2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Modelling of UVC Monitor Class as a Finite State Machine for a Packet-Based Interface

Djordje Velickovic, Verification Engineer, Veriest Solutions, Nis, Serbia (djordjev@veriests.com)
Milos Mitic, Verification Engineer, Veriest Solutions, Belgrade, Serbia (milosmi@veriests.com)

***Abstract*— This paper intends to present the advantages of modelling standard UVC monitor class for packet-based interface as a finite state machine. It will do so by explaining in depth, the concept of UVC monitor class using the FSM model on an open-source Ethernet core IP TX interface. It compares it to a traditionally coded monitor class with emphasis on the advantages of the proposed solution in debugging such interface and coding of protocol violation checkers. It also shows that all these advantages are achieved without adding degradation in CPU usage percentage and simulation time.**

Keywords—UVM, Monitor, Packet-based interface, Finite State Machine, Ethernet, Debug

I. Introduction

The development of good quality verification components and IPs are usually taking substantial resources during the verification process on a wide range of projects. Although the introduction of verification methodologies over the years standardized the process for VIP development, it is still, more than often, crucial to allocate a sizeable amount of time (or resources) for this purpose in project planning. A cornerstone in a development process of a reliable and capable Verification IP will be, in most cases, proper monitor modelling.

To create a monitor which covers the functionality of an interface which is being verified in the best way possible and to compliment that with objective needs for a project in progress, can be quite a serious task. Modelling of a UVC monitor class can be particularly challenging for packet-based interfaces. This comes from two major problems in the monitoring of the packet-based interface. A packet of such an interface will usually be comprised of multiple frames and each frame will contain multiple chunks of data. This can result in very long packets, where data carries different information based on its place in the packet. In parallel to monitoring the data and placing it in the correct frame of the packet, a verification engineer who is developing the monitor needs to pay extra attention to interface signals behavior during the valid packet transfer and out of it. The effort needed for the development of a proper monitor of packet-based interface scales immensely with packet complexity and length.

The proposed solution for a Modelling of a UVC Monitor Class as a Finite State Machine for a Packet-Based interface aims to standardize the coding process of such monitor class and therefore shortens the development time needed for its completion (especially for engineers who are lacking experience working with packet-based interfaces). Additionally, this approach should also shorten the time used to debug potential issues found in a module being verified. It gives an engineer an extra tool to work with as an FSM model of the packet can help in pinpointing the issue in the erroneous packet.

II. Concept of Modelling of UVC Monitor Class as a Finite State Machine for a Packet-Based Interface

The concept relies on the fact that packets for interfaces in question, by definition, have multiple frames (header, payload, preamble...). The idea behind this implementation is to introduce a variable into the monitor of an enumerated type defined to represent each of the packet frames definition plus the IDLE state which is set as default and means that there is no valid packet transfer on the interface. When the monitor detects that the packet was started it will change the value of the variable to match the start of the packet frame. In the same way, each frame of the packet will be modelled with an assigned value to a variable that matches. State-jumping will completely mimic standard finite state machine implementation with case structure (see “Figure 1- Case Structure of FSM Monitor and Packet Frames”) and added functionality for each defined state. As well, in each state, a condition needs to be met, if the flow can progress to the next frame in the packet. When a packet reaches the end of the frame, the FSM variable will be set to the IDLE state and the item which was populated through all frames of the packet with appropriate data collected from the physical interface will be ready for use. Once the variable is in the IDLE state it will wait for the start frame conditions of another frame to start the cycle again.

