Adopting UVM for safety Verification requirements

Srinivasan Venkatarmanan, VerifWorks Pvt.Ltd
Hemakiran Kolli, CVC Pvt.Ltd
Gurubasappa Kinagi, VerifWorks Pvt.Ltd
Satinder Paul Singh, Cogknit GmbH

accellera SYSTEMS INITIATIVE

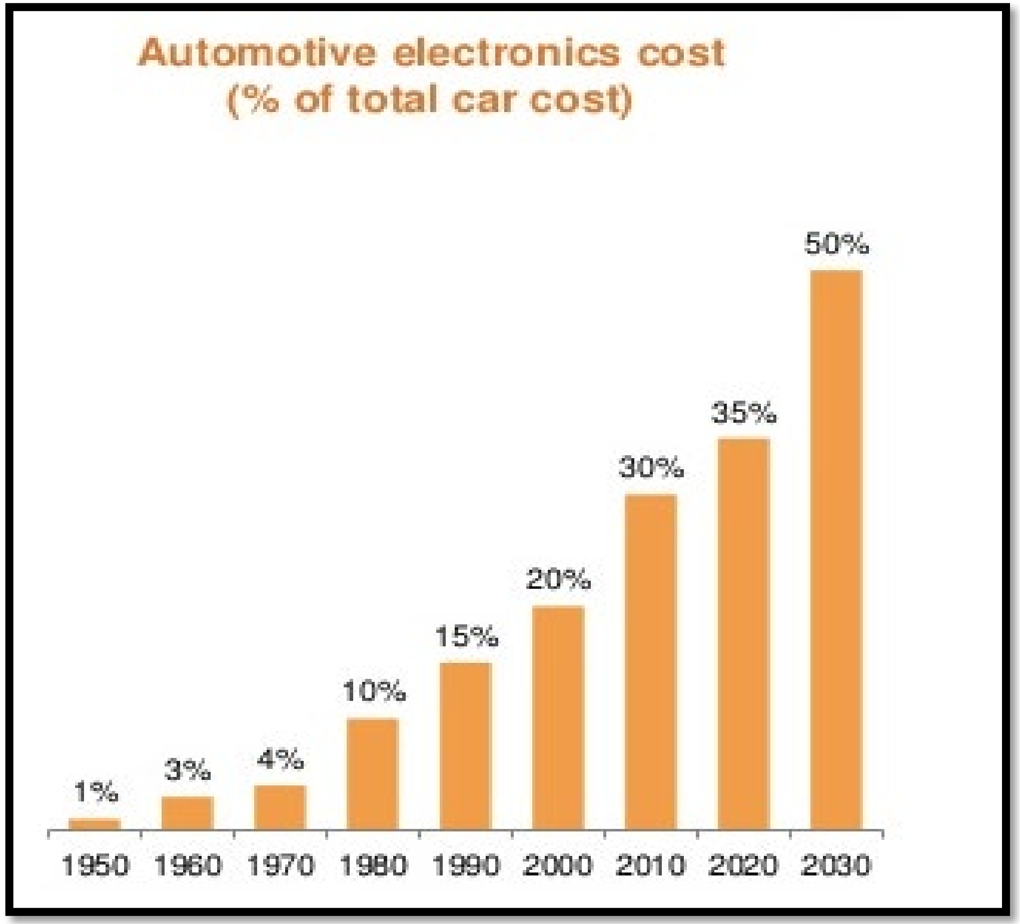
2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Agenda

- Introduction
  - Safety critical application
  - Verifying safety critical designs
- Introduction to Go2UVM
- Deploying Go2UVM in Safety Verification
  - Directed error injection
  - Random fault injection
  - Using log predictors
- Conclusion

Safety critical Application

- Automotive is a safety critical application
- Few safety applications include
  - Air bags
  - Anti-lock Brake System
  - Electronic stability control
  - Adaptive cruise control
  - Emergency breaking assist



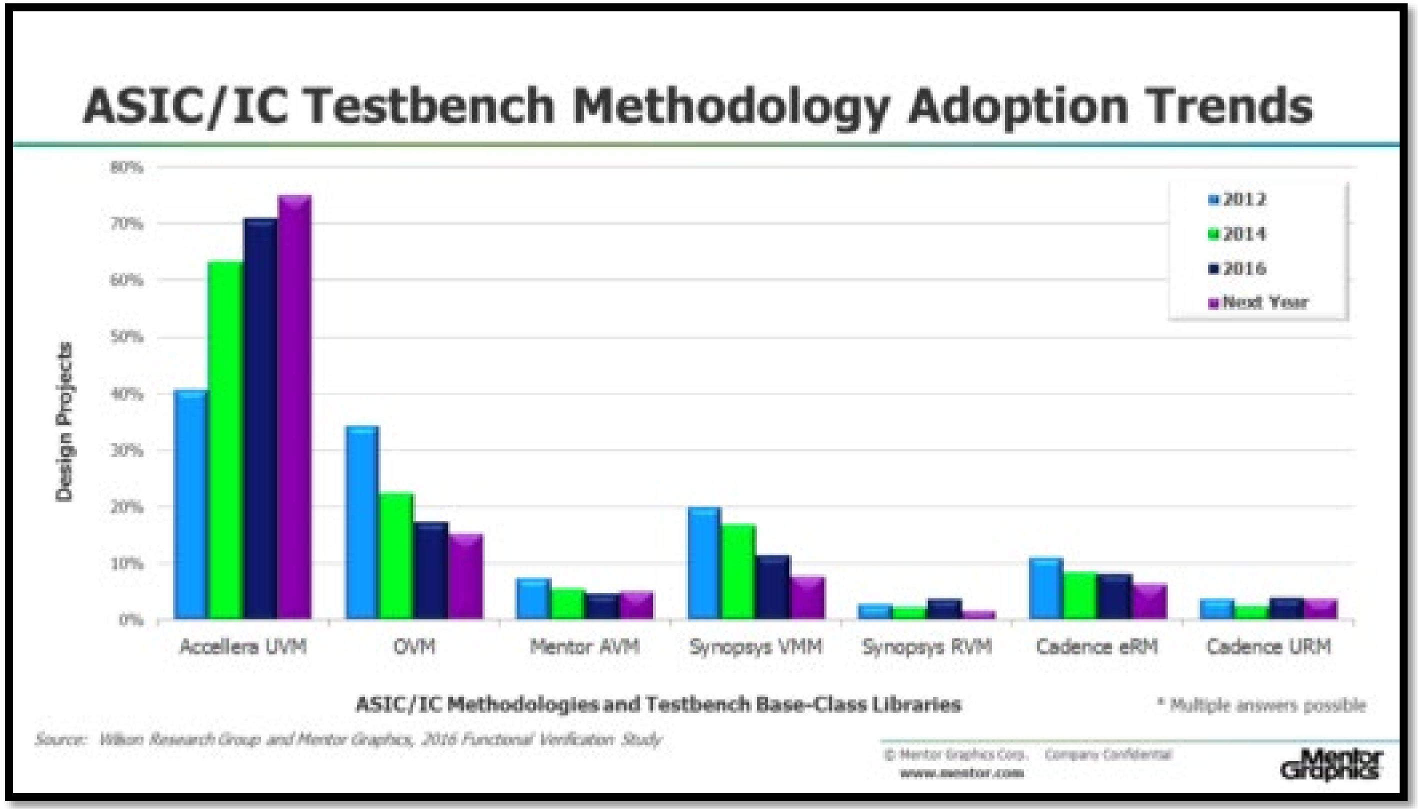
Source: PWC Analysis

Verifying safety critical designs

- Key requirements for functional verification of safety critical designs
  - Simulation of the unaltered design under test (DUT)
  - Fault injection at random points
  - Reuse of the existing functional verification environment with support for System Verilog, Universal Verification Methodology (UVM)
  - Support for multiple fault types, including single event upset (SEU), stuck-at-0/stuck-at-1, and single event
  - Log prediction to create self-checking error tests

