
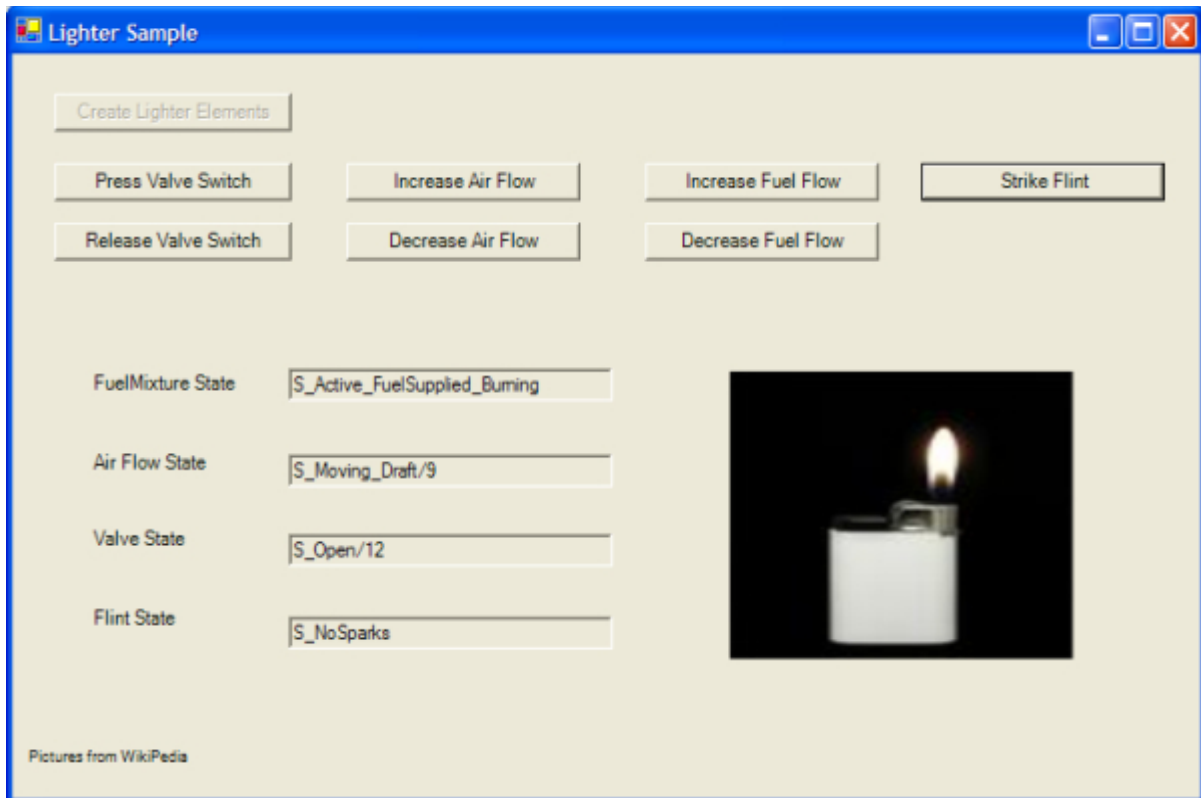


-  Download demo project - 593 Kb
- First Article: [StateProto Beta - State Chart Designer for Qf4Net](#)
- Second Article: [StateProto - Executing Multiple StateMachines](#)



- Introduction
- The Sample
- Ports - the means of communication
- The Lighter
 - Starting up
- Creating
- Opening the valve
- Less stable conditions...
- The Frame
- Summary
- Coming Up

Introduction

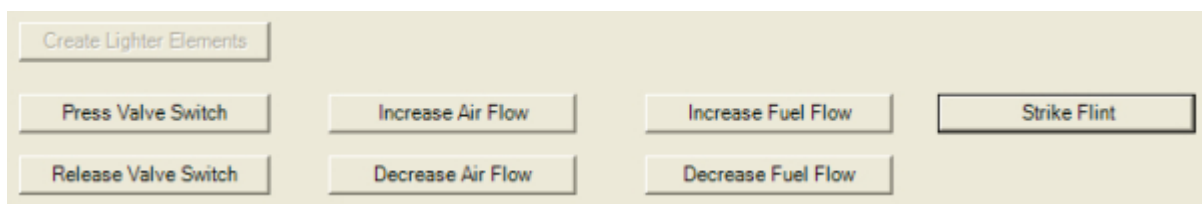
In the last article on stateProto I introduced the mechanisms needed to run one or more instances of a state machine. I also showed some basic animation of the statemachine. In this article the question is - how does one enable different state machine instances to communicate with each other (and possibly with the outside world)?

