AMS System-Level Verification and Validation using UVM in SystemC and SystemC-AMS: Automotive Use Cases


Abstract—The automotive trends is to increase the electronics systems inside vehicles. The complexity of such systems is rising with the number of components involved on the one hand and on the other hand on the tighter interaction between these components, being analog, digital hardware or software. The verification of Electronic Control Systems (ECU) becomes more and more challenging. In this paper we show that the Universal Verification Methodology (UVM), initially developed for digital systems, which consists in clearly distinguishing between the test scenario, described in an abstract way, the Device Under test (DUT) and the test environment that translates the test to the DUT interface, can be extended to analog and mixed signal systems. We introduce the UVM-SystemC-AMS library implemented above SystemC and its AMS extension SystemC-AMS. This approach is used to verify two ECUs from automotive industry. The first use case shows how UVM can be used for the simulation-based verification of a complex mixed-signal design. The second use case shows how UVM can be used on a Hardware In the Loop (HIL) system to verify/validate a FPGA prototype.

Index Terms— Design Under Test (DUT), Electronic Control Unit (ECU), Electronic System Level (ESL), Hardware In the Loop (HIL), system simulation, SystemC, SystemC-AMS, Timed Data Flow (TDF), Transaction Level Modeling (TLM), Universal Verification Methodology (UVM), system verification, Virtual Prototyping (VP).

I. INTRODUCTION

The complexity of electronic systems for the automotive industry is increasing. Such systems are designed to exploit tight interaction between the physical world, captured or controlled through sensors and actuators, and the digital hardware (HW) and software (SW) world. Electronic Control Units (ECU) in cars are fully heterogeneous systems, implementing analog power electronics and low-voltage electronics, controlled by software running on an embedded processor. To master the complexity of the ECU design, a variety of Electronic System Level (ESL) design methods and languages have arisen. To that purpose, the SystemC language standard [1][2] has been extended with powerful Analog and Mixed Signal (AMS) modeling capabilities [3], addressing functional and architecture level.

Yet, as great effort was made towards efficient analog and mixed system level design and modeling technologies [4][5][6][7][8][9][8], the system-level verification are left behind. Coverage-driven verification of digital IP has become more mature since the introduction of the Universal Verification Methodology (UVM) standard [10], implemented in SystemVerilog [11]. The principle is to build structured test-benches with reusable verification components. Following the UVM approach, the tests are designed in a hierarchical and modular way, using similar abstraction levels than the device under test (DUT) itself. This includes abstraction such as Transaction Level Modeling (TLM) for the test sequences [12], combined with accurate, signal interface to the DUT. We have introduced the UVM-SystemC and AMS extensions, called here UVM-SystemC-AMS, to enable the development of virtual prototypes at TLM abstraction with a structured test bench methodology like UVM. The goal is to perform extensive verification of the embedded application. Yet, one application will often require a high computational effort, which can take up to hours or days. When the DUT model includes RTL level descriptions, the simulation-based verification may become impractical.

This is where the UVM approach helps to efficiently translate the test bench built for simulation verification into real hardware prototyping validation using laboratory measurements. Using UVM-SystemC-AMS we were able to establish a tight coupling between the virtual prototyping simulation and the Hardware In the Loop (HIL) laboratory validation leading to an increase in speed and bug detection. Two use cases will show the applicability of UVM-SystemC-AMS for industrial scale applications. The first use case shows how UVM can be used for the simulation-based...
verification of a complex SystemC-AMS design, which uses Timed Data Flow (TDF) interfaces. The second use case shows how UVM can be used on a HIL system to verify/validate a FPGA prototype.

The paper is organized as follows. Section II describes the related work in the areas of methodologies and verification tools for AMS-ESL. Section III presents our approach in terms of layered verification environment architecture based on UVM, SystemC and SystemC-AMS. Section IV introduces the UVM-SystemC-AMS library and its main features. Section V describes the constrained randomization of real values as well as the functional coverage API illustrated with a SPI-controlled filter example. Section VI presents the creation of a verification environment in UVM-SystemC-AMS for an automotive verification use case. Section VII presents an automotive validation use case. Section VIII concludes the paper.

