Adopting UVM for safety verification requirements

- Tips & tricks packaged as a reusable library

Srinivasan Venkataramanan, Chief Technology Officer, VerifWorks, Bengaluru, India
(*srini@verifworks.com*)

Hema Kiran, ASIC Design Verification Engineer, VerifWorks, Bengaluru, India

(*hemakiran@verifworks.com*)

Guru, Lead Verification Engineer, VerifWorks

Satinder Paul Singh, Chief Technology Officer, Cogknit Semantics Pvt Ltd, Munich

(satinder.singh@cogknit.com)

***Abstract*—Safety in electronic circuits is taking a central stage with growing automotive and IoT systems. In the world of hardware designs using ASIC and/or FPGA, the Accellera UVM [1] has become the de-facto approach to verify the functionality of these designs at the pre-silicon stage. The complexity of UVM at times is overwhelming for smaller design teams especially in IoT domain (as the teams are very small usually). In this paper, we describe some of the easy-to-use layers we have developed on top of standard UVM such as random fault injection, easy error scenario creation, ability to predict log file contents based on injected errors etc.**

***Keywords*—UVM, Safety, VLSI, ASIC, FPGA, error injection, log prediction, Go2UVM [3]**

I. INTRODUCTION

A. SAFETY CRITICAL APPLICATIONS

Automotive is a safety critical application. The complexity of electronics systems for the automotive industry is increasing. These systems are characterized by having close interaction between software (SW), hardware (HW) and analog components. Automotive systems are a heterogeneous system, since it contains digital, analog low-voltage and high-voltage electronics, combined with software running on an embedded processor. Furthermore, automotive systems are safety critical systems, and as a consequence, verification of all related requirements is mandatory. Therefore, to design these safety-critical systems, the engineers more often require a dynamic and versatile functional verification environment of the hardware architecture. Few safety applications include

- Air bags
- Anti-lock Brake System
- Electronic stability control
- Adaptive cruise control
- Emergency braking assist
- Blind-spot monitoring
- Lane-departure warning
- Rear cross-traffic detection
- Pedestrian detection
- Traffic sign recognition

B. Verifying Safety Critical designs

Seamless reuse of functional and mixed-signal verification environments is key to accelerate the time to develop safety verification. Hence using standard verification approaches like UVM is the preferred approach even for simple block level verification by designers of these systems. Below are key requirements for easy verification of safety critical designs from a functional verification point of view.

- Fault injection at random points
- Reuse of the existing functional verification environment (with support for SystemVerilog, Universal Verification Methodology (UVM), and e).
- Simulation of the unaltered design under test (DUT).
- Support for multiple fault types, including single event upset (SEU), stuck-at-0/stuck-at-1, and single event.
- Quick creation of corner cases in simulation
- Log prediction to create self-checking error tests

Safety verification is a complex process with many different requirements. Usually, it requires a multitude of technologies and tools to thoroughly verify safety critical designs. We share some of our experience in achieving a partial set of such requirements using a Go2UVM layer on top of standard UVM. We would like to acknowledge that complete safety verification cannot be achieved using these techniques alone.

C. Universal Verification Methodology (UVM)

Universal Verification Methodology (UVM), as defined by the IEEE 1800.2, is getting adopted widely across ASIC design teams. UVM is also getting its flavor in SystemC [2] so that the Electronic System Level (ESL) community can move to UVM-based approach for verifying models. Given the standard way of information flow, well-defined test sequencing (in terms of phasing for instance), UVM is attractive to almost all electronics design teams.

Given the technical commonalities across many design verification tasks, one could imagine that some portions of UVM are applicable to these tasks, while some advanced UVM features may not be so appealing to them. In this paper, we present our experience in adopting UVM for safety verification and being able to deploy UVM via a convenience layer around the standard UVM named *Go2UVM*. *Go2UVM* is not a script to generate a set of files that users can fill-in, rather it is an OOP layer around standard UVM hiding all the glory details of UVM and providing the first-time UVM users an easy to use procedural interface to UVM. *Go2UVM* is open-sourced, sits on top of standard UVM and hence is 100% in-line with UVM philosophy of test creation. There are also several "apps" being developed to speed up the process of creating templates for *Go2UVM* to make the industry move to UVM the fastest way!

II. Standard UVM use model

A. Brief introduction to UVM

UVM is a methodology for the functional verification of digital hardware, primarily using simulation. The hardware or system to be verified would typically be described using Verilog, SystemVerilog, VHDL or SystemC at any appropriate abstraction level. This could be behavioral, register transfer level, or gate level. UVM is explicitly simulation-oriented, but UVM can also be used alongside assertion-based verification, hardware acceleration or emulation.

UVM test benches are more than traditional HDL test benches, which might wiggle a few pins on the design-under-test (DUT) and rely on the designer to inspect a waveform diagram to verify correct operation. UVM test benches are complete *verification environments* composed of reusable verification components, and used as part of a complete methodology of constrained random, coverage-driven, verification.

B. *Challenges with adopting UVM for smaller teams*

However, the full-fledged UVM can be perceived as an overkill for some specific tasks and in certain classes of designs. There are also smaller teams with no formal training on advanced, software centric approach being promoted by UVM. The other group of electronic designs includes FPGA designs with the majority of them using simpler, linear and procedural test bench styles. Also, FPGA teams often look for template creation tools as add-ons to their EDA tools to speed up their test bench creation process. There are also cases in ASIC design flow, wherein the test inputs come as a stream of structured data (such as DFT patterns) wherein teams are finding it difficult to leverage on well-defined UVM approach. Last but not the least, various university students across the globe are looking for a quick start into industry standard UVM and they do not have access to all the sophisticated training and hand-holding needed to get started. Given that the future DV engineers emerge from these universities, it is imperative for the industry to get the students started UVM as early as possible

C. *Adopting UVM for safety applications*

As noted earlier, standard UVM makes first-timers feel bit out-of-place. And with safety applications on the rise, many teams need additional layers on top of standard UVM to make some tasks easier. There are three key areas that we have adopted UVM to ease safety verification challenges.