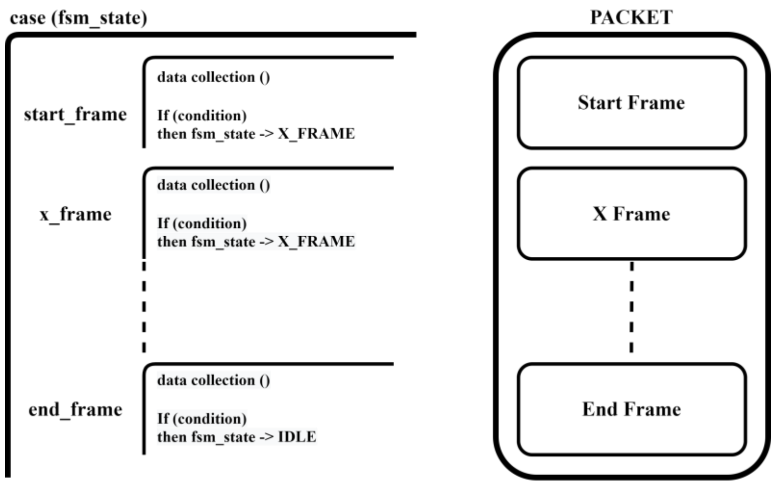

Figure 1 – Case Structure of FSM Monitor and Packet Frames

The greatest benefit of implementing the monitor this way comes with the simplicity of coding the protocol checkers. Once the packet state is modelled correctly, part of the monitor dedicated to protocol violation checks just needs to track the activities on each of the respective signals of the interface in parallel. The idea is to create independent processes which will be tracking changes of signal values on the interface. When a change in signal value is detected it is compared against the modelled value of the packet frame and then checked if the detected transition is allowed by interface definition.

Also, debugging is an important benefit of implementing the UVC monitor utilizing the FSM concept. When dealing with complex and long packets usually is not easy to understand where the potential problem actually occurred and an engineer who is debugging the interface behavior will need deep knowledge of interface functionality to be proficient in this process. The option of having the FSM model of the packet as a debug aid will allow engineers with limited knowledge of interface functionality to find the issue in a faulty packet.

III. IMPLEMENTATION

Following implementation of UVC monitor class as an FSM is done using UVM[1]. This means that the monitor in question is functioning as a part of the agent developed according to the UVM and is intended to be used in a verification environment developed using the same methodology.

A. *Definition of interface used for showcasing the implementation*

For the purpose of this paper, the implementation of a monitor using FSM for a packet-based interface will be showcased on an interface that presents the TX side of an open-source Ethernet IP core[2]. The definition of interface signals is given in Table 1.

Table 1 – Open-source Ethernet IP Core TX Interface

Name	***Width***	***Direction***	***Description***
MRxDV	*1 bit*	*input*	*Valid indication*
MRxD	*4 bits*	*input*	*Data bus*
r_Pro	*1 bit*	*input*	*Promiscuous mode control signal - all frames are received regardless of their destination address*

Name	*Width*	*Direction*	*Description*
r_Bro	*1 bit*	*input*	*Broadcast mode control signal - all frames containing broadcast addresses are rejected*
MAC	*48 bits*	*input*	*MAC address of the used Ethernet MAC IP Core*
MRxClk	*1 bit*	*input*	*Clock*
Reset;	*1 bit*	*input*	*Reset*

When MRxDV is asserted, data on an MRxD bus is valid. Control signals r_Bro, r_Pro, and MAC are set at the start of each packet. It is assumed that r_Bro, r_Pro, and MAC should not change value during valid packet transfer. Packets sent through an MRxD bus should follow the pattern shown in Figure 2.

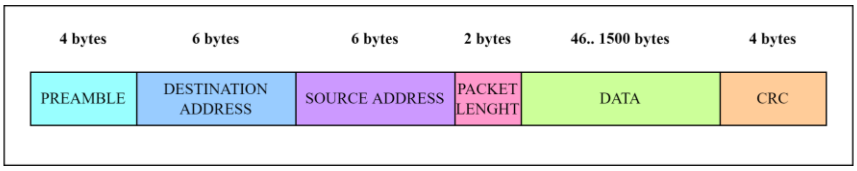

Figure 2 – Packet's Frames

B. *Coding the FSM model inside the monitor class*

The coding of the FSM model for the monitor class is divided into two tasks, detect_packet and collect_packet. Task detect_packet has the functionality to monitor the activity of the MRxDV signal and detect when valid data arrived on the data bus and started packet transmission. When valid data is detected, under this task, the FSM variable value is changed from IDLE to PREAMBLE, a new item is created and static item values (r_Pro, r_Bro, and MAC) are populated with corresponding signal values from the interface. Then, the collect_packet task is called. When there is no valid data on a bus, the FSM variable is in the IDLE state. Implementation of the detect_packet task is shown below:

```
virtual task detect_packet();

    fsm = IDLE;

    forever begin
        if (vif.slv_cb.MRxDV == 1'b1) begin
            fsm = PREAMBLE;
            item = uvm_eth_vip_item::type_id::create("item");
            item.MAC= vif.slv_cb.MAC;
            item.r_Pro = vif.slv_cb.r_Pro;
            item.r_Bro = vif.slv_cb.r_Bro;
            collect_packet();
        end
    else
        @vif.slv_cb;
    end
endtask
```

