Get Hooked

Changing an existing eVC[1] for new project requirements is a grueling task. It's really painful for any eVC developer to witness his creation being torn apart by an end user, but a little prudence from the developer can result into longevity of eVC.
Hooks are very useful in these kinds of situations. The hook mechanism in AOP allows the user to hook into and modify the behavior of the class with very less effort[2]. Hook methods are methods initialized in a base class that can be placed at specific points in the eVC where future modifications are certain.
As an example lets assume that we develop a scoreboard for an eVC. The requirement is to catch specific data from the transaction. The transaction packet is as shown below.

```
<'
type parity_t : [EVEN, ODD];
type scoreboard_kind_t : [PROJECT1, PROJECT2];
type scoreboard_side_t : [PRIMARY, SECONDARY];
//Transaction packet
struct packet_s {
  frame_size : uint;
    keep frame_size == 8;
  frame_bits[frame_size] : list of bit;
  parity : parity_t;
    keep parity == calc_parity(frame_bits).as_a(parity_t);
  calc_parity(frame_bits:list of bit) :bit is{
    return frame_bits.sum(it)[0:0];
  };
};
'>
```

We'll build a simple monitor that generates this packet and send it across to the scoreboard. The monitor uses method port to send the data to the scoreboard.
The monitor part is shown below.

```
<'
//Type definition for method port
method_type hookup_method_t (data_pkt: packet_s);
//Monitor Unit
unit monitor {
  // port to send data out
  send: out method_port of hookup_method_t is instance;
  monitor_pkt()@sys.any is {
    for i from 1 to 10 {
    // generates and sends data to port
    var data := new packet_s;
    gen data;
    send$(data);
    wait cycle;
    };
  wait[1];
  stop_run();
  };
  run() is also {
    start monitor_pkt ();
  };
};
'>
```

The scoreboard is an interesting place in this scenario. There may be many ways to implement the scoreboard such that to satisfy its intent. We can develop the scoreboard with all the required functionality in one shot. Going ahead in this direction, we close all doors for future modifications of the scoreboard. There is also one better way by which

we keep open considerable space for future development or modifications of the scoreboard. In the rest of the section we'll implement scoreboard in the latter way. Here we develop the scoreboard as shown below.

```
<'
//Scoreboard unit
unit scoreboard {
  sb_kind : scoreboard_kind_t;
  sb_side : scoreboard_side_t;
  !processed_pkt1 : list of packet_s;
  !processed_pkt2 : list of packet_s;

  //sequencer pointer
  sb_seq_p : sb_sequencer;
  add_sb_pkts(data_pkt:packet_s, sb_side:scoreboard_side_t,
  sb_kind:scoreboard_kind_t) is empty;
};
'>
```

The scoreboard has two user defined types, sb_side and sb_kind. The sb_side is used to define the type of scoreboard i.e. PRIMARY or SECONDARY and the sb_kind is used to define where the scoreboard is used i.e. PROJECT1 or PROJECT2. It also has a sequencer pointer called 'sb_seq_p' and an empty base class called 'add_sb_pkts'. This is just a skeletal structure of the scoreboard. No functionality has been added yet.
Next step is to extend the scoreboard and add the functionality. The behavior of the base class is implemented in this part of the scoreboard. We have two packet arrays called processed_pkt1 and processed_pkt2. The scoreboard uses these to store the processed packets as per the requirement. Here in this implementation the primary scoreboard uses processed_pkt1 list and the secondary scoreboard uses processed_pkt2 list to store the relevant packet.

```
//Scorebaord with functionality added
<'
extend scoreboard {
  add_sb_pkts(data_pkt:packet_s, sb_side:scoreboard_side_t,
  sb_kind:scoreboard_kind_t) is{
    if(sb_side==PRIMARY){
      processed_pkt1.add(data_pkt);
      out(sb_kind," -> ",sb_side," SB"," Run:",sys.time," :
      ",processed_pkt1);
    }else if(sb_side==SECONDARY){
      processed_pkt2.add(data_pkt);
      out(sb_kind," -> ",sb_side," SB"," Run:",sys.time," :
      ",processed_pkt2);
    };
  };
};
'>
```

Now we have reached the most interesting section of this article called Sequencer. A Sequencer is a set of hook methods that are called in a defined order[2]. The scoreboard sequencer is a separate unit that has the scoreboard instantiated in it. It is here that we insert hook methods. The sequencer controls the scoreboard through these hooks. This is called hooking!!!

