2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Modelling Finite-State Machines in the Verification Environment using Software Design Patterns

Darko M. Tomušilović, Veriest Vtool, Belgrade, Serbia (darkot@thevtool.com)
Mihajlo Z. Minović, Veriest Vtool, Belgrade, Serbia (mihajlom@thevtool.com)

***Abstract*—Software design pattern is a software development technique representing a solution to a typical problem repeatedly found over a development cycle. This paper shows how State, Singleton, Mediator and Template method design patterns can be utilized to efficiently model Finite-State Machines in the Verification environment. The proposed implementation improves code readability and reusability on both active generation side and passive checking and coverage collection side. It also facilitates the maintainability, by localizing the changes upon the addition of new FSM states.**

Keywords—FSM, UVM, Design patterns

I. INTRODUCTION

A finite-state machine (FSM) or simply a state machine, is an abstract machine that consists of a list of states it can be in, including the initial state, and the conditions for the transition between the states, according to the input stimulus. Each state is characterized by a set of corresponding outputs.

Accomplishing the task of FSM verification, as a part of the electronic system functional verification process, assumes leading the state machine between the possible states using the proper input stimulus and checking that the output signals are properly driven. Finally, it is necessary to collect coverage on all states and state transitions, but as well as on high level scenarios, for which the coverage is not built-in. To achieve all that, one common approach verification engineers might opt for is develop a DUT reference model within the verification environment. However, as the project cycle is shortened whereas the design complexity increases, verification engineers face several typical challenges. To reduce the effort, it is critical to have reusability in mind, as well as to provide an easily modifiable solution, with a view to facilitate addition or removal of certain features.

The UVM methodology [1] is introduced with the goal to help overcoming some of these challenges. It defines a framework comprising a set of components, which have a consistent role in attaining the requirements posed by Metric-Driven Verification principles. Main considered concepts are reusable constrained random stimulus generation, automated self-checking environment and functional coverage model. By using the most widespread implementation of the methodology, in SystemVerilog language, the developer is provided with the language constructs dedicated to the functional verification, such as assertions and covergroups. But since SystemVerilog is in essence an object oriented language, it also lends itself for application of design patterns as a well-established technique from the software development field, as it will be shown in this paper.

II. FSM IMPLEMENTATION

Two possible FSM implementations will be shown using the example state machine presented in Figure 1. After the power-on reset is released, the device goes through the initialization stage, whose completion is marked by a dedicated counter expiration. After the initialization is performed, the device operates in the run mode. Using a software switch, implemented as a register field, the device can be lead into the clock gating state, in which the configuration clock is not provided to certain functional blocks. The device exits the clock gating state upon an issued interrupt request. Using a hardware switch, the device can be put into switch-off sleep mode. After the switch is on, the device needs to go through the initialization state before being operational again.

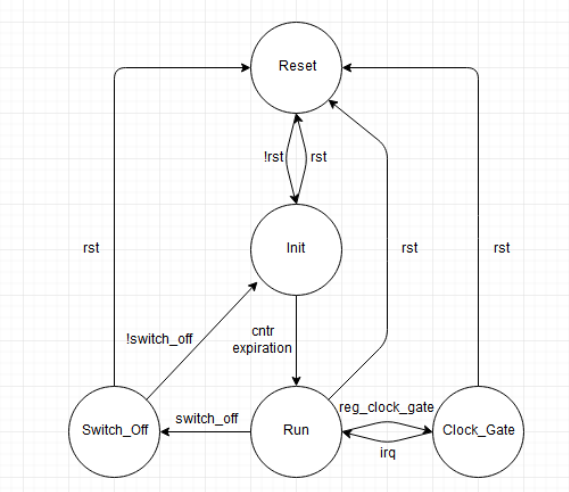

Figure 1. Modelled device state machine

A. *Tightly coupled FSM implementation*

When modelling the FSM, one typical implementation is the most obvious one: enumerating the states and using a simple variable to represent the current state. All the logic that performs actions corresponding to a certain state and updates the variable which denotes the current state is typically placed within a huge "if" or "switch/case" statement, conditioned by the current state. The principle is illustrated by a code excerpt shown in Figure 2.

This approach suffers from a key drawback that several independent tasks are coupled together. The same piece of code represents the FSM static structure, its general behavior reliant upon the current state, as well as its dynamic part. Even in the lower-level language, such as VHDL, these tasks are separated into multiple processes. Without proper role-splitting, the code quality is affected in terms that the same logic is not straightforward to be incorporated into a new project. Also, as the code complexity grows, the solution is prone to the code duplication.

