# Reuse doesn’t come for free - learnings from a UVM deployment

Sumeet Gulati, Senior Techincal Leader, NXP Semiconductors, Bangalore, India
(sumeet.gulati@nxp.com)

Srinivasan Venkataramanan, Chief Technology Officer, CVC Pvt.Ltd, Bangalore, India
(srini@cvcblr.com)

Azhar Ahammad, ASIC Design Verification Engineer,CVC Pvt.Ltd, Bangalore,
India(azhar@cvcblr.com)

Ketki Gosavi, Trainee, NXP Semiconductors, Bangalore, India (ketki.gosavi@gmail.com)

Saumya Anvekar, Senior Design Engineer, NXP Semiconductors, Bangalore, India
(saumya.anvekar@nxp.com)

***Abstract*— Mobile audio chips are critical for modern day electronic hand held devices including cell-phones, tablet laptops etc. Given the very fast turn-around time of these systems, a lot of design IPs gets reused in these systems. IP reuse on the design (at Register Transfer Level - RTL) domain was made possible through various initiatives such as CoReUse (Ref: 5), RMM (Reuse Methodology Manual) etc. for more than a decade now. However, the same level of reuse has not yet been achieved on the verification of these IPs (Intellectual Property), sub-systems and SoCs (System-on-Chip). While SystemVerilog and UVM (Universal Verification Methodology) has provided solid framework to build test benches and a set of guidelines for keeping them reusable, in practice there is lot more yet to be done to achieve good amount of reusability. In one of our projects, verification of these designs at IP level was previously done through legacy test benches written in TCL (Tool Command Language) and other languages. New blocks are being verified using modern SystemVerilog based benches with UVM framework. As our team adopted SystemVerilog& UVM slowly into the design cycle we stumbled upon few interesting use cases that prevented reuse from IP level to SoC. While UVM has good coding guidelines and features for enabling reuse throughout this project we realized that "reuse" is much more than just base classes and features like factory, register models in UVM. In this paper we present some of our deep learnings earned during working in the trenches deploying UVM. We also share our deep analysis on the register model randomization features, the ability to control randomness from top level etc.**

*Keywords— UVM, Reuse, UVM Registers, RAL (Register Abstraction Layer) Models, Optimal Design Partitioning, Randomization, SoC, SystemVerilog, Constraints*

## I. INTRODUCTION(STYLE: HEADING 1)

Design and Verification of modern day mobile audio SoCs (System-on-Chip) and associated IPs (Intellectual Property) is very challenging, and innovative task for engineers! UVM (Universal Verification Methodology) standard has become very widely adopted for verifying complex designs at various levels of abstraction. In a nutshell, UVM defines a set of templates and a set of coding guidelines to keep verification environment reusable across levels of verification, across projects etc. However, as we deployed UVM into few IPs in our recent project, we realized there is more to reuse than just what UVM prescribes. In this paper we will share some of our finer learnings during the process. We also leveraged on several UVM features that are not well spoken about such as register model reuse, factory override with and without replace etc.

## II. SOC ARCHITECTURE

Our chip is a mobile audio Class D amplifier SoC and is part of our family of products that reuse several IPs over many years. Figure-1 below shows a higher level view of the SoC DUT (Design Under Test).

