Introduction to SystemDotNet

By Carsten Wulff (wulff@iet.ntnu.no)

Mixed-Signal Simulation

For a long time simulation of integrated circuits was divided into two camps; one for analog simulation that used SPICE and one for digital simulation that used VHDL or Verilog. In the resent years a need for behavioral simulation of large mixed-signal systems has spanned a new class of simulators. The major contenders are SystemC, VHDL-AMS and Verilog-AMS. We use SystemC for behavioral simulation of mixed-signal systems in several courses, but since SystemC is based on C++ the learning threshold for a new student is quite high. With much experience with .NET technology and the supremacy of this technology when it comes to seamless integration between windows application and web application we decided to create a new mixed-signal simulator. The simulator is based on the principles of SystemC where the simulator is compiled with the circuit description. It is also based on the article ".NET framework - a solution for the next generation tools for system-level modeling and simulation" [1].

SystemDotNet Simulation Core

The simulation core can roughly be divided into 4 portions a shown in Figure 1; Event queue and simulator control, signals, modules and output writers. The classes in Figure 1 are the essential classes in the simulation core.

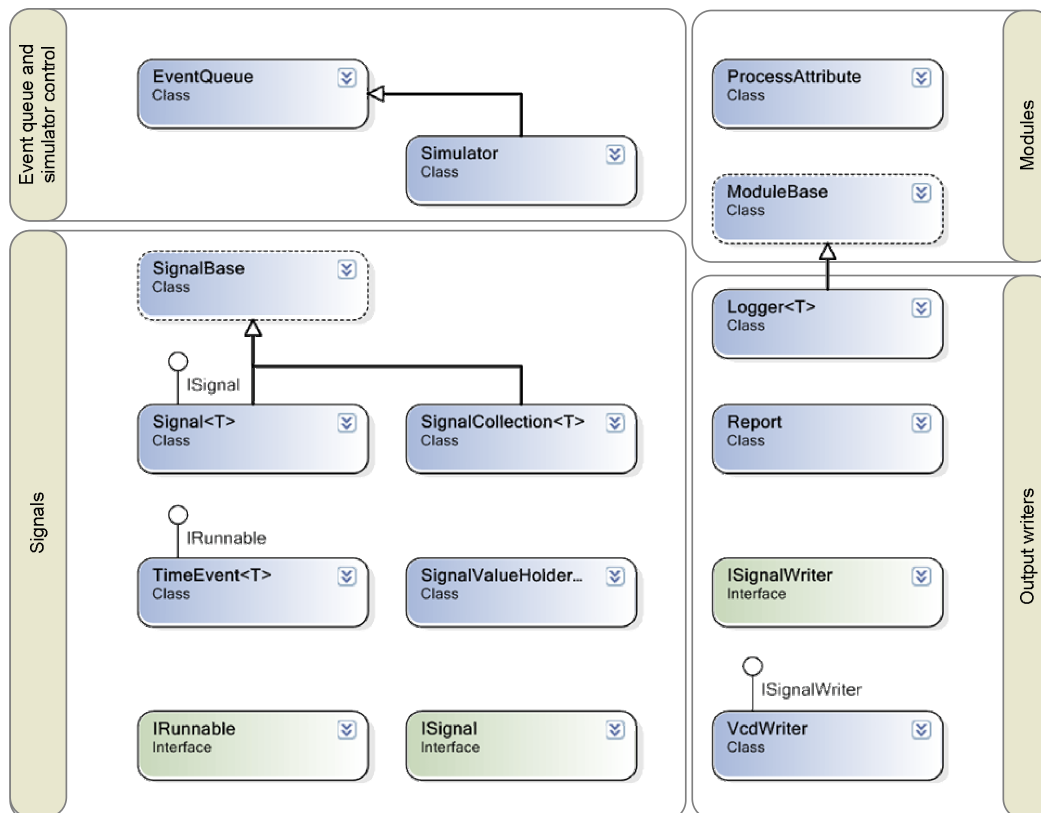

Figure 1 Simulation core overview

Basic principal of simulation core

In an event driven simulator we have three main objects; event queue, signals and stuff that happens when signals change. Think of the event queue as grand old man that has full control over when things should change. Every time we write to a signal we ask the old man to put the change into his queue and execute it at the right time. The old man has a master clock that everything follows, the algorithm used in SystemDotNet can be seen in Figure 2. We start by setting the current time to next time, the first time this happens the next time is 0. We then check if there are any events scheduled for this time step, if there are we remove them from the event queue en execute them. Next we find the next time step if there is one and the process starts all over again. Unlike most event driven simulators the event queue in SystemDotNet is not a sorted list. The event queue is a named collection. Figure 3 shows visualization. A conventional queue contains events that are inserted sorted into the queue, for each step the simulator pops an event and runs it. In SystemDotNet all events that happen at the same time are stored in a list. This list is referenced by the time the event is to occur. We have experimented with a conventional queue but it did not give a speed advantage for the circuits we tried.