What can be done to achieve the decoupling between the system structure and its behavior is use State design pattern.

```
class FSMExample;
  local fsm_t currentState;

  function new(fsm_t initState);  currentState = initState;  endfunction

  function void doAction(Input inputs);
    case (currentState)
      fsm_reset: begin
        doActionForState_reset(inputs);
        currentState = calculateNewState(currentState, inputs); end
      fsm_init: begin
        doActionForState_init(inputs);
        currentState = calculateNewState(currentState, inputs); end
      …
    endcase
  endfunction

  protected function void doActionForState_reset(Input inputs); endfunction
  protected function void doActionForState_init(Input inputs);  endfunction
   …
  local function fsm_t calculateNewState(fsm_t currentState, Input inputs); endfunction
endclass
```

Figure 2. Tightly coupled FSM implementation

B. Loosely coupled FSM implemenation based on the State design pattern

Design patterns [2] define a set of classes and relations between them, which interact in order to resolve a common problem faced during software development. They target system creation, structure and its behavior.

In Object-oriented programming terminology, the State pattern[2] is a behavioral software design pattern, used to model finite-state machines, decouple them from the rest of the system and provide simple interface to them. The overall block diagram is shown in Figure 3, while the UML class diagram displayed in Figure 4 demonstrates a more detailed view of the necessary classes.

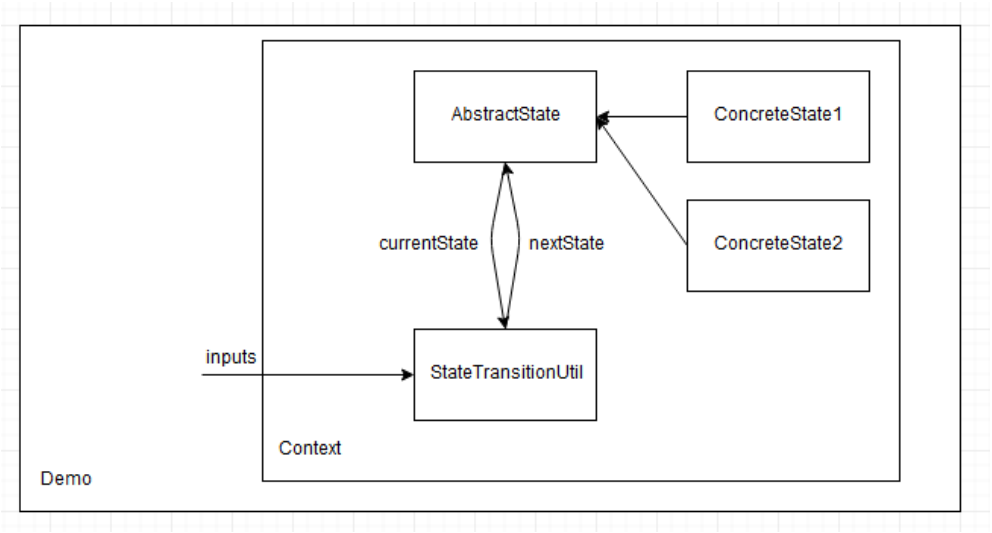

Figure 3. State design pattern - Overall block diagram

The state machine model is encapsulated within a context class, through which the communication with the rest of the verification environment is supposed to be performed. In the environment developed according to the UVM methodology, the context class is to be instantiated within the UVM monitor class, who provides it with values of observed state machine input signals and represents its client, using the method *doAction()*.

Every state of a state machine contains a set of common features and actions. Abstract base class State defines these common operations. As some of them, such us updating the expected values of the output signals and calculation of the next state, appear in a well-defined order, they are appropriate to be modelled using Template method design pattern[2]. The *doAction* method of the abstract State class is a Template method - a concrete method that defines the sequence of operations, some of which can be abstract - left to be implemented in the concrete derived classes. The abstract State class is shown in Figure 6. It is referenced in the context class.

Different states of the state machine are modelled as classes derived from the State base class. They provide implementation to the abstract operations defined in the abstract class, defining a state-specific behavior. As one object is sufficient to represent each state, the derived states are implemented as singletons - classes that can have only one instance object[2]. The implementation of the concrete classes is presented in Figure 7.

Finally, it is necessary to create a logic for changing the current state, updating the reference within the context class. The place where the state transitions should be defined are out of scope of the State pattern. Defining it in the context class is not the optimal solution because the intent is to decouple it from the system behavior. Defining it in the State base class is not a good option either, because in that case the base class would have to be aware of all its derived classes. By defining it in each State derived classes, the context class is decoupled from the state transition logic and the new State subclasses can be easily added. The disadvantage is that each State derived class has coupling to its siblings, which introduces undesired dependencies between subclasses.