UVM – fastest growing methodology

- Source: Independent survey by Wilson group
  - Sponsored by Mentor Graphics



ASIC/IC Testbench Methodology Adoption Trends

* Multiple answers possible

Source: Wilson Research Group and Mentor Graphics, 2016 Functional Verification Study

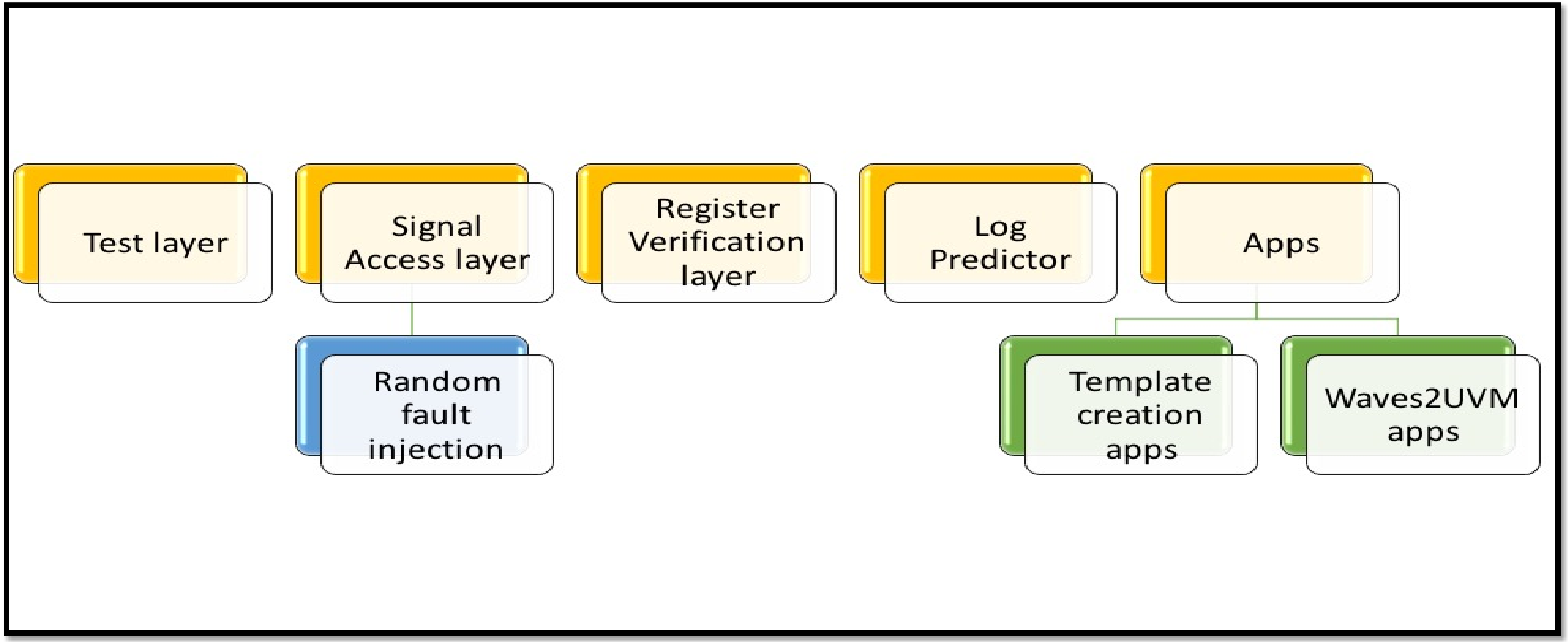
© Mentor Graphics Corp. Company Confidential
www.mentor.com

What is *Go2UVM*?

- SystemVerilog package
- TCL “apps” to auto-create Go2UVM files
- Package on top of Standard UVM framework
- Two primary goals:
  - Simplify UVM for first-time users
  - Extend standard UVM to add specific features
- Simplifying UVM adoption:
  - Go2UVM base Test from *uvm_test* class
  - Hides phasing, objection, name-parent hook-up etc.
- Extended features
  - Fault injection
  - Log predictor
  - Checker library
  - Built-in UVCs for Registers, Low power verification etc.

