

ENHANCING SYSTEM VERILOG WITH AOP CONCEPT

Mimic AOP-like functionality using OOP techniques in System Verilog

■ NISHITH SHUKLA

Aspect-oriented programming (AOP) is a well-established technique for adding, modifying or enhancing functionality to a class or environment. It has become *de-facto* standard for developing reusable verification environment or component.

This article explains semantics differences between AOP and object-oriented programming (OOP) and, where relevant, presents how similar functionality can be obtained using System Verilog.

Fundamentals

Despite having many commonalities, 'e' and System Verilog programming languages differ significantly in certain critical areas. For instance, 'e' supports AOP techniques, while System Verilog supports OOP techniques.

AOP is popular because it's very easy to maintain the scattered code. Programmers can easily express cross-cutting concerns in standalone modules called 'Aspect.' OOP doesn't allow you to decompose a problem into all of its concerns. You can only encapsulate some of the concerns. It really doesn't allow you to deal with cross-cutting concerns. Making OOP do something which it doesn't naturally do, needs some extra efforts and thinking.

It's understandable that the ultimate difference between AOP and OOP comes down to organisation of the code. In AOP the code is organised by functionality, while in OOP the code is organised by classes and objects. The

easiest way to understand this concept is to take a packet written in 'e' and then see how it would be written in System Verilog:

Sample 'e' code:

```
<'
type packetType : [DBTRANS];
struct TransBaseT {
    isFirstTrans    : bool;
    kind            : packetType;
    keep isFirstTrans == false;

    processTrans() is {
    }

    print_me() is {
        message(LOW,"-----");
        message(LOW,"Packet Infomration");
        message(LOW,"-----");
    }
}
'>

<'
extend DBTRANS TransBaseT {
    %addr        : uint(bits:32);
    %data        : uint(bits:32);
    %wr_rd       : bool;
    keep addr in [0x0000..0xffff];
    keep wr_rd = select { 50: true,
                        50: false
    }

    print_me() is also {
        message(LOW,"Addr= ",addr, "Data=",
            data,"WR_RD=",wr_rd);
    }

    processTrans() is only {
        message(LOW,"Processing Databus
            Transaction");

        print_me();
    }
}
'>
```

Equivalent System Verilog code:

```
class TransBaseT;
    rand bool isFirstTrans;
    constraint C1 {
        isFirstTrans == false;
    }

    virtual function processTrans();
    endfunction

    virtual function printMe();
    endfunction
endclass

class DbTransT extends TransBaseT;
    rand bit [31:0] addr;
    rand bit [31:0] data[];
    rand bool wr_rd;
    constraint C2 { addr inside
        [32'h0:32'hffff] ;
        data.size() inside
        [32'h0:32'hfff]
    }

    constraint C3 { wr_rd dist { 50 :
true, 50 : false }

    virtual function printMe();
        $display("DbTransT : Addr=%h
Data = %h Wr_Rd=%h",
            addr,data,wr_rd);
    endfunction

    virtual function processTrans();
        $display("Processing DbTransT
Packet")
        printMe();
    endfunction
endclass
```

In classic OOP, you should instantiate the extended class not the base class. If you extend a class, the base class and classes which have already been instantiated will not see changes. Nor would you want them to because the risk is very high that one would carelessly extend a class and corrupt the functionality of all instantiated classes.

Through abstract factory patterns, OOP offers the option to have extended class correctly modify the functionality of the base class and previously instantiated classes.

Polymorphism in System Verilog

Like 'e', System Verilog classes are not automatically constructed and therefore do not use memory before these are explicitly constructed by calling new:

```
TransBaseT obj_TransBaseT = new; //
Handle to instance of Base Class
DbTransT obj_DbReqT = new; //Handle
to instance of Derived Class

obj_TransBaseT.processTrans(); //
Calls the base class method with the
// constraints in base class
obj_DbReqT.processTrans();// Calls
the derived class method with the
// constraints in derived class
```

Even though one is calling the function from base class, it has a handle to derived class. This is known as 'polymorphism.' OOP users use polymorphism to create factory patterns that mimic AOP functionality of 'e'. In System Verilog, a method called using polymorphism must be declared as virtual.