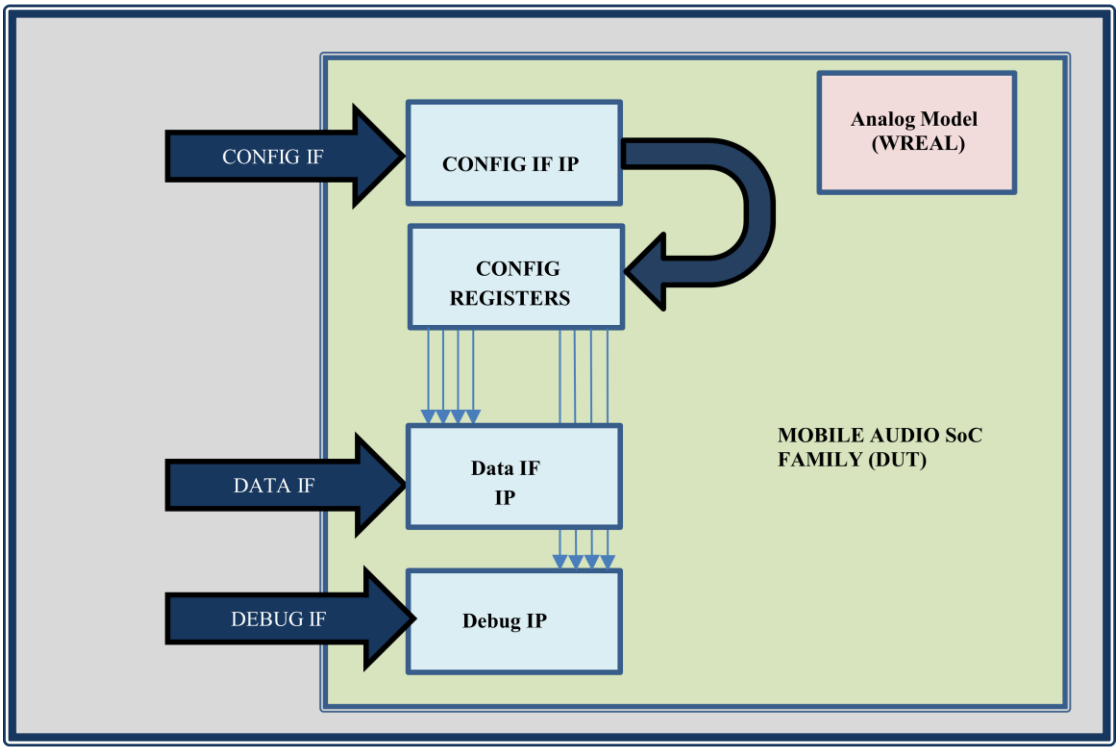


Figure 1. SoC architecture higher level view

The actual Mixed Signal SoC has many more interfaces and one of which is I2C (Inter-Integrated Circuit) interface, but for the sake of this paper we will use the simplified view as above. Also the design is heavy Analog & Mixed Signal (AMS) in nature with WREAL (Wire-REAL) models being used to describe the analog portions. In this work we will focus on the digital part though.

## III. VERIFICATION REQUIREMENTS

With the Mixed Signal SoC being fairly complex, the individual IPs are verified in a stand-alone IP verification environment and then integrated to the top level SoC verification environment. Also given that this Mixed Signal SoC is part of a family of products, several IPs get reused across generations of the chip. One of the key requirements for this reuse is to be able to abstract individual IP level configuration sequences at the SoC level. The individual IP owners understand each IP deeply and the top level integrator may not be fully aware of complex sequences for each IP to configure correctly at top level. This means that the IP level sequences shall be maximally portable. And given that SoC level verification will have different scenarios than IP level, the IP configuration sequences shall be flexible enough to be tweaked from top level should the need arise; For instance a sophisticated test at SoC level may want to turn off all but 2 register randomization in a UVM sequence for an IP.

## IV. INITIAL IP VERIFICATION PARTITIONING

Given that this design is part of family of chips, legacy test environments exist based on VHDL (VHSIC (Very High-Speed Integrated Circuit) Hardware Design Language), VERILOG and driven by TCL (Tool Command Language) test cases. While our team understood the advantages of SystemVerilog and UVM based approach, the changes had to be done incrementally. So the first cut verification for an IP was done as shown in Figure 2 below following exactly how it was partitioned in the previous generations. Using a generic name as DATA IF IP (Data Interface IP) the figure shows two important input interfaces to the DUT:

- I2C based configuration values

- Data interface driving audio data (a custom interface)