- Random fault injection in a UVM simulation
- Flexible control of UVCs/VIPs to inject pre-defined errors
- Log predictors to control UVM error messaging mechanism during such error simulations

II. MAKING UVM EASY-TO-USE

A. *Introduction to Go2UVM*

In simple words, *Go2UVM* is an open-source, SystemVerilog package around Accellera UVM base class library. It provides a test layer around standard UVM to hide the common complexities such as:

- Phasing (*reset_phase*, *main_*phase etc.)
- Objection mechanism (A must in UVM to get even a simple stimulus through to the DUT)
- Multiple layers of components, that at times, smaller designs may not require
- Hierarchical component hook-ups via UVM's preferred *uvm_component :: new (string name, uvm_component parent*) pattern

An overview of various layers and apps associated with Go2UVM is shown below in Figure – 1.

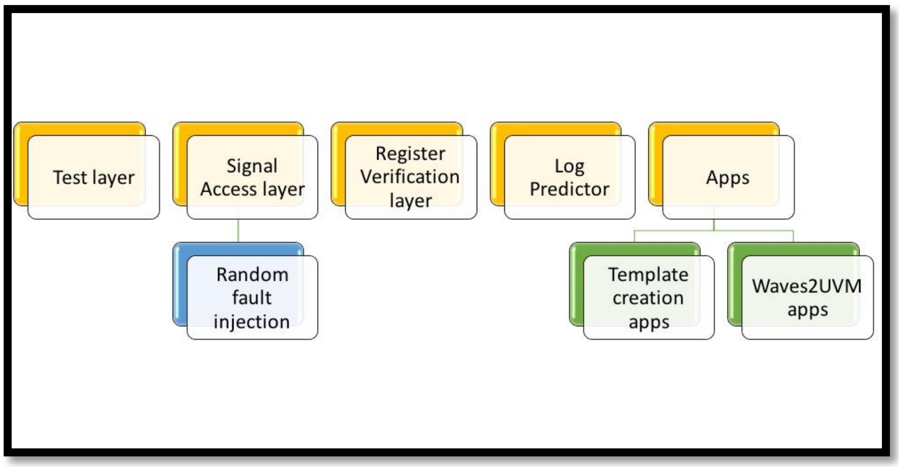


Figure-1 Various layers and apps in Go2UVM

B. *Test layer*

As mentioned earlier, *Go2UVM* is a SystemVerilog package around standard UVM. It extends the uvm_test base class as shown in UML diagram below:

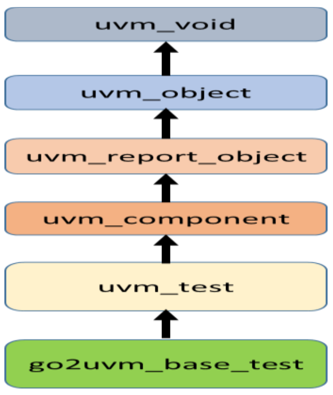
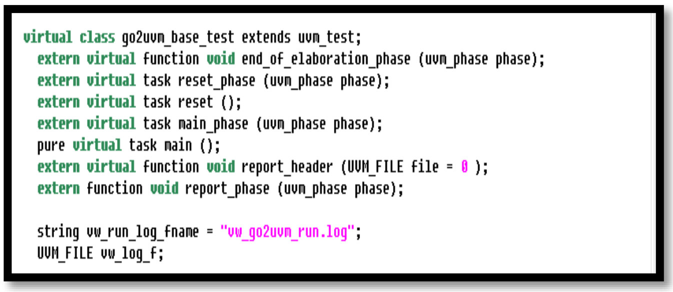


Figure-2 Go2UVM UML diagram

Following the general idea of standard object-oriented paradigm Go2UVM base test contains common code that needs to be repeated by every test/user. It is declared as *virtual class* indicating that it is an abstract class and shall be extended by the user with a concrete class. In UVM even to add a simple trace (a series of input signal wiggling is generally called a "trace"), the following needs to be done:

- Use *uvm_test* (and typically more UVM components, for now, focus on test)
- Hierarchically connect to the UVM framework via *parent* argument of the constructor
- Create a task that performs a reset of the given DUT (consider that all hardware designs typically require reset and the reset is a time-consuming task).
- Create a task that performs the intended DUT signal wiggling (more sophisticated transaction based approach is next step in UVM)
- Invoke the above 2 tasks to perform *reset* and the *main* functionality inside standard UVM phasing (So that UVM BCL can invoke these tasks at appropriate time in simulation)
- While performing any time-consuming action within UVM framework, make sure to raise and drop objections.

