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

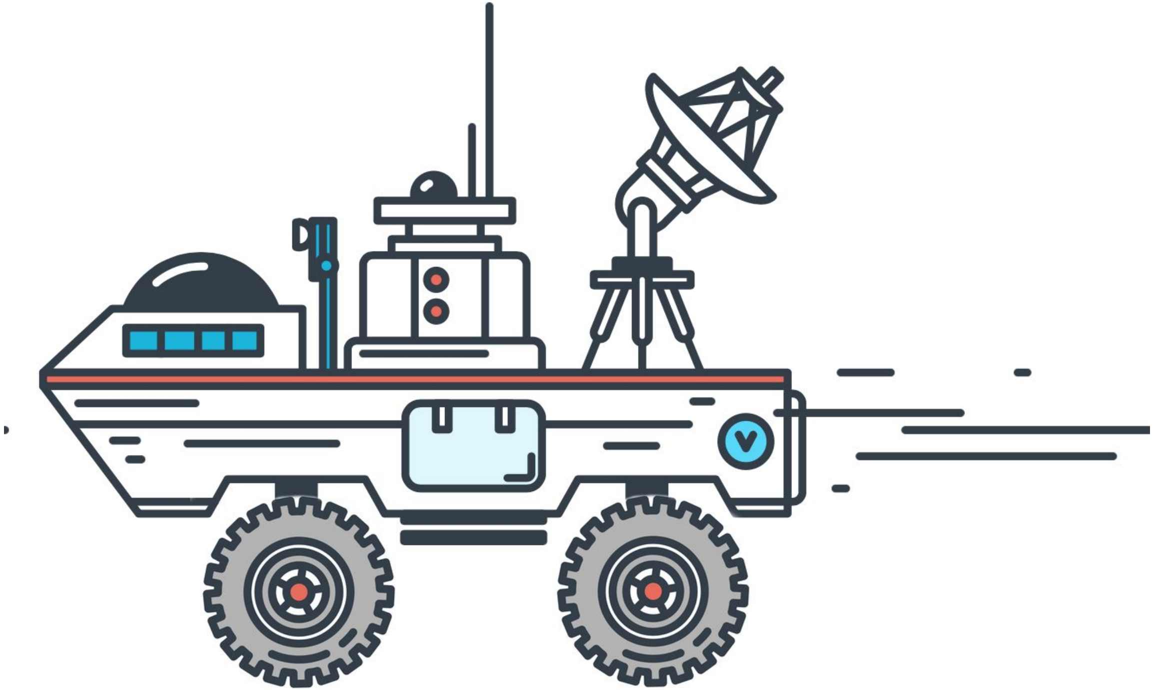
Darko M. Tomušilović

vtool smart verification

SYSTEMS INITIATIVE

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

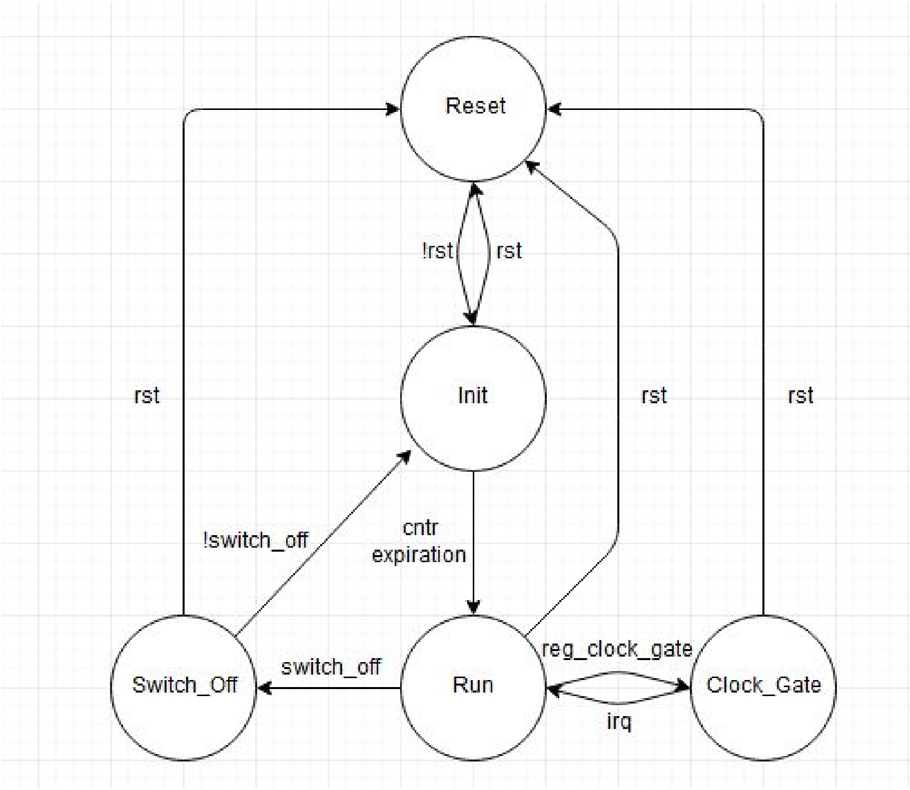
# State Machines are everywhere - And They have to be verified

