
40          //Begin[[ClassBodyCode]]
41          //End[[ClassBodyCode]]
```

Illustration 3: Start of Class Definition and ClassBodyCode block

Notice that the Sample1 hsm derives off LQHsm (which is derived off QHsm from Qf4net) and implements the ISigSample1 interface.

This interface shows up later as:

2 If there is some compelling reason to add this – it will not be hard to do.
3 Although there are many other usability items that I would like to add.

```
319     public interface ISigSample1
320     {
321         void Sigb_to_a (object data);
322         void Sigt_1_to_2 (object data);
323         void Sigt_2_to_3 (object data);
324         void Sigt_3_to_4 (object data);
325         void Sigt_4_to_4 (object data);
326         void Sigt_4_to_ha (object data);
327     }
```

Illustration 4: State Chart Specific Signals Interface

which allows you to more easily raise the specific signals to the state machine that it knows about.

Also, take note of the //Begin[[ClassBodyCode]] and //End[[ClassBodyCode]]. This format **//Begin[[*something*]]** and **//End[[*something*]]** is reserved for use by StateProto and defines code sections within the generated file that the developer can freely add code to. As long as the csharp file is saved – any regeneration after that from the state model will maintain the code within these blocks. The ClassBodyCode section is the main section within the class for the developer to place all business action and guard methods.

What does a state get generated to?

```
189
190     #region State Outer_HoldingA_First
191     protected static int s_trans_t_1_to_2_Outer_HoldingA_First_2_Outer_HoldingA_Second = s_TransitionChainStore.GetOpenSlot ();
192     [StateMethod ("Outer_HoldingA_First")]
193     protected virtual QState S_Outer_HoldingA_First (IQEvent ev){
194         switch (ev.QSignal){
195         case QSignals.Entry: {
196             LogStateEvent (StateLogType.Entry, s_Outer_HoldingA_First);
197         } return null;
198         case QSignals.Exit: {
199             LogStateEvent (StateLogType.Exit, s_Outer_HoldingA_First);
200         } return null;
201         case QualifiedSample1Signals.t_1_to_2: {
202             LogStateEvent (StateLogType.EventTransition, s_Outer_HoldingA_First, s_Outer_HoldingA_Second, "t_1_to_2", "t_1_to_2");
203             TransitionTo (s_Outer_HoldingA_Second, s_trans_t_1_to_2_Outer_HoldingA_First_2_Outer_HoldingA_Second);
204             return null;
205         }  // t_1_to_2
206         } // switch
207
208         return s_Outer_HoldingA;
209     } // S_Outer_HoldingA_First
210     #endregion
```

Illustration 5: Code generate for state First

This is the First (three levels of nesting) state which has HoldingA and then Outer as successive parent states. The three states are used to derive the method name viz., S_Outer_HoldingA_First.

This has some repercussions:

1. All characters in state names must also be valid as part of a method name in C#.
2. To make it easier to discern – it is also probably best that an underscore is not used in state names.

The First state has only one transition which takes us from First to Second. Notice the LogStateEvent(...) calls that allows for some level of instrumentation of the state machine.

The case selectors QualifiedSample1Signals.t_1_to_2 are constants that are defined as follows:

```
328     public class QualifiedSample1Signals
329     {
330         public const string b_to_a = "b_to_a";
331         public const string t_1_to_2 = "t_1_to_2";
332         public const string t_2_to_3 = "t_2_to_3";
333         public const string t_3_to_4 = "t_3_to_4";
334         public const string t_4_to_4 = "t_4_to_4";
335         public const string t_4_to_ha = "t_4_to_ha";
336     }
337     public class Sample1Signals
338     {
339         public const string b_to_a = "b_to_a";
340         public const string t_1_to_2 = "t_1_to_2";
341         public const string t_2_to_3 = "t_2_to_3";
342         public const string t_3_to_4 = "t_3_to_4";
343         public const string t_4_to_4 = "t_4_to_4";
344         public const string t_4_to_ha = "t_4_to_ha";
345     }
```

Illustration 6: State specific Signals as Constants

The difference between the QualifiedSample1Signals and the Sample1Signals classes will be made evident later.

A Qf4net user might notice that the signals are strings instead of ints or an enum. Strings give the benefit of readability of the switch statements with the disadvantage of a slight reduction in performance[4].