The above steps are done using a base class named *go2uvm_base_test*. The prototypes of relevant methods are shown in the code snippet below:

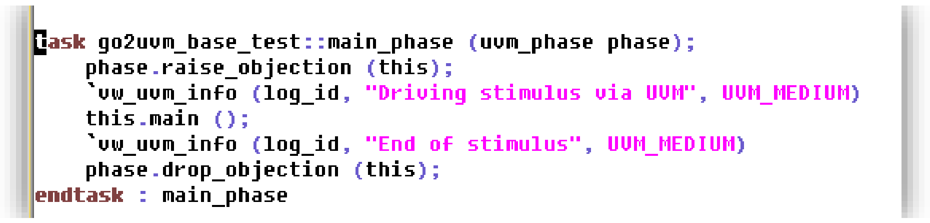


```
virtual class go2uvm_base_test extends uvm_test;
  extern virtual function void end_of_elaboration_phase (uvm_phase phase);
  extern virtual task reset_phase (uvm_phase phase);
  extern virtual task reset ();
  extern virtual task main_phase (uvm_phase phase);
  pure virtual task main ();
  extern virtual function void report_header (UVM_FILE file = 0 );
  extern function void report_phase (uvm_phase phase);

  string vw_run_log_fname = "vw_go2uvm_run.log";
  UVM_FILE vw_log_f;
```

Figure-3 Go2UVM base test code snippet

From an end user perspective (read it as UVM unaware engineer), this class adds 2 methods: *reset()* and *main()* – both of them are user extendable methods. The *go2uvm_base_test* invokes these methods inside standard UVM phasing with proper objection raising and dropping under-the-hood.

This is one of the very common mistakes a first-time UVM user gets and Go2UVM takes care of this. The other common mistake is to declare a task and not calling it. While UVM handles this cleverly with standard phrasing, the code to declare any phase is little more than what a first-timer would like it to be. It is augmented by the fact that all UVM phases take *uvm_phase phase* as argument and majority of them don't use this argument – this is confusing to many users indeed. In *Go2UVM* we simplify this by hiding the phases from user code and instead requiring them to fill-in two simple tasks *reset & main* and the base class invokes them inside the correct phase as shown below:



```
task go2uvm_base_test::main_phase (uvm_phase phase);
    phase.raise_objection (this);
    `vw_uvm_info (log_id, "Driving stimulus via UVM", UVM_MEDIUM)
    this.main ();
    `vw_uvm_info (log_id, "End of stimulus", UVM_MEDIUM)
    phase.drop_objection (this);
endtask : main_phase
```

Figure-4 Go2UVM test main phase

What if a user forgets to fill-in ***task main()*** in derived class? Well, this is why *Go2UVM* package declares this method as *pure virtual. T*he standard definition:

A ***pure virtual function*** *or* ***pure virtual method*** *is a virtual function that is required to be implemented by a concrete, derived class*

Hence with *Go2UVM* if one forgets to implement *task main()* in derived class, a compiler would flag this almost instantaneously:

```
../unit_test_src/vw_ahb_lite_cip_go2uvm_test.svi : (22, 27): Cannot declare class
vw_ahb_lite_cip_test as non abstract class due to not implemented pure virtual methods:main()
```

Figure-5 Go2UVM test Compilation Error

It takes care of the standard UVM requirement of component hierarchy hook-up via *name* & *parent* under the hood so that first-time users do not need to bother about it. More details with the full source code can be found at Go2UVM on GitHub.

The idea is to provide the fastest way for an engineer to get started with UVM. Since it is 100% IEEE 1800 (SystemVerilog) and IEEE 1800.2 (UVM) compatible, users can easily start with *Go2UVM* and move to a full-fledged UVM environment, as they get mature with the technology. *Go2UVM* is well tested on all major simulators from popular EDA vendors.

C. Arbitrary Signal Access Layer

UVM defines a structured approach to signal accesses and recommends to restrict them to driver and monitor layer typically. While this is very useful for reusable pieces of verification such as the agent, environment etc. at test layer occasionally needs to access a set of DUT signals at-ease. This could be to test certain corner cases (such as setting a large FIFO with almost-empty level quickly), the complex interaction of signals etc. This is also a preferred approach for RTL designers trying to quickly augment verification and/or trying to verify their designs for a set of "known-worry states" before handing their RTL to full-fledged Design-Verification (DV) team. Strictly speaking, UVM does not prevent this explicitly via a rule/guideline. Such tests are usually not intended for reuse and are usually done to avoid long simulation cycles etc.

While the above requirements are quite common across design teams, SystemVerilog language (IEEE 1800-2015) has certain restrictions on hierarchical accesses, especially when done from classes enclosed inside a package. Hence if a UVM sequence (made available as part of a SystemVerilog package) wants to access an arbitrary set of signals inside the DUT, it is prohibited as per the language semantics. There are also scenarios wherein the design is in mixed HDLs such as Verilog, SystemVerilog & VHDL.

A robust workaround for this problem is to use VPI/VHPI and special APIs provided by EDA vendors. Some of the common APIs for this purpose are provided below in Table.

Vendor	**API name(s)**
Aldec	$signal_agent, $force
Cadence	$nc_force, $nc_release
Mentor	$signal_force, $signa_spy
Synopsys	$hdl_xmr

Table-1 Vendor and API Names

Detailed usage of each API is beyond the scope of this paper and readers are encouraged to refer to their tool manuals for the same. In general, these APIs are harder (than what an average engineer would look for) to use. Also, many a times users prefer to keep their code vendor neutral as much as possible.

Go2UVM library has a set of handy APIs and a base class for this purpose. It provides a base class named *go2uvm_sig_access* that has necessary APIs to wrap the vendor specific APIs listed in Table above. These APIs inside *go2uvm_sig_access* are declared static and hence can be accessed without having to create a handle and an object. This class contains support for various vendor APIs and manages to switch between tools via built-in text macros (such as `ifdef VCS, `ifdef INCA etc.). This keeps user code portable across tools, yet provides the necessary APIs to allow arbitrary signal access from anywhere in the testbench code (UVM test, UVM sequence etc.).
To make things even more convenient for a typical RTL engineer, a convenience API is provided as shown in Table-2 below.