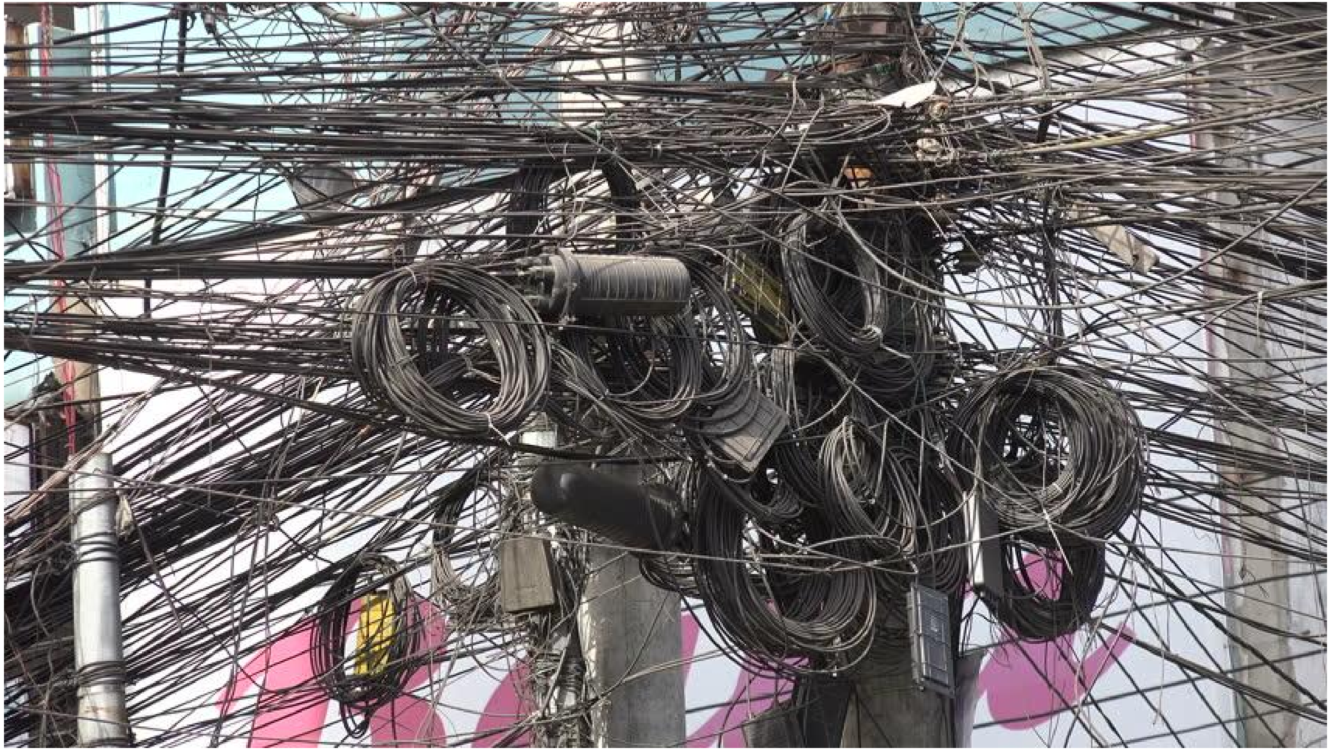


# Introduction

- FSM verification process
  - Achieve state transitions using the proper input stimulus
  - Check that the output signals are properly driven
  - Collect coverage (state, state transition, coverage on higher-level scenarios)
- FSM reference model
- Goal: reusable, modifiable solution
- Introduce main UVM concepts
- Introduce design patterns

# Example State Machine





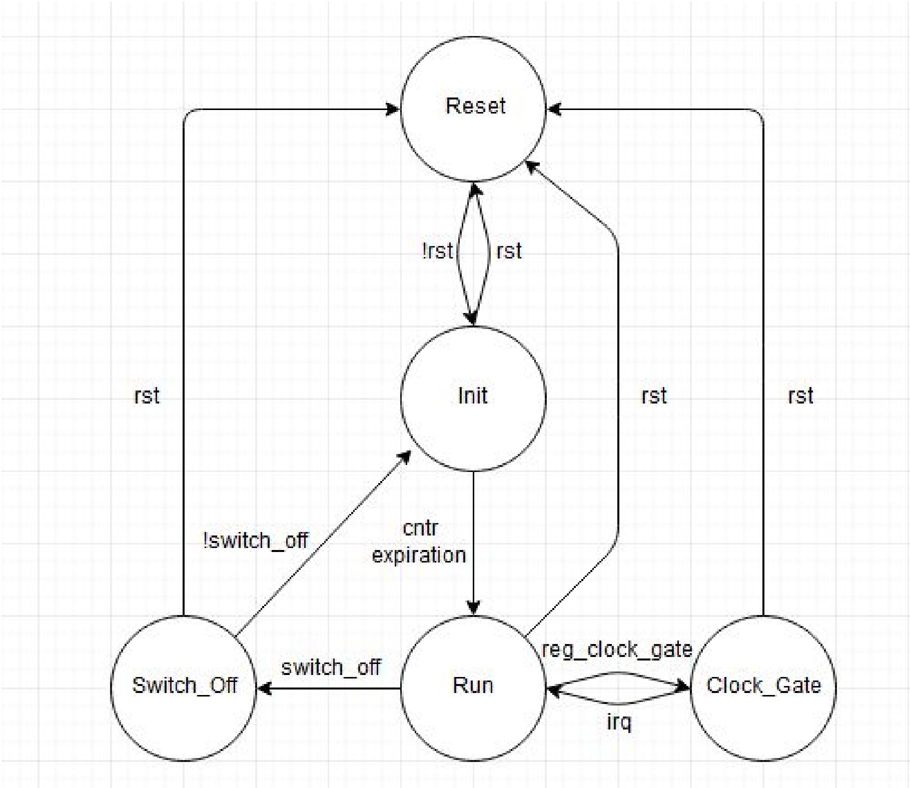
# Tightly coupled FSM implementation Overview