II. STATE OF THE ART AND CONTRIBUTION

The trend to start earlier with system-verification in the design cycle, in combination with the growing complexity of the actual design implementation (and its virtual prototype counterpart), emphasizes the need to apply a proven verification methodology. Therefore there have been many attempts in the past to create structured and reusable verification environments in SystemC/C++ (Figure 1). In [13], the Open Verification Methodology (OVM) formed the basis for creating a SystemC-equivalent verification environment. The System Verification Methodology [14] was based on OVM-SystemC, donated by Cadence to the community [15]. Mentor Graphics released a SystemC version as part of their Advanced Verification Methodology (AVM) [16]. Also Synopsys introduced as part of their Verification Methodology Manual (VMM) a class library in SystemC [17][18]. More recent studies address the need to support a true multi-language verification environment in SystemVerilog, SystemC and C [19].

Figure 1: Evolution of Verification Methodologies

However, all these initiatives do not fully comply with the methods defined in the UVM standard, primarily because they are built on the former AVM, OVM, and VMM technologies. The consolidation into a single UVM standard resulted in major changes. As a consequence, the user has to deal with the incompatibilities related to simulation semantics and language constructs. Especially the move from OVM to UVM significantly changed the way components deal with the phasing mechanism and how the end-of-test is managed. To avoid legacy concepts and constructs in modern test benches, migration to UVM standard compatible implementations should be encouraged.

Alternative solutions are proposed in [20][21][22] to address the multi-language integration challenges found in today’s verification environments, by defining a set of coding guidelines centered around TLM communication. However, the creation of reusable verification components and integration in a test bench is much more than having an agreed communication method; additional elements like test bench configuration and reuse of test sequences do require a more holistic view on UVM and its principles, and justifies making these concepts available in other languages.

Therefore an up-to-date and UVM standard compliant language definition and reference implementation is needed in SystemC/C++, which not only gives the user community a semantically and syntactically correct implementation of UVM, but also the same user experience in terms of the UVM “look & feel”. Especially the latter aspect would facilitate UVM users to start using SystemC for system-level and hardware/software co-verification, or make SystemC or software experts more familiar with the powerful UVM concepts to advance in the verification practice [23].

The recent C++ based extension for the analog mixed signal (AMS) simulation in SystemC-AMS [3] has opened the perspective for electronic mixed domain system simulation and extension to the verification domain and test coverage. There is an actual need to extend actual UVM digital oriented methodology and implementation to new construction to support the analog domain usable for the Model of Execution (MoC) of SystemC-AMS.

Ultimately, this will benefit the entire design community, where verification and system-level design practices come closer together. It will serve the universal objectives of the UVM, addressing the need for having a common verification platform, including hardware prototyping in which C-based test sequences or verification components in UVM-SystemC are reused in HIL simulation or Rapid Control Prototyping (RCP) [24].

UVM is a verification framework that allows the creation of test benches based on a constrained random stimulus principle. Instead of testing the DUT with directed test sequences, random stimulus is applied, which is shaped by constraints so that the randomly generated values are valid stimulus. As the input stimulus is randomly generated, it is very important to observe which data has been sent to the DUT, to make sure that all design corners have been tested during a verification regression run. Therefore, functional coverage can be used which allows defining our own coverage goals.

The UVM standard and associated class library implementation in SystemVerilog does not define the constructs for randomization and functional coverage, because these concepts are intrinsically part of the SystemVerilog
standard, defined in IEEE Std. 180 [11]. In a similar way, UVM in SystemC will not introduce such constructs for randomization and coverage, but will make use of dedicated libraries for this purpose. Several good attempts have been made to support constrained randomization for SystemC, such as the SystemC Verification library (SCV) [25] and the Constrained Random Verification Environment for SystemC (CRAVE) [26]. Also different libraries that add functional coverage to SystemC have been proposed in [12][27][28][29]. Furthermore, various commercial, proprietary or vendor-specific solutions are available.

