2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Qualification of a Verification IP under Requirement Based Verification standards

An approach to the verification of the verification

Francois Cerisier, AEDVICES Consulting, Grenoble, France, francois.cerisier@aedvices.com
Adrien Carmagnat, AEDVICES Consulting, Grenoble, France, adrien.carmagnat@aedvices.com
Alessandro Basili, Melexis, Bevaix, Switzerland, abs@melexis.com
Gilles Curchod, Melexis, Bevaix, Switzerland, gcr@melexis.com

***Abstract*—As metric driven verification has become the de-facto methodology used in the micro-electronics industry to verify consumer application designs, more and more safety related industries are looking into integrating UVM into their safety requirement standards such as DO254, ISO26262, IEC 61508 or similar. However, these industries face the challenge of making "random" generation (even though constrained) compatible with requirement traceability. Further to this, safety critical standards demand that the verification environment (tools, bench, …) is qualified. The verification IPs (VIP) as part of the verification environment therefore need to be qualified so that we ensure the VIPs are not letting protocol violations escaping the assertions nor protocol checkers. This verification of the Verification IPs also requires applying the traceable verification techniques to the VIP itself.**

This paper presents an approach taken to verify a VIP for the automotive SENT protocol targeted for automotive applications under ISO26262 standards. We describe the missing part of UVM for VIP qualification, the report library we developed in order to expect UVM_ERROR under error conditions and compare regression results with and without using this library.

Keywords—Metrics Driven Verification, Requirement Based Verification, Assertions, Coverage, UVM, VIP, Qualification, ISO26262

I. INTRODUCTION

While metrics driven verification methodologies, today mainly supported by UVM, have proven to be efficient in finding hard-to-find bugs, it remains mostly untrusted by safety-critical standards. There are several reasons for this, starting from the industry background, the lack of understanding of what "constrained random" really is, to the trust that safety engineers have in the verification tools, the VIPs and in the testbenches.

For safety-critical standards, trust is built through qualification processes. These processes are mostly defined in the safety standards as testing or qualifying the verification tools[1][2][3] and these tests should remain traceable in regards to the item to be verified up to the higher level requirement.

Somehow, any conscientious verification engineer performs some kind of "untraceable" qualification by making sure an assertion fires an error upon an unexpected behavior. This is however mostly done during live development by changing a line of code to make the error to occur, and then reverting to the working code.

About 15 years ago, the start-up Certess (since acquired by Synopsys), has introduced the mutation testing with Certitude©[4] to the EDA industry to provide automatic qualification of the verification environment. This mutation testing approach proves to be effective in finding areas of the design that are not properly verified. However, if this is a valid approach to qualify the verification of a given design[5], this gives little indication on the quality of the VIP itself, nor on the validity of each single assertion in regards to the protocol and independently of any design.

As we are considering the VIP to be tested, a potential approach would be to apply mutation testing to the VIP itself. But this will lead to a few issues that need to be resolved:

- Mutation testing has been developed for the qualification of the verification. The mutations are therefore applied to design constructs (modules, processes, signals), and need to be adapted to a UVM verification IP, implemented with SystemVerilog code, functions, object-oriented classes and assertions.
- From the nature of automatic mutation testing, it is very difficult to map the result to a requirement-based approach, in particular, when we think of the traceability of the qualification of each assertion in regard to the protocol and its related requirements.
- Mutations are good to highlight missing checkers. It is quite hard to understand how good they are terms of checking the quality of an assertion and its limits.

VIPs, as part of the verification toolset, therefore need to be independently qualified. On this matter, safety standards such as ISO 26262 and DO254 define the verification tools as being tools that can fail to detect an existing error in the designs[2][3] and special care should be taking in this qualification.

This qualification of the VIPs therefore needs to be performed in an organized and strict approach, so that each qualification test can be reported to a requirement of the protocol.