- The most obvious approach
- State enumeration
- A huge “if” or “switch/case” statement conditioned by the current state
- Drawbacks:
  - Independent tasks coupled together
  - Not straightforward for reuse
  - Code duplication

# Tightly coupled FSM implementation
# Code example

```
class FSMExample;
	local fsm_t currentState;
	function void doAction(Input inputs);
	  case (currentState)
		fsm_reset: begin
			doActionForState_reset(inputs);
			… // checkers, coverage, register model update, etc.
			currentState = calculateNewState(currentState, inputs);
		end
		...
	  endcase
	endfunction
endclass
```

# Example State Machine

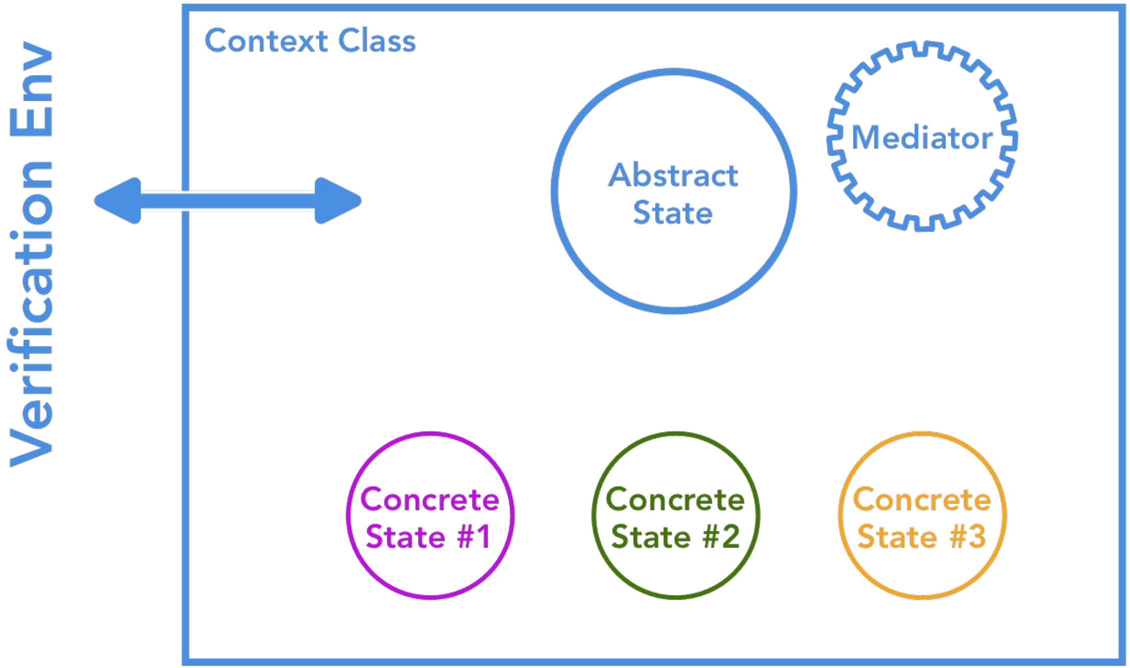


# dīvide et īmpera

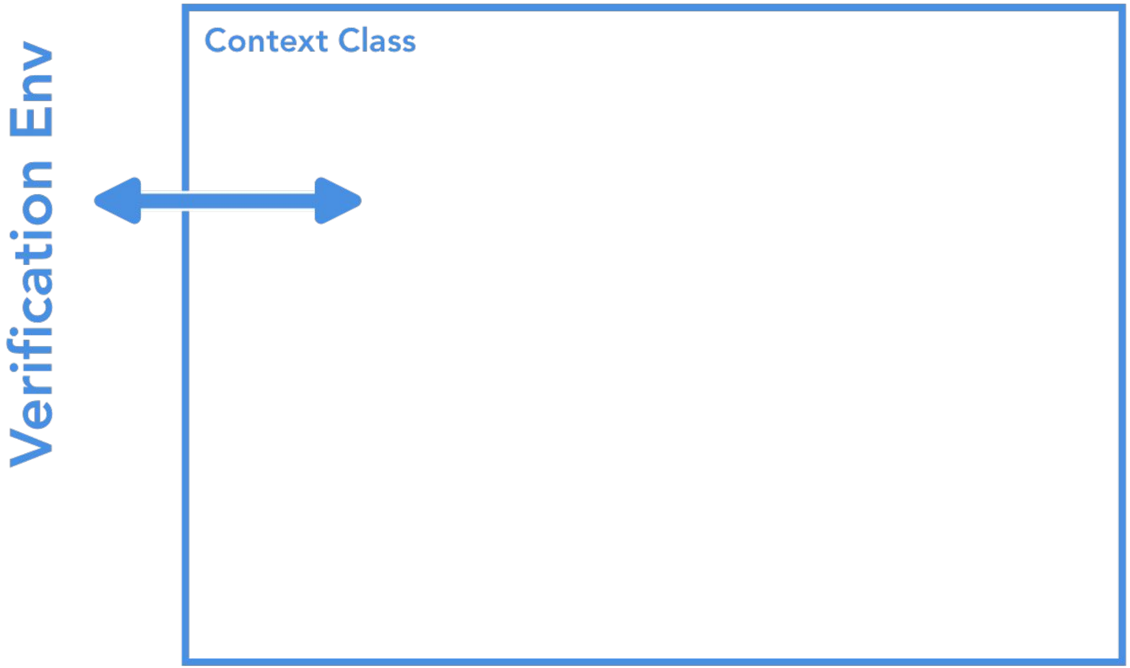
*Divide and Conquer*

# Loosely coupled FSM implementation Overview

- State design pattern
  - Model state machines, decouple them from the rest of the system, provide simple interface to them
- Context class
- Abstract State base class
- Concrete State classes
- State transition logic