API	**Description**	**Example**
g2u_force	Forces given value on a target signal (Overrides any other existing driver for that signal)	g2u_force ("/sprot_go2uvm/sprot_0/byte_val", 22);
g2u_release	Releases an existing force on a target signal	g2u_release ("/sprot_go2uvm/sprot_0/byte_val");

g2u_deposit	Deposits given value on a target signal (Without overriding any existing drivers)	g2u_deposit ("/sprot_go2uvm/sprot_0/byte_val", 22);

Table-2 Go2UVM APIs

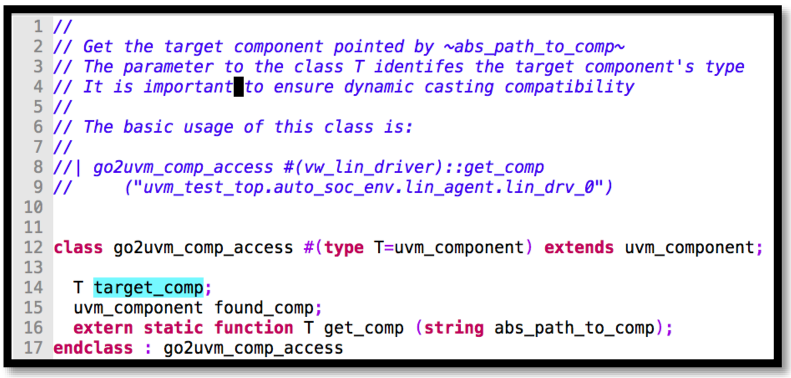
III. Deploying Go2UVM in Safety Verification

Safety verification is a complex process with many different requirements. Usually, it requires a multitude of technologies and tools to thoroughly verify safety critical designs. We share some of our experience in achieving a partial set of such requirements using a Go2UVM layer on top of standard UVM. We would like to acknowledge that complete safety verification cannot be achieved using these techniques alone.

A. Directed Error injection

Most of the safety critical designs are error tolerant in the sense that they should be able to detect a set of well-defined error scenarios and be able to recover from such error scenarios as soon as the normal scenarios reappear. This is critical for the verification process to ensure this sequence of normal → Error → Normal is tested (and other similar sequences). In a typical IoT system, there are several peripherals all connected via a system bus and interact. Individual peripherals support various error scenarios and have individual recovery mechanisms built-in to the design. From a verification perspective (using UVM), each of these peripherals has a UVC and the individual UVCs support error injection through configurations. In a well-architected UVC, such error injection shall be supported per transaction (or frame) than just being a one-time configuration. However, reaching out to each of these UVC agent/drivers at times becomes a challenge from a typical UVM sequence. In UVM, components are hierarchically connected with an unambiguous hierarchical name to each instance of every component (agent/sequencer/driver etc.). However, UVM sequences are not hierarchy aware. Hence injecting errors from a virtual sequence on various peripherals becomes a challenge.

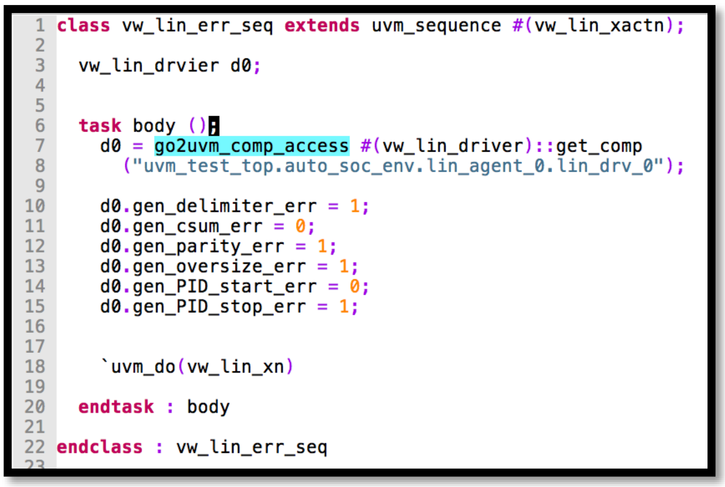
We used UVM's hierarchy APIs to solve this problem. Go2UVM provides a convenience layer to access any component instance through its hierarchical path. It is a parameterized class with the parameter indicating the target component type. Basic usage and the class API is shown in Figure-6 below.



```
//
// 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
```

Figure-6 Go2UVM component access layer

Using the ***go2uvm_comp_access #(comp)::get_comp*** API, given the complete hierarchical path to a component, we can get a handle to that component during the simulation. Once a handle to the target component is found, its configurations can be changed to inject errors from a UVM sequence. Figure-7 below shows a typical use case of this API for a LIN peripheral UVC error generation scenario.



```
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
```

Figure-7 Go2UVM component access API use case

B. *Random fault injection*

One of the common requirements in safety verification is an injection of random faults across critical parts of the design and observe the behavior under such scenarios. The word "fault" is loosely used in this context and does not refer to the classical fault-model based testability aspects (though there are some similarities between them). In a typical UVM framework, this can be interpreted as "random value injection" on "arbitrary signals". In a more stringent random fault injection for safety verification, mutation based techniques are used along with formal verification techniques to choose relevant parts to inject the fault ([4], [5]).

Consider a sample automotive SoC as shown in Figure-8 below.

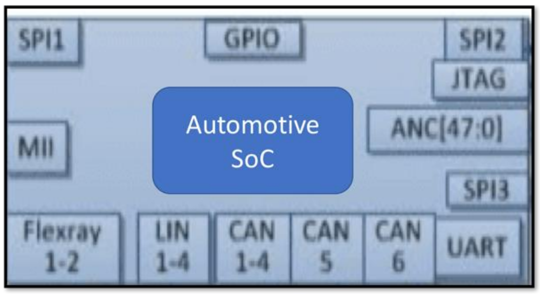


Figure-8 Sample Automotive SOC

At a pure UVM based simulation level, we could narrow it down to injecting random values for a set of safety critical signals inside the design. With traditional UVM based flow with multiple UVCs, a verification environment can be built as shown in Figure-9 below.