There is a fear though, that this becomes endless. If we verify the VIP, then do we need to verify the verification of the VIP? Furthermore, it is well established that VIP strength in finding design bugs comes to their ability to generate thousands or millions of transactions with random attributes to cover hard to find design corner cases. Do we need do the same to qualify a VIP then?

It appears though that we can concentrate this qualification activity on a limited number of points.

II. QUALIFICATION APPROACH

A. Verification Plan of the Verification IP

When we think of the qualification of a VIP, we can think of the VIP as our design-under-test (DUT). The VIP becomes our DUT and to some extends the qualification activity becomes the verification of this DUT (the VIP).

Any VIP should provide the high-level features:

- Ability to control and generate stimulus for the given protocol
- Ability to report transaction/protocol coverage
- Ability to check the protocol (mostly states, transitions, timings)

In a strict qualification process, we would need to qualify each of these aspects in a traceable process.

Logically, we may think that we need to verify all aspects of the operational modes in order to ensure that the VIP has the ability to cover the full range of the protocol and that no assertions are triggered within this functional range. Although debatable, we could however claim that the use of the verification IP in the verification of a design validates its ability to generate the required stimulus. As safety standards require tools to be qualified on a per project basis, this last point can be considered valid[3].

The protocol violation conditions, on their side, are really the key parts that are required to be properly qualified. Our goal is therefore to verify that any protocol deviation actually fires an assertion and/or a UVM_ERROR. The protocol checkers then become our features to be verified.

However, taking the specification of the VIP as an entry point for the verification plan development cannot be the right approach. For instance, if assertions are missing and are neither implemented nor specified, it is likely that they will be missed by the qualification analysis while still making a good qualification result. Hence the starting point of this analysis should be the protocol specification itself (in our case the SENT protocol specification – JAE 2716™ used for automotive sensor data transmission to the electronic control unit – ECU)[6][7].

The analysis should then concentrate on two aspects:

- The operational / functional mode of the protocol, in order to eliminate false negative checkers
- The protocol violation conditions, in order to eliminate false positive checkers

To illustrate, let’s take some examples from the SENT protocol specification.

As a very a quick introduction, this protocol provides means to transmit automotive sensor data on a 1-line communication channel as a series of pulses, where the duration of each pulse (measured between falling edges) represent actual data being transmitted (as a PWM protocol). A SENT protocol frame consists of:

- A calibration pulse (defined as a pulse of 56 time units or ticks)
- A series of pulses representing 4-bit data or nibbles (with duration from 12 to 27 ticks, where 12 ticks represent the value of 0 and 27 the value of 15)
    - A start nibble with 2-bit communication and 2-bit status information
    - 1 to 6 data nibbles for sensor data communication
    - 1 checksum nibble

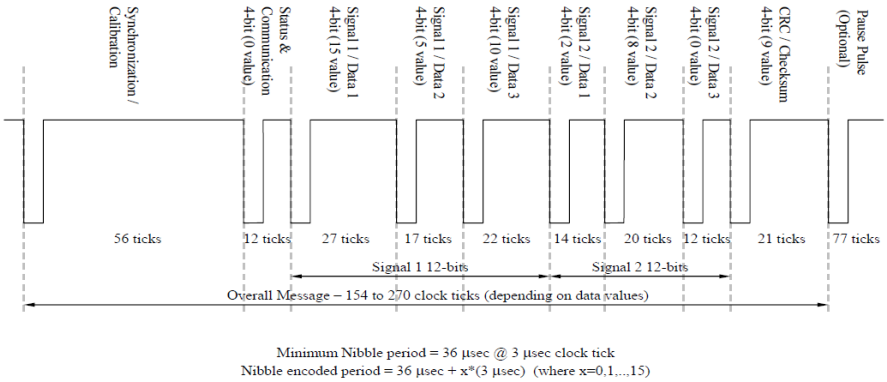

Figure 1 - *SAE J2716™ SENT specification sample - Example encoding scheme for two 12-bit data*

We will now further analyze a specific sample of the SENT specification:

5.2.3 Transmission Properties of Nibble Pulse

- Minimum pulse period is 12 clock ticks.
- More than 4 clock ticks driven low time (all remaining clock ticks driven high).
- Each nibble count is 1 clock tick (0 – 15 counts ⇒ 0 – 45 µs at a 3 µs clock tick).
- Minimum nibble pulse period (transmission value of 0) = 12 clock ticks (36 µs at a 3 µs clock tick).
- Maximum nibble pulse period (transmission value of 15) = 12 + 15 = 27 clock ticks (36 + 45 = 81 µs at a 3 µs clock tick).

Figure 2 – SAE J2716™ SENT protocol specification sample

As a verification engineer, this should lead us to verify the design under all valid conditions:

- data pulse duration is between 12 and 27 ticks
- low time of this pulse is 4 or more

Analyzing this section, we should also question ourselves what the maximum low time value to expect as this specific point is not clearly specified. By construction, the low time cannot be greater than the duration of the pulse itself, and we deduce from other part of the specification that the high time is at least 1 tick (which is an indivisible time). So, we deduced that the maximum low time is the actual pulse length minus one.

We should therefore ensure the following coverage of low time and pulse length is reached:

Coverpoint	**Bins**
data_pulse_length	All integer values from 12 to 27
data_pulse_low_time	All integer values from 4 to data_pulse_length-1
cross data_pulse_low_time, nibble_value	For each nibble value 0 to 15, all integer values from 4 to data_pulse_length-1

In terms of qualification however, we should also ensure that any protocol violation will lead to an error. In this specific case the VIP should report an error under the following conditions:

- The duration of this pulse is strictly less than 12
- The duration of this pulse is strictly more than 27
- The duration of the low time of the pulse is strictly less than 4.
- The duration of the low time of the pulse is strictly more than data_pulse_length-1

Our qualification of the VIP should then create these 4 scenarios in error condition tests and ensure that we have an assertion that reports these errors.

The following snapshot shows a sample of the associated qualification test plan using the Mentor Graphics Questa® VM Testplan Excel Add-In. This qualification plan is written so that we:

- Ensure that no errors are reported under valid conditions
- Ensure that errors are reported under invalid conditions

Sectic	Title	type	Covers	Description	Link	Type
1	Global Verification Plan for SENT V	title				
1.1	Calibration Pulse	title				Test
1.1.1	Calibration Length	VTC	SAE J2716 - Section 5.2.2	Note : Since protocols allows a ±20% clock period variation a calibration pulse would be considered as valid if its length is inside [45;67] ticks length. **Goals**: - verify that VIP reports an error when calibration length is outside of valid value **Test Scenarios:** - For various SENT frames with valid low time with different length for the calibration pulse. - Cover cases with length = 56 - Cover case with length 45, 46, 47, 65, 66, 67 and other value inside [45;67]. **Checks:** - Check that assertions are not triggered when calibration length is inside [45 ; 67] - Check that assertions are triggered if calibration calibration length is < 45 or > 67	frame_cg:calibration_pulse	CoverPoint
1.1.2	Calibration low time	VTC	SAE J2716 - Section 5.2.2	**Goals**: - verify that VIP reports an error when the five clock ticks of calibration pulse are not driven low **Test Scenarios:** - For various SENT frames with different low time for the calibration pulse. - Cover cases with low time = 1, 2 ,3 and 4 - Cover cases with low time = 5, 54, 55 and other arbitrary values. **Checks:** - Check that assertions are triggered if calibration low time is 1, 2, or 3. - Check that assertions are not triggered if calibration low time is > 4.	invalid_calibration_low_time_test_seq frame_cg:calibration_pulse frame_value_cg:low_tu SENT_ASSERT_SYNC_LOW_MIN_TIME QUALIF_ASSERT_CHECK_EXPECTED_ERRORS	Test CoverPoint CoverPoint Assertion Assertion
1.2	Nibble	title				
1.2.1	Nibble length	VTC	SAE J2716 - Section 5.2.3	**Goals**: - verify that VIP reports an error when nibble length is outside [12:27] clock ticks (SAE J2716 - Section 5.2.3 - Transmission properties of Nibble Pulses) **Test Scenarios:** - For various SENT frames with different data values. - Covers full range of data values between 0 and 15. - For various SENT frames with invalid data nibbles. - cover cases with nibble length = 11, 28 and some other values < 11 and > 28. **Checks:** -Check that no assertions are fired when data are valid. -Check that assertioons fire an error when length <= 11 and >= 28.	invalid_nibble_length_test_seq nibble_cg:length	Test CoverPoint