Inside of collect_packet task state machine logic is implemented which follows packet definition by frames. With each subsequent cycle with valid data on a bus, data will be sampled and stored in the corresponding item field (preamble, dest_addr, source_addr…). The state jumping mechanism is regulated by counters for each modelled state. These need to be initialized to values taken from the packet definition. For example, the preamble consists of 4 bytes. Since the data bus is 4 bits wide, the preamble will be driven in 8 valid cycles and therefore preamble_cnt is initialized to value 8. Following the same principle, all counters are initialized (dest_addr_cnt = 12, src_addr_cnt = 12, length_cnt = 4…). The counter is decremented with each valid data cycle and when reaches value 0, the FSM variable is set to the next frame value. Implementation of the collect_packet task is shown below:

```
virtual task collect_frame ();

    while(fsm != IDLE) begin
        if (vif.slv_cb.MRxDV == 1'b1) begin
            case (fsm)
                PREAMBLE:
                    begin
                        item.preamble[(preamble_cnt-1)*4 +: 4] = vif.slv_cb.MRxD;
                        preamble_cnt--;
                        if (preamble_cnt == 0)begin
                            fsm =  DEST_ADDR;
                            preamble_cnt = 8;
                        end
                    end
                DEST_ADDR:
                    begin
                        item.dest_addr[(dest_add_cnt-1)*4 +: 4] = vif.slv_cb.MRxD;
                        dest_add_cnt--;
                        if (dest_add_cnt == 0)begin
                            fsm =  SRC_ADDR;
                            dest_add_cnt = 12;
                        end
                    end
                SRC_ADDR:
                    begin
                        item.source_addr[(src_add_cnt-1)*4 +: 4] = vif.slv_cb.MRxD;
                        src_add_cnt--;
                        if (src_add_cnt == 0)begin
                            fsm =  DATA_LENGHT;
                            src_add_cnt = 12;
                        end
                    end
                DATA_LENGHT:
                    begin
                        item.p_length[(lnght_cnt-1)*4 +: 4] = vif.slv_cb.MRxD;
                        lnght_cnt--;
                        if (lnght_cnt == 0)begin
                            fsm =  DATA;
                            lnght_cnt = 4;
                            data_cnt = item.p_length*2;
                        end
                    end
                DATA:
                    begin
                        item.data.push_back(vif.slv_cb.MRxD);
                        data_cnt--;
                        if (data_cnt == 0)begin
                            fsm =  CRC;
                            data_cnt = 0;
                        end
                    end
                CRC:
                    begin
                        item.CRC[(crc_cnt-1)*4 +: 4] = vif.slv_cb.MRxD;
                        crc_cnt--;
                        if (crc_cnt == 0)begin
                            fsm =  IDLE;
                            crc_cnt = 8;
                            a_port.write(item);
                        end
                    end
            endcase
        end
        @vif.slv_cb;
    end
endtask
```

Information about the counter for data state (data_cnt) is not predefined by packet definition. Instead, it's being fetched from packet length information during packet transfer.

When the FSM variable reaches the CRC state, which is the last state by packet definition, CRC data is collected, and the FSM variable is reset to the IDLE state. When that occurs, the item is completely populated with necessary data and, since flow reached the end of the packet, is ready to be sent via the monitor analysis port.

C. *Implementation of protocol checkers*

As mentioned before, the existence of modelled FSM values in real-time makes the coding of protocol checkers simple and standardized. The concept relies on the detection of any change in the value of a monitored signal. Implementation of the check_signal task is shown below:

```
virtual task check_signals();
    fork
        begin
            forever begin
                @(posedge vif.r_Pro, negedge vif.r_Pro);
                r_Pro_signal_chk: assert(fsm == IDLE)else
                    `uvm_error("check_signals", $sformatf("Illegal r_Pro change detected during packet state : %s",fsm.name));
            end
        end
        begin
            forever begin
                @(posedge vif.r_Bro, negedge vif.r_Bro);
                r_Bro_signal_chk: assert(fsm == IDLE)else
                    `uvm_error("check_signals", $sformatf("Illegal r_Bro change detected during packet state : %s",fsm.name));
            end
        end
        begin
            forever begin
                @(vif.MAC);
                MAC_signal_chk: assert(fsm == IDLE)else
                    `uvm_error("check_signals", $sformatf("Illegal MAC change detected during packet state : %s",fsm.name()));
            end
        end
    join_none
endtask
```

When change is detected, the current FSM modelled value is checked against packet states in which change of value is legal. In the case of the interface defined above, it was assumed that signals r_Bro, r_Pro, and MAC should not change if a valid data transaction is ongoing. Therefore, any detected change on these signals when modelled FSM variable is not in IDLE state will be considered as a protocol violation and flagged with UVM_ERROR.

