A P P N O T E S SM

ADMS_Signals: Nets of User-defined Type in Standard SystemVerilog for Event-driven Analog Modeling

By: Kenneth Bakalar

Last Modified: February 16, 2010

A common requirement in digital-dominated mixed-signal verification is the need for purely event-driven (“real number”, or RN) models that imitate Spice or AMS blocks at low fidelity but high speed. Resolved record types are commonly used for this modeling style in VHDL-based flows. Unfortunately, SystemVerilog defines only one resolved net type, the *logic* type. A second, non-standard net type, *wreal*, has been borrowed from Verilog-AMS and, with proprietary extensions, added to some implementations of SystemVerilog. *wreal* is a single real value with a small, fixed set of resolution functions. It solves only a subset of the problems commonly encountered in event-driven analog modeling.

In contrast, the ADMS_signals approach is completely general and extensible while still conforming strictly to the IEEE SystemVerilog standard. The stored data type can be any type that is legal in SystemVerilog, including arrays and structs (nested to arbitrary depth) and even class instances (objects). The resolution function is a user-supplied SystemVerilog function. Different networks in the same design hierarchy may be given distinct stored type and resolution function.

Introduction to ADMS_signals

The package ADMS_signals provides two classes: *SignalBase* and *Driver*.

The user first declares a subclass of *SignalBase* customized for a particular stored type and resolution function. For example, here is the declaration of a signal with real values that uses an averaging resolution function:

```
import ADMS_signals::*;
typedef real Stored;
class aReal extends SignalBase #(Stored);
  function new(Stored initialValue = 0.0);
    val = initialValue;
  endfunction
  function Stored resolve(Stored arg[]);
    real val = 0.0;
    foreach(arg[i]) val += arg[i];
      val = val / arg.size();
    return val;
  endfunction
```

Copyright © 2010 Mentor Graphics Corporation

Trademarks that appear in Mentor Graphics product publications that are not owned by Mentor Graphics are trademarks of their respective owners.

```
    function bit equal(Stored oldValue, Stored newValue);
      equal = (oldValue == newValue);
    endfunction;
  endclass
```

The portions in black are common to all signals. The portions in red are customized for this particular stored type and resolution function.

The import statement makes the contents of package *ADMS_signals* visible. The name of the new type is *aReal*. The value of a signal is stored in the variable *val*. A new *aReal* instance gets the initial value 0.0 by default. The body of the resolution function *resolve* computes the average of an array of *aReals*. The function *equal* is used to test whether the value of an *aReal* driver has changed. If it has, *resolve* is called with an array containing all of the values of the drivers of the signal.

The user can declare any number of signals of type *aReal* in his model:

```
aReal larry= new (1234), moe = new (0.0), curly = new ;
```

The numbers in parenthesis are the initial values of the signals. *curly* gets the default initial value 0.0 from the *new* function in the declaration of *aReal*.

The user must separately declare drivers for the signals:

```
Driver #(real) larry1= new(larry), larry2 = new(larry);
Driver #(real) moe1= new(moe), moe2 = new(moe);
```

Drivers of a signal may be driven with calls to the *drive* method from always and initial blocks, tasks and functions. Here are some sequential statements in an initial block that drive a signal, and an instance of a module with *aReal* ports:

```
initial begin
  #10 larry1.drive(12);
  #5  larry2.drive(6);
…
end
subc(larry, moe);
```

The initial block waits for 10 time units to elapse and then drives *larry1*'s signal (*larry*) with 12. A driver is not activated until the first time it is driven. Before that, it does not contribute anything.
If this is the first time *larry* has been driven, after 10 time units the value of *larry* will change from 1234 to 12. Then 5 units later at time 15, the second statement activates driver *larry2* and causes the value of *larry* to change to (12+6) / 2 = 9.