Figure 2 – TestPlan Sample in Excel using Questa® VM Testplan Tracking Excel Add-In

The VIP test plan is therefore built from the specification and its requirements. For each of the specification aspects, the verification Plan of the VIP defines:

- conditions to be covered in order to hit the valid case boundaries
- conditions to be covered in order to hit the invalid case boundaries
- the expected assertions to be fired under the invalid conditions and the related check (expected error)

B. VIP Qualification Environment

In order to qualify a VIP, we need to build a testbench which instantiate the VIP connected to a symmetrical agent. The connected agent need to be able to provide the following features:

- Generate valid requests in nominal mode
- Generate valid responses in nominal mode
- Ability to generate invalid requests on-demand
- Ability to generate invalid responses on-demand

A short-cut would be to connect the master and the slave of the same VIP, but this is however not a valid case to qualify the VIP under safety standards, unless the master and the slave parts of the VIP are developed by two different persons, do not share any code (especially on the checker parts) and only either the master or the slave needs to be qualified.

In our case, we opted in the development of a small behavioral model of the protocol which has the ability to generate both valid and invalid frames with control of each single timing of the protocol. A test sequence can therefore inject specific errors in the protocol.

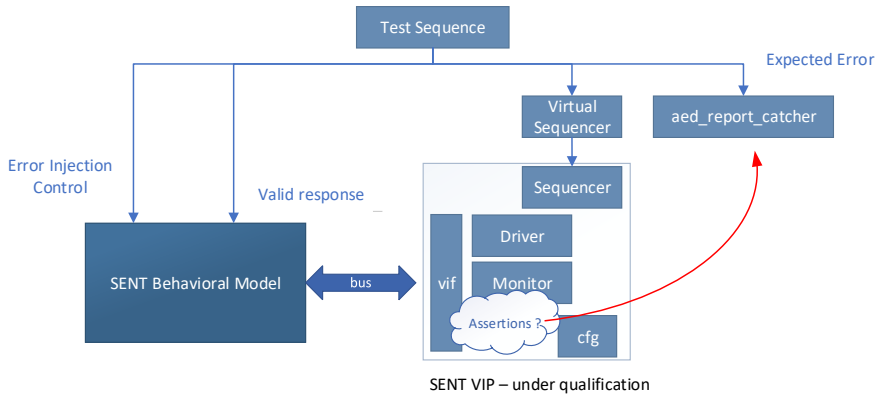

Figure 3 - VIP Verification Environment

In this environment, in order to build a test to check an error condition, the test sequence needs to:

- Specify which error it is trying to verify
- Configure the behavioural model to generate the corresponding error condition.
- Wait for the supposed time necessary to detect the error
- Check the error has been detected by the VIP in the timeframe.

C. *UVM Report extension to expect errors*

As described in the verification plan, the qualification of the VIP requires us to expect the VIP to report errors.

We need however to precisely distinguish the errors that are expected, which will make the test to pass, and the errors that are not expected, that will make our test to fail.

UVM provides interesting error reporting mechanisms, using the uvm_report_catcher class and the `uvm_error macros. But UVM does not provide natively the notion of “expected” errors. Thankfully, the uvm_report_catcher class provides easy ways to extend the UVM reporting mechanism and to catch any expected errors. We can then think of developing a qualification library that will provide these features in a more generic way.