Focus of this paper

Go2UVM in a nutshell



Test layer

Signal Access layer

Random fault injection

Register Verification layer

Log Predictor

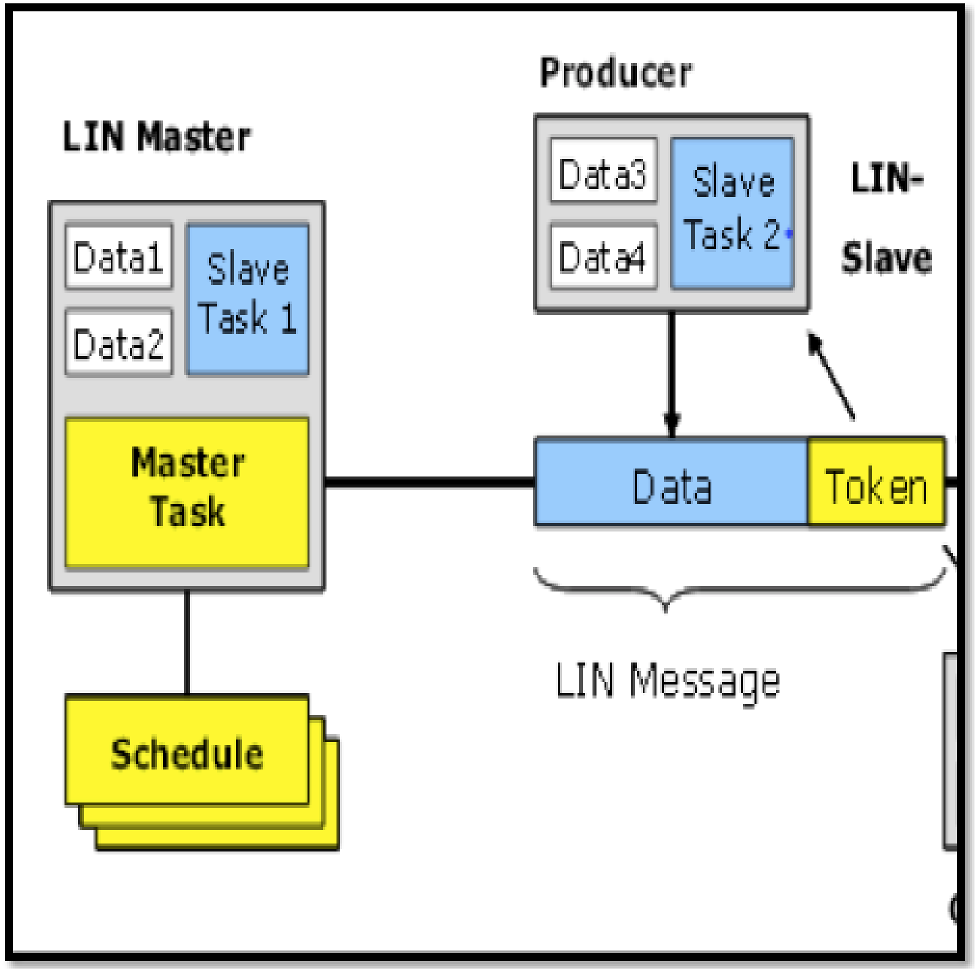
Apps

Template creation apps

Waves2UVM apps

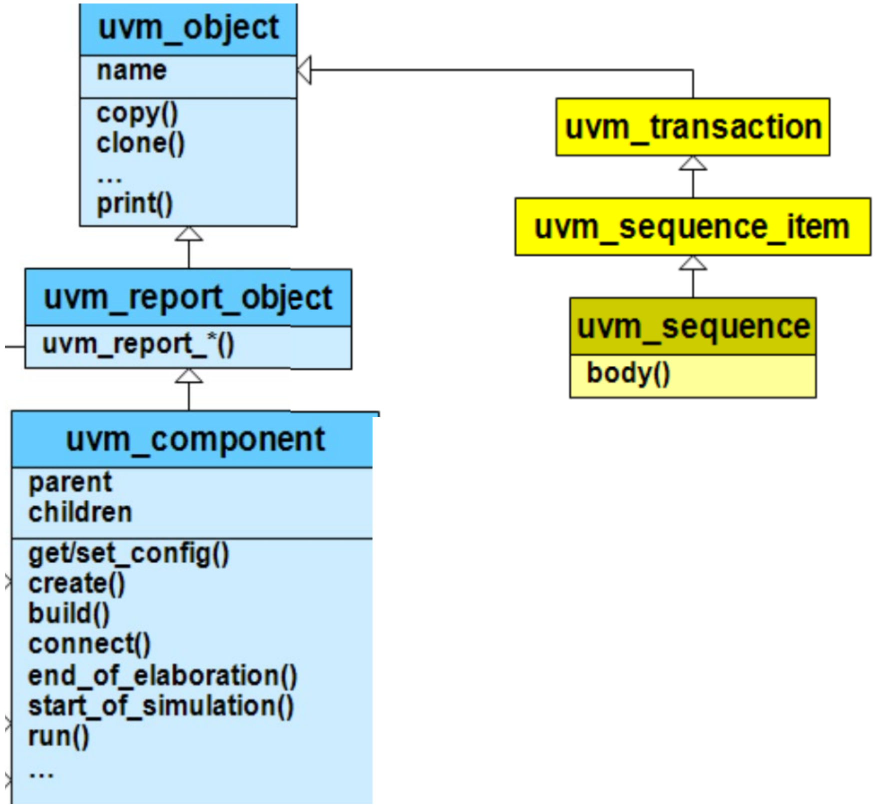
Directed Error injection

- Typical UVCs contain several error injection capabilities
  - Some are part of transaction
  - Some are part of components/drivers
- Consider LIN protocol
- Typical errors:
  - Delimiter err
  - Checksum err
  - PID start/stop err
  - Parity err
  - Oversize err etc.
- UVC has knobs to control these error generation



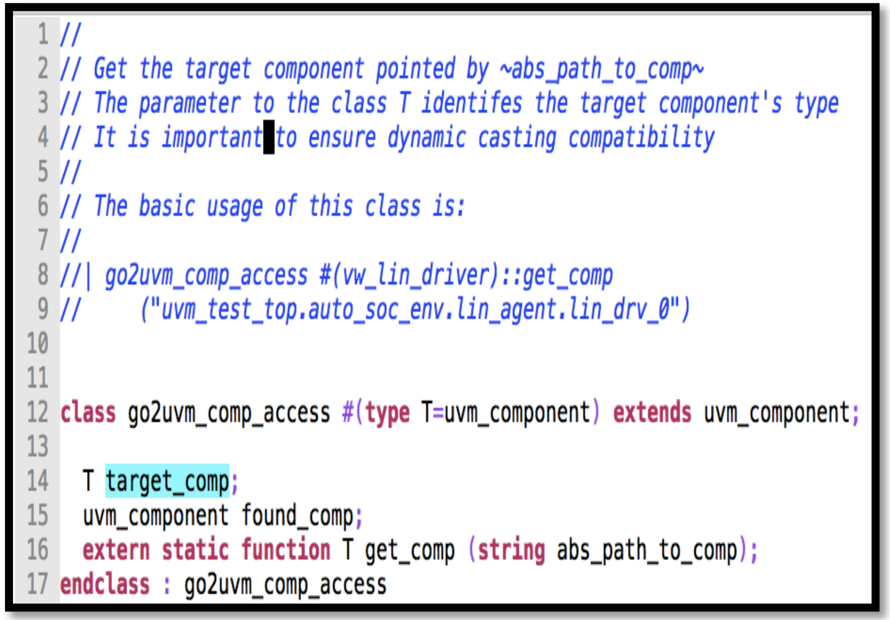
LIN Error control in UVM framework

- UVM base class – 2 main class trees
  - Components (Hierarchical)
  - Transaction/SEQ (Not hier aware)
- Often users need to write Virtual sequences to develop test scenarios
- Tweaks knobs in Driver/Agent from uvm_sequence::body()
- Out-of-the-box UVM does not support this
  - As sequences are not hierarchy aware
  - uvm_root class has API needed for this