The sample code for calling the derived class method from base class object is given below:

```
obj_TransBaseT = obj_DbReqT;//
Assigning handle of derived class to
base class
obj_TransBaseT.processTrans(); //
Will call derived class method
```

Controlling factory patterns

Let's say you have a new class 'IdbTransT' derived from 'DbTransT' and you want to instantiate 'IdbTransT' throughout design under the following two conditions:

1. You want changes reflected everywhere in design
2. You want 'DbTransT' to be completely replaced functionality-wise with 'IdbTransT,' but you don't want to modify base class 'DbTransT'

In 'e,' you can extend the class without changing the name and that changes every instantiation globally. In System Verilog, you can use factory patterns.

The sample code for factory patterns in System Verilog is:

```
class IdbTransT extends DbTransT;
    rand bit [31:0] channelEnables;
    rand bool      isMySpace;
    rand bit [3:0] spcEnable;
    constraint C2 { addr inside
[32'h000f:32'hf000]}
    constraint C4 { channelEnables
inside [32'h1:32'hff] }
    function printMe();
        $display("DbTransT : Addr= %h Data
= %h Wr_Rd=%h ChEn = %h isMySpc = %h
SpcEn = %h", addr,data,wr_rd,channel
Enables,isMySpace,spcEnables);
    endfunction
    function processTrans();
        $display("Processing IdbTransT
Packet")
        printMe();
    endfunction
endclass

class StimGenT #
(type PACKET = TransBaseT);
task run;
    PACKET pkt;
    for (int i=0;i<5;i++) begin
        pkt = new;
        assert(pkt.randomize());
        pkt.processTrans();
    end
endtask : run
endclass : StimGenT

module top;
    StimGenT #(TransBaseT) obj1 = new;
    stim_gen #(IdbTransT) obj2 = new;

    initial
    begin
        obj1.run;
        obj2.run;
    end
endmodule : top
```

'when' inheritance in OOP

The sample code for 'when' inheritance in 'e' language is:

```
<'
extend DBTRANS TransBaseT {
    when SMALL TransBaseT {
        keep data.size() in [0..10];
    }
}
'>
```

The equivalent code in System Verilog would be:

```
class LargeDbTransT extends DbTransT;
    constraint c5 {
        data.size() inside
        [32'h50:32'h100]
    }
    function processTrans();
        printMe();
    endfunction
endclass
```

The following code illustrates that based on the random value of 'PktSize,' you get appropriate extended class handle which is assigned to the base class:

```
typedef enum {SMALL,MEDIUM,LARGE}
PktSize;

class stim_fact; //Intermediate
Class
    static function TransBaseT
create_stim(PktSize ps) ;
    case (ps)
        SMALL : begin
            DbTransT s = new;
            return s;
        end
        MEDIUM : begin
            IdbTransT s = new;
            return s;
        end
        LARGE : begin
            LargeDbTransT s = new;
            return s;
        end
    endcase
    endfunction : create_stim
endclass : stim_fact
```

This functionality can be achieved with the post_randomize() function also.

'is also' in OOP

When you want to add more functionality to a function, the 'is also' extension in 'e' is quite handy.

In OOP, by inserting virtual, you can decide whether a method can be modified or not.

Not only that, the original method still exists whether or not you have modified it, and any changes you make only apply to that particular sub-class.

The System Verilog code for 'is also' functionality is:

```
class DbTransT extends TransBaseT;
    virtual function processTrans();
        super.processTrans();
        //Local functionality goes here
    endfunction
endclass

class IdbTransT extends DbTransT;
    function processTrans();
        super.processTrans();
        //Local functionality goes here
```

```
    endfunction
endclass

DbTransT e = new;
IdbTransT t1 = new;
TransBaseT p;
TransBaseT p2 = new;

p2.processTrans(); //Original
Version
e.processTrans(); //DbTransT Version
t1.processTrans(); //IDB Version

p = e;

p.processTrans(); //DbTrans Version
```

OOP provides the flexibility of keeping the methods in both the base and all extended classes. By using the power of polymorphism, you can choose and pick which

function to call.

The bottomline

As discussed in the article, System Verilog allows you to do all the things that AOP does—it's simply a matter of understanding it. It provides painless introduction to OOP, allowing you to use as little or much of OOP as you feel comfortable with, so understanding it is not difficult for verification engineers.

Comparing AOP with OOP, it is very obvious that the main difference is in organisation of the code. In AOP you organise your code by functionality, while in OOP you organise your code by classes and objects. ●

The author is a project leader with more than seven years of experience in Chip/ASIC division of elfnchips