Taking all of the above into account, the authors suggest the state transition logic be defined in a separate utility class. This will not remove the need for the awareness of all possible derived states, but will localize the transition logic and decouple the rest of the system from it. The solution closely resembles Mediator pattern[2],

which defines a centralized mediator object through which the classes communicate, rather than interact among themselves. The usage of a mediator object loosens the coupling between the classes and therefore improves the code maintainability. The mediator utility class responsible for determining the state transitions, based on the current state and the input stimulus, is shown in Figure 8.

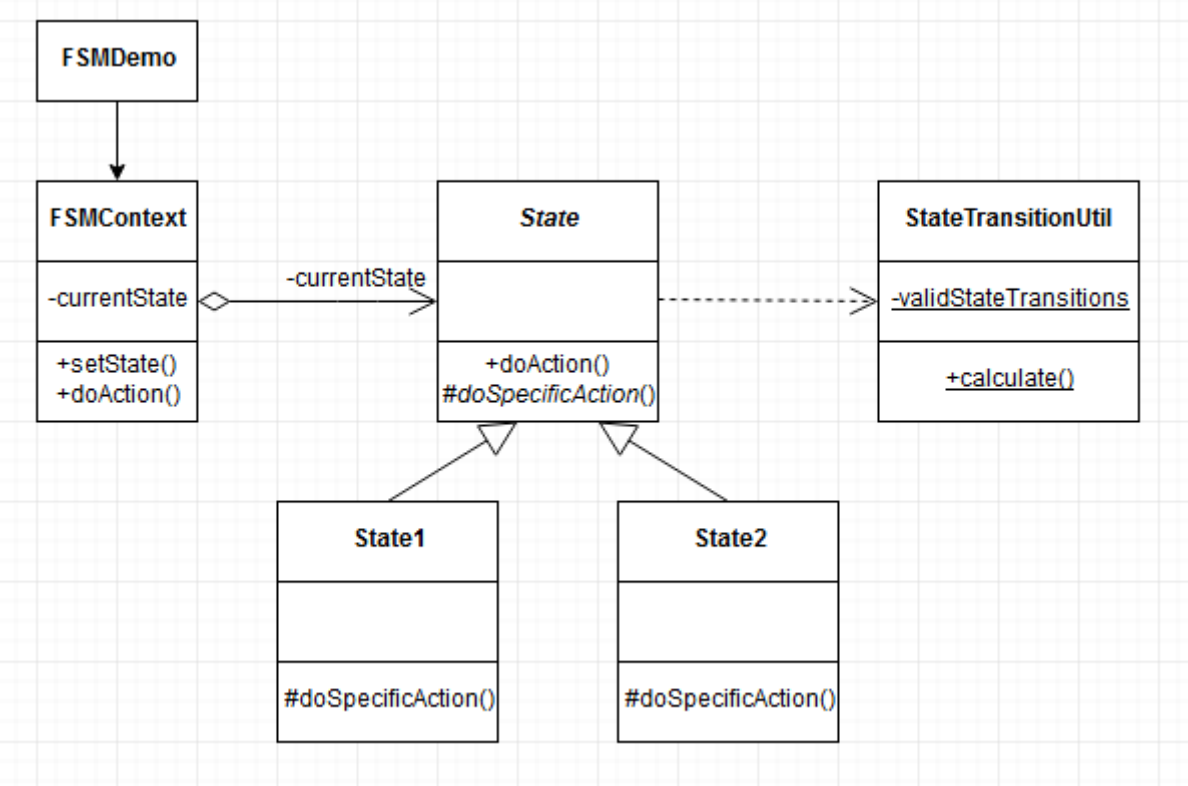

Figure 4. State design pattern – UML class diagram

```
class FSMContext;
  local State currentState;

  function new(State initialState);
    currentState = initialState;
  endfunction

  function void setState(State s);
    currentState = s;
  endfunction

  function void doAction(Input inputs);
    currentState.doAction(this, inputs);
  endfunction
endclass
```