Verification Env
Context Class
Abstract State
Mediator
Concrete State #1
Concrete State #2
Concrete State #3

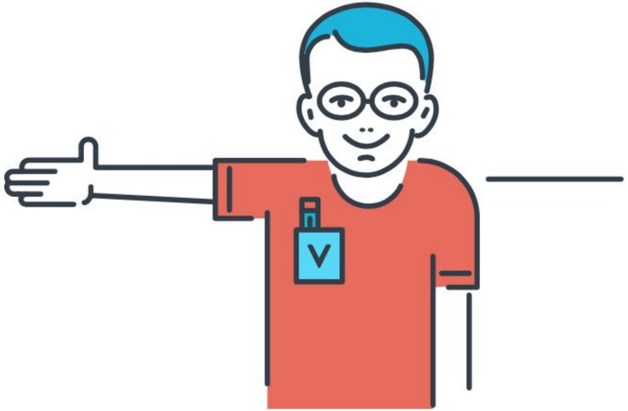

Verification Env
Context Class

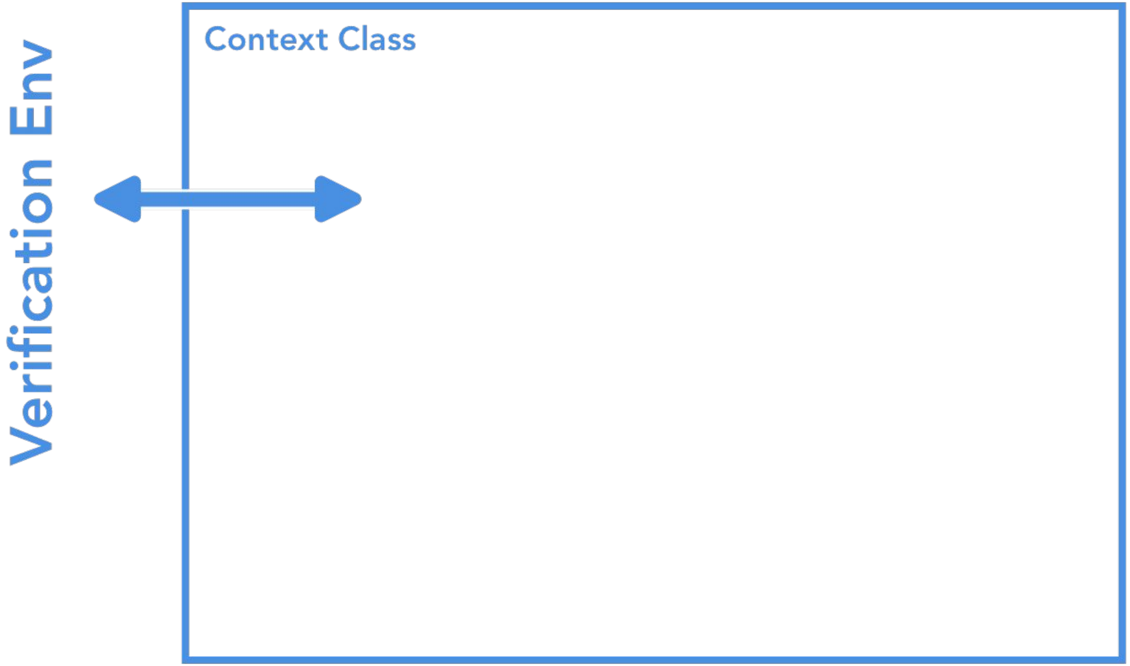
# Loosely coupled FSM implementation
# Context class

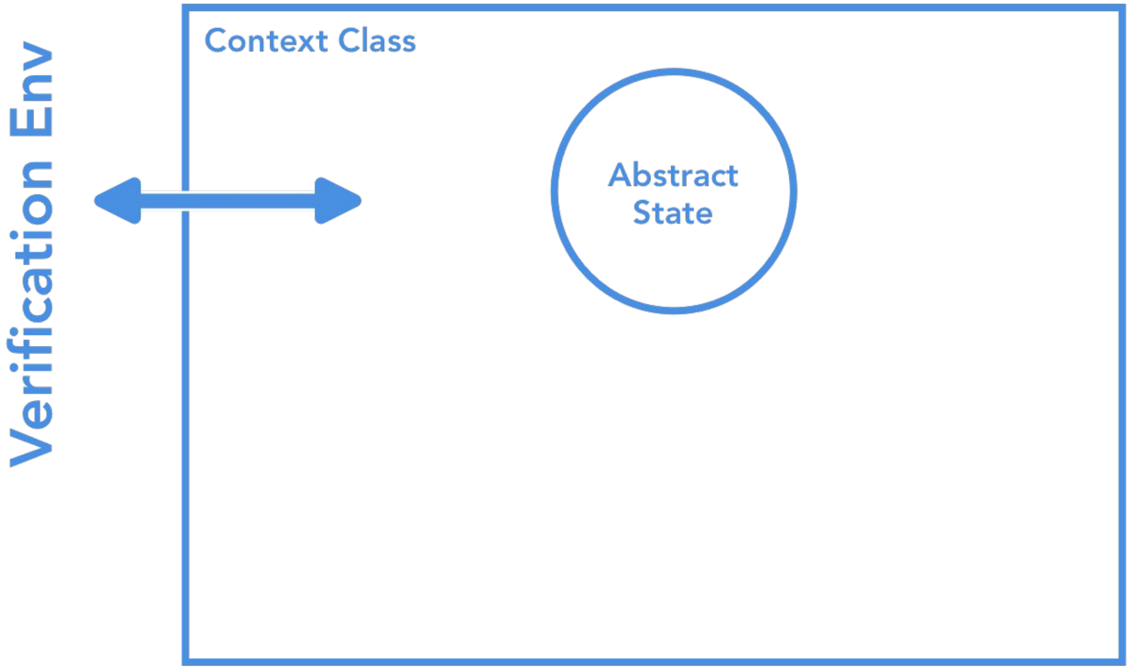
- State design pattern
  - Model state machines, decouple them from the rest of the system, provide simple interface to them
- Context class
  - Communicates with the rest of the Verification environment
  - Provided with the observed values of the input signals
- Abstract State base class
- Concrete State classes
- State transition logic