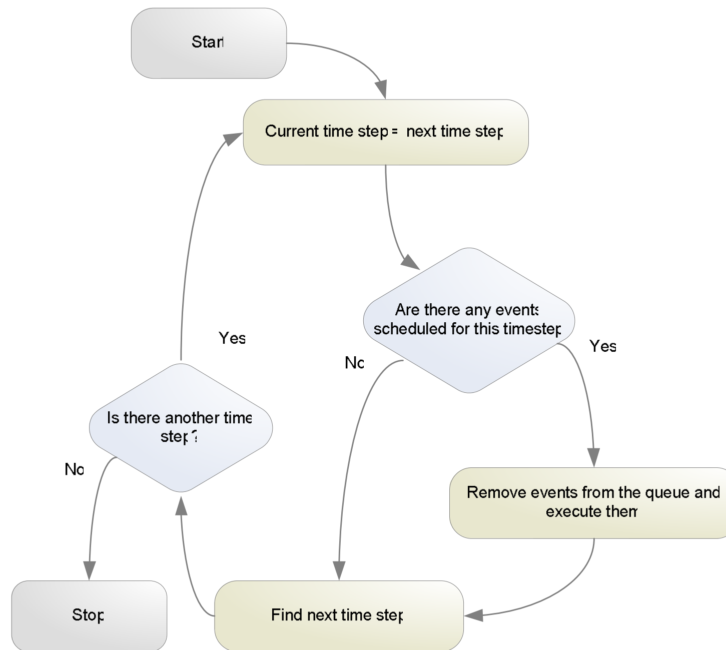

Figure 2 SystemDotNet algorithm

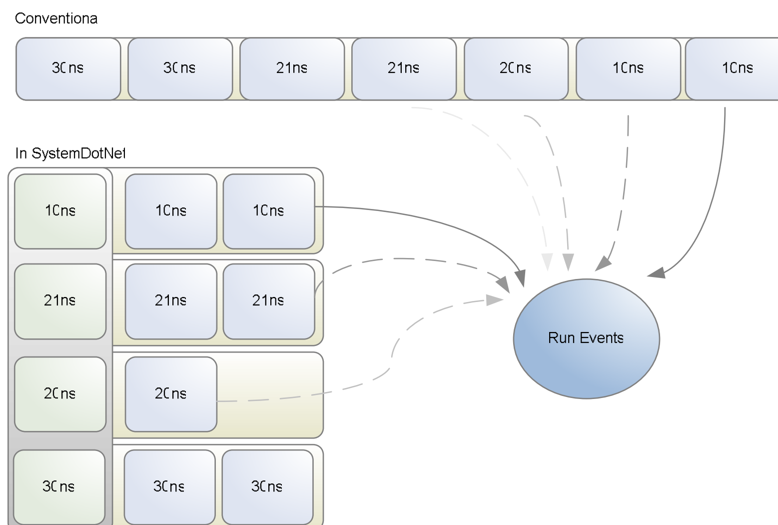

Figure 3 Event queue principle

The objects in the event queue are of type `TimeEvent`, which inherit `IRunnable`. this class holds a `SignalValueHolder` and an object. `SignalValueHolder` is a class used in `Signal` and `SignalCollection` to hold the value of the signal. When the `TimeEvent` is executed the value of `SignalValueHolder` is set to the object. This fires a changed event on the signal where the value has changed. This event must in some way be wired to processes/methods (stuff that happens when a signal change). In SystemDotNet this event

listening can be done in two ways; direct or indirect. To better understand lets look at two modules that use these methods. Modules are classes that define a circuit cell (or block). The rules for Modules are; they must inherit `ModuleBase` or a subclass of `ModuleBase` and all signals available outside the class (often called ports) must be public fields. The code is show in Figure 4 and Figure 5. If we look at Figure 4 we can see the definition of a module. We first have a line with the class name and the inheritance (`ModuleBase`). The definition of the `Clk` signal follows after which we have the constructor. In this constructor we add the `Next()` method to the `Clk.Changed` event. This means that when the `Clk.Changed` event is called, the `Next()` method is run. The method `First()` is overridden from ModuleBase and is run before the simulation starts. This is to make sure that the event queue has some events before it starts, otherwise it will stop immediately. In Figure 5 you can see there is no constructor, but a new definition has been added over the `Next()` method, `[Process("Clk")]`. This is called an attribute and is a C# native construct. Attributes can be used to add Meta information to methods. The ProcessAttribute marks the method as a process and the `"Clk"` parameter tells the simulator to try to connect this method to the change event of the signal `Clk`. The two methods are equivalent, but the indirect listening is the preferred one because it is simpler to write and more visually pleasing. The last block in Figure 1 is the output writers. Two main output formats are implemented in SystemDotNet, VCD (IEEE Std 1364–2001) and CSV (Comma Separated Values). The VCD files can be opened in ModelSim or GTKWave. The CSV files can be parsed using Excel or Matlab.

```
public class Clock : ModuleBase
{
    public Signal<bool> Clk = new Signal<bool>(); //Clk signal

    public Clock()
    {
        Clk.Changed += new EmptyHandler(Next); //Direct listening
    }

    public override void First()
    {
        Clk.Write(true, 200);
    }

    public void Next()
    {
        Clk.Write(!Clk.Read(), 200);
    }
}
```