An always block can be made sensitive to changes in the value of a signal, and the value of a signal can be read at any time:

```
always @(larry.changed) begin
  $display("New value for 'larry' detected: %f", larry.value);
  …
end;
```

Signals can be passed by reference through ports. A module may have ports of the new signal type, and may declare additional, independent drivers for the signals:

```
module subc (ref aReal a,b);
  Driver #(real) b1 = new(b);
```

```
    always @(a.changed) begin
      #3 b1.drive(a1.val);
    end;
  endmodule;
```

Every signal has an associated event named *changed*, which is triggered each time the value of the signal changes. The instance of module *subc* above will drive the new value of *larry* on to *moe* 3 time units after *larry* changes. In the example, *moe* changes at times 10+3 and again at time 15+3.

Some additional details:

- Neither a driver nor a signal can be the target of a continuous assignment statement
- A signal cannot be driven directly. Only drivers have the *drive* method.
- It is possible to change the value of a driver or signal directly, but it probably won't have the effect you expect.
    - If you change the value of a signal (e.g., **`larry.val = 345;`**), the change will persist until the next time the resolution function is called. That will happen when some driver of *larry* is driven with a new value.
    - If you change the value of a driver (e.g. **`moe1.driven_value = 123;)`** then the new value will not take effect until the next time the resolution function is called. That will happen when some driver of *moe* is driven with a new value. If that driver is *moe1* itself, then the assignment to *driven_value* will be overwritten and will have no consequential effect.

A realistic case: AvgWreal

The package *AvgReal* is a more realistic example using *ADMS_signals*. The package *wreal_signals* defines a stored type *Realxz*, which is a struct with two fields: a real *value*, and a *strength*. The *strength* is one of the following literals, varying left to right from strongest to weakest: *unknown, supply, strong, pull, weak, highz*. Realxz is used in the declaration of a *SignalBase* subclass *AvgWreal*. The resolution function finds the subset of driver with greatest strength and returns their average. Other drivers are ignored.

If all drivers drive with a single strength — strength *strong*, for example — the value of an *AvgWreal* signal is the average of all its drivers. You can disable a particular driver by driving it with strength *highz* (equivalent to a disconnect or Z) and cause the resolved value to indicate a conflict by driving with *unknown* strength (analogous to the X logic strength).

A subclass of *Driver* called *Sdriver* illustrates how to make this special case easier to use. *Sdriver* includes three methods: *drive*, inherited form the base class, which takes a *Realxz* argument, and *sdrive*, *zdrive* and *xdrive* which take a single *real* argument with the desired value. *sdrive* always drives with strength *strong*; *zdrive* drives with strength *highz* and *xdrive* drives with strength *unknown*.

```
package wreal_signals;
  import adms_signals::*;
  export ADMS_signals::Driver;
  typedef enum {unknown, supply, strong, pull, weak, highz} StrengthType;
  typedef struct {real value; StrengthType strength;} Realxz;

  class AvgWreal extends SignalBase #(Realxz);
    function new(Realxz initialValue = '{0.0, unknown});
      val = initialValue;
    endfunction
    function Realxz resolve(Realxz arg[]);
```

```
      real values[StrengthType] = '{default: 0.0};
      integer cnt[StrengthType] = '{default: 0};
      foreach (arg[i]) begin
        values[arg[i].strength] += arg[i].value;
        cnt[arg[i].strength] += 1;
      end
      foreach (cnt[i])
        if (cnt[i] > 0) return '{values[i]/cnt[i], i};
    endfunction
    function bit equal(Realxz oldValue, Realxz newValue);
      equal = (oldValue == newValue);
    endfunction
  endclass

  class Sdriver extends Driver #(Realxz);
    function new(SignalBase #(Realxz) owner = null);
      super.new(owner);
    endfunction
    task sdrive (real arg);
      super.drive('{arg, strong});
    endtask;
    task zdrive (real arg = 0.0);
      super.drive('{arg, highz});
    endtask
      task xdrive (real arg = 0.0);
      super.drive('{arg, unknown});
    endtask
endclass
endpackage : wreal_signals;;
```