# Loosely coupled FSM implementation
# Context class - Code example

```
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
```




Verification Env
Context Class

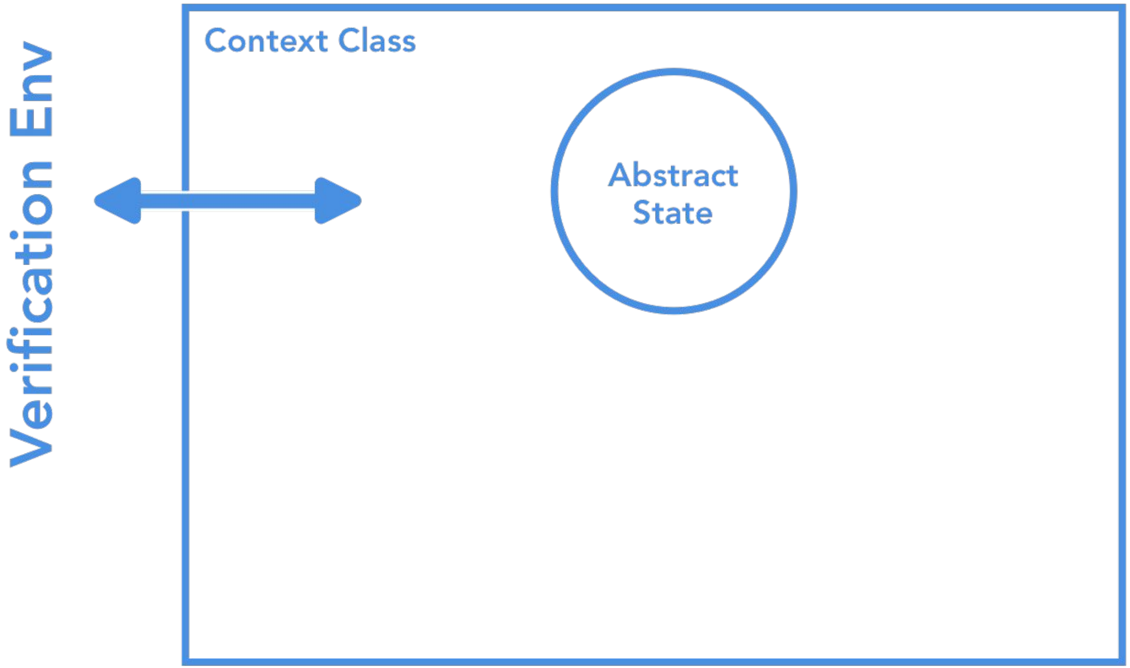

Verification Env
Context Class
Abstract
State

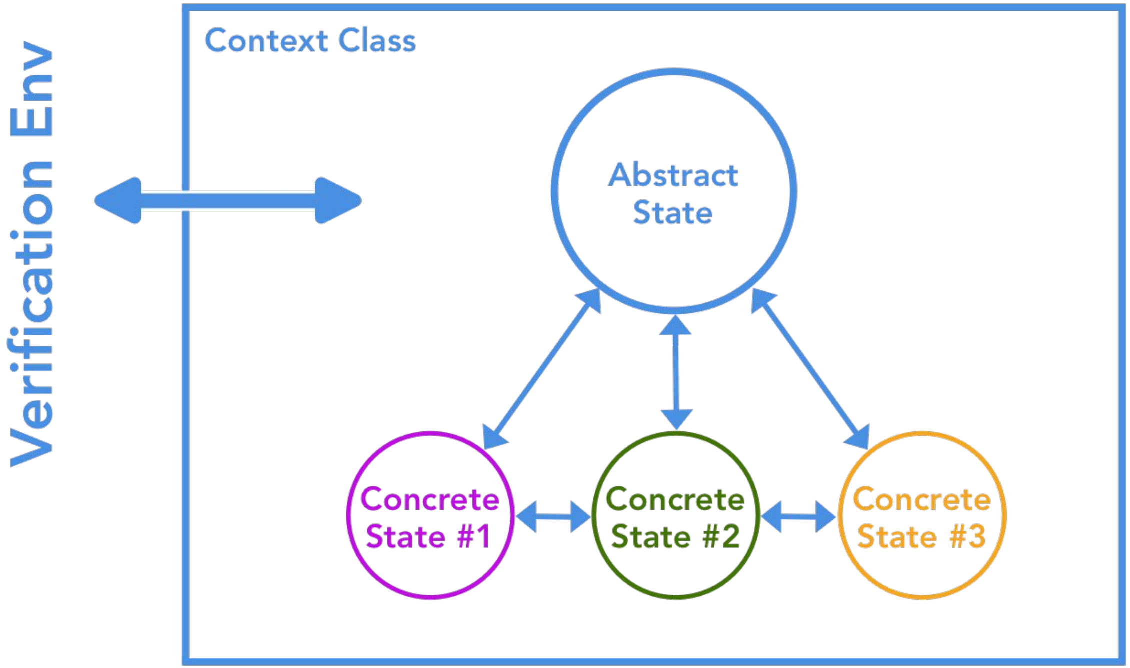
# Loosely coupled FSM implementation Abstract State class