The Sample

While trying to think of a reasonably simple but interesting sample to put together for this article the ideas always seemed to gravitate towards some physical process - including things like coin operated turnstiles, payed for parking, brake system operation, human-elevator interaction, etcetera. The problem is that these can turn into complex and boring samples.

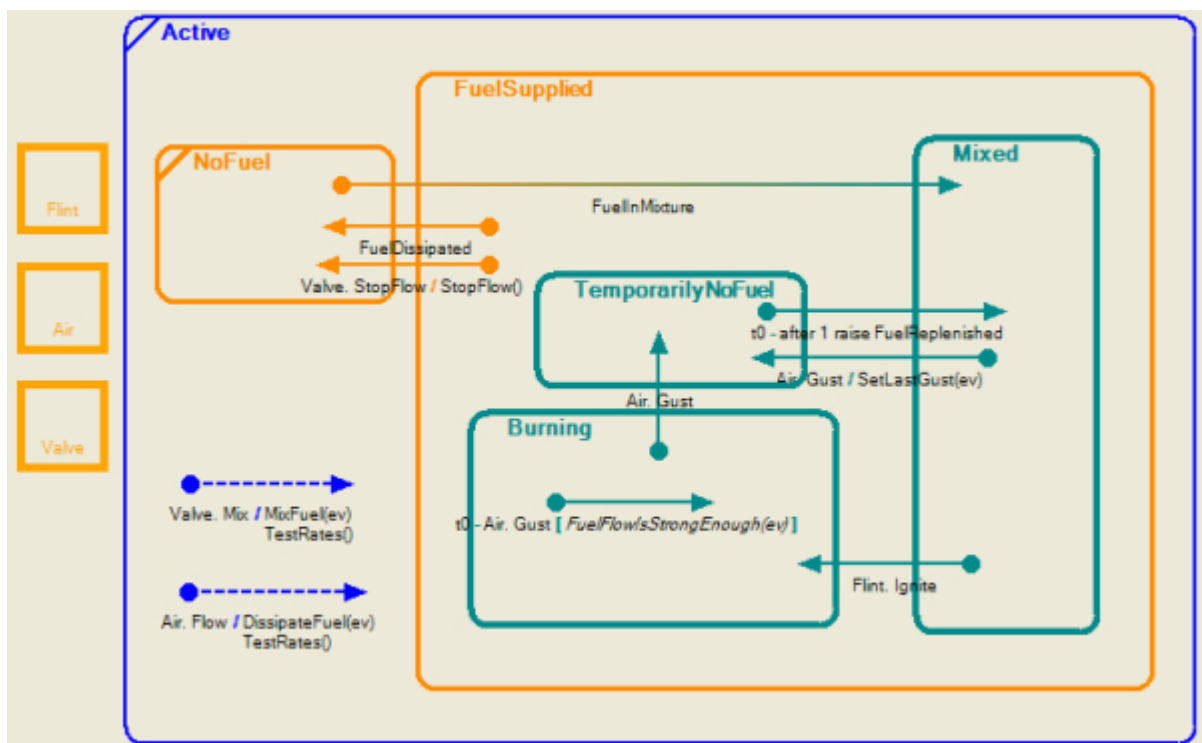
I decided that as a South African I'll think about it further while putting some meat on the braai. To light the braai I bought a small lighter and this led me to thinking about the various interactions that results in the production of a flame. The simplest model I could come up with includes the following domain objects:

1. the **valve** that supplies the fuel,
2. **air** which provides the oxygen source,
3. the resulting **fuel/air** mixture
4. and the **flint** which provides the spark to ignite the fuel/air mixture.