An Example That Uses *AvgWreal*

Here is a test case for *AvgWreal* and the output it generates. The example illustrates a number of different coding styles a user might employ in a real design:

```
module top1;
  import wreal_signals::*;
  Realxz init = '{0.0, weak};
  Realxz  XX ='{0.0, unknown};
  Realxz  ZZ ='{0.0, highz};

  AvgWreal a = new(init);
  Driver #(Realxz) a1 = new(a), a2 = new(a);
  Driver #(Realxz) a3 = new(a), a4 = new(a);
  Sdriver a21=new(a), a22=new(a);
  initial begin
    #10 a1.drive('{10.0, strong});
    #10 a2.drive('{20, strong});
    #10 a21.sdrive(30);
    #10 a22.sdrive(20);
    #10 a21.zdrive();
    #10 a3.drive('{30, supply});
    #10 a4.drive('{60, supply});
    #10 a1.drive(XX);
    #10 a1.drive(ZZ);
  end
  always @a.changed $display ("'a' changed at %t to %f %s",
    $time, a.val.value, a.val.strength.name());
endmodule
```

This is the output that *top1* generates during simulation:

```
# 'a' changed at                   10 to 10.000000 strong
# 'a' changed at                   20 to 15.000000 strong
# 'a' changed at                   30 to 20.000000 strong
# 'a' changed at                   50 to 16.666667 strong
# 'a' changed at                   60 to 30.000000 supply
# 'a' changed at                   70 to 45.000000 supply
# 'a' changed at                   80 to 0.000000 unknown
# 'a' changed at                   90 to 45.000000 supply
```

Definition of the Package *ADMS_signals*

The package *ADMS_signals* defines two classes, each parameterized by the type of the data to be stored in the signals and drivers of the instantiated classes.

The constructor of class *Driver* stores a pointer to the signal it will drive, then pushes itself on to the *drivers* list of that signal. The method *drive* stores a new value for the driver and calls the method *ResolveDrivers* of the owner signal if the new value is not identical to the old. The flag *active* is zero until the first time *drive* is called for a newly minted driver.

SignalBase declares the list *drivers* of all of the drivers of the signal. The method *resolveDrivers* assembles the value fields of all of the drivers into an array and passes it to the function *resolve*, which must be overridden in the subclass. Drivers that have never been driven, as indicated by the state of the *active* bit, are ignored by *resolveDrivers*.

The event *changed* will be triggered if the new resolved value is different from the current stored value. The definition of “different” is contained the function *equal* which, with *resolve*, must be declared in the user-defined subclass that defines a new kind of resolved signal.

```
package ADMS_signals;
  typedef class SignalBase;

   class Driver #(type BaseType = bit);
    SignalBase #(BaseType) ownerSignal;
    BaseType driven_value; // input to resolution
    protected bit active;
    function new(SignalBase #(BaseType) owner = null);
      ownerSignal = owner;
      active = 0;
      owner.drivers.push_back(this);
    endfunction
    task drive(BaseType driving_value);
      if (!active | driven_value != driving_value) begin
        active = 1;
        driven_value = driving_value;
        ownerSignal.resolveDrivers();
      end
    endtask
  endclass : Driver
  virtual class SignalBase #(type BaseType = bit);
    Driver #(BaseType) drivers[$];
    BaseType val;
    event changed;
    pure virtual  function BaseType resolve(BaseType arg[]);
    pure virtual  function bit equal(BaseType oldValue, newValue);
    protected task automatic resolveDrivers();
      BaseType dvals[$];
      BaseType new_value;
      foreach (drivers[i])
        if (drivers[i].active)
          dvals.push_back(drivers[i].driven_value);
      new_value = resolve(dvals);
      if (!equal(new_value,val)) ->changed;
      val = new_value;
    endtask
  endclass : SignalBase
endpackage : ADMS_signals;
```

Copyright © 2010 Mentor Graphics Corporation

Trademarks that appear in Mentor Graphics product publications that are not owned by Mentor Graphics are trademarks of their respective owners.