In order to verify our VIP, we need to be able to:

- Catch/detect when expected errors are fired.
    - When we expect at least N errors
    - When we expect at most M errors
    - After the source of the error
    - Before the recovery condition following the error.
- Mark the expected errors as expected, making the test to pass.
- Continue to mark non-expected errors as making the test to fail.

These errors can be caught per ID, message or context. It is also possible to demote an expected error as a warning so that the final UVM report summary only counts the non-expected UVM_ERRORs. In the end, a qualification summary is provided so that the test can be marked pass or failed depending on this summary.

The proposed API for this library provides the following features:

- Specify UVM error ID or message to catch
- Specify the minimum number of errors to expect for the specified ID/Message
- Specify the maximum number of errors to expect for the specified ID/Message
- Start / End the analysis time window in which errors are expected
- Specify the action to take when an error is caught: for example, demote to UVM_WARNING
- Ability to group expected errors into a group to deal with common start / end events and common actions

The following code presents how the assertion ID SENT_ASSERT_MASTER_LOW_TIME and SENT_ASSERT_CRC_VALUE are captured between the call of start() and stop() of the analysis window. In this case, we want to capture that 1 and only 1 (min = max = 1) UVM_ERROR for each expected ID in the analysis window. When the expected error is detected, it is demoted to a UVM_WARNING so that the UVM reports does not count this expected error.

```
// Enable full report at the end of test.
aed_report_catcher::final_report_mode ( AED_REPORT_FULL );

// Get a new group of message to catch and demote.
msg_grp1 = aed_report_catcher::type_id::get_report_action_group("Group-1");

// Set message ID to catch, its minimum and maximum expected occurrences
msg_grp1.expect_error_by_id(
       .name          ( "QUALIF_ASSERT_SYNC_LOW_MIN_TIME" ),
       .id            ( "SENT_ASSERT_SYNC_LOW_MIN_TIME" ),
       .min           ( 1 ),
       .max           ( 1 ));

// Set message ID to ignore within the same time window
// Same as calling expect_error_by_id with minimum=0 and no maximum.
msg_grp1.ignore_error_by_id(.id ( "SENT_ASSERT_CRC_VALUE" ));

// Set action to perform on these errors: Demote to Warning.
msg_grp1.demote_error ( UVM_WARNING );

// Start the expected error time window
// alternatively, an event pointer could have been passed to the group creation.
msg_grp1.start();

// Create the error condition
`uvm_do_on_with(req, sensor_sequencer, { req.invalid_sync_pulse == 1;} );

// Stop the expected error window, no more error expected
msg_grp1.stop();
// At this point, if the expected error message are not caught between
// a minimum and maximum amount, an UVM_ERROR is issued.
```

In the above code, we expect the UVM_ERROR with the ID "SENT_ASSERT_SYNC_LOW_MIN_TIME" to be fired between the start() and the stop() time execution:

- At least once
- At most once

If this error is actually not caught or is fired more than once, a UVM_ERROR will be issued by the stop() method and the test will be marked as failed.

The assertion ID "SENT_ASSERT_CRC_VALUE" on its side has no expectation, as it may or may not fail depending on the actual data and variation of the synchronization. We therefore mark it to be caught with no minimum and with no maximum. This is not an issue as this is not the assertion which is being qualified.