- State design pattern
  - Model state machines, decouple them from the rest of the system, provide simple interface to them
- Context class
- Abstract State base class
  - Features and actions common to every state of a state machine
  - Main behavior modelled using Template method design pattern
- Concrete State classes
- State transition logic

# Loosely coupled FSM implementation
# Abstract State class - Code example

```
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);
endclass
```


Verification Env
Context Class
Abstract
State

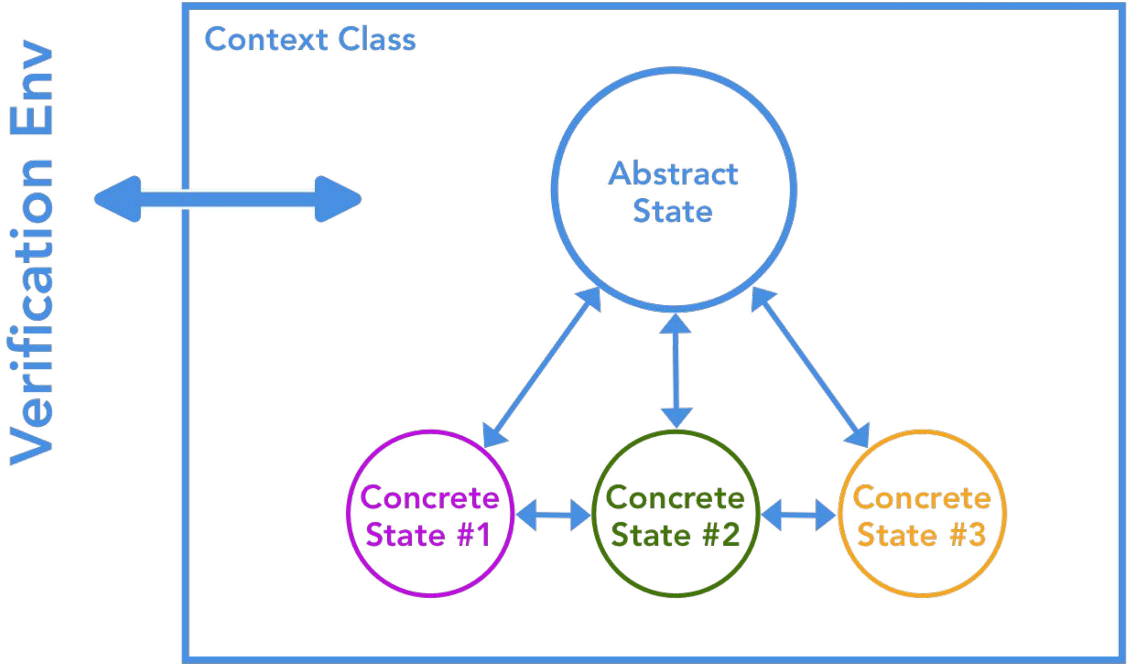

Verification Env
Context Class
Abstract State
Concrete State #1
Concrete State #2
Concrete State #3

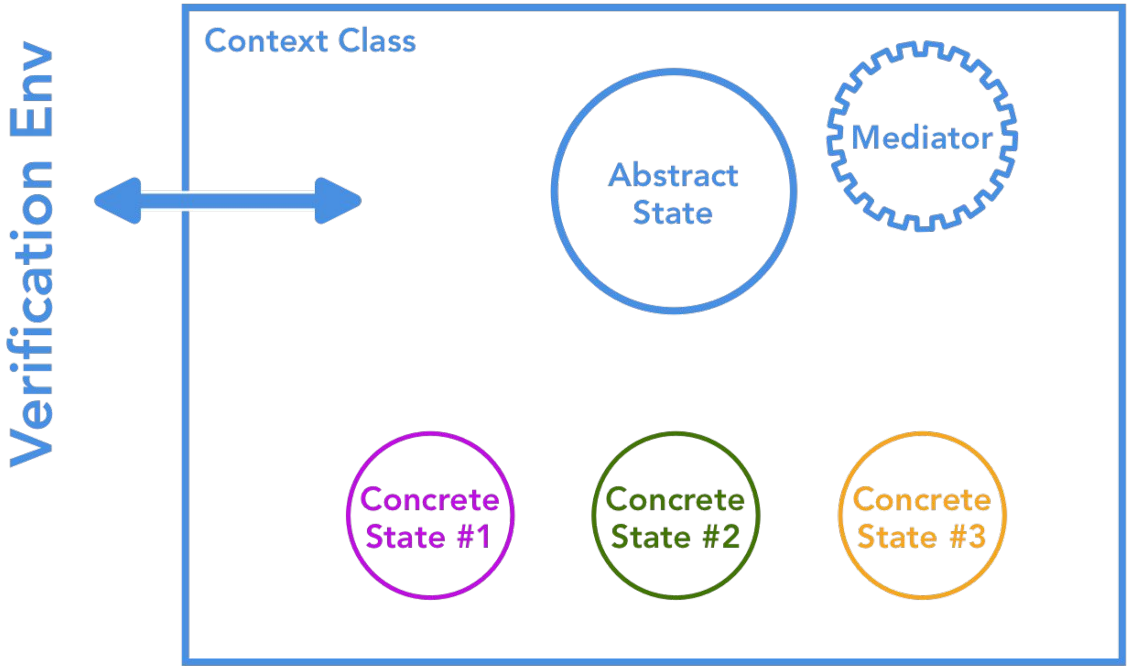
# Loosely coupled FSM implementation Concrete State class