In the following we will present our contributions: a methodology based on UVM and SystemC-AMS allowing to test mixed-signal designs at different abstraction levels. We will illustrate how the test descriptions are reusable also for the laboratory and prototype validation. To support the real analog value, we will introduce an API for randomization, which supports constraints randomization of continuous distribution functions as well as an API for coverage real of real value.

III. UVM PRINCIPLES FOR AMS SYSTEMS

The AMS system-level verification methodology is based on three UVM basic principles that have been extended to AMS systems since the first presentation of UVM-SystemC library [23]. The first one consists of a clear separation between tests, test bench and AMS DUT implementation. The second one is the definition of abstract and reusable tests scenarios. Creating a sequence of commands, which can drive an analog signal pattern, forms a test scenario. The third one concerns the test bench, or verification environment. It is the translator between the abstract test scenario and the DUT.

A. Layered approach

UVM-SystemC-AMS follows a layered approach, inspired by [17], where levels of abstraction are introduced to distinguish the test scenario form the actual verification environment on which sequences are executed. Figure 2 illustrates the 3-level top-down refinement of test sequence stimuli as well as the bottom-up reconstruction of performance indicators for verification. The virtual sequences are extracted from a scenario database (the test) and propagated to the test bench. The virtual sequencer controls all sequences at top-level. The sequencers control the sequence of data transactions and send them to drivers. The drivers translate the sequence of data transaction to Discrete Event (DE) signals for the digital IPs and to SystemC-AMS TDF samples for AMS IPs and send them to DUT ports. The monitors collect the output signal from the DUT and store them. From a relevant set of output signals, the analysis utility functions translate them to transactions. Figures of merit are computed from these outputs signal transactions as well as the associated collected input stimuli transactions. The scoreboard compares the results for the DUT with the golden model reference ones.

Figure 2: UVM based layering

B. UVM components

We take advantage of the already existing principles of UVM used in the digital domain to clearly separate the analog/mixed signal test, the test bench and the actual implementation of the DUT exhibiting AMS behaviors. Universal Verification Components (UVCs) are the primitive building blocks to create the test bench or verification environment. Figure 3 illustrates a typical example. At the top level of the hierarchy, two UVCs (called here VIP1 and VIP2) are connected to the virtual sequencer using TLM communication (see Figure 2 and ⊗ in Figure 3).

A UVC contains one or more agents. The corner stone of UVC is the agent (○ in Figure 3), which instantiates the sequencer, the driver (Figure 2 and ⊗ in Figure 3) and the monitor (Figure 2 and ⊗ in Figure 3). The agent receives sequential request and converts them into low-level data at the DUT interface. Agents may be active or passive. Active agents drive signals to the DUT and thus instantiates a sequencer and a driver. Passive agents do not drive DUT and thus do not instantiate any sequencer or driver. Whether active or passive, agents may instantiate monitors to collect results.

Figure 3: UVM-SystemC-AMS test environment
The sequences encapsulate sequence items, also called transactions. The sequences are not part of the test environment hierarchy. They are UVM-SystemC-AMS objects that are mapped on one or more sequencers. The sequencer reacts to the orders given by the driver by getting sequence item and delivering it to the driver. The driver requests transactions (sequence item) and translate them to one or more physical signals.

All components can be configured using the UVM configuration database.

Functional coverage can be collected by adding coverage models at different levels of abstractions shown in green in Figure 3. Coverage requesting signal values is collected in a monitor (© in Figure 3). More abstract coverage estimation is collected based on transactions, which a monitor provides through an analysis port (Figure 2 and © in Figure 3). The subscribers (© in Figure 3) are supplied with such kind of collected information. This allows functional coverage related to the overall checking of the verification goals to be performed by the scoreboard.