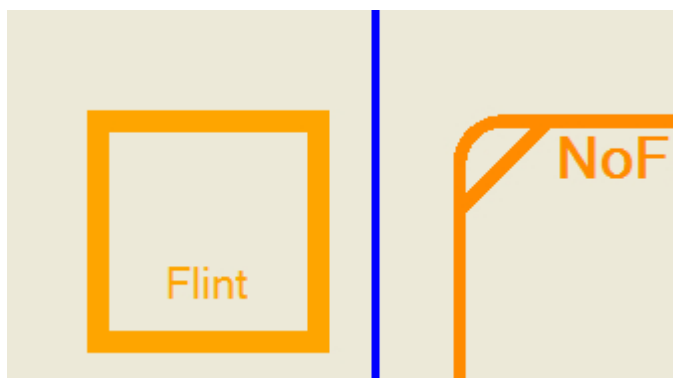
Controls of Demo

Ports - the means of communication



Fuel Mixture Hsm (see ports on the left hand side)

State machines can indicate sources and sinks of events with the use of ports. A port is a placeholder to which another statemachine can be attached that will enable the two state machines to interact.

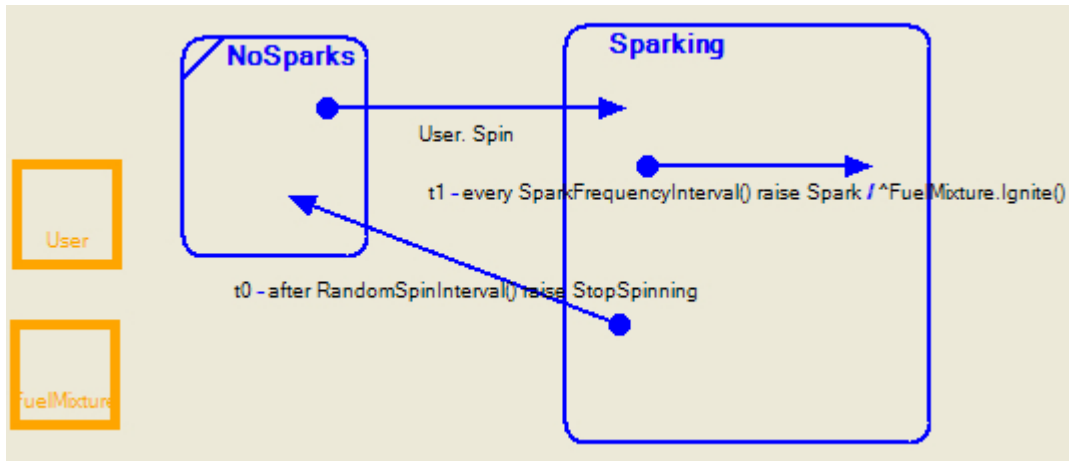


Flint Port of Fuel Mixture Hsm

One state machine can send a message to another state machine via the port that represents the

target state machine via the following expression:

`^TargetStateMachinePort.SignalName(ArgumentToTheSignal)`. For example, when the Flint state machine wants to send a message to the FuelMixture state machine then it sends `^FuelMixture.Ignite()` (note that the `^` means **send**, `FuelMixture` is the port and `Ignite` is the signal to be sent - without any arguments).



Flint Hsm: notice the FuelMixture Port. Also in the **Sparking** state notice the `^FuelMixture.Ignite()` in the action.

If you do not understand the terms State, Signal and Action then please refer to Miro Samek's article referenced below.

Ports work as complementary pairs between state machines - if StateMachine `Flint` wants to send a message to StateMachine `FuelMixture` then `Flint` might have a **FuelMixture** port and `FuelMixture` might have an **Flint** port. It is important to note that the actual naming of the ports is not really of importance here - as we could have called the **Flint** port the **IgnitionSource** port instead.

```

1  class Flint : LQHsm {
2      IQPort FuelMixture;
3  }
4  class FuelMixture : LQHsm {
5      IQPort Flint;
6  }
7
  
```

What is useful about this pattern of using ports as intermediaries is that we can swop out one `Flint` implementation for another without `FuelMixture` being any the wiser. Also, at a later stage - the two state machines could be running in separate processes or even on different machines and neither would be the wiser (assuming latency is not an issue). While not necessarily very useful for `Flint` and `FuelMixture` - the extensibility of this mechanism will be of use to us in the future.