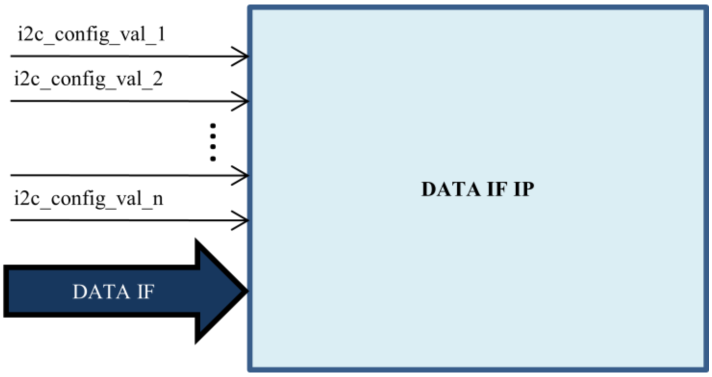


Figure 2. Block diagram of a single IP

At the RTL level of DATA_IF_IP the I2C configuration values are fed in as direct inputs. At the SoC level these signals shall be connected to an I2C control register block in RTL as shown in Figure 4 in next section.

## V. Design Partitoning at IP Level to Enable Verifiaction Reuse

One of the significant limitations of legacy environment was the inability to truly reuse IP level scenarios directly at SoC level. This is because the IP level sequences were directly tweaking the control/config pin values than via the I2C configuration registers. This led to duplication of all register programming sequences per IP at SoC level leading to:

- Redundant work
- Wasted debug cycles due to wrong configurations at SoC level (All SoC integrators are not familiar with every IP sequence, for instance)

So our team realized that mere use of UVM with SystemVerilog alone doesn't always enable reuse. After some brainstorming involving verification lead and expert UVM consultant, we decided to re-partition the design at IP level and include SoC Level I2C control block with relevant or all registers at every IP level DUT as well. A modified partition looks as in Figure 3.

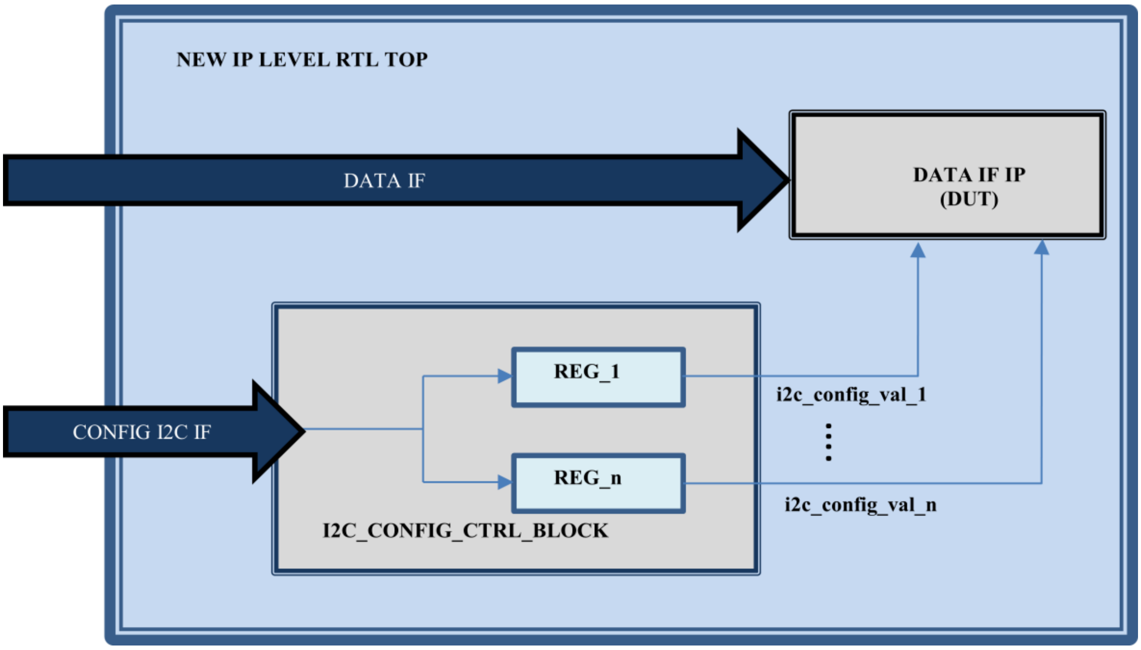


Figure 3. Re-partitioned Data Interface IP including I2C control logic

## VI. NEW IP LEVEL VERIFICATION ENVIRONMENT

With the design re-partitioned as in previous section, we used the same Data IF UVC (Universal Verification Component) as used in the previous environment and added a new I2C UVC that our team has built for top level SoC environment. The new verification setup at IP level looks as shown in Figure - 4.1