Entry, Exit and Transition Actions

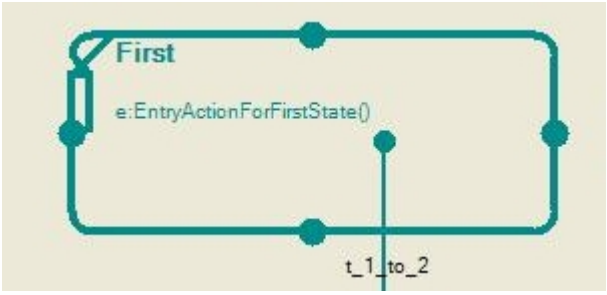

Illustration 7: Entry Action

An entry action shows as **e:** prefixed text within a state. The preference is that whatever is entered as the action code is a method call although it can also be any other valid C# code. Please take note the sizing handles that are displayed on the currently selected state.

4 This is the biggest change to the underlying Qf4net library. Some other additions were made to the QEvent. Most remaining additions were done as an extension library.

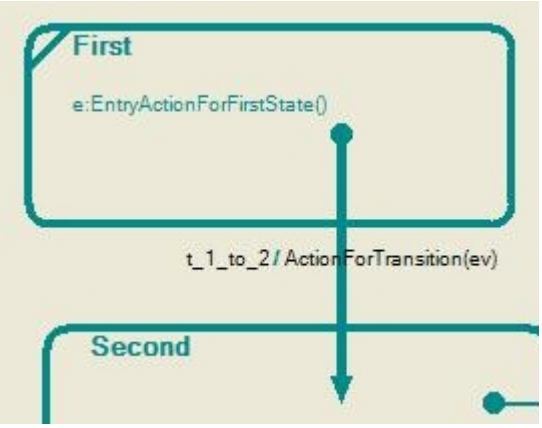

Illustration 8: Transition Action

The situation is similar for transitions. An action can be placed after the / in a transition. The recommendation again is that this be a method call. The implementation for the method call must be done in the ClassBodyCode block discussed earlier. Please take note here that the currently selected transition is bolder than any other transition on the drawing.

An exit action will be displayed with an **x:** prefix within the state glyph.

Conditional Transitions

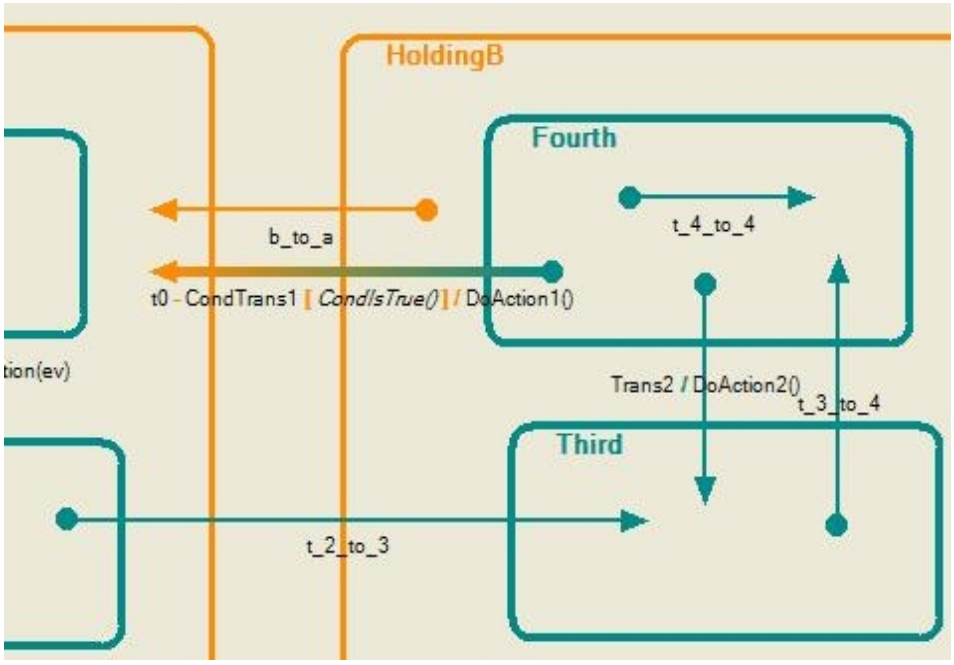

Illustration 9: Condition on CondTrans1 transition

```
262
263 -       #region State Outer_HoldingB_Fourth
264         protected static int s_trans_t_4_to_4_Outer_HoldingB_Four
265         protected static int s_trans_t0_CondTrans1_Outer_HoldingB
266         protected static int s_trans_Trans2_Outer_HoldingB_Fourth
267         [StateMethod ("Outer_HoldingB_Fourth")]
268 -       protected virtual QState S_Outer_HoldingB_Fourth (IQEvent
269 -           switch (ev.QSignal){
270 -           case QSignals.Entry: {
271                 LogStateEvent (StateLogType.Entry, s_Outer_HoldingB
272             } return null;
273 -           case QSignals.Exit: {
274                 LogStateEvent (StateLogType.Exit, s_Outer_HoldingB_
275             } return null;
276 -           case QualifiedSample1Signals.CondTrans1: {