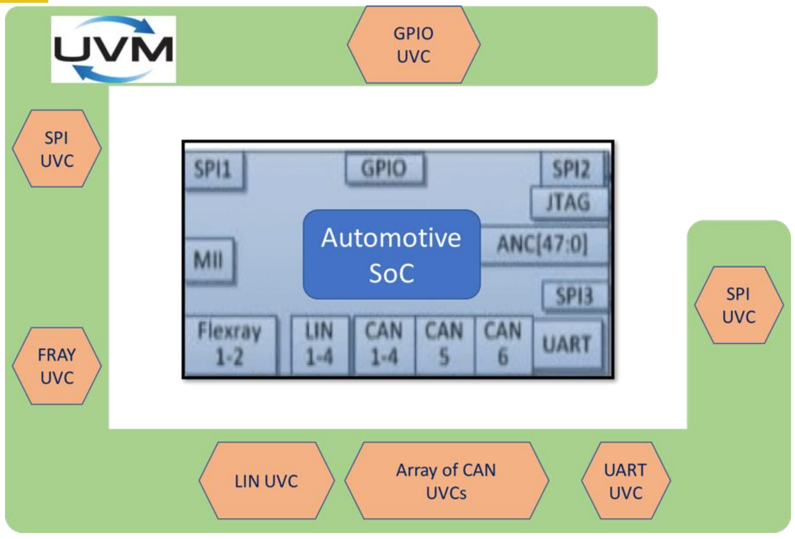
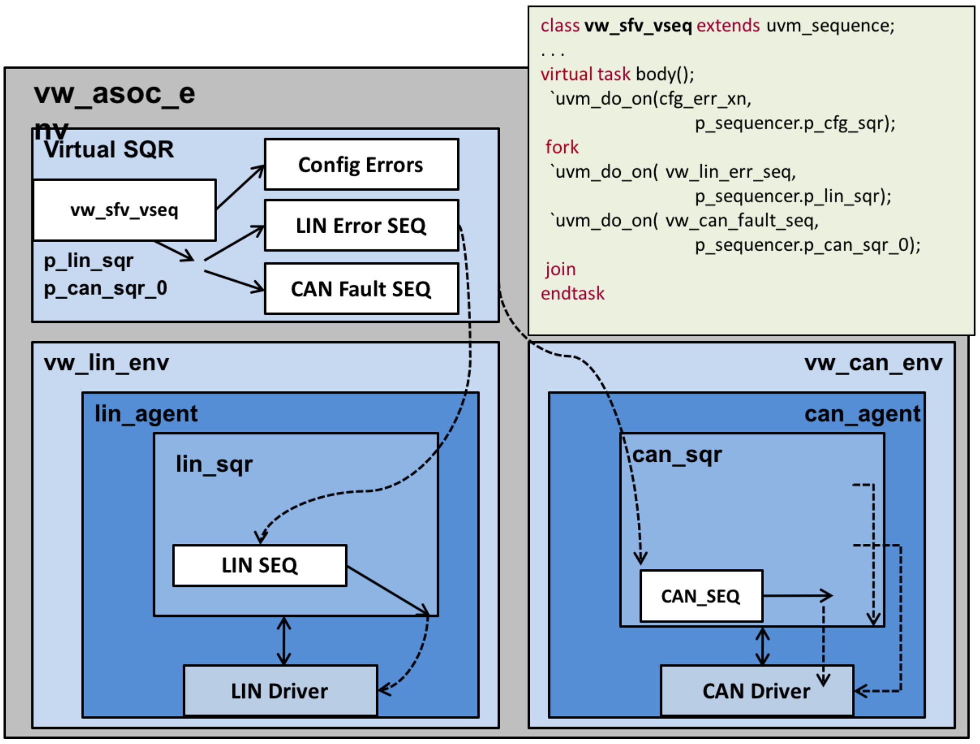


Figure-9 Verification Environment with multiple UVCs

A typical random fault spreads across interfaces and deep inside the design. To stick to standard UVM and still be able to inject these errors one has to create a virtual sequence and multiple sub-sequences and spawn them on corresponding sequencers as shown in Figure-10 below.