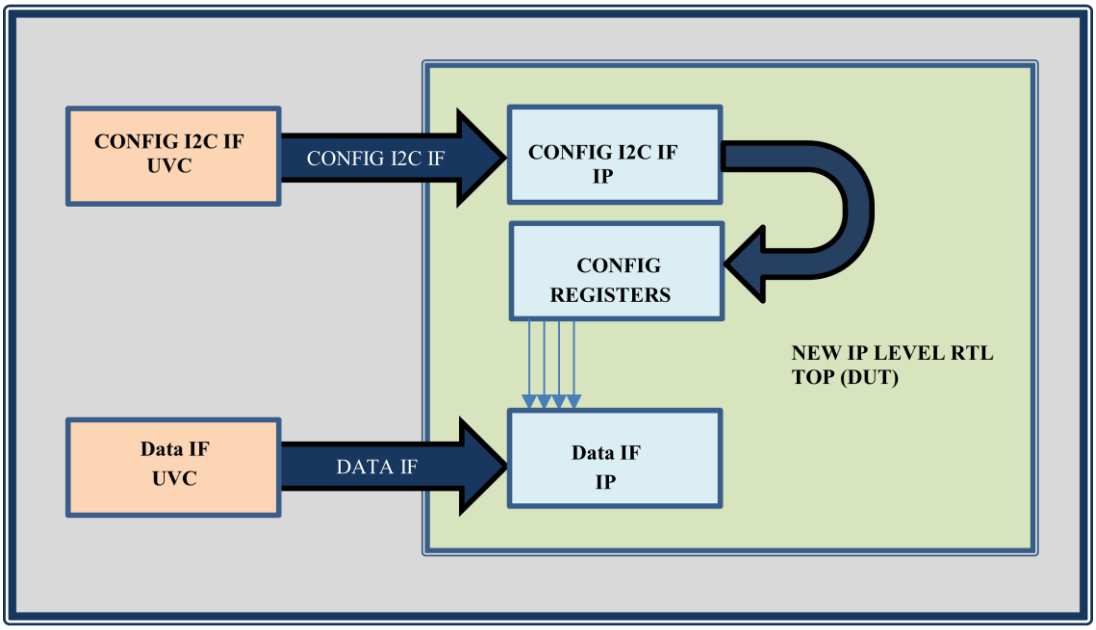
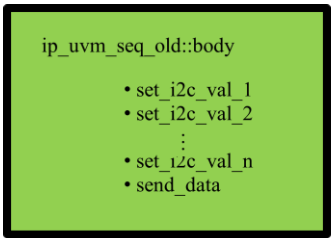


Figure 4. Modified IP level verification environment with 2 UVCs

Note that the motivation to do this re-partitioning was reusability of IP level sequences than the UVCs themselves. Let's see how this is achieved. Let's first look at how the old UVM sequence looked for IP level. Figure-5 below shows a pseudo-code for old IP level UVM sequence:



```
ip_uvm_seq_old::body
          • set_i2c_val_1
          • set_i2c_val_2
                 ⋮
          • set_i2c_val_n
          • send_data
```

Figure 5. Old IP level sequence

The sequence body simply specifies the configuration values for needed i2c_val_n inputs. A simple, dummy UVM driver attached to this sequence (via sequencer) then drives the configuration, control and data interface. Note that there was no real I2C interface involved at IP level in the previous setup. This worked fine with IP level verification. However, since at SoC level there is a real I2C interface, the basic sequences developed at IP level were not reusable at SoC level Config I2C IF UVC driver, before.