A port is a concrete implementation of an IQPort interface. This interface is defined as follows:

```

1  /// <summary>
2  /// IQPort - single state machine accessor.
3  /// </summary>
4  public interface IQPort
5  {
6      string Name { get; }
7      void Send (IQEvent ev);
8      event QEventHandler QEvents;
9
10     void Receive (IQPort fromPort, IQEvent ev);
11 }
12
13 public delegate void QEventHandler (IQPort port, IQEvent ev);
  
```

This interface can be split into two behavioural components.

- The eventing portion that is used by the sending state machine. `Send` is called which results in the `QEvents` event sink being called - any code that registers with `QEvents` will then

receive the event being sent.

```
1      void Send (IQEvent ev);
2      event QEventHandler QEvents;
```

- It just so happens that the `Receive` method fits the call signature of the `QEventHandler` delegate perfectly. All that is necessary is for the state machine on the other side to register its complementary port's `Receive` method to the sender's `QEvents` sink.

```
1      void Receive (IQPort fromPort, IQEvent ev);
```

Thus for `Flint` and `FuelMixture` above we can link the two up as follows:

```
1      Flint flint = new Flint("flint1", lifeCycleManager);
2      FuelMixture fuelMixture = new FuelMixture("mix1", lifeCycleManager);
3
4      // Setup flint to be able to call FuelMixture.Send(ev1)
5      // This will result in fuelMixture's Flint.Receive() being called
6      // with the message ev1.
7      flint.FuelMixture.QEvents += new QEventHandler(fuelMixture.Flint.Receive);
8
9      // Setup fuelMixture to be able to call Flint.Send(ev2).
10     // This will result in flint.FuelMixture.Receive() being called
11     // with the message ev2.
12     fuelMixture.Flint.QEvents += new QEventHandler(flint.FuelMixture.Receive);
13
14     // activate the two state machines
15     flint.init();
16     fuelMixture.init();
```

As soon as `Receive` is called the message gets placed on the receiving port's owning state machines queue - with a message that is qualified with source information. It is even possible to use ports for sending events to and receiving events from some external (non state machine) actor in the system.

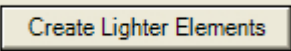
Sending a message from one State Machine to another

The simplest way to send a message from one Hsm to another is to get a direct reference to the other Hsm and call its `AsyncDispatch(ev)` method. The problem with this approach is modularity. Direct access to the second Hsm might cause the developer to attempt to directly call exposed properties, etc on this instance. A port prevents such mishaps. Also, a direct Hsm reference would, at a later stage, make it more difficult to isolate the interacting state machines by placing each into their own application domains (or even into separate process spaces on different machines).

The answer is a simple interface defined by the port which takes on the role of a pipe for data transmission. To keep things simple each individual port supports data flow in one direction only.

The Lighter

Starting up

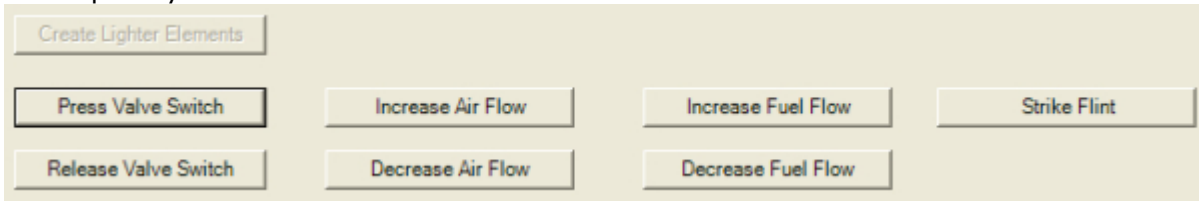


Enabled Create Button

On starting the sample application - you will find yourself presented with a simple gui with one enabled button **"Create Lighter Elements"**.