- State design pattern
  - Model state machines, decouple them from the rest of the system, provide simple interface to them
- Context class
- Abstract State base class
- Concrete State classes
  - Define a state-specific behavior
  - Modelled using Singleton design pattern
- State transition logic

# Loosely coupled FSM implementation Concrete State class - Code example

```
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.iso_expected <= 0;
    endfunction
    virtual function void doSpecificSeqAction(FSMContext cntxt, Input inputs); endfunction
endclass
```


Verification Env
Context Class
Abstract State
Concrete State #1
Concrete State #2
Concrete State #3


Verification Env
Context Class
Abstract State
Mediator
Concrete State #1
Concrete State #2
Concrete State #3

# Loosely coupled FSM implementation Mediator class

- State design pattern
  - Model state machines, decouple them from the rest of the system, provide simple interface to them
- Context class
- Abstract State base class
- Concrete State classes
- State transition logic
  - Modelled using Mediator design pattern
  - Mediator utility class
  - Localization, decoupling, improved code maintainability

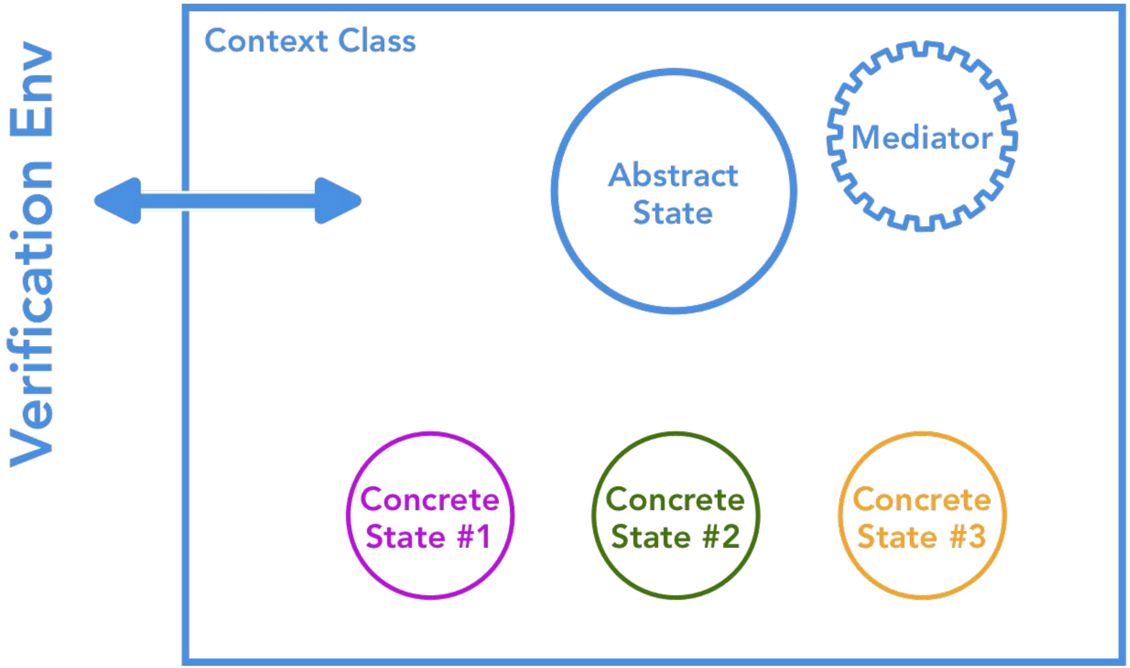
accellera SYSTEMS INITIATIVE

# Loosely coupled FSM implementation Mediator class - Code example

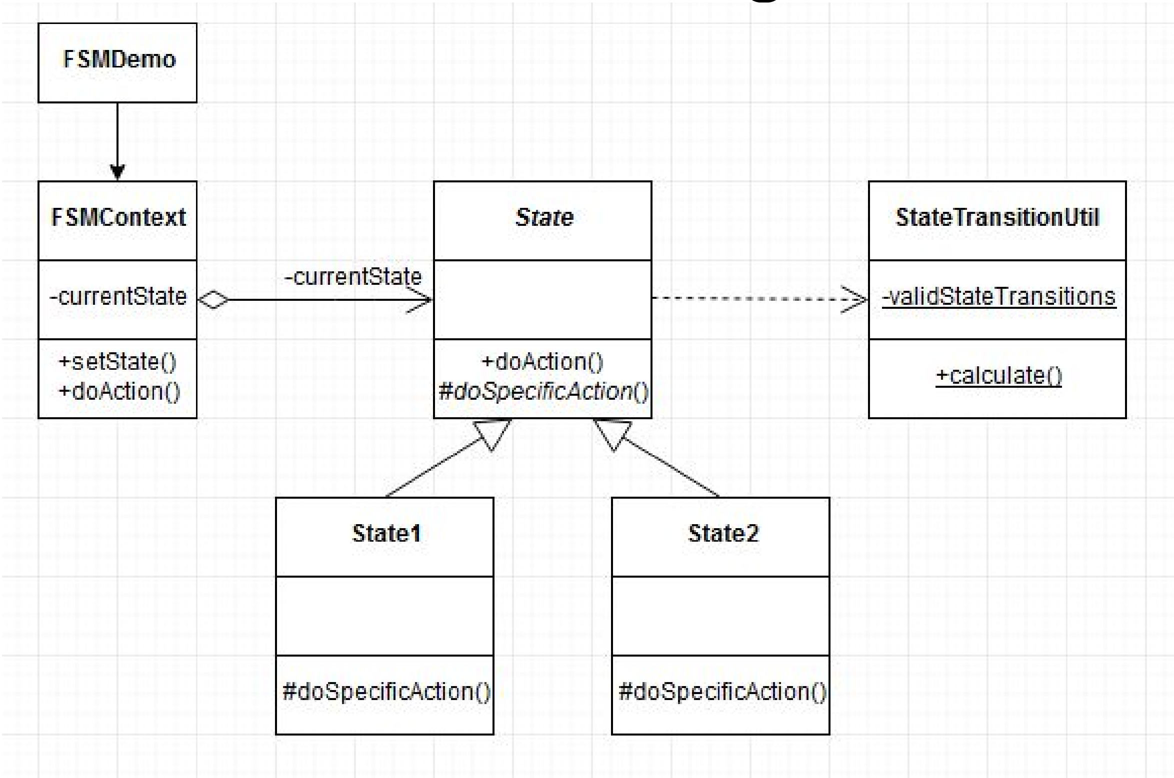
```
class StateTransitionUtil;
    local static State validStateTransitions[State][$];
    static function void init();
        validStateTransitions[ResetState::Instance()] = { ResetState::Instance(),
                                                           InitState::Instance()};
        …
    endfunction
    static function State calculate(State currentState, Input inputs);
        …
        nextState = calculateNextState(currentState, inputs);