In the end, the final report will summarize which errors have been caught and if expected errors were found. Non-expected errors will also be reported.

```
# ====== AED DEMOTER END OF SIMULATION REPORT =====
#
# UVM_ERROR short_path/aed_report_utils/src/sv/aed_report_catcher.sv (314) @ 78933000:
uvm_test_top.sequencer@@sent_invalid_test_seq
 [MIN COUNT NOT REACHED] Filter : QUALIF_ASSERT_SYNC_LOW_MIN_TIME
#  Number of caught error is lower than min count :
#   Actual count :          0        Min expected count :          1
# ____________________________________________________________
# Catcher Name                    ,   Expected  ,    Actual  ,     Status
# QUALIF_ASSERT_SYNC_LOW_MIN_TIME     , [  1:  1 ]  ,       0    ,      FAIL
# IGNORE_FILTER_SENT_ASSERT_CRC_VALUE , [   >= 0 ]  ,       1    ,      OK
# ____________________________________________________________
#
# report_catcher : Group-1: FAILED    ; 0 error while expecting at least          1
#
```

```
# ================================================
```

As we can see from the end of simulation report, one error has been caught (`SENT_ASSERT_CRC_VALUE`) and has been ignored while it hasn't caught any message with the ID "`SENT_ASSERT_ SYNC_LOW_MIN_TIME`". As we didn't meet the expected error, the test does not behave as we expect: the end of simulation therefore reports a UVM_ERROR [MIN COUNT NOT REACHED].

D. Implementation Overview

The implementation is based on the uvm_report_catcher class.

The class aed_report_catcher extends the UVM class uvm_report_catcher and in particular its catch() method which is the one used to process UVM messages and errors through uvm_reports. At this point, the reporting attributes are available (client, severity, verbosity, id, message, file name, line in file...) to determine if the current processed message is expected one or not and needs to be filtered.

In our case, we made the choice to group a list of error messages that need to be handled in the same way. The user can therefore specify the list of IDs which need to be expected, demoted or ignored between two events without having to call the start/end methods for each of them.

Once set, each time the aed_report_catcher come across a report message that fits a report_filter it will update the count of this filter. If the count surpasses the max_count set in filter or group it will immediately issue an uvm_error. The min_count is compared to the current count when a filter or a group monitoring is stopped.

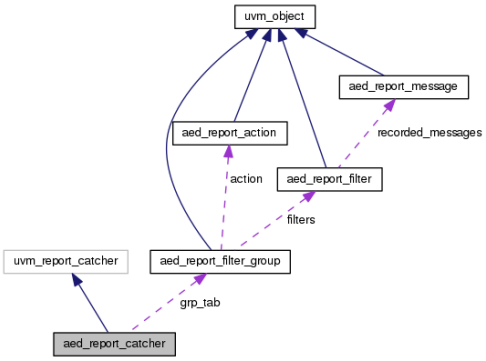

Figure 4 – report_catcher collaboration diagram generated from Doxygen

III. RESULTS

The following graphs shows the same regression with and without the report catcher activated. This highlights the fact that about half of tests are actually checking invalid protocol conditions and the VIP reports errors. When the expected errors are demoted, the associated tests are automatically marked as passed, saving lots of analysis of the logs.