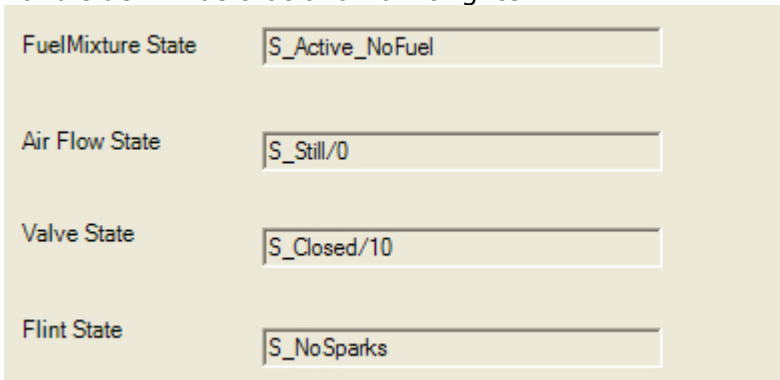
Creating

On clicking this button - it will become disabled - and all the other event input buttons will be subsequently enabled.



Event Input Buttons are all enabled

The elements of the lighter scenario will be in their default startup states. The image on the right hand side will be that of an unlit lighter.



Default Startup State of elements

You will notice that the states in the diagram all start with S_. This is in fact the method name in the generated code. I will leave this out in the text as I discuss these state names.

- **FuelMixture** - Active_NoFuel /fuel=0 /air=0 -- Fuel/Air Mixture is in the active state but no fuel is present.
- **Air Flow** - Still/0 -- The air is currently still with a flow rate of 0 m/s.
- **Valve** - Closed/0 -- The valve is in a closed position with a resulting 0 flow rate.
- **Flint** - NoSparks -- The flint is not being struck and as a result has no sparks.

Some things to note (although this is probably stating the obvious):

1. Even if the flint were to be spun and thus creating sparks - there is no fuel mixture - so no flame will result.
2. If the Valve Switch were pressed - fuel would start flowing with a default flow rate of 10m/s (the middle fuel flow rate in this app).
3. If the air flow rate is faster than the fuel flow rate then the fuel would be dissipated and no flame would result if sparks were to fly
4. If a flame was burning and the air flow got too fast or the fuel flow rate is reduced then the flame would go out.
5. If the Air pressure were increased then the air flow would reach "**gusty**" conditions. This would mean that the flow rate would on average be higher than mere "**drafty**" conditions.
 - Also, ever so often, a gust of higher intensity would come by.
6. So even if the fuel air mix might be surviving the faster gustier flow rates - it might not survive these interim turbulent gusts and would be temporarily depleted of fuel.
7. Very irritating conditions to try to light a braai :-).

Opening the valve

Pressing the valve open "**Press Valve Switch**" will change the states to look as follows:

Highlighted elements have changed state.

- **FuelMixture** - Active_FuelSupplied_Mixed /fuel=10 /air=0 -- Fuel/Air Mixture is in the active state, fuel is being supplied and the mixture is fuel rich (i.e. mixed).
- **Air Flow** - Still/0.
- **Valve** - Open/10 -- The valve has been opened and the fuel is flowing out at 10m/s.
- **Flint** - NoSparks.

Striking the flint now will result in the following change:

- **FuelMixture** - Active_FuelSupplied_Burning /fuel=10 /air=0 -- Fuel/Air Mixture is in the active state, fuel is being supplied and we have a flame!
- **Air Flow** - Still/0.
- **Valve** - Open/10.
- **Flint** - NoSparks -- the flint enters the S_Sparking state and then automatically times out after a random interval returning to the S_NoSparks state. In fact the first time I struck the flint it timed out so fast that it did not even spark. "*Sparking*" is also initiated by a timer (with an *every* expression) which could happen a number of times while the flint is in the Sparking state.



Lighter will remain stably in flame for as long as the Valve is open and the Air is still.

Less stable conditions...

Hitting the "**Increase Air Flow**" results in a less stable flame. It will move about in the drafty air - but because the initial fuel flow started at 10m/s - the flame will not go out. Decreasing the fuel flow rate "**Decrease Fuel Flow**" twice results in a situation where the flame could be blown out by the drafty conditions.