277 -               if (CondIsTrue()) {
278                     DoAction1();
279                     LogStateEvent (StateLogType.EventTransition, s_
280                     TransitionTo (s_Outer_HoldingA, s_trans_t0_Cond
281                     return null;
282                 }
283             } break; // CondTrans1
284 -           case QualifiedSample1Signals.t_4_to_4: {
285                 LogStateEvent (StateLogType.EventTransition, s_Oute
286                 TransitionTo (s_Outer_HoldingB_Fourth, s_trans_t_4_
287                 return null;
288             }  // t_4_to_4
289 -           case QualifiedSample1Signals.Trans2: {
290                 DoAction2();
291                 LogStateEvent (StateLogType.EventTransition, s_Oute
292                 TransitionTo (s_Outer_HoldingB_Third, s_trans_Trans
293                 return null;
294             }  // Trans2
295             } // switch
296
297             return s_Outer_HoldingB;
298         } // S_Outer_HoldingB_Fourth
299         #endregion
```

Illustration 10: Code generated for CondTrans1 transition has if test

The t4_to_ha transition is changed to respond to a CondTrans1 signal and set with a condition that must be true before allowing the DoAction1() to be executed and the transition itself to be taken to state HoldingA.

The condition (such as CondIsTrue()) is also referred to as a Guard.

Code generation produces a standard “if” test within the CondTrans1 case of the event switch in state Fourth. Notice how the DoAction1() and actual TransitionTo(...) are held within the if block.

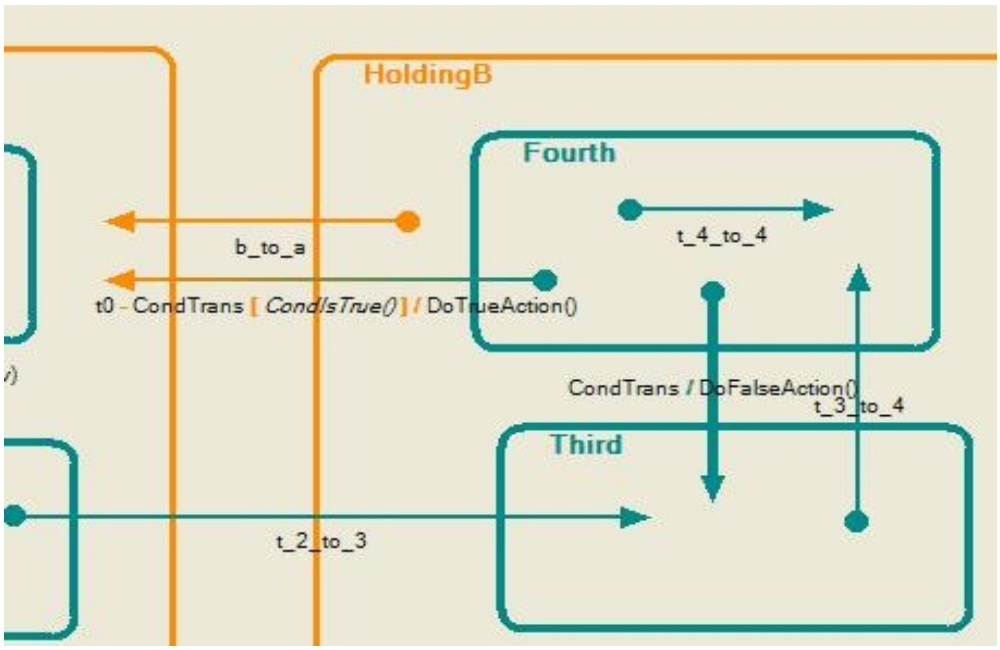

Illustration 11: Two CondTrans signals -- one with Guard Condition

More than one transition can receive the same signal from within a single state. The provisor is that each of these transitions bar one must have a guard condition. The guard conditions should be mutually exclusive. As I said, all bar one – the one without a condition becomes the final else part of the "if" tests.