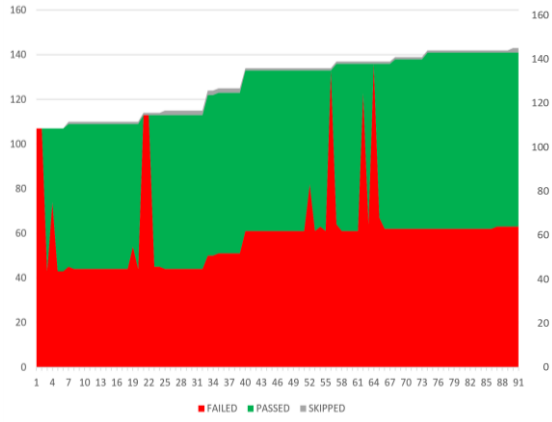

Figure 6 - Test Suite without Expected Error Catcher

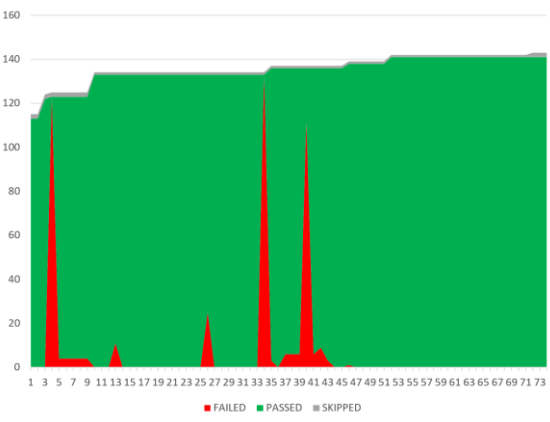

Figure 5 - Test Suite with Expected Error Catcher

As seen in these results, without a proper way to check the expected error, it will be very time consuming to go and dig into the reports to checks that the errors we see are actually the expected errors.

In the end, the activity performed on this SENT VIP has highlighted a few conditions that were not strictly checked by the original assertions:

- Some assertions were missing and were discovered from an independent analysis of the spec and actual qualification tests.
- Some assertions were found to be not called in all the cases of the error they were checking.

Additionally, by automating the expected error catching, this makes the regression able to detect any change that can impact the quality of the checkers.

IV. LIMITATIONS

Unfortunately, the approach has some limitations. These are not critical as long as they are well understood:

1- In the same way as for the design vs verification, for the approach to really make sense, a specification analysis should be performed from a different person than the VIP developers.
2- Building a qualification plan, is much more consuming than setting up mutation testing, but you will get traceability in return. The number of required tests is however much more limited than in mutation testing.
3- When testing an error condition, we known that an error is expected, but we may not know in advance which one will be fired. This requires debug and iterations in order to actually set up the demoter properly without taking the risk of demoting unwanted or too many assertions.
4- The UVM catcher is only able to catch UVM messages. The library cannot therefore be used to qualify third party VIP that do not use UVM_ERROR in their assertions.

CONCLUSION

The proposed approach to qualify a verification IP is mostly based on applying verification techniques to the verification IP as the object to be verified. As for design verification, this requires a qualification plan, a testbench and tests that will exercise or respond to the VIP. The major difference with design verification is that we really need to think of invalid cases; this can be a very huge task and requires in depth analysis and verification background to be conducted properly.

In the end, the goal is to check the conditions at the limits of the specification. Making sure that valid limits are reached by the VIP without errors, while going beyond the limit makes the VIP to report errors. As these errors are the actual expected results, we have to clearly identify the good errors from the bad errors. Thankfully, UVM library allows to easily extend its reporting capability to meet our goals. The remaining challenging part is now to make people distinguish between good and bad errors.

ACKNOWLEDGMENT

Special thanks to Melexis mix signal design team in Bevaix, Switzerland who allowed this development.

REFERENCES

[1] Dr. Andrew J. Kornecki a, nd Dr. Janusz Zalewski, “The Qualification of Software Development Tools From the DO-178B Certification Perspective” , CROSSTALK The Journal of Defense Software Engineering, Software Engineering Technology

[2] Certification Authorities Software Team (CAST) Position Paper CAST-27 CLARIFICATIONS ON THE USE OF RTCA DOCUMENT DO-254 AND EUROCAE DOCUMENT ED-80, DESIGN ASSURANCE GUIDANCE FOR AIRBORNE ELECTRONIC HARDWARE, https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-27.pdf

[3] Qualifying Software Tools According to ISO 26262, Mirko Conrad, Patrick Munier, Frank Rauch, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.454.3782&rep=rep1&type=pdf

[4] SYNOPSYS Certitude Datasheet, https://www.synopsys.com/verification/simulation/certitude.html

[5] Coverage Discounting: Improved Testbench Qualification by Combining Mutation Analysis with Functional Coverage, Nicole Lesperance, Peter Lisherness, and Kwang-Ting Cheng

[6] SENT—Single Edge Nibble Transmission for Automotive Applications J2716_201001, https://www.sae.org/standards/content/j2716_201001/

[7] A Tutorial for the Digital SENT Interface, Tim White https://zh.idt.com/document/whp/tutorial-digital-sent-interface-zssc416xzssc417x