C. Reuse

UVM test environments in SystemC can be reused for verification at different levels of abstractions as well as for validation in the laboratory. UVM-SystemC-AMS test environment can be used to test a design from a system level description down to an RTL-description of the design, as SystemC can be coupled with different simulators [30][31]. Furthermore, as the C++ language can be easily compiled to different hardware platforms it is also possible to reuse UVM test environments for laboratory based validation. SystemC can run e.g. on Hardware in the Loop Simulator systems [32][33], an example of possible reuse of a test environment is shown in Figure 4. The application use case shown in Section VII demonstrates the reuse possibilities of UVM for SystemC.

D. Test execution and verification features

In order to have a consistent test bench execution flow, the UVM-SystemC and UVM-SystemC-AMS use phases to order the major steps that take place during simulation.

Figure 5: UVM-SystemC-AMS common and runtime phases

There are three groups of phases, which are executed in the following order:

1. Pre-run phases (build_phase, connect_phase): The build_phase is executed at the start of the UVM testbench simulation and their overall purpose is to construct, configure the test bench component hierarchy. During the build_phase UVM components are indirectly constructed using the factory pattern. The connect_phase is used to interconnect all the components.

2. Runtime phase (run_phase): The test bench stimulus is generated and executed during the runtime phases, which follow the build phases. In run_phase and run-time phases, we execute the test scenarios by performing the configuration of the DUT and applying primary test stimulus to DUT. These runtime phases are all spawned SystemC processes and can consume time, unlike the other phases, which are untimed function calls. All UVM components using the run-time schedule are synchronized with respect to the pre-defined phases in the schedule.

3. Post-run phases (extract_phase, report_phase and final_phase): where the results of the test case are collected and reported. The 4 post-run phases are used to post-process the results after the execution of the test scenario of the DUT.

E. TDF and Discrete Event synchronisation

Virtual sequences, sequences, and sampled signals represent the actual data sent to or received from the DUT at different time scales. To correctly synchronize the UVCs involved in the test environment and the DUT, it is necessary to synchronize the related SystemC and SystemC-AMS modules.

In Figure 6, an UVM-SystemC-AMS driver component consisting of two modules is presented. The first component is a SystemC TLM adaptor component. A process of this component reads the timestamps of the transactions and writes (schedules) the data of the transaction in the second delta cycle, one resolution time step before the timestamp of the transaction. This guarantees that the data will be read at the correct time by the TDF module, since SystemC-AMS reads event driven SystemC signals always at the first delta cycle of the current time.
This pipelined synchronization method is presented in Figure 7.

![Figure 7: Pipelined synchronization process for scoreboard](https://example.com/figure7.png)

Between each sequence item, the sequencer/driver waits for a period $T_0$ (which can be variable). The monitoring thread waits for the same period, so that the emission of stimuli and the reconstruction of performance indicators are kept synchronous during the simulation, for scoreboard comparisons. Once a new sequence item has been propagated to the driver (at the very beginning of the sequence item period), the corresponding TDF driver module uses the transaction description to generate the TDF stimuli for the DUT. For instance, if the sequence item defines a new operating frequency for the wave generator, the corresponding TDF processing function reacts immediately (i.e. within a time step $T_0$) to the driver (at the very beginning of the sequence item). The corresponding driver module, once a period $T_0$ has elapsed, will generate the appropriate low level generated stimuli. When scheduled, the thread associated to the monitor gains direct access to these performance values, which will remain valid for the next $T_0$ period.

### IV. UVM-SystemC-AMS Library Implementation

In this section we present the UVM-SystemC-AMS library and describe the main features with code snippets, therefore extending the UVM-SystemC library version of [23].