Figure 5. State design pattern based FSM implementation - Context class

```
virtual class State;
  function void doAction(FSMContext cntxt, Input inputs);
    State nextState;

    doSpecificSeqAction(cntxt, inputs);
    nextState = StateTransitionUtil::calculate(this, inputs);
    cntxt.setState(nextState);
    nextState.doSpecificCombAction(cntxt, inputs);
  endfunction

  pure virtual function void doSpecificCombAction(FSMContext cntxt, Input inputs);
  pure virtual function void doSpecificSeqAction(FSMContext cntxt, Input inputs);

  pure virtual function fsm_t getStateId();
endclass
```

Figure 6. State design pattern based FSM implementation - Abstract State class

```
class RunState extends State;
  local static RunState inst = null;

  protected function new();  endfunction

  static function RunState Instance();
    if (inst == null)
      inst = new();
    return inst;
  endfunction

  virtual function void doSpecificCombAction(FSMContext cntxt, Input inputs);
    inputs.vif0.irq              <= 0;
    inputs.vif0.iso_expected  <= 0;
    inputs.vif0.clkg_expected <= 0;
  endfunction

  virtual function void doSpecificSeqAction(FSMContext cntxt, Input inputs);
  endfunction

  virtual function fsm_t getStateId();
    return fsm_run;
  endfunction
endclass

class ResetState extends State;
class InitState extends State;
class Clock_GateState extends State;
class Switch_OffState extends State;
class ErrorState extends State;
```

Figure 7. State design pattern based FSM implementation - Concrete State classes

```
class StateTransitionUtil;
  local static State validStateTransitions[State][$];
  local static CovergroupWrapper cgWrapper;

  static function void init();
    validStateTransitions[ResetState::Instance()]   = { ResetState::Instance(), InitState::Instance()};
    …
    cgWrapper = new();
  endfunction

  static function State calculate(State currentState, Input inputs);
    State nextState = null;
    State result    = null;
    State nextValid[$];

    nextState = calculateNextState(currentState, inputs);

    nextValid = validStateTransitions[currentState].find(x) with ( x == nextState );
    if (nextValid.size() != 0) begin
      cgWrapper.sample(currentState.getStateId(), nextState.getStateId());
      return nextState;
    end
    else begin
      return ErrorState::Instance();
    end
  endfunction

  extern static function State calculateNextState(State currentState, Input inputs);
endclass
```

Figure 8. State design pattern based FSM implementation - State transition utility

* *Checkers*

One of the key tasks in the process of FSM verification is assuring that the correct values are driven at the FSM output. As shown in Figure 7, each derived state class updates the expected values of relevant output signals, which facilitates the implementation of the required assertions. As the code which performs the prediction is encapsulated within a derived class, the addition of a new state is very simplified. Basically, it would be necessary to provide a set of signal that a certain state has an impact on. Also, the removal of a certain state is straightforward, as a well-encapsulated and decoupled code does not affect the rest of the verification environment. Examples that show comparison between the expected and the observed signal values are provided in Figure 9. As part of checking mechanism, the derived classes can also interact with the UVM register model, updating the expected values of status registers, which are compared against the values controlled within the DUT upon a read access.

* *Functional coverage*

The state and state transition coverage is implemented within a covergroup wrapper class, providing coverage creation on demand. The wrapper class is instantiated within the state transition utility class, where the sampling is also performed. The covergroup wrapper class is shown in Figure 10. The coverage model can be easily extended to include additional factors into the coverage metric.

* *Generation*

On top of giving contribution on the passive checking and collection side, good FSM modelling in the verification environment can facilitate stimuli generation. In the shown example, each transition between states is associated with a dedicated *uvm_sequence*, which provides a convenient way of developing testcases which lead the state machine from one state to another. In the UVM context, each sequence represents a virtual sequence which coordinates activity on multiple interfaces – register access, reset pin driving, etc. in order to achieve state transitions. On top of that, a graph traversing algorithm is utilized do generate random scenarios. As an input, the user provides a list of states they want to be entered during the testcase and the testcase creates a random sequence of transitions which eventually lead the state machine into the desired states. This is demonstrated in Figures 11, 12 and 13. Relevant waveform is shown in Figure 14. The developed sequences can also be reused across testcases, in combination with sequences that target other features to increase the stress which is applied on the DUT.

```
logic iso_observed, iso_expected;
logic clkg_observed, clkg_expected;

property iso;
    @(posedge clock) iso_observed == iso_expected;
endproperty
assert property (iso);

property clkg;
  @(posedge clock) clkg_observed == clkg_expected;
endproperty
assert property (clkg);
```