- **FuelMixture** - Active_FuelSupplied_Mixed or Active_NoFuel /fuel=8 /air=8 -- The previously burning mixture could have its flame extinguished by the moving air.
- **Air Flow** - Moving_Draft/8 -- Even though I show /8 here - this speed is continuously (every second or two) changing.
- **Valve** - Open/8 -- Decrease fuel flow was clicked twice which took the flow rate to 8m/s.
- **Flint** - NoSparks.

If you were to strike the flint while the fuelmixture was "Mixed" - you might stand a chance that the mixture takes flame - but could soon be extinguished again. Increasing the fuel flow rate would increase your chances at getting a stable flame again.

Even less stable conditions would result if the air flow were to increase to gusty conditions. This implies a higher speed air flow rate. The air flow rate (like its drafty counterpart) will also change randomly every 1 to 2 seconds. But this higher flow rate is not the only nuance of this air flow state.

Ever so often - a higher (but short lived) turbulent gust would come along which could temporarily deplete the fuel - but because the valve is still open, and if the normal air flow rate is less than the fuel flow rate - would be replenished a second or two later. This depletion scenario will have the following states:

- **FuelMixture** - Active_FuelSupplied_Mixed or Active_NoFuel /fuel=8 /air=12 /lastgust=0 -- The previously burning mixture could have its flame extinguished by the moving air.
 - Notice the **/lastgust=0** which will indicate the value of the last gust of air.
- At a fuel rate of 8 the mixture would indicate Active_NoFuel.

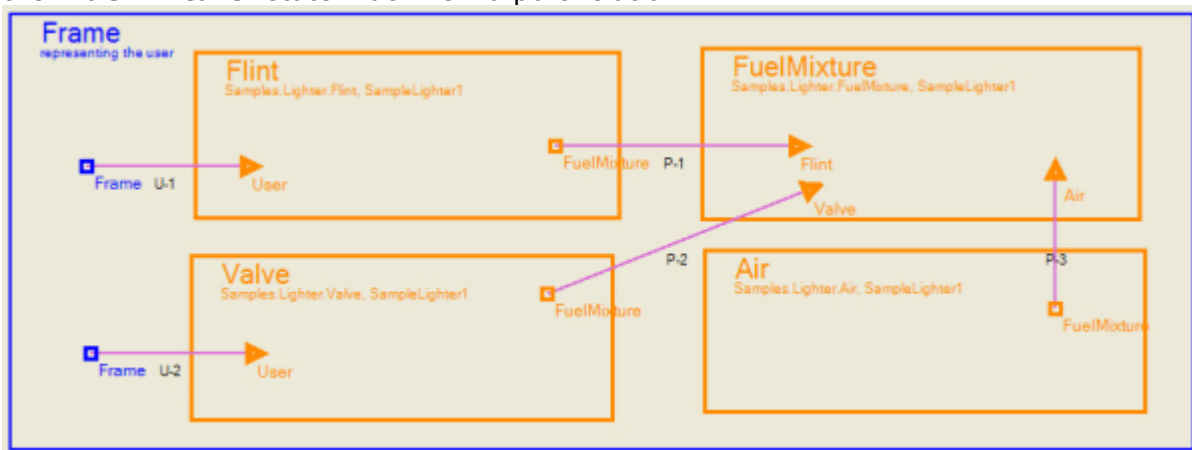
- At a fuel rate of more than the current air rate the mixture would indicate Active_FuelSupplied_Mixed.
- Note that /lastgust would only update if the fuel flow rate is more than the air flow rate at the time of the gust.
- **Air Flow** - Moving_Gust/12 -- The /12 flow rate could go as high as 14. But the interim "gusts" could go up to 18.
- **Valve** - Open/8 -- Decrease fuel flow was clicked twice which took the flow rate to 8m/s.
- **Flint** - NoSparks.

Moving the fuel flow rate to 20 and striking the flint would result in a flame that will keep on going in this sample (there is no countdown of the fuel amount at the moment).