#### A. The agent

Listing 1 below shows the creation of an UVM component with the user-defined name `vip_agent` in UVM-SystemC-AMS. An agent encapsulates the components, which are necessary to drive and monitor the (physical) signals to (or from) the DUT. Typically, it contains three components: a sequencer, a driver and a monitor. The agent can also contain analysis functionality for basic coverage and checking, but this is not shown in this simple example.

```c++
// Listing 1: UVM-SystemC-AMS agent

class vip_agent : public uvm_agent {
public:
  vip_sequencer<vip_trans>* sequencer;
  vip_driver<vip_trans>* driver;
  vip_monitor* monitor;

  uvm_component_utils(vip_agent)
  vip_agent( uvm_name name )
  : uvm_agent(name), sequencer(0), driver(0), monitor(0) {}

  virtual void build_phase( uvm_phase& phase ) {
    uvm_agent::build_phase(phase);
    if ( get_is_active() == UVM_ACTIVE ) {
      sequencer = vip_sequencer<vip_trans>::type_id::create("sequencer", this);
      driver = vip_driver<vip_trans>::type_id::create("driver", this);
      assert(sequencer);
      assert(driver);
    }
    monitor = vip_monitor::type_id::create("monitor", this);
    assert(monitor);
  }

  virtual void connect_phase( uvm_phase& phase ) {
    if ( get_is_active() == UVM_ACTIVE )
      // connect sequencer to driver
      driver->seq_item_port.connect(sequencer->seq_item_export);
  }
};
```

#### B. The sequences

The UVM sequence item is used as template argument for the creation of the actual sequence, as shown in Listing 2. A sequence is derived from template class `uvm_sequence` (Line 3). The macro `UVM_OBJECT_PARAM_UTILS` supports the registration of template classes with multiple arguments, which are derived from `uvm_object` (Line 11).

```c++
// Listing 2: UVM-SystemC-AMS sequence

template <typename REQ = uvm_sequence_item, typename RSP = REQ>
class sequence : public uvm_sequence<REQ,RSP> {
public:
  sequence( const std::string& name )
  : uvm_sequence<REQ,RSP>( name ) {}

  UVM_OBJECT_PARAM_UTILS(sequence<REQ,RSP>);

  virtual void pre_body() {
    if ( starting_phase != NULL )
      starting_phase->raise_objection(this);
  }

  virtual void body() {
    REQ* req,
    RSP* rsp;
    ...
    start_item(req);
    // req->randomize(); //optional randomization call
    finish_item(req);
    get_response(rsp);
  }
};
```
The callback function body is used to implement the user-specific test scenario (Line 18). The member functions start_item and finish_item are called to negotiate and then send the sequence to the sequencer (Lines 22 and 24). The member function randomize, as defined as part of the compatibility layer to the SCV or CRAVE library (Line 23, see Section V). The member function body is automatically called from the higher level in the test bench, for example by explicitly calling the member function start in a virtual sequence. Alternatively, a sequence can be started implicitly by defining a default sequence in a test, along with specifying the component and phase where it should be executed. The latter approach is presented in section IV.D, Listing 4.

The callback functions pre_body and post_body (Line 13 and 30) are called before and after the callback body, respectively, to raise and to drop an objection only when the sequence has no parent sequence. For this purpose, the data member starting_phase is used, offering a handle to the default sequence (Line 15 and 32).

### C. The test bench

The test bench is defined as the complete verification environment which instantiates and configures the universal verification components (UVCs), scoreboard, and virtual sequencer if necessary. The UVCs are sub-environments in the test bench, which contain one or more agents.

Listing 3 shows the implementation of the test bench. It uses the base class uvm_env (Line 1). The UVCs and scoreboard are instantiated using the factory (Lines 15, 17, 21, 24). This facilitates component overriding from the test scenario. The configuration database is used to configure each agent in the UVC as being active or passive, by means of the global function set_config_int (Lines 19 and 20).

```cpp
Listing 3: UVM-SystemC-AMS test bench
```

### D. The test

Each UVM test is defined as a dedicated test class derived from class uvm_test, as shown in Listing 4 (Line 1). It instantiates the testbench (Line 11) and defines the default sequence which will be executed on the virtual sequencer in the run_phase (Lines 14-17 and section IV.B). The factory member function set_type_override_by_type is used to override the original UVM driver in the agent by a new driver (Listing 4, lines 18-20). The types of the original and new driver are obtained by calling the static member function get_type.