Go2UVM Component access feature

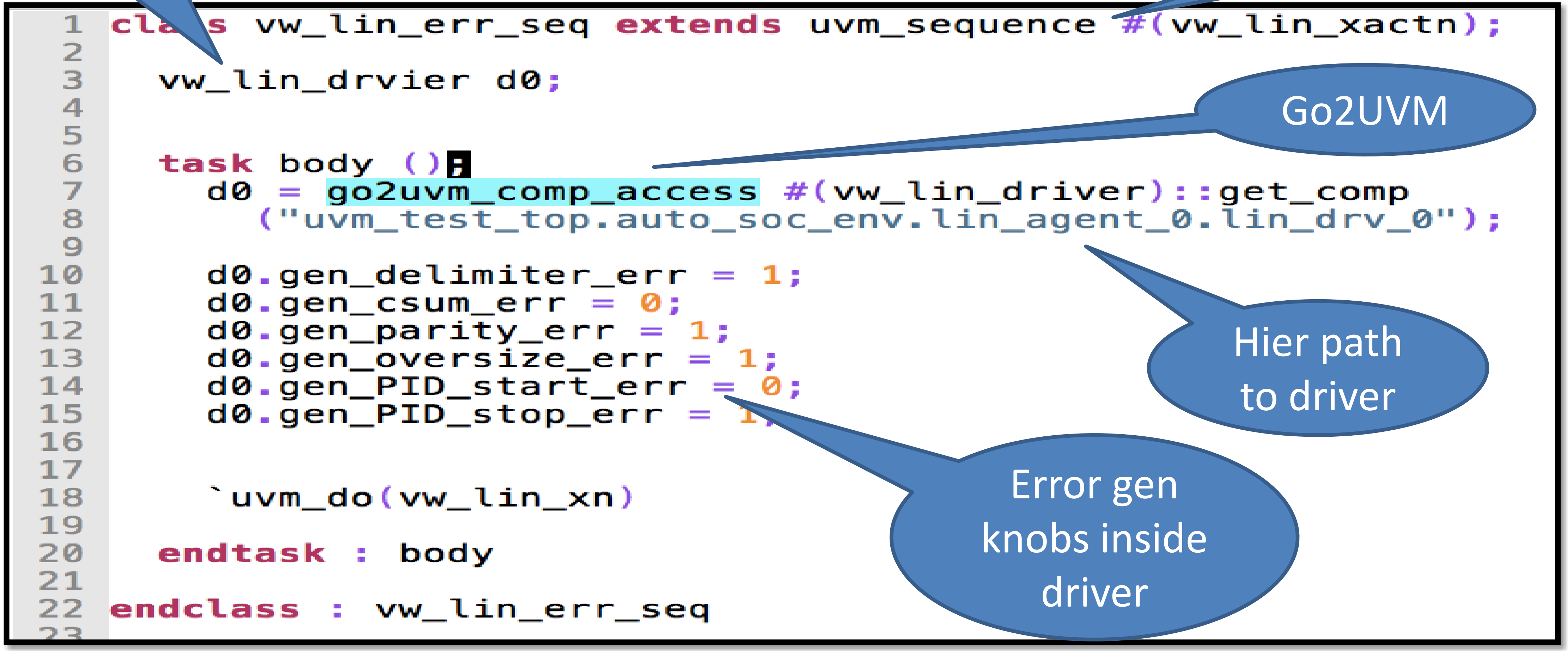
- Base class: ***go2uvm_comp_access***
- An OOP layer around *uvm_component*
- Has a static function ***get_comp()***
- Uses *uvm_root::find* API
  - Makes it easy to use for end-users
  - Built-in error checking for wrong hierarchy specification
  - Hides dynamic casting ($cast) from end-user



```
//
// Get the target component pointed by ~abs_path_to_comp~
// The parameter to the class T identifes the target component's type
// It is important to ensure dynamic casting compatibility
//
// The basic usage of this class is:
//
//| go2uvm_comp_access #(vw_lin_driver)::get_comp
//     ("uvm_test_top.auto_soc_env.lin_agent.lin_drv_0")


class go2uvm_comp_access #(type T=uvm_component) extends uvm_component;

  T target_comp;
  uvm_component found_comp;
  extern static function T get_comp (string abs_path_to_comp);
endclass : go2uvm_comp_access
```

Go2UVM component access layer

Using Go2UVM comp_access



```
class vw_lin_err_seq extends uvm_sequence #(vw_lin_xactn);

  vw_lin_drvier d0;


  task body ();
    d0 = go2uvm_comp_access #(vw_lin_driver)::get_comp
      ("uvm_test_top.auto_soc_env.lin_agent_0.lin_drv_0");

    d0.gen_delimiter_err = 1;
    d0.gen_csum_err = 0;
    d0.gen_parity_err = 1;
    d0.gen_oversize_err = 1;
    d0.gen_PID_start_err = 0;
    d0.gen_PID_stop_err = 1;


    `uvm_do(vw_lin_xn)

  endtask : body

endclass : vw_lin_err_seq
```

LIN driver

UVM SEQ

Go2UVM

Hier path to driver

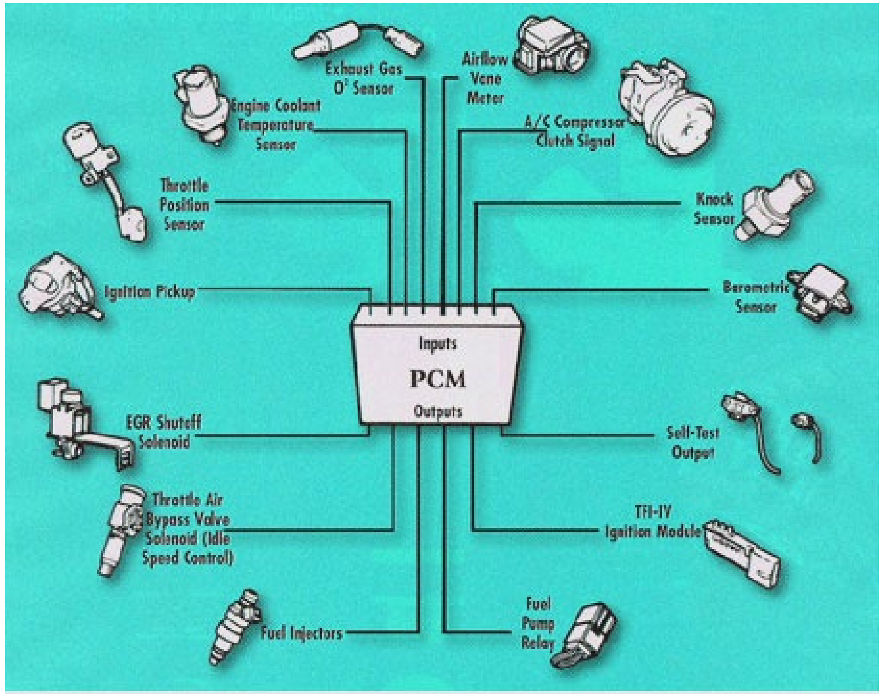
Error gen knobs inside driver

Directed Error injection - summary

- Error injection is important for safety verification in UVM
- Standard UVM sequences are “hierarchy-unaware”
- Error injection scenarios coded as sequences in UVM
- Typical UVC has error injection control knobs inside agent/driver
- Having access to those knobs from a sequence is very useful
- Go2UVM makes it easier to access any component from anywhere
- Built-in debug messages help with wrong usage

Need for Fault injection in safety verification

- Consider a typical car’s Power-train Control Module (PCM)
  - Takes inputs from various sensors
  - Controls several vital parts of a car
- Need to verify many fault scenarios
- Fault injection is essential to mimic real life scenarios