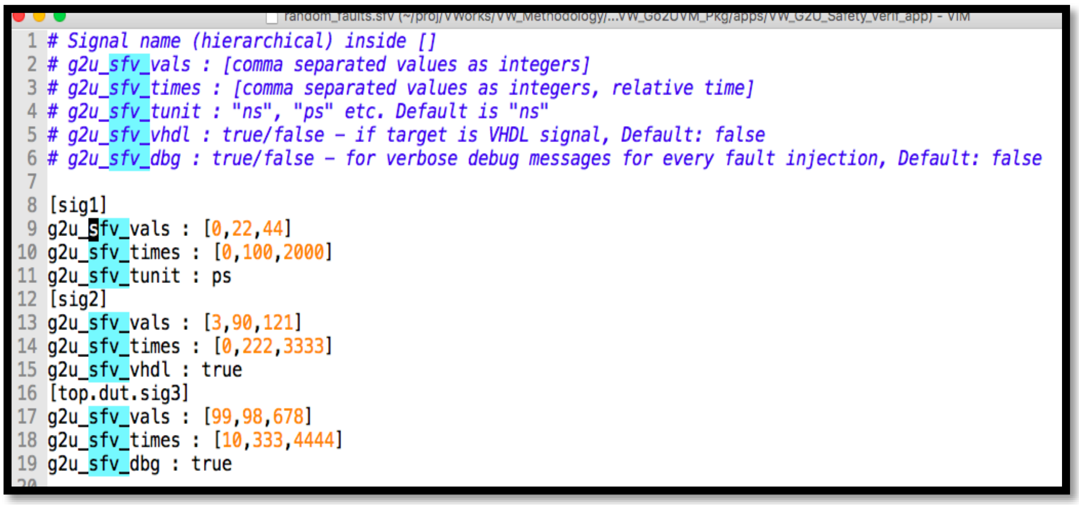
```
class vw_sfv_vseq extends uvm_sequence;
. . .
virtual task body();
 `uvm_do_on(cfg_err_xn,
                    p_sequencer.p_cfg_sqr);
 fork
 `uvm_do_on( vw_lin_err_seq,
                    p_sequencer.p_lin_sqr);
 `uvm_do_on( vw_can_fault_seq,
                    p_sequencer.p_can_sqr_0);
 join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

Figure-10 Virtual Sequence and Multiple Sub-sequences

The above is just an indicative setup and can get very complex should the number of fault injected signals were to rise. Also given that such faults are random in nature, one has to create a flexible framework to be able to somehow randomly generate such virtual sequences and hope that the corresponding driver indeed has access to those signals of interest. A typical UVM setup accesses only primary input, outputs signals and not the internal signals/state elements.

In our setup, this is achieved by building on the basic Go2UVM's signal access layer (see section III-B above). A plain text file with fault information is created. This file is referred to as SFV (**S**a**F**ety **V**erification) file. The user is expected to provide a set of signals that are of interest to random fault injection.

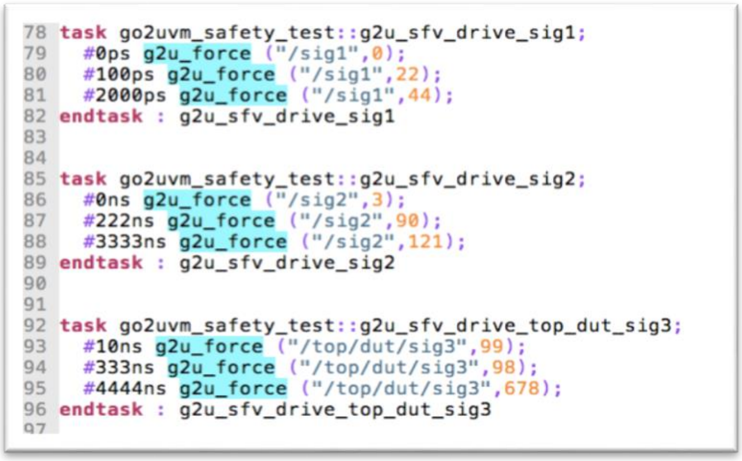
A sample SFV file is shown below in Figure-11.


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
```

Figure-11 Sample SFV File

A Python app developed on top of Go2UVM converts SFV file to a typical Go2UVM test as shown in Figure-12 below.

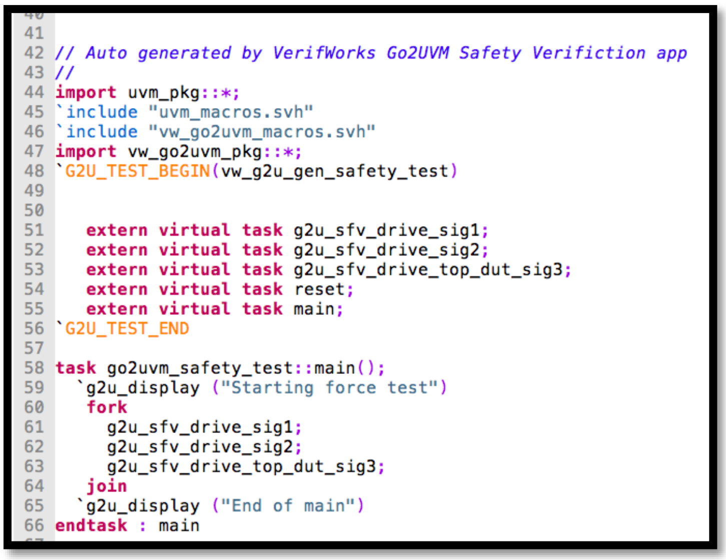


```
task go2uvm_safety_test::g2u_sfv_drive_sig1;
  #0ps g2u_force ("/sig1",0);
  #100ps g2u_force ("/sig1",22);
  #2000ps g2u_force ("/sig1",44);
endtask : g2u_sfv_drive_sig1


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
```

Figure-12 Sample Go2UVM SFV Test

As can be seen in Figure-12 above, a separate task is generated per safety critical signal listed in the input SFV file. The timescale can also be modified as per design requirement in the SFV file.

Go2UVM test's entry point is a task named ***main()*** as explained in Section III-A. This SFV app injects faults on each identified signal in parallel with ***main()*** task spawning all these tasks as shown in Figure-13 below.



```
// 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
```

Figure-13 Go2UVM *main()* task for SFV app

The key advantage of this Go2UVM layer on top of standard UVM is the ease-of-use and quick fault injection for a given set of arbitrary signals. In a traditional, strict UVM flow, this would require multiple interfaces, drivers, sequencers, sequences and virtual sequences. Instead with Go2UVM, this becomes a single component that injects faults at arbitrary design locations.

C. Using log predictors

UVM provides a way to consistently log/report messages during the simulation. Each message can be classified with a severity and is associated with an ID. UVM also provides a set of APIs to control and configure messages based on severity, ID etc. In a typical UVM test run, at the end of the simulation, a summary of these messages is printed. It is grouped under two headings viz:

1. Severity based summary
2. ID based summary

The severity based summary is intended to serve as an indication of a successful test run – i.e. a test is considered a PASS if there are no ERRORs and FATALs, and is considered a FAILURE otherwise. In this context, we refer to UVM_ERROR and UVM_FATAL severities specifically.