The result from the scoreboard checking is extracted in the extract_phase and updates the local data member test_pass. The actual pass/fail reporting is done in the callback report_phase using the available UVM macros for reporting information and errors.
These extensions concern especially:

- The identification of the different UVM objects (test, test-bench, UVC, agent, scoreboard, monitor, driver and sequencer)
- The interconnection of the DUT to the UVCs with the help of virtual interfaces
- The configuration of the platform using the UVM configuration database and agents of configuration

The assembly described in IP-XACT, reproduces the same architecture of the UVM test environment and then offers the same level of reusability. At the end, the generator will provide the SystemC and SystemC-AMS code (header and cpp files) of the different layers: top, test and test environment but also the header of the virtual sequencer containing references to the sequencers instantiated in the different UVCs. The generation is based on templates, one per output files, to introduce some flexibility. Text part (header, comments, fix library) is dumped without any modification and key words are replaced by meta-data elaborated in an internal model from the IP-XACT description. Figure 8 illustrates an example of a generator. The left column shows the template used. The right column shows the generated code.

E. The top-level

The main program, also called top-level, uses the SystemC **sc_main** function and contains the DUT, the interfaces connected to the DUT, and the definition of the test, as shown in Listing 5. The interfaces are stored in the configuration database to be used by the UVC drivers and monitors to connect to the DUT (Lines 6-9).

F. IP-XACT

IEEE1685 IP-XACT standard [34][35] enables an efficient assembly and configuration of the structural layers of the test environment (test bench, test and top level elements) by generating the relevant SystemC and SystemC-AMS views necessary to conduct verification. It provides also a unified repository to exchange and share compatible components from multiple companies or services.

In order to capture the UVM properties, we have introduced some extensions in the schema of the IP-XACT standard. These extensions concern especially:

- The identification of the different UVM objects (test, test-bench, UVC, agent, scoreboard, monitor, driver and sequencer)
- The interconnection of the DUT to the UVCs with the help of virtual interfaces
- The configuration of the platform using the UVM configuration database and agents of configuration

V. CONSTRAINED RANDOMIZATION AND COVERAGE

To cope with the complexity of ESL, random sequences/stimuli are used to increase the coverage of the DUT. As several attempts exist to the power of SystemC to digital verification such as SCV [25] or CRAVE [26], we have proposed an API for randomization and coverage as a compatibility layer, which can either use SCV or CRAVE as a back end solver. To correctly handle the analog signals found in AMS systems, new functions have been introduced to generate random real values, following continuous distribution functions, which are subject to constraints. More over the functional coverage API provides the functions covergroup, coverpoint and bins offered by SystemVerilog, which are extended to handle real values using intervals (Table I).

<table>
<thead>
<tr>
<th>Functionality</th>
<th>UVM-SystemC-AMS (scvx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage model</td>
<td>scvx_covergroup</td>
</tr>
<tr>
<td>Coverage point</td>
<td>scvx_coverpoint</td>
</tr>
<tr>
<td>Coverage state bins</td>
<td>bins()</td>
</tr>
<tr>
<td>Illegal bins</td>
<td>illegal_bins()</td>
</tr>
<tr>
<td>Ignore bins</td>
<td>ignore_bins()</td>
</tr>
<tr>
<td>Triggers sampling the covergroup</td>
<td>sample()</td>
</tr>
</tbody>
</table>

Table I: Basic Language constructs for functional coverage in UVM-SystemC-AMS (scvx)
Table II: Continuous Distribution Function available for real values in UVM-SystemC-AMS (scvx)