The Frame

The `LighterFrame` is the class in which the four interacting state machines are created and through which the user/gui can interact with them.

The `LighterFrame` diagram below shows this encapsulation of the state machines within the frame. It also shows the port relations between the various state machines and the direction of signal flow. For example - the "**FuelMixture**" port on the "**Flint**" state machine is related to the "**Flint**" port on the "**FuelMixture**" state machine via port relation **P-1**.



The `LighterFrame` is a containing class for interaction between the user/gui and the state machines.

The port relations U-1 and U-2 does not really exist as two ports that are linked via their `QEvents`. The "**User**" port in the "**Valve**" state machine, for example, allows us to model a user's interaction with the state machine - while on the user sending side - we call the `Receive` method on the Valve's User port directly as show in line 3:

```

1     public void PressValve()
2     {
3         _Valve.User.Receive (null, new QEvent (ValveSignals.Press));
4     }

```

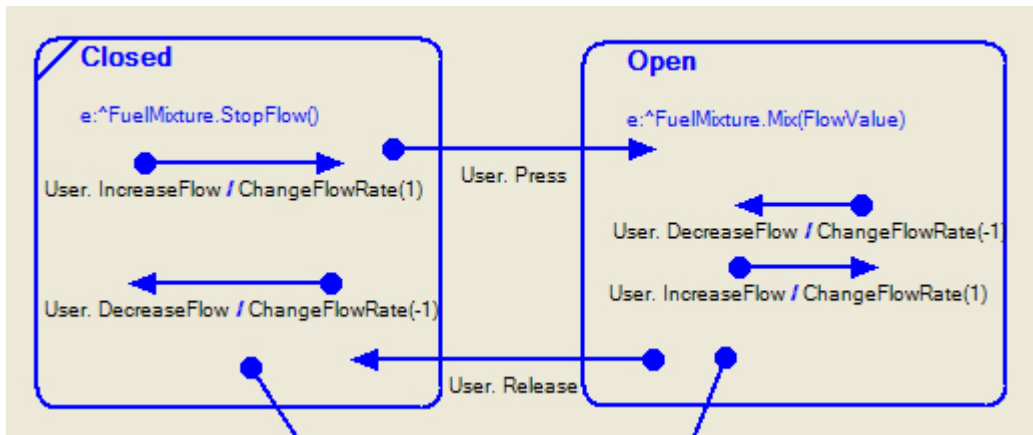
The advantage of this is that the generated code is fully qualified within the Valve state machine:

```

1     protected virtual QState S_Closed (IQEvent ev){
2
3         // ..... other code removed .....
4
5         case QualifiedValveSignals.User_Press: {
6             LogStateEvent (StateLogType.EventTransition, s_Closed, s_Open, "User.Press",
7 "User.Press");
8             TransitionTo (s_Open, s_trans_User_Press_Closed_2_Open);
9             return null;
10            } // User.Press
11
12            // ..... other code removed .....
13        } // S_Closed

```

Notice the `QualifiedValveSignals.User_Press` which means that the state machine is reacting to a more specific event than just **Press** but is taking the signal source **"User"** into account as well.