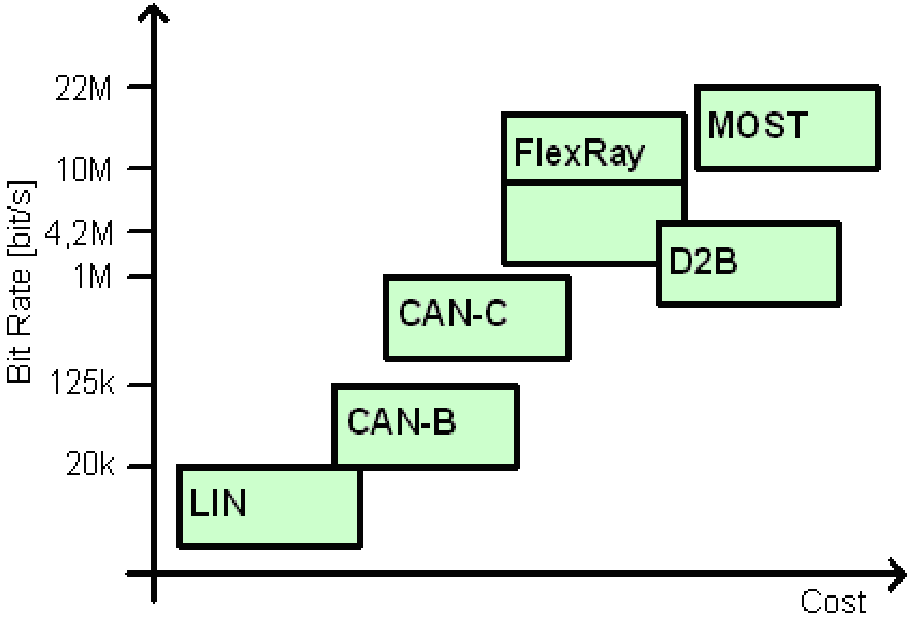


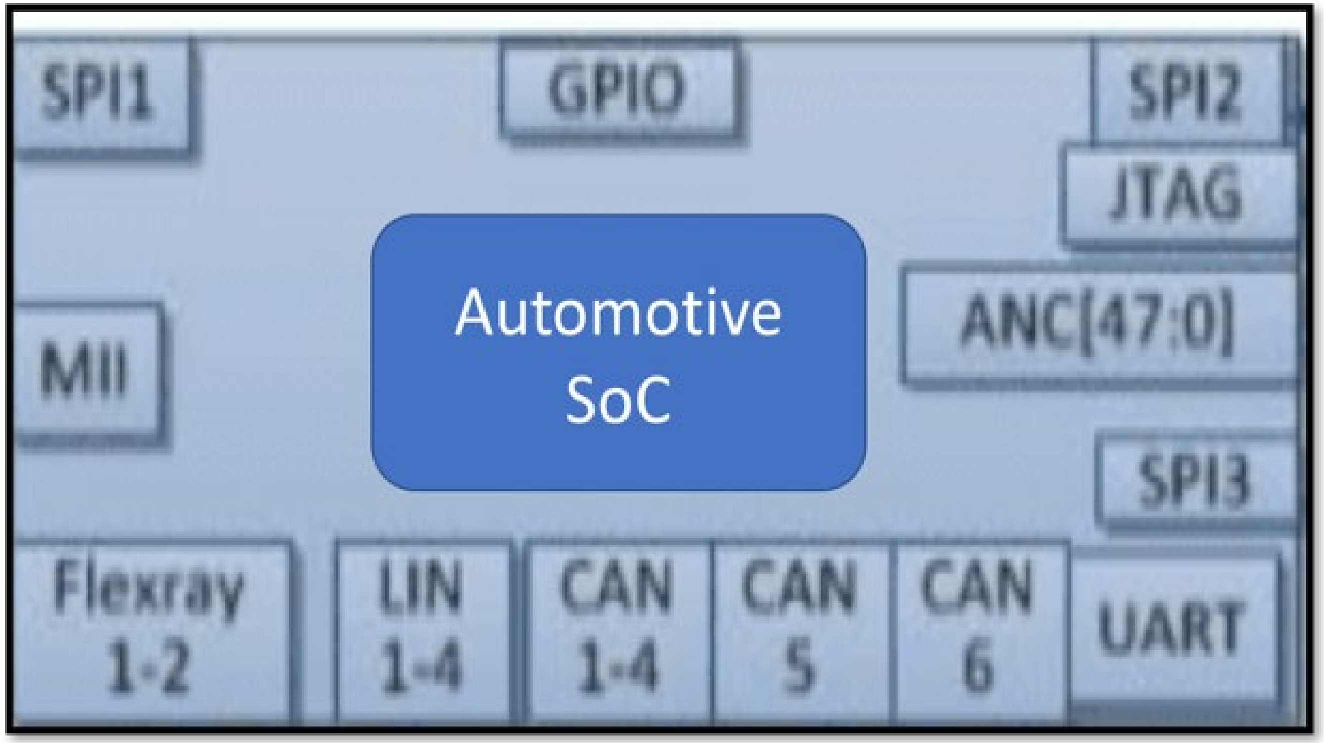
The PCM's job is to manage the powertrain. This includes the engine's ignition system, fuel injection system and emission controls. The PCM receives input from a wide variety of sensors and switches.

Ford™ PCM module

Automotive SoC example

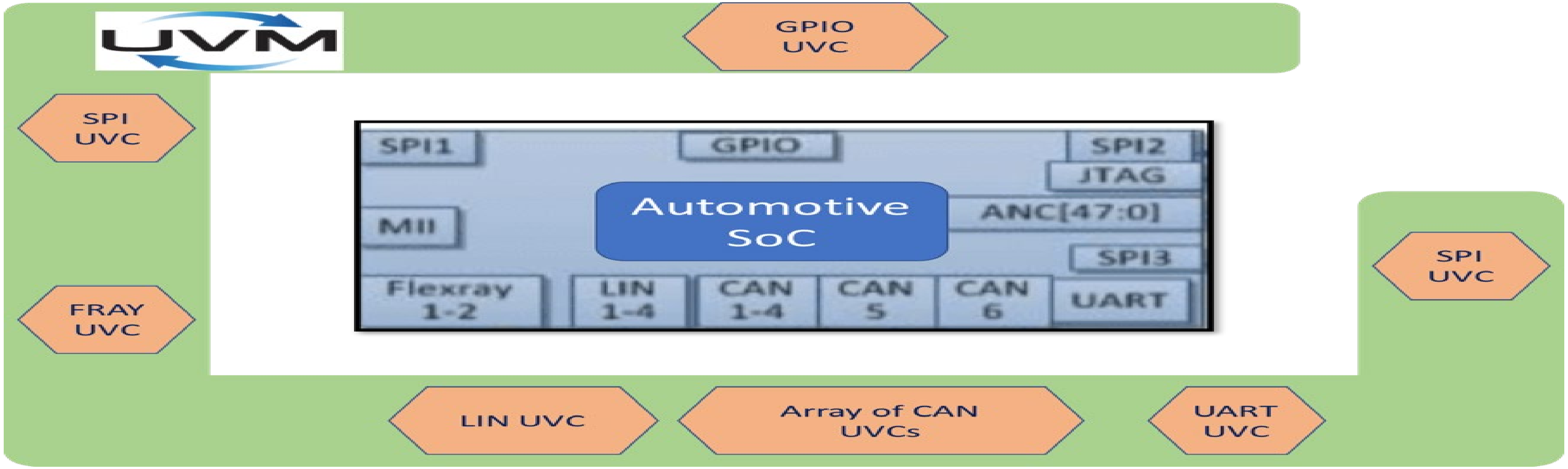
- Has multiple interfaces to control various part of the automotive