As discussed in previous section about the re-partitioning for verification reuse, the new design partition included I2C configuration block in RTL. Following that change to the DUT partition, the new, modified IP level verification environment now included a register model. With necessary UVM RAL (Register Abstraction Layer) model added to individual IP level verification environment, the IP level sequences programmed I2C registers that are needed to configure the specific IP (in exact order as per individual IP requirements). A typical IP level sequence involved writes to I2C block to achieve the same scenario as shown in Figure-5. Modified IP level sequence with I2C block is shown below in pseudo-code in Figure-6 below:

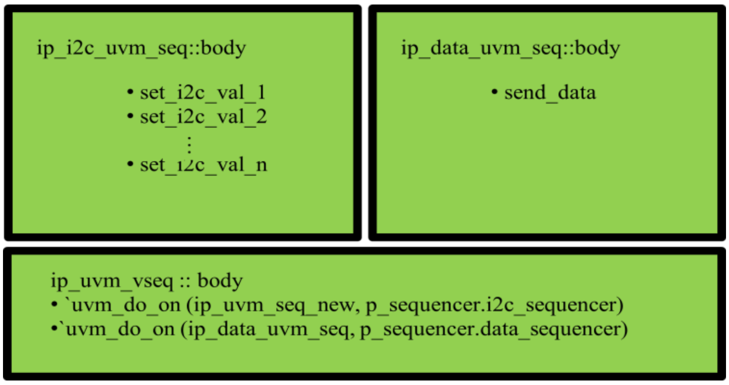


Figure 6. New modified IP level sequence with I2C model

With the new IP level sequences using I2C interface, even the IP level UVM sequences turned out to be virtual sequences with I2C sequence running on a real I2C sequencer and a data sequence running on a data interface sequencer.  The IP level sequences now mimic the programmer's model of the IP and hence is easy for reviewing by system architects as a side benefit.

At the SoC level, these I2C RAL sequences were plug-and-playable as the I2C interface was visible at the top level design. On the UVM front, RAL provides an easy way to manage the offset of individual register blocks by handling multiple register maps at SoC level.

## VII. USING CREATE() INSTEAD OF NEW() IN UVM MODELS

Given the upfront guidelines set by management to keep the code reusable for future changes, the engineering team took care to ensure all UVM coding guidelines were followed. There are two significant phases during the project where these guidelines had to be ensured - first during the environment development and the other during developing sequences. We will elaborate one significant guideline here. In UVM, factory provides the necessary infrastructure to keep code reusable. In a nutshell the factory involves three steps:

1. Registering the classes with factory table
2. Consulting the factory table during construction
3. Setting overrides on need basis

The first step is achieved through the use of handy macros: `uvm_component_utils_begin (i2c_driver) and `uvm_object_utils (data_if_config_reg) - one for registering components and the other for sequences/register models. There are code generators available to do this part automatically (Ref: 3).

The second step is to be done during construction of every object within UVM and is very critical to keep the code reusable. Below is the standard constructor of an uvm_component:

function new (string name, uvm_component parent);

UVM recommends not to change the prototype of this constructor in any derived class as the UVM base class would call this new() internally during object creation. User code should instead call a proxy method named create() that in-turn calls the new() after consulting any overrides set by end user code.  At a high level, this create has an “if” statement to go and check in the factory table (as populated by step-1 above) to see if there was an override and returns derived object or the base object otherwise. A pseudo code describing this behavior is shown below:

```
if (factory_override_table.exists(vlb_drvr))
   create = derived_class::new();
else
create = base::new();
```

A typical usage of create() instead of new() looks as shown below:

```
vl_ctrl_reg = vl_ctrl::type_id::create("vl_ctrl_reg");
```

With the create() routine, we now have a mechanism to set an override in a table/registry and swap a base class with a derived class. This is core to reusing components and transactions in UVM including register models. For instance a test writer can replace the basic register with a derived class having additional constrains on need basis.