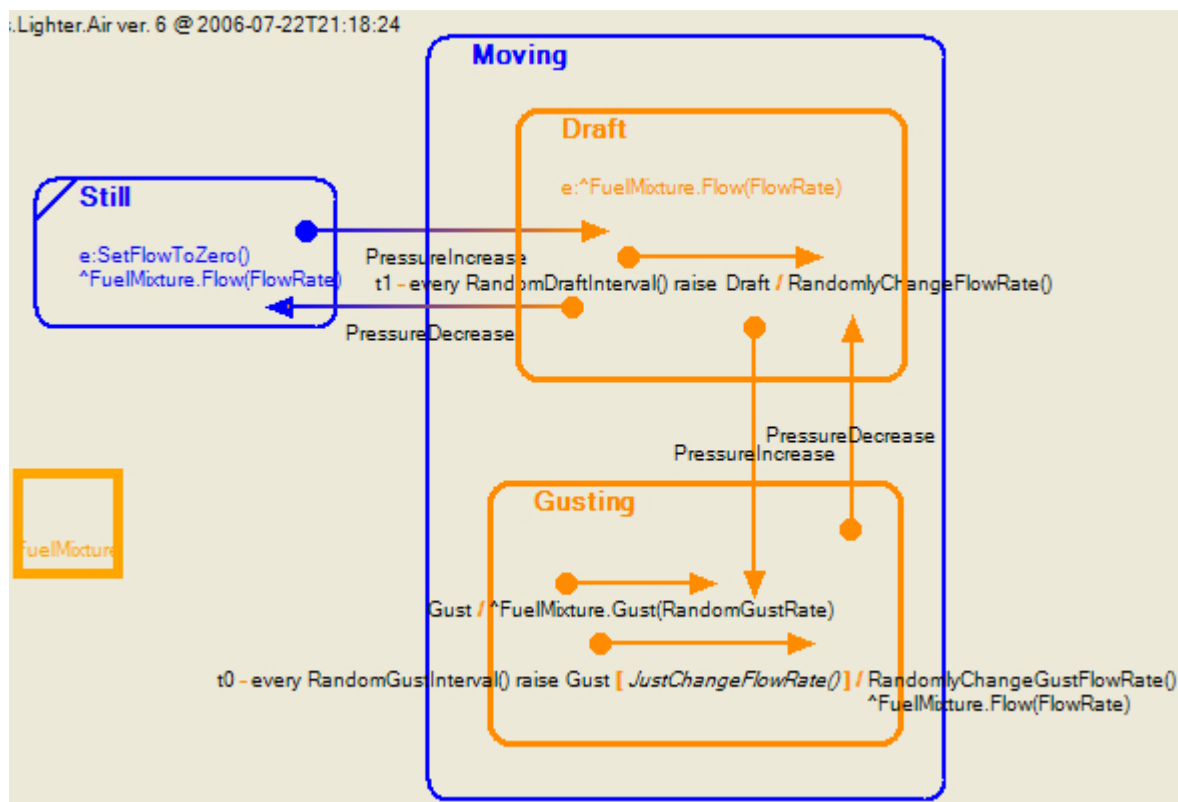


Valve reacts to **Press** event from within the **Closed** state only if the source is the **User** port.

On the other hand...

On the other hand - the Air state machine also needs user input to Increase and Decrease the air speed. It, however, does not have a User port. In order to send it the PressureIncrease signal - the frame simply calls the `SigPressureIncrease` directly.

```
1 public void IncreaseAirFlow()
2 {
3     _Air.SigPressureIncrease (null);
4 }
```



Air state machine will react to PressureIncrease and PressureDecrease from any source.

Though simpler - this mechanism would have to change if we wanted to simulate the user by adding a **"User"** state machine into the system. Even so, I showed both methods just for illustration - it is more important that the four state machines interact via their ports. The user to state machine interaction can be done either way.

Summary

Ports make interactions between related state machines cleaner and simpler, while allowing for future enhancements such as process space isolation of individual state machines.

Using Ports also means that I can swop out one implementation of the Valve state machine with another - as long as the message protocol is the same.

I tried to chose a fairly simple interaction scenario - but as I developed it - I found that the actual interaction scenario's can become quite complex as each individual element adds variability to their behaviour.

Coming Up

1. Saving the State Machine for later rehydration.

History

Third article for stateProto beta release showing how state machines send messages to each other.

References

Miro Samek's Quantum Hierarchical State Machine technology can be found at <http://www.quantum-leaps.com/>

Dr Rainer Hessmer's QF4Net port can be found at <http://www.hessmer.org/dev/qhsm/> and at sourceforge at <http://sourceforge.net/projects/qf4net/> which Dr Hessmer agreed for me to publish.

StateProto can be found at <http://sourceforge.net/projects/stateproto/>

Miro Samek explains QHsm <http://www.quantum-leaps.com/writings/samek0311.pdf>

CSharp Code Formatter at <http://www.manoli.net/csharpformat/>

Wink at <http://www.debugmode.com/wink/>

First Article: [StateProto Beta - State Chart Designer for Qf4Net](#)

Second Article: [StateProto - Executing Multiple StateMachines](#)