Automotive SoC Verif env with UVM

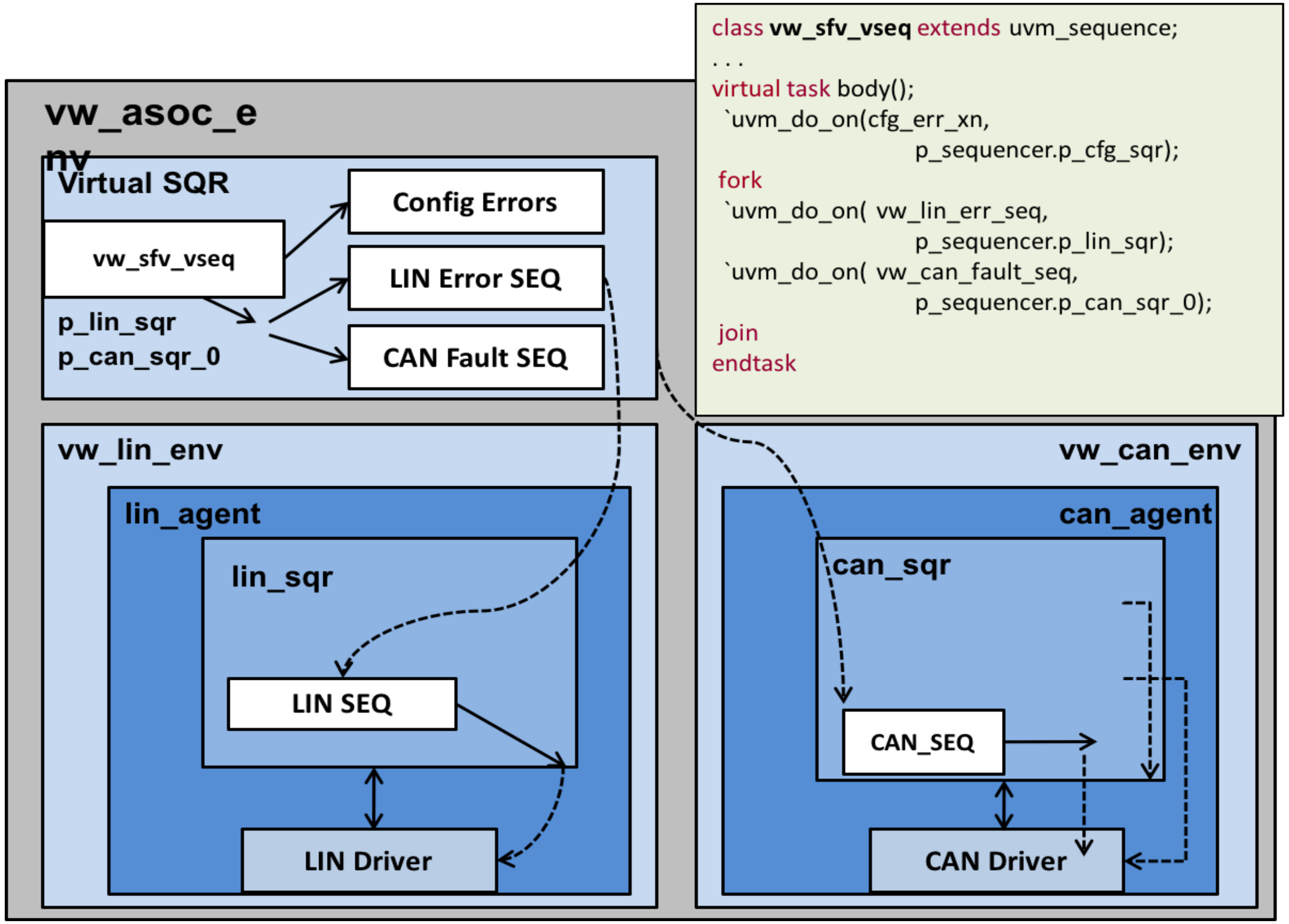
- Traditional UVM based flow with multiple UVCs, a verification environment can be built as shown in Figure below.



Verification Environment with multiple UVCs

Virtual sequences for Automotive SoC

- Multiple UVCs
- Virtual SEQ
  - Control individual interfaces
  - Error generation
  - Orchestrates various IP interactions
- Well-understood, well-deployed use model
- Adding random faults
  - Tricky!

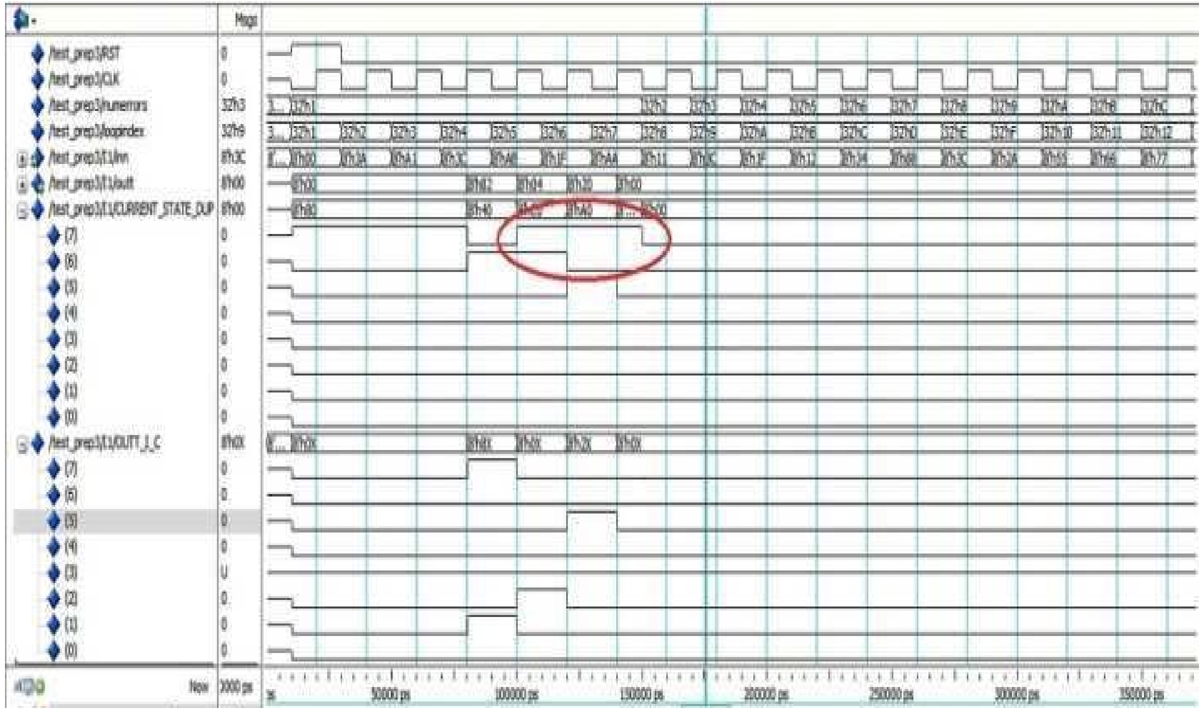


```
class vw_sfv_vseq extends uvm_sequence;
. . .
virtual task body();
 `uvm_do_on(cfg_err_xn,
                p_sequencer.p_cfg_sqr);
 fork
 `uvm_do_on( vw_lin_err_seq,
                p_sequencer.p_lin_sqr);
 `uvm_do_on( vw_can_fault_seq,
                p_sequencer.p_can_sqr_0);
 join
endtask
```

Virtual Sequence and Multiple Sub-sequence

Fault injection in UVM simulation

- Regular traffic (via UVM virtual SEQ)
- Faults == random values on select signals
- Typically spread across the design
  - Hard to decide upfront
  - Difficult to code as “SV Interface”
- Occasional occurrence
  - Not very frequent