In a safety critical verification flow, the above interpretation needs an extension. It is so because, by definition, in a safety verification error injection is included and is a MUST (unlike in many other classes of designs wherein error injection is considered as negative testing and is not as critical as the positive tests). Also in some of the safety verification flows, requirement tracing is a must and some of the requirements are on the error injection, recovery etc. So, in a safety verification with UVM the following interpretation is adopted:

- A test is considered PASS:
  - If there are NO ERRORs (and FATALs) <or>
  - If the errors are as expected (as in error injection tests)
- A test is considered FAIL:
  - If there are ERRORs (and FATALs) <or>
  - If there are NO ERRORs, but it is an error injection test

Go2UVM provides a special base class named ***go2uvm_log_predcitor***. It provides relevant APIs for users to be able to specify what errors are to be predicted/expected. Some of the features of this API are listed in Table-3 below:

API	**Description**	**Features**
g2u_log_predict	Predicts a message from UVM reporting with given ID, Message and SEVERITY	Wildcard support Time window selection

Table-3 go2uvm_log_predict API Features

Consider AHB protocol requirement [6] on *htrans* signal as shown in Figure-14 below:

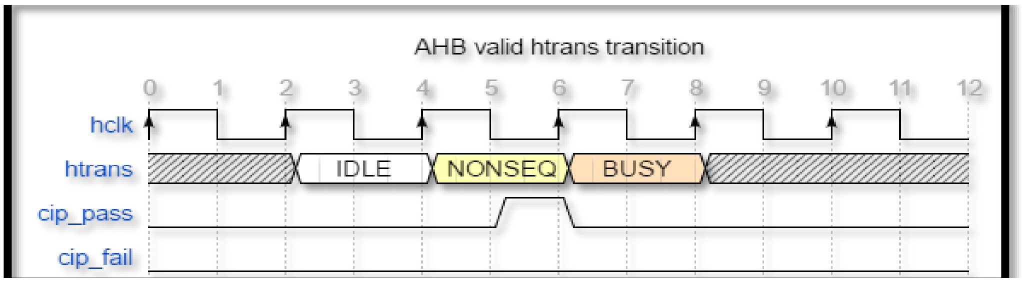


Figure-14 AHB htrans requirement

In an error injection scenario, we might inject an error as shown in Figure-15 below:

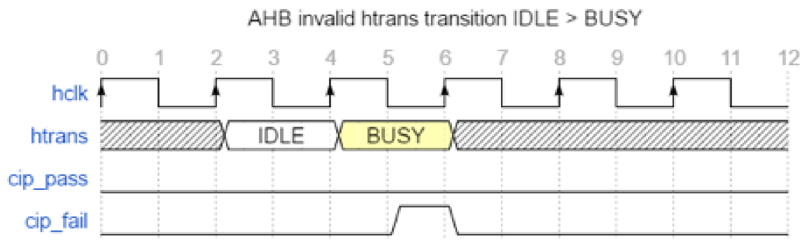
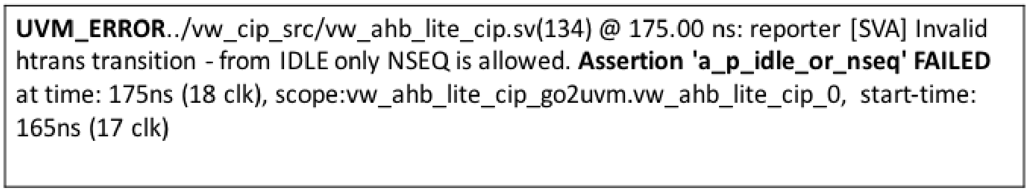


Figure-15 AHB htrans invalid transition (error injection test)

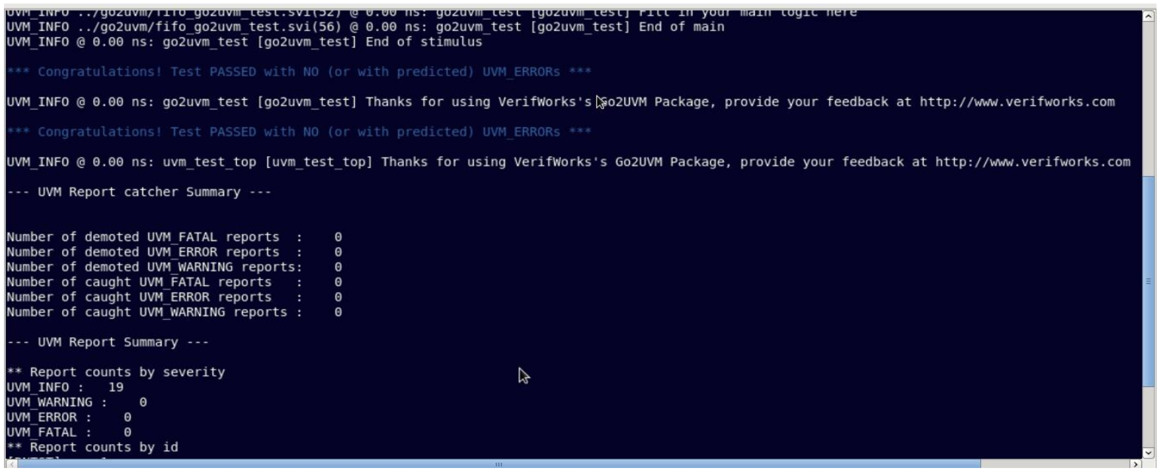
At clock tick 6, we would expect the assertion to fire. If we run the trace as-is, it reports a UVM_ERROR like shown below:



UVM_ERROR../vw_cip_src/vw_ahb_lite_cip.sv(134) @ 175.00 ns: reporter [SVA] Invalid htrans transition - from IDLE only NSEQ is allowed. **Assertion 'a_p_idle_or_nseq' FAILED** at time: 175ns (18 clk), scope:vw_ahb_lite_cip_go2uvm.vw_ahb_lite_cip_0, start-time: 165ns (17 clk)

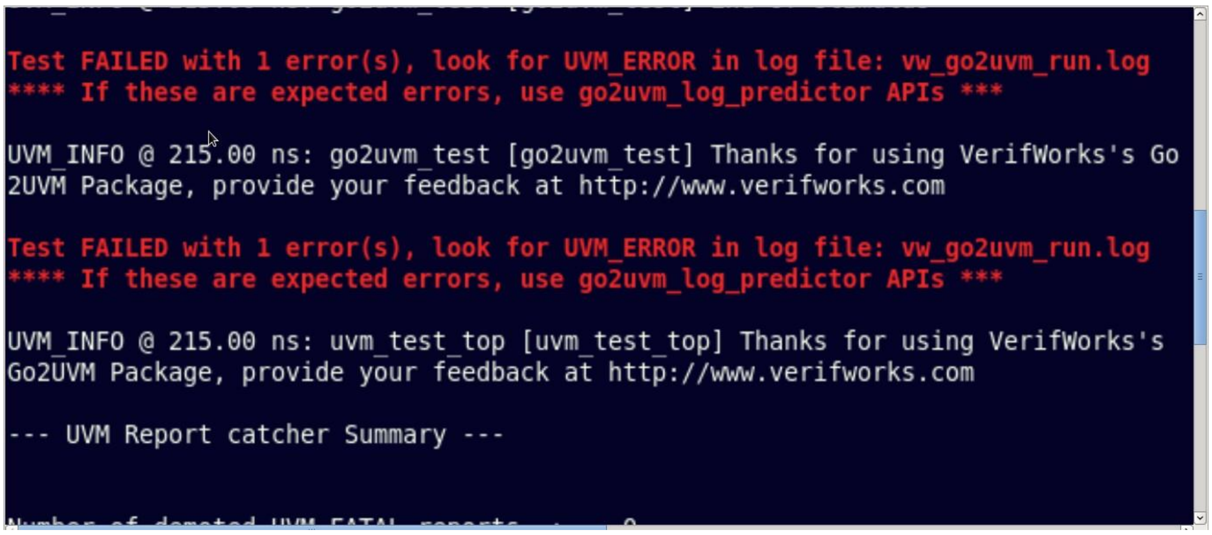
Figure-16 Sample UVM ERROR due to FAIL trace (error injection test)

Without the ***go2uvm_log_predictor*** the above error injection test shall be classified as FAILURE. However, that would be a false alarm as the test indeed injected this specific error. Hence the absence of such UVM_ERROR in this simulation run should be considered a FAILURE.



```
UVM_INFO ../go2uvm/fifo_go2uvm_test.svl(52) @ 0.00 ns: go2uvm_test [go2uvm_test] Fill in your main logic here
UVM_INFO ../go2uvm/fifo_go2uvm_test.svl(56) @ 0.00 ns: go2uvm_test [go2uvm_test] End of main
UVM_INFO @ 0.00 ns: go2uvm_test [go2uvm_test] End of stimulus

*** Congratulations! Test PASSED with NO (or with predicted) UVM_ERRORs ***

UVM_INFO @ 0.00 ns: go2uvm_test [go2uvm_test] Thanks for using VerifWorks's Go2UVM Package, provide your feedback at http://www.verifworks.com

*** Congratulations! Test PASSED with NO (or with predicted) UVM_ERRORs ***

UVM_INFO @ 0.00 ns: uvm_test_top [uvm_test_top] Thanks for using VerifWorks's Go2UVM Package, provide your feedback at http://www.verifworks.com

--- UVM Report catcher Summary ---


Number of demoted UVM_FATAL reports  :    0
Number of demoted UVM_ERROR reports  :    0
Number of demoted UVM_WARNING reports:    0
Number of caught UVM_FATAL reports   :    0
Number of caught UVM_ERROR reports   :    0
Number of caught UVM_WARNING reports :    0

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   19
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
```

Figure-17 Sample Pass Trace



```
Test FAILED with 1 error(s), look for UVM_ERROR in log file: vw_go2uvm_run.log
**** If these are expected errors, use go2uvm_log_predictor APIs ***

UVM_INFO @ 215.00 ns: go2uvm_test [go2uvm_test] Thanks for using VerifWorks's Go
2UVM Package, provide your feedback at http://www.verifworks.com

Test FAILED with 1 error(s), look for UVM_ERROR in log file: vw_go2uvm_run.log
**** If these are expected errors, use go2uvm_log_predictor APIs ***

UVM_INFO @ 215.00 ns: uvm_test_top [uvm_test_top] Thanks for using VerifWorks's
Go2UVM Package, provide your feedback at http://www.verifworks.com

--- UVM Report catcher Summary ---
```

Figure-18 Sample Fail Trace

IV. Conclusion

Safety verification is a challenging task. It involves a multitude of technologies and 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. In this paper, we shared our experience of using Go2UVM for safety verification challenges in a simulation context.

References

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

[2] UVM SystemC: http://accellera.org/activities/working-groups/systemc-verification

[3] *Go2UVM*: http://www.go2uvm.org

[4] SystemVerilog Assertions Handbook, http://verifnews.org/publications/books/

[5] VerifWorks: http://www.verifworks.com

[6] ARM AMBA Specifications: https://www.arm.com/products/system-ip/amba-specifications.php