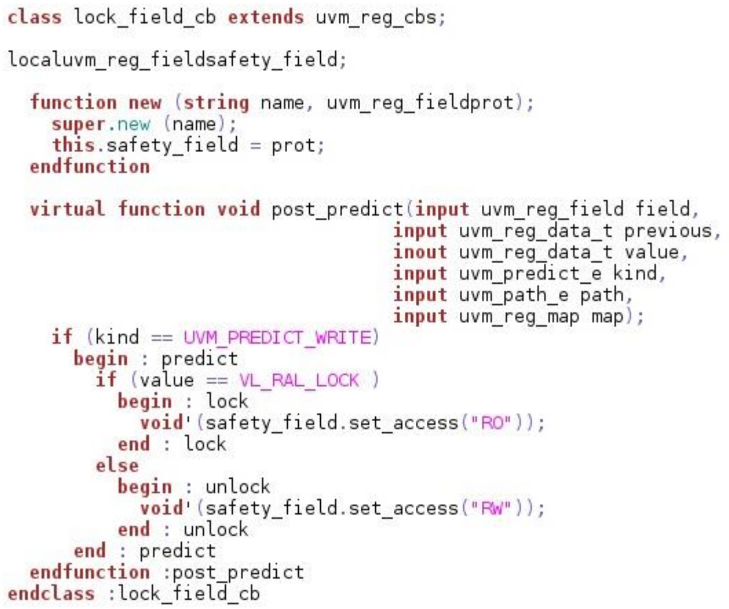
```
reg_test::set_type_override_by_type (.original_type(vl_ctrl_reg::get_type()),
                                      .override_type(vl_ctrl_derived_reg::get_type()));
```

At SoC level, similar factory override can be done to leverage on individual IP owner’s deep know-how on the IP programming sequence and yet tailor to a SoC verification scenario.

## VIII. TWEAKING UVM RAL MODEL PREDICTORS

There is some amount of hype around UVM that tends to imply - UVM knows how to verify your system - however the reality is that UVM only provides a reusable framework and users need to write good amount of code on top to make it usable. A classical case is UVM RAL predictors. There is some amount of "auto prediction" that is supported by UVM out-of-the-box. However given modern designs having very advanced register features including volatile registers, safety controlled registers etc. this auto-prediction does not work in such cases.

In UVM terminology, access to a register/field is set once during the model generation to be RW/RO/WO etc. If there is a change in that access policy dynamically during a simulation that needs to be modeled by user code. This becomes critical for the predictor to work well in our environment as there are safety related registers that change access privileges based on some control register settings. UVM provides API to achieve this via uvm_reg_field::set_access(). It becomes interesting when such change in access is dependent on some other register's setting. Our team used post_predict callback in UVM registers to model this. A pseudo-code to achieve this behaviors is shown below. Assume that we have a register named "vlb_control_reg" and another register named "vlb_safety_reg" that needs to be locked/unlocked based on the vlb_control value.



```
class lock_field_cb extends uvm_reg_cbs;

localuvm_reg_fieldsafety_field;

  function new (string name, uvm_reg_fieldprot);
    super.new (name);
    this.safety_field = prot;
  endfunction

  virtual function void post_predict(input uvm_reg_field field,
                                   input uvm_reg_data_t previous,
                                   inout uvm_reg_data_t value,
                                   input uvm_predict_e kind,
                                   input uvm_path_e path,
                                   input uvm_reg_map map);
    if (kind == UVM_PREDICT_WRITE)
      begin : predict
        if (value == VL_RAL_LOCK )
          begin : lock
            void'(safety_field.set_access("RO"));
          end : lock
        else
          begin : unlock
            void'(safety_field.set_access("RW"));
          end : unlock
      end : predict
  endfunction :post_predict
endclass :lock_field_cb
```

The above callback models the dynamic control to the field named "safety_field" based on value of another register content. The above callback is then integrated to the register model during the model configure step as shown in the pseudo-code below:

```
function void configure(vlb_control_regcontrol_reg, vlb_safety_regdata_reg);
  lock_field_cblock_cb;
  lock_cb = new("lock_cb", data_reg.safety_field);
  uvm_reg_field_cb::add(control_reg.ctrl_field, lock_cb);
endfunction : configure
```

## IX. PRELIMINARY RESULT, FUTURE WORK & CONCLUSIONS

While this effort is still in progress we hereby provide some of our early stage results and projected benefits in the near future.

- IP level verification has to be thought upfront on requirements of future reusability.
- It is possible that some of the IP level work has to be re-factored, re-engineered at a small cost – keeping in view of the bigger benefits that SoC level verification will yield.
- The significant benefit we see is the ability to leverage on IP level knowledge directly at SoC level in the form of UVM sequences.
- Several real life learnings were