Signal Access API in Go2UVM

```
class go2uvm_sig_access extends uvm_object;
  `uvm_object_utils(go2uvm_sig_access)

 extern static function void g2u_force (string sig_name,
   logic [`VW_G2U_SIG_MAX_W-1:0] sig_val,
   bit verbose = 1,
   bit is_vhdl_sig = 0);

 extern static function void g2u_deposit (string sig_name,
   logic [`VW_G2U_SIG_MAX_W-1:0] sig_val,
   bit verbose = 1,
   bit is_vhdl_sig = 0);

 extern static function void g2u_release(string sig_name,
   bit verbose = 1,
   bit is_vhdl_sig = 0);
endclass : go2uvm_sig_access
```

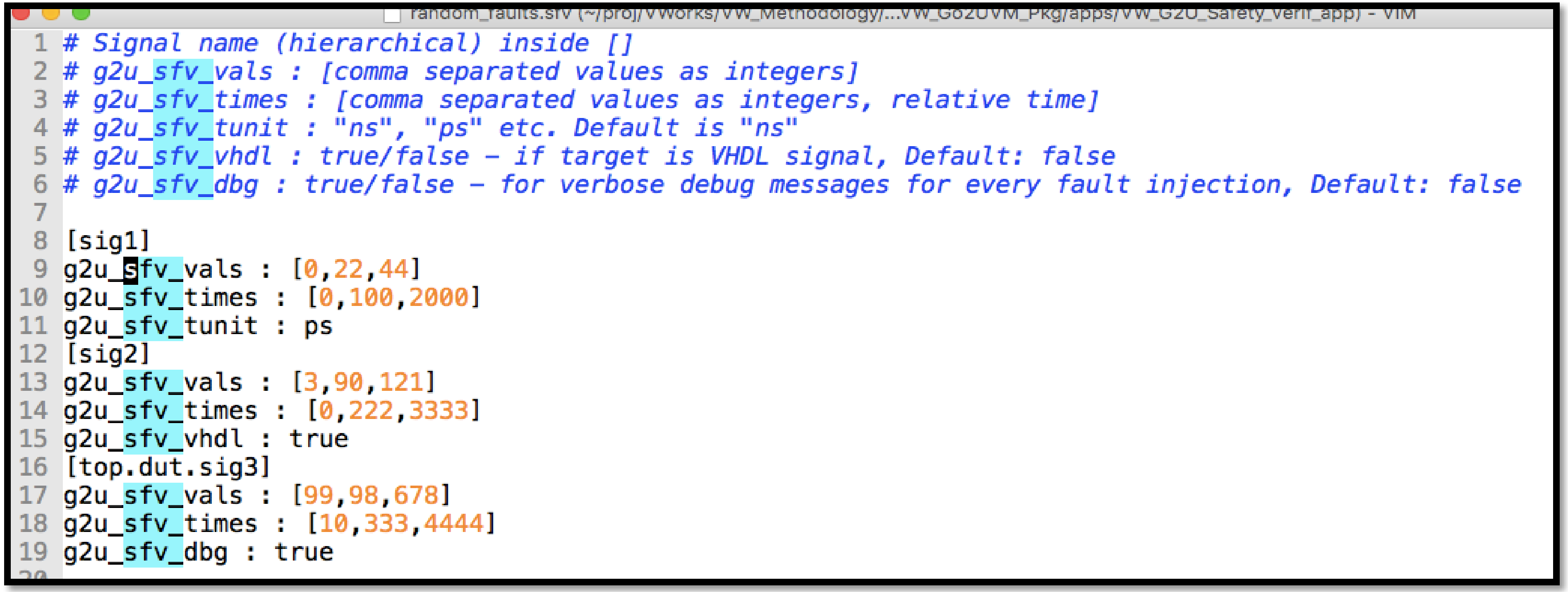
Signal access layer Go2UVM

- Go2UVM has a signal access layer
- Uses simulator’s force/release API
- Works across HDL boundary
- Handy technique for sideband drives:
  - PLL output
  - GLS reset etc.

```
class go2uvm_sig_access extends uvm_object;
  `uvm_object_utils(go2uvm_sig_access)

 extern static function void g2u_force (string sig_name,
   logic [`VW_G2U_SIG_MAX_W-1:0] sig_val,
   bit verbose = 1,
   bit is_vhdl_sig = 0);

 extern static function void g2u_deposit (
   string sig_name,
   logic [`VW_G2U_SIG_MAX_W-1:0] sig_val,
   bit verbose = 1,
   bit is_vhdl_sig = 0);

 extern static function void g2u_release(string sig_name,
   bit verbose = 1,
   bit is_vhdl_sig = 0);
endclass : go2uvm_sig_access
```

Fault injection with Go2UVM

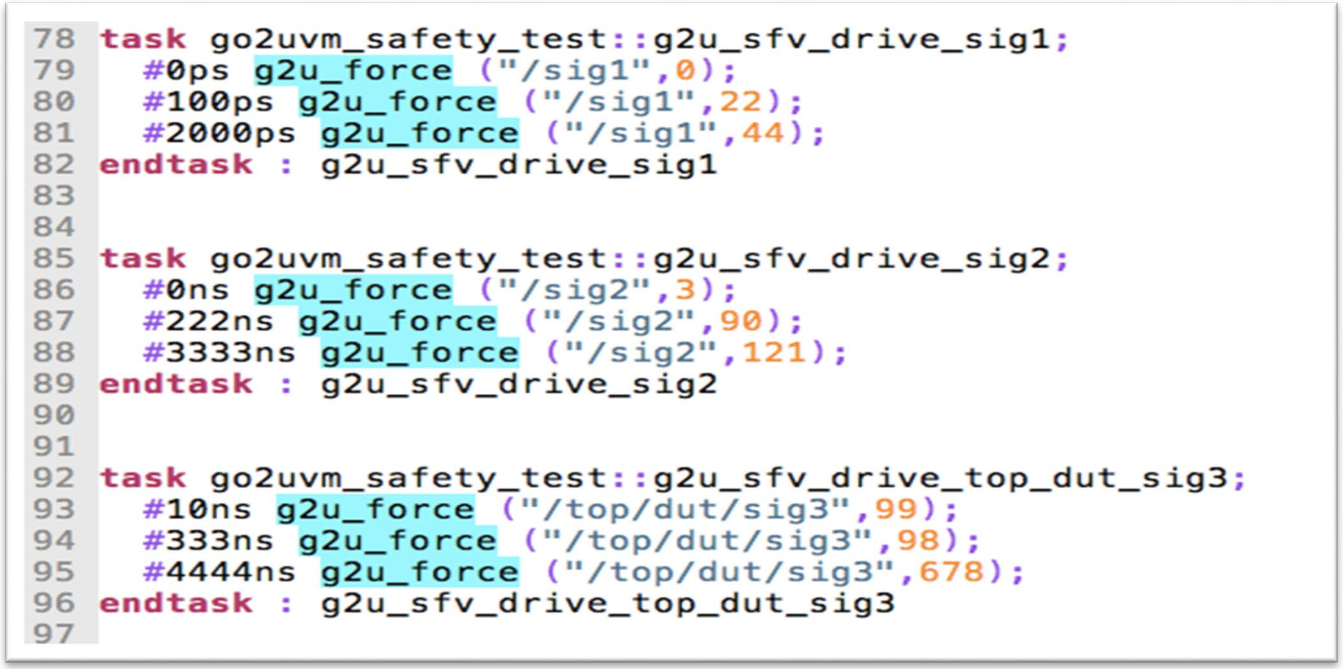
- Go2UVM’s signal access layer is extended for fault injection
- A new app named “SaFety Verification” (SFV) is developed


random_faults.sfv (~/proj/VWorks/VW_Methodology/...VW_Go2UVM_Pkg/apps/VW_G2U_Safety_verif_app) - VIM