Figure 9. Implemented SystemVerilog assertions

```
class CovergroupWrapper;
  fsm_t currentStateId;
  fsm_t nextStateId;

  covergroup state_cg();
    coverpoint currentStateId { ignore_bins ignore_val = { ErrorState::Instance().getStateId() }; }
    coverpoint nextStateId    { ignore_bins ignore_val = { ErrorState::Instance().getStateId() }; }

    cross currentStateId, nextStateId {
      ignore_bins reset_ignore = binsof(currentStateId) intersect { ResetState::Instance().getStateId() } &&
                                 binsof(nextStateId)    intersect { RunState::Instance().getStateId(),
                                                                    Clock_GateState::Instance().getStateId(),
                                                                    Switch_OffState::Instance().getStateId() } ;
    … }
  endgroup

  function new(); state_cg = new(); endfunction

  virtual function void sample(fsm_t currentStateId, fsm_t nextStateId);
    this.currentStateId = currentStateId;
    this.nextStateId    = nextStateId;
    state_cg.sample();
  endfunction
endclass
```

Figure 10. State and state transition coverage

```
class run_to_reset_sequence extends uvm_sequence;
  `uvm_object_utils(run_to_reset_sequence)
  `uvm_declare_p_sequencer(virtual_sequencer)

  drive_reset_sequence reset_seq;

  function new(string name="run_to_reset_sequence");
    super.new(name);
  endfunction

  virtual task body();
    `uvm_do(drive_reset_seq)
  endtask
endclass
```

Figure 11. A *uvm_sequence* corresponding to a state transition

```
State enterState[$] = { Clock_GateState::Instance(), InitState::Instance(), Switch_OffState::Instance() };
```

Figure 12. A testcase developer-defined sequence of states to be entered during a testcase

```
Transition from Reset to Init
Transition from Init to Reset
Transition from Reset to Init
Transition from Init to Run
Transition from Run to Clock_Gate
Transition from Clock_Gate to Reset
Transition from Reset to Init
Transition from Init to Run
Transition from Run to Reset
Transition from Reset to Init
Transition from Init to Run
Transition from Run to Clock_Gate
Transition from Clock_Gate to Reset
Transition from Reset to Init
Transition from Init to Run
Transition from Run to Switch_Off
```

Figure 13. A random generated sequence of states eventually entering the defined sequence of states – log output

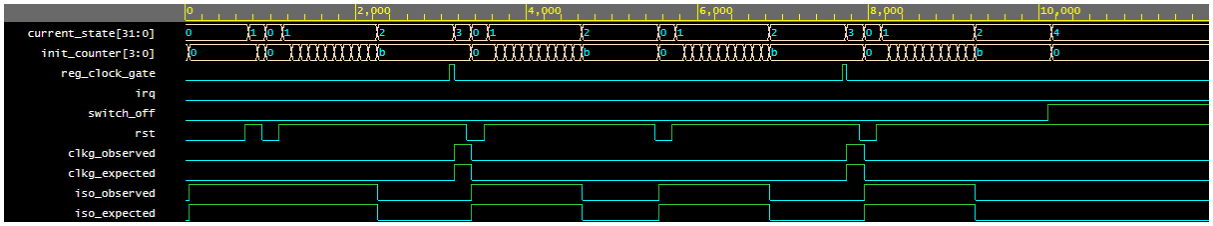

Figure 14. A random generated sequence of states eventually entering the defined sequence of states - waveform

III. SUMMARY

The solution natively developed for the software modelling, can easily be integrated within a UVM environment, offering the verification engineers a powerful way of verifying FSMs. On the generation side, the relevant tasks can be provided to invoke UVM sequences leading the FSM from one state to another. What's more, the methodology is very beneficial on the passive side. SystemVerilog assertions developed to check the output signals values are not polluted by the information about the current state and can, therefore, be reused across several projects. The logic to calculate the expected values of the output signals is localized within the derived states and can also be reused. The state transition utility class can be extended to perform the transition coverage collection. The ability to add or remove a state without significant code change, also giving valuable contribution to the code reusability, is one more argument in favor of using the aforementioned implementation for the FSM verification. Due to its scalability, the solution especially gains advantage over the classic "case enum" solution in the process of verification of large scale systems with complex state machines. The solution also typically better scales than formal FSM analysis techniques, which provide exhaustive verification based on the set of formal properties to be satisfied, but are mainly applicable to smaller safety-critical systems as they are prone to state space explosion with the number of state variables increasing.

ACKNOWLEDGMENT

The authors would like to thank Hagai Arbel of Veriest Vtool organization for his support during the implementation of the aforementioned solutions.

REFERENCES

[1] http://accellera.org/downloads/standards/uvm

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1994.