<table>
<thead>
<tr>
<th>Distribution function</th>
<th>UVM-SystemC-AMS (scvx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal distribution</td>
<td>scvx_normal_distribution</td>
</tr>
<tr>
<td>Uniform distribution</td>
<td>scvx_uniform_distribution</td>
</tr>
<tr>
<td>Bernoulli distribution</td>
<td>scvx_bernoulli_distribution</td>
</tr>
<tr>
<td>Piece-wise linear probability distribution function</td>
<td>scvx_piecewise_linear_probability_distribution</td>
</tr>
<tr>
<td>Discretized probability distribution function</td>
<td>Scvx_discrete_probability_distribution</td>
</tr>
</tbody>
</table>

Figure 9 shows the histogram of the normal distribution with real values. To estimate the coverage, the range of each random variable is divided into intervals. The number of draws (tries which have fulfilled the constraints) is computed in each interval. The functional coverage is based on these figures.

![Figure 9: Histogram of a Normal distribution featuring real values](image)

The random variable $a$ is declared in Listing 6. As an example, the member function `set_distribution` defines the distribution `uniform_real`, requiring the parameters min (15.3) and max (40.2) for random variable $a$.

```cpp
scvx::scvx_rand<real> a;
a.set_distribution(scvx_distribution::uniform_real(15.3, 40.2));
```

Listing 6: Setting a distribution function for a random variable

To illustrate the randomization for real values with constraints, a simple case is presented in Figure 10. The DUT is a SPI-controlled programmable filter. In order to verify the DUT with UVM-SystemC-AMS, 4 UVCs have been instantiated in the test environment: a sinusoidal signal generator (U1), an input monitor U2, a power supply generator (U3) and a SPI driver to generate SPI compliant commands (U4).

![Figure 10: Test of a programmable filter with the proposed UVM-SystemC-AMS](image)

Connection between UVCs and the DUT follow the interface configuration mechanism provided by UVM-SystemC-AMS. The scoreboard is responsible for comparing the expected performance of the filter (here the gain) with the actual one.

![Figure 11: Simulation result, showing SPI commands, power supply, input/output of the DUT and performance indicator (gain). The input signal is a sinusoidal one with randomized frequency. The performance indicator is the gain of the filter. The green crosses show the DUT gain after each sequence item. The red curve shows the reference value computed in the scoreboard.](image)

Figure 11 presents a simulation result of the SPI controlled programmable filter. Note that the first 4 signals at the top of Figure 11 use a dedicated timescale, showing the setting of the SPI commands. The next curves use another timescale, dedicated to illustrate the randomization of real values. The input stimuli of the programmable filter is a sinusoidal signal with randomized frequency. Each frequency corresponds to a certain sequence item. At the end of each sequence item, the gain of the filter is computed from a set of collected input and output signals and the result (green crosses in Figure 11) is sent to the scoreboard. To estimate the coverage of the test, the frequency range has been divided into 12 intervals and a minimum number of frequencies is required in each interval. At the beginning of the test execution, the coverage is low, as shown in Listing 7 whereas at the end of the test, the coverage has increased, though below 100% in our example to illustrate the UVM report.
For detecting and handling system errors, safety critical systems typically contain various mechanisms creating arbitrary analog vector interfaces the test bench contains analog interface stimulated by driver modules and SystemC models depending on the accuracy required. As TDF data flow models or as electrical linear network mechanical components are modeled in SystemC. The analog components and the RTL level and transistor level representations allow a significantly higher simulation performance than specification uses abstract behavioral simulation developed to meet the system requirements, i.e. to enable the complete system to work as intended, i.e. to verify that these mechanisms operate as intended. In order to enable tests simulating the corresponding failure scenarios, the test bench implements several drivers for injecting failures into the system.

Figure 12 shows the overall system and test bench including error injection for emulating a short.

The availability of a well structured library of verification components based on a standardized methodology significantly enabled to develop easily further tests and test benches, thereby reducing time and cost for test development.

Further the UVM methodology enabled the construction of fully automated regression test suites.

The tests have been written in a way enabling to re-use them for the validation of RTL design when it becomes available by coupling the HDL simulator simulating the system and a SystemC simulator simulating the test bench.