IV. BENEFITS AND RESULTS

A. *Gains in debug*

Presented concept of modelling a UVC monitor for packet-based interface as an FSM gives significant benefits in debug process.

When dealing with erroneous behavior on a packet-based interface, an engineer usually needs to analyze fairly complex packets, sometimes, with more than a thousand valid data cycles. This can be quite a challenging task and it might take substantial time until the issue is located. This is particularly the case when debugging is conducted on the verification component used in the verification of a module that does any kind of packet generation or data manipulation within the existing packet. In order to find an issue in a faulty packet, an engineer will have to dive deep into the interface definition and understand signal behavior in detail. Even after that, tracking data visually on waveforms is almost impossible in some cases of extra-long packets.

Option to push modelled FSM value to waveform alongside interface signals and track how packet progresses, presents a powerful tool in the visual analysis of a packet. An example of this is given in Figure 3 where a packet, of previously defined ethernet core interface, is showcased alongside modelled FSM variable.

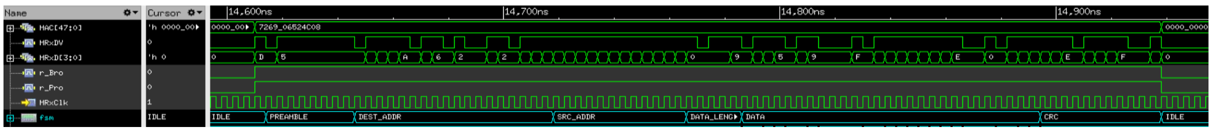

Figure 3 – Interface and FSM Variable

Furthermore, each packet state counter implemented in the monitor solution can be used for tracking data on the waveform. The existence of these counters in the monitor makes various tools' built-in cycle counter feature almost completely redundant, as they are much less cumbersome to use (no need for setting cursors and comparing the cycle numbers with valid signals).

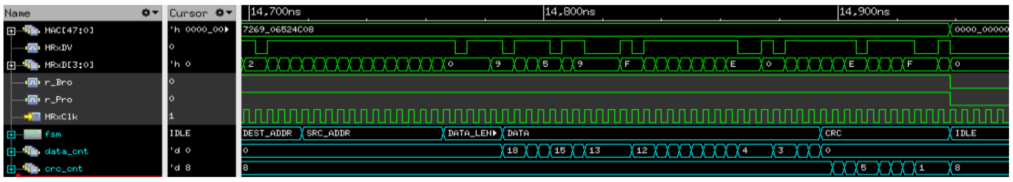

Figure 4 – Interface and Data Counters

B. *Performance metrics*

One of the main concerns during the initial development of a UVC monitor which uses the FSM model concept was how the addition of the FSM model will affect the performance of the monitor. With the aim to investigate this topic, a dedicated testing environment was created.

The testing environment encapsulated two agents based on the TX side of the Ethernet IP core interface presented earlier, developed according to the UVM. One agent was set as Master and the other as Slave and they were connected via simple interconnect module. The slave agent was configured as PASSIVE since only the TX side of the interface was implemented. A simple sequence was created to send the desired number of packets from the Master to the Slave agent.

To secure proper control for performance measurement, another monitor was coded. The second monitor was developed in a much more traditional way for this kind of interface: When valid packet transmission is detected, monitor flow starts looping through cycles following the packet definition and collecting data. Part of the implementation of this monitor is given below:

```
//preamble
for (int i = 7; i>=0; i--)begin
    if (vif.slv_cb.MRxDV == 1'b1)begin
        item.preamble[(i)*4 +: 4] = vif.slv_cb.MRxD;
        @vif.slv_cb;
    end
    else begin
        i++;
        @vif.slv_cb;
    end
end
//destination_address
for (int i = 11; i>=0; i--)begin
    if (vif.slv_cb.MRxDV == 1'b1)begin
        item.dest_addr[(i)*4 +: 4] = vif.slv_cb.MRxD;
        @vif.slv_cb;
          .
          .
          .
```

For the testing scenario, the data length of each item, as well as the delays between packets, were kept at fixed values with the goal to minimize the effects of randomization on experiment results. During testing, two simulation parameters are measured, CPU usage percentage and simulation time. Parameters measurement results were compared when agents used the FSM monitor and the regular one. All results presented are rounded average values deduced from regression of simulation runs. CPU usage and simulation time results are generated using the -perfstat feature of the Xcelium [3] simulator.