```
261
262 -       #region State Outer_HoldingB_Fourth
263         protected static int s_trans_CondTrans_Outer_HoldingB_Fourth_2_Outer_
264         protected static int s_trans_t_4_to_4_Outer_HoldingB_Fourth_2_Outer_
265         protected static int s_trans_t0_CondTrans_Outer_HoldingB_Fourth_2_Ou
266         [StateMethod ("Outer_HoldingB_Fourth")]
267 -       protected virtual QState S_Outer_HoldingB_Fourth (IQEvent ev){
268 -           switch (ev.QSignal){
269 -           case QSignals.Entry: {
270                 LogStateEvent (StateLogType.Entry, s_Outer_HoldingB_Fourth);
271             } return null;
272 -           case QSignals.Exit: {
273                 LogStateEvent (StateLogType.Exit, s_Outer_HoldingB_Fourth);
274             } return null;
275 -           case QualifiedSample1Signals.CondTrans: {
276 -               if (CondIsTrue()) {
277                     DoTrueAction();
278                     LogStateEvent (StateLogType.EventTransition, s_Outer_Holdi
279                     TransitionTo (s_Outer_HoldingA, s_trans_t0_CondTrans_Outer_
280                     return null;
281                 }
282 -               else {
283                     DoFalseAction();
284                     LogStateEvent (StateLogType.EventTransition, s_Outer_Holdi
285                     TransitionTo (s_Outer_HoldingB_Third, s_trans_CondTrans_Out
286                     return null;
287                 }
288             } // CondTrans
289 -           case QualifiedSample1Signals.t_4_to_4: {
290                 LogStateEvent (StateLogType.EventTransition, s_Outer_HoldingB_
291                 TransitionTo (s_Outer_HoldingB_Fourth, s_trans_t_4_to_4_Outer_
292                 return null;
293             } // t_4_to_4
294             } // switch
295
296             return s_Outer_HoldingB;
297         } // S_Outer_HoldingB_Fourth
298         #endregion
```

Illustration 12: Code generated for 2 CondTrans transitions forming if-else blocks

TimeOut Transitions

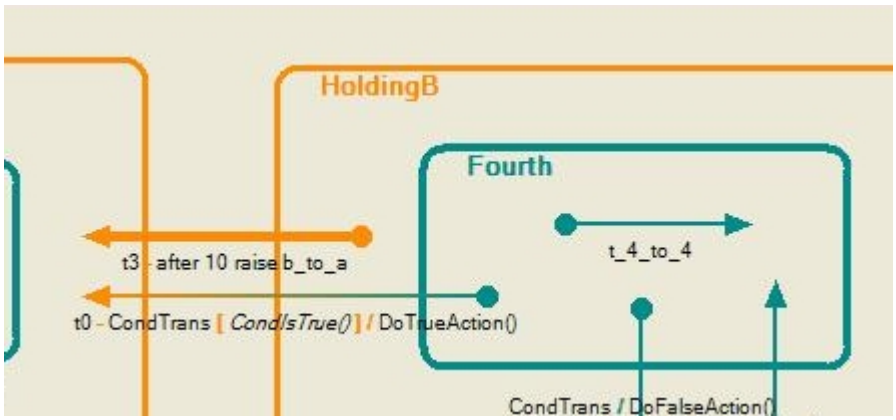

Illustration 13: TimeOut -- after 10 second raise the b_to_a signal

A timeout can be set against a transition based on a timeout expression of either after

TIMEPERIOD or every TIMEPERIOD. The semantic in this code generator is that the timeout is defined as a timeout interval that raises a properly named signal after the timeout TIMEPERIOD.

```
238
239 -       #region State Outer_HoldingB
240         protected static int s_trans_t3_b_to_a_Outer_HoldingB_2_Outer_HoldingA = s_
241         [StateMethod ("Outer_HoldingB")]
242 -       protected virtual QState S_Outer_HoldingB (IQEvent ev){
243 -           switch (ev.QSignal){
244 -           case QSignals.Entry: {
245                 LogStateEvent (StateLogType.Entry, s_Outer_HoldingB);
246                 SetTimeOut ("Outer_HoldingB_t3_b_to_a", TimeSpan.FromSeconds (10), ne
247             } return null;
248 -           case QSignals.Exit: {
249                 ClearTimeOut ("Outer_HoldingB_t3_b_to_a");
250                 LogStateEvent (StateLogType.Exit, s_Outer_HoldingB);
251             } return null;
252 -           case QualifiedSample1Signals.b_to_a: {
253                 LogStateEvent (StateLogType.EventTransition, s_Outer_HoldingB, s_Oute
254                 TransitionTo (s_Outer_HoldingA, s_trans_t3_b_to_a_Outer_HoldingB_2_Out
255                 return null;
256             }  // b_to_a
257             } // switch
258
259             return s_Outer;
260         } // S_Outer_HoldingB
261         #endregion
```

Future Articles

1. What to do to actually run class Sample1. This looks at EventManagers and Runners.
2. Getting the Source – where to checkout etc.
3. Explaining the innards.
4. Other processing goals.