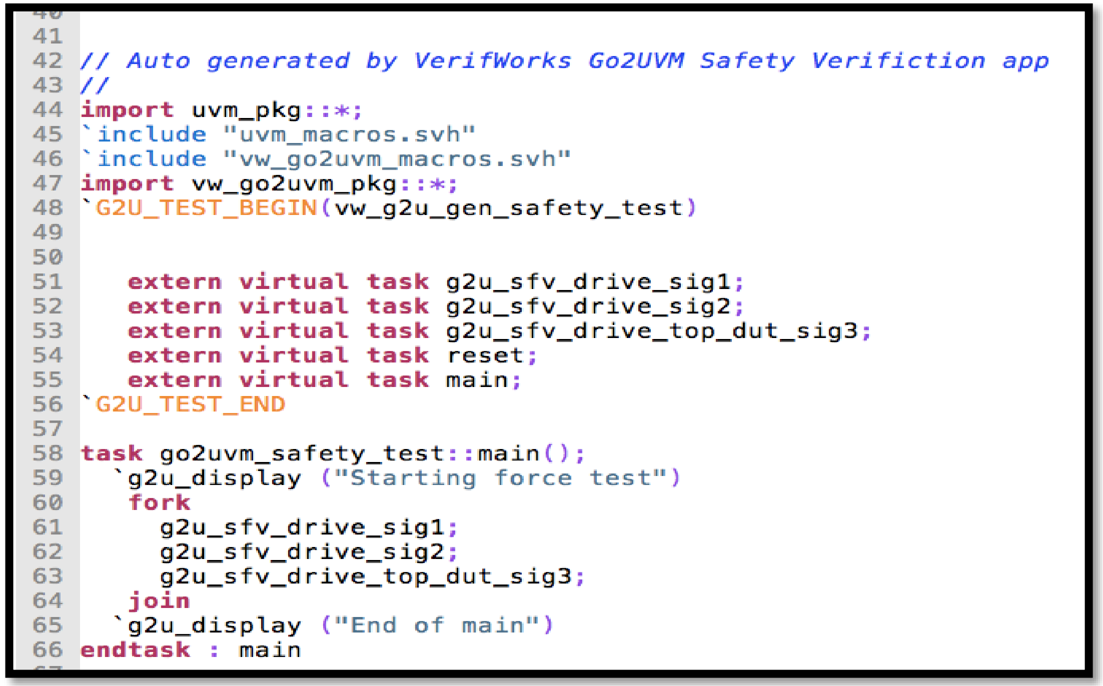
```
# Signal name (hierarchical) inside []
# g2u_sfv_vals : [comma separated values as integers]
# g2u_sfv_times : [comma separated values as integers, relative time]
# g2u_sfv_tunit : "ns", "ps" etc. Default is "ns"
# g2u_sfv_vhdl : true/false - if target is VHDL signal, Default: false
# g2u_sfv_dbg : true/false - for verbose debug messages for every fault injection, Default: false

[sig1]
g2u_sfv_vals : [0,22,44]
g2u_sfv_times : [0,100,2000]
g2u_sfv_tunit : ps
[sig2]
g2u_sfv_vals : [3,90,121]
g2u_sfv_times : [0,222,3333]
g2u_sfv_vhdl : true
[top.dut.sig3]
g2u_sfv_vals : [99,98,678]
g2u_sfv_times : [10,333,4444]
g2u_sfv_dbg : true
```

Using SFV in UVM test/sequence

- SFV → Go2UVM test via an app
- Can generate a SEQ as well – to be used in a virtual sequence



```
task go2uvm_safety_test::g2u_sfv_drive_sig1;
  #0ps g2u_force ("/sig1",0);
  #100ps g2u_force ("/sig1",22);
  #2000ps g2u_force ("/sig1",44);
endtask : g2u_sfv_drive_sig1


task go2uvm_safety_test::g2u_sfv_drive_sig2;
  #0ns g2u_force ("/sig2",3);
  #222ns g2u_force ("/sig2",90);
  #3333ns g2u_force ("/sig2",121);
endtask : g2u_sfv_drive_sig2


task go2uvm_safety_test::g2u_sfv_drive_top_dut_sig3;
  #10ns g2u_force ("/top/dut/sig3",99);
  #333ns g2u_force ("/top/dut/sig3",98);
  #4444ns g2u_force ("/top/dut/sig3",678);
endtask : g2u_sfv_drive_top_dut_sig3

```



```

// Auto generated by VerifWorks Go2UVM Safety Verifiction app
//
import uvm_pkg::*;
`include "uvm_macros.svh"
`include "vw_go2uvm_macros.svh"
import vw_go2uvm_pkg::*;
`G2U_TEST_BEGIN(vw_g2u_gen_safety_test)


   extern virtual task g2u_sfv_drive_sig1;
   extern virtual task g2u_sfv_drive_sig2;
   extern virtual task g2u_sfv_drive_top_dut_sig3;
   extern virtual task reset;
   extern virtual task main;
`G2U_TEST_END

task go2uvm_safety_test::main();
  `g2u_display ("Starting force test")
   fork
     g2u_sfv_drive_sig1;
     g2u_sfv_drive_sig2;
     g2u_sfv_drive_top_dut_sig3;
   join
  `g2u_display ("End of main")
endtask : main
```

Sample Go2UVM SFV Test

Log predictors in Go2UVM

- Fault injection leads to random failures
  - Expected to be caught by assertions, UVM monitors/scoreboards
  - Predicting such errors is key to ensure quality
- UVM has “reg_predcitor”
- Go2UVM adds a “log_predictor”
- Motivated by Mock frameworks in SW
  - Mockito, EasyMock etc.
  - SVUnit’s uvm_report_mock
- Ability to “predict” error/warning/info in LOG file

Go2UVM log predictor

- Go2UVM adds a base class: ***go2uvm_log_predictor***
- Has static method:
  - *go2uvm_log_predict (uvm_severity SEV, string ID, string msg, time start_t = 0, time end_t =0);*
- User can specify Severity, ID etc. and let the final test status account for these
- Can also control start & end time of the prediction

Conclusion

- Safety verification is challenging task
- Ivolves multitude of technologies & tools such as simulation, formal, mutation based etc.
- UVM is the most adopted verification methodology for simulations.
- However, for safety verification few additional features are needed on top of standard UVM
  - Go2UVM comp_access → Directed error injeciton
  - Go2UVM SFV → Safety Verification layer/app
  - Go2UVM Log predictor → Ability to build self-checking tests