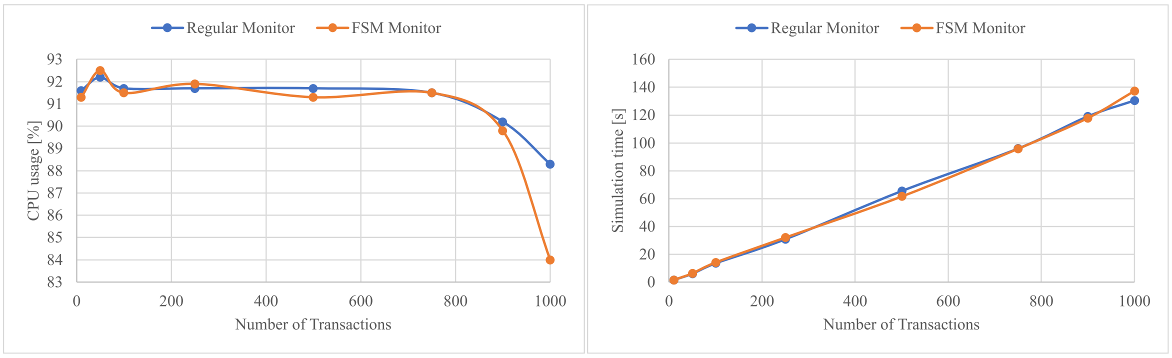

Figure 5 – FSM vs. Regular monitor performance metrics

On the graphs above (Figure 5) it is plotted how CPU usage percentage (left) and simulation time (right) are behaving with the number of transactions ran during simulation for both monitors. It can be noticed that for this experiment, results in terms of simulation time are fairly close between the two monitors. With a higher number of transactions ran in a single test, a small spike in simulation time can be spotted for simulation with the FSM monitor but an increase in time does not exceed 10% on thousand transactions. Contrary to that, collected metrics on CPU usage suggest that the FSM monitor used less percentage of available computing power than the regular monitor with the highest difference of 4% on 1000 transaction runs. Gathered results are indicating that modeling the monitor class of packet-based interface as an FSM would not deteriorate the performance of the simulation in which such monitor is used, and it even has the potential to improve them.

During the experiment, memory usage was also measured. Results were identical for both types of monitors.

V. Conclusion

As mentioned, monitoring of packet-based interfaces can be a challenging task, especially for interfaces with complex packet definitions. The proposed solution of modelling monitor class as an FSM for packet-based interfaces tends to standardize the coding of a such monitor. It even offers a certain amount of code reusability between interfaces with similar packet definitions. An FSM structure would remain the same across different monitors and can be copied and later modified to fit the new packet definition of an interface when reused. A similar type of standardization would come with the implementation of protocol violation checkers. As shown in the paper, coding these checkers can be quite an easy task, once an FSM variable is modelled correctly.

The benefits of the presented solution during a debug of a monitored interface are substantial. In modern ASIC functional verification, debugging is the process that statistically takes more resources than coding. Therefore, during verification component development, verification engineers should code their VIP with as many debug capabilities as possible. Modelled FSM variable in a monitor which shows the current state of the packet will speed up the process of debug of complex packets immensely.

All the mentioned benefits of the presented monitor concept are achieved without noticeable degradation in performances when compared to a traditionally coded monitor per conducted experiments.

Modelling of a UVC monitor Class as an FSM for a packet-based interface certainly gives an interesting perspective on the problem of monitoring for packet-based interfaces and can probably become even more efficient and applicable with future upgrades and extensions.

REFERENCES

[1] Universal Verification Methodology (UVM) 1.2 Class Reference, 2011 - 2014 Accellera Systems Initiative (Accellera)

[2] Igor Mohor - IgorM@opencores.org "Ethernet IP Core Design Document", Rev. 0.4 ,October 29, 2002

[3] Xcelium XRUN User Guide - Product Version 19.09, September 2019