The scoreboard sequencer as implemented below, has an in method_port to receive the data from the monitor, scoreboard types and hook methods namely check_packet() and valid_packet(). It also has a sequencer method named as get_pkts().

```
<'
```

```
//Scoreboard sequencer
unit sb_sequencer{
  sb_kind : scoreboard_kind_t;
  sb_side : scoreboard_side_t;
  !do_not_add : bool;
  sb_p : scoreboard is instance;
    keep sb_p.sb_seq_p==me;
    keep sb_p.sb_kind == sb_kind;
  //port to get data
  get_pkts : in method_port of hookup_method_t is instance;
  check_packet(p:packet_s) is empty;
  valid_packet(p:packet_s) is empty;
  get_pkts(p:packet_s) is {
    check_packet(p);
    valid_packet(p);
   };
};
'>
```

The hook methods are defined as empty initially. We can add functionality to the hook methods as per the requirements later. The check_packet() method filters away unwanted packets, and the valid_packet() method adds the useful packets to PRIMARY or SECONDARY scoreboard. This functionality is strictly based on protocol assumptions. Let's assume for PROJECT1, we need only packets whose frame is not in the range 0x0A-0xFF. The packets that fall only in this category are added to the scoreboard. Moreover the packets generated with even parity are added to the PRIMARY scoreboard and the packets generated with odd parity are added to the SECONDARY scoreboard. The implementation is as shown below.

```
//PROJECT1 Scoreboard Sequencer
<'
extend PROJECT1 sb_sequencer{

  check_packet(p:packet_s) is also{
    if(p.frame_bits[7:0] not in [0xA0..0xFF]){
      do_not_add = TRUE;
    }else{
      do_not_add = FALSE;
    };
  };
  valid_packet(p:packet_s) is also{
    if(!do_not_add){
      if(p.parity==EVEN){
        sb_side = PRIMARY;
        sb_p.add_sb_pkts(p,sb_side,sb_kind);
        message(HIGH,"Adding data to the SB->PRIMARY");
      }else{
        sb_side = SECONDARY;
        sb_p.add_sb_pkts(p,sb_side,sb_kind);
       message(HIGH,"Adding data to the SB->SECONDARY");
      };
    }else{
       message(LOW,"Invalid pkt not added to the SB-> ", p);
    };
  };
};
'>
```

Sometime later we start PROJECT2 with a different set of requirements. Here we need only packets whose frame is in range 0x0A-0xFF. The packets that fall only in this category are added to the scoreboard. There is no need to modify the existing scoreboard

to implement the additional requirement. We can extend the scoreboard sequencer with the sb_side_t type and proceed further. In this way we can preserve the existing functionality intact and add additional functionality.

```
//PROJECT2 Scoreboard Sequencer
<'
extend PROJECT2 sb_sequencer{
  check_packet(p:packet_s) is also{
  if(p.frame_bits[7:0] in [0xA0..0xFF]){
    do_not_add = TRUE;
  }else{
    do_not_add = FALSE;
  };
};

valid_packet(p:packet_s) is also{
  if(!do_not_add){
    if(p.parity==EVEN){
      sb_side = PRIMARY;
      sb_p.add_sb_pkts(p,sb_side,sb_kind);
      message(HIGH,"Adding data to the SB->PRIMARY");
    }else{
      sb_side = SECONDARY;
      sb_p.add_sb_pkts(p,sb_side,sb_kind);
      message(HIGH,"Adding data to the SB->SECONDARY");
    };
  }else{
    message(LOW,"Not adding invalid pkt to the SB-> ", p);
  };
 };
};
'>
```

The scoreboard can be configured for PROJECT1 or PROJECT2 at the higher level where the scoreboard sequencer is instantiated.

```
<'
extend sys {
  my_monitor: monitor is instance;
  sb_seq : sb_sequencer is instance;
     keep sb_seq.sb_kind == PROJECT1; //or PROJECT2
     keep bind(my_monitor.send,sb_seq.get_pkts);
};
'>
```

Hence by using a couple of hooks we have successfully implemented a scoreboard for two different types of requirement without doing any modification in the main scoreboard unit.

References:

[1] e – Reusable Methodology

[2] Aspect-Oriented Programming with the e Verification Language, By David Robinson

M.P. Rashid
clickonrashy@gmail.com