        …  // Check whether the transition is valid
        return nextState;
    endfunction
endclass
```

# Loosely coupled FSM implementation Summary

- State design pattern
  – Model state machines, decouple them from the rest of the system, provide simple interface to them
- Context class
- Abstract State base class
- Derived state classes
- State transition logic


Verification Env
Context Class
Abstract State
Mediator
Concrete State #1
Concrete State #2
Concrete State #3

# Loosely coupled FSM implementation UML class diagram




vvv
vtool smart verification
V

# Loosely coupled FSM implementation
# Checkers implementation

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

property iso;
        @(posedge clock) iso_observed == iso_expected; // FSM output vs FSM reference model output
endproperty
assert property (iso);

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

# Loosely coupled FSM implementation Functional coverage considerations

```
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
```

# Loosely coupled FSM implementation Generation side

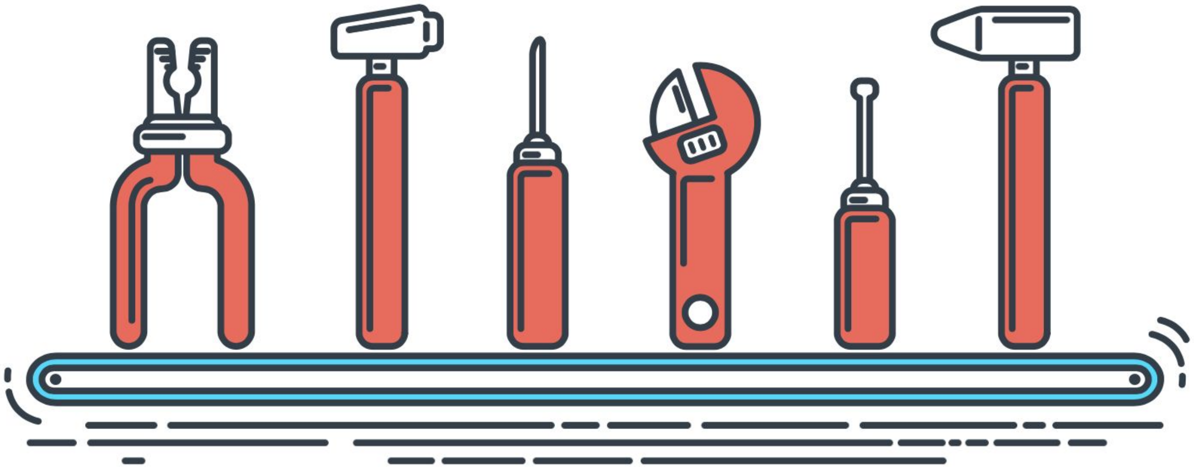
- A dedicated uvm_sequence associated with each state transition
- Graph traversing algorithm to generate random scenarios
- Input: user-provided list of states to be entered during a testcase

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

- Output: a random sequence of transitions leading the state machine into the desired states
- The developed sequences can be reused across the testcases, to stress the designed logic

# Summary

- The solution beneficial on active (generation) and passive (checking and coverage collection) side
- Improves the code quality
- More scalable solution compared to other common approaches (“case enum”, formal FSM analysis techniques)



Questions?

Thanks!