VII. VALIDATION USE CASE

The methodology was applied on an airbag SoC, as an example for an AMS product, with high complexity and full integration, to show, how UVM can be used on a HIL system to verify a FPGA prototype.

For an effective HIL simulation a tight coupling between
the verification of the virtual prototype and laboratory validation of the DUT with the HIL had to be established, as depicted in the workflow in Figure 13. Thereby the re-use of the test environment, with its test and check sequences are critical to allow an early move of the implementation onto the prototype to accelerate and aid verification and validation.

The DUT evolves from a behavioral SystemC-AMS description to a mixed abstraction model. It is then synthesized towards an FPGA prototype and results in the first hardware setup [39].

While the DUT evolves, the test environment should remain as similar as possible to guarantee functional correct testing throughout the steps. This constancy also ensures that a minimal effort is spend on test environment creation. Figure 14 illustrates the implemented software and hardware layers of the verification and laboratory validation architecture and emphasizes the re-use.

As depicted in Figure 14, the test environment is based on UVM-SystemC represented by the three upper most layers. The high abstraction test description provides test vectors to the specific UVM agent via the virtual sequencer, which in turn provides specific driver and monitor components (not shown) to connect the test environment to the corresponding DUT implementation. During the verification, the monitor and drivers are function implementations, connecting and wrapping the modeled DUT via interfaces to the UVM-SystemC test environment. The connection in case of validation has to connect through the specific HW of the HIL-Tester.

The HIL-Tester can directly build and run the native UVM-SystemC based test environment. It was implemented using an ARM based Zynq-SoC [40], build on a Zedboard [41], running a real-time Linux scheduling the test environment.

Figure 15 demonstrates the laboratory setup of the airbag SoC DUT FPGA prototype. To realize the HIL-Tester concept, the main micro-controller is replaced now by the Zedboard with the Xilinx Zynq based ARM-SoC, on which the UVM-SystemC test bench is re-used to emulate the physical stimulus to drive the DUT FPGA Prototype.

With this implementation, it was possible to shorten and focus the effort of a root-cause analysis in the virtual prototype through the re-use of the test sequences obtained from the HIL simulation within the computer based verification. As the test sequences are re-used, including the test vectors, a direct mapping between the verification and validation activities becomes possible.

Including the HIL in the verification process increased the speed of verification and the bug detection possibilities at an early point in time, thus reducing development costs.

The UVM-SystemC-AMS test bench can be directly re-used and it can additionally be extended for long-term tests and stress tests, which are impractical to do with classical system computer-based simulation.

With the new setup, the whole prototype system is checked at real-time speed against real sensor network used later in the application instead of sensor model simulation.

VIII. CONCLUSION

We have explained how a unified methodology for the verification of systems having interwoven AMS/RF, HW/SW and non-electrical sub-systems and functions can be achieved. As of today, there only exist verification methodologies addressing either AMS/RF or digital HW/SW functions. The clear separation of the DUT and its verification description, which has been established in the last years for complex digital systems, has been extended to analogue mixed signal ones. Thus the essential unification of analogue and digital verification is made possible. We have shown that the components and the scenarios designed for the simulation-based verification can be reused for the validation based on measurements of the complex DUT.

To support the new methodology inspired by UVM, we have introduced language constructs and generic verification components in SystemC and its AMS extensions to create application scenarios consisting of stimuli generation and response checking. We have called this library UVM-SystemC-AMS. The generic verification components with
their reuse features have been illustrated by two industrial use cases: the first use one showing UVM-SystemC-AMS based verification of a complex SystemC-AMS design, the second one showing the verification/validation with HIL using a FPGA prototype. Moreover, by extending the IP-XACT packaging process for facilitating its exploitation in industrial design flows, we aim to push the reuse operation.

The implemented UVM-SystemC-AMS library introduced in this paper is under Standardization within the Accellera Systems Initiative.

REFERENCES