Figure 4 Direct Listening

```
public class Clock : ModuleBase
{
    public Signal<bool> Clk = new Signal<bool>(); //Clk signal

    public override void First()
    {
        Clk.Write(true, 200);
    }

    [Process("Clk")]      //Indirect listening
    public void Next()
    {
        Clk.Write(!Clk.Read(), 200);
    }
}
```

Figure 5 Indirect Listening

Pipeline Simulation

A behavioral description of a pipeline analog to digital converter was created in SystemDotNet. Pipeline converters are a popular architecture for analog to digital conversion. This class of converters can suffer from mismatch during production that leads to gain errors in multiplying stages of the converter. These gain errors (especially from the first stage) reduce the signal to noise ratio of the converter significantly. Using the web application the user can easily see how the signal to noise ratio changes as we change the gain error. Figure 6 shows the user interface and the result for ideal pipeline converter. Figure 7 shows the result for 10 % gain error in the first stage, we can clearly see the degradation of the signal to noise ratio. The signal is the spike at 5MHz.

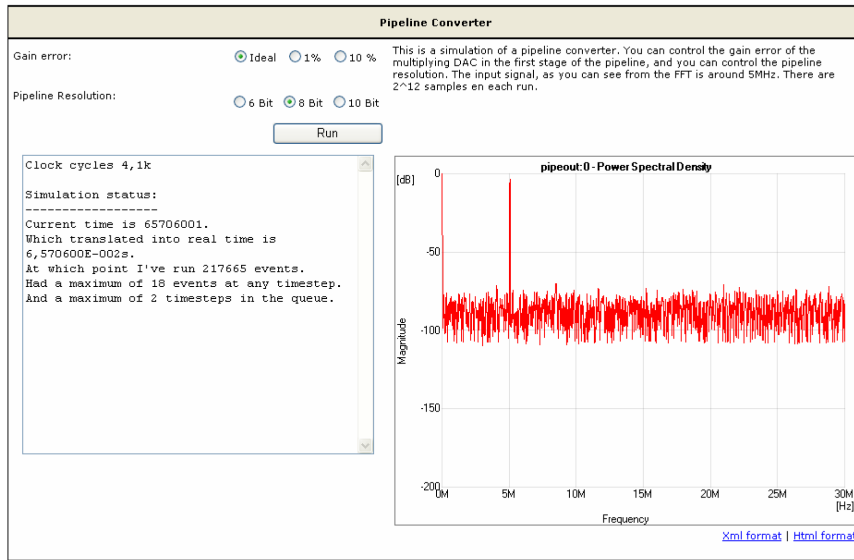

Figure 6 Pipeline ADC converter Ideal

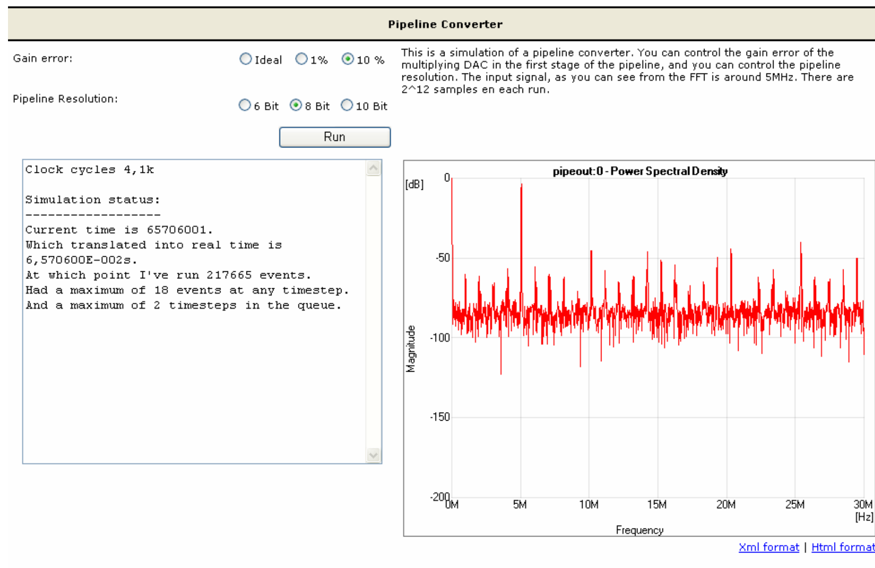

Figure 7 Pipeline ADC converter 10% gain error first stage

References

[1] Lapalme, J.; Aboulhamid, E.M.; Nicolescu, G.; Charest, L.; Boyer, F.R.; David, J.P.; Bois, G. "*.NET framework - a solution for the next generation tools for system-level modeling and simulation*", Design, Automation and Test in

Europe Conference and Exhibition, 2004. Proceedings , Volume: 1 , 16-20 Feb. 2004. Pages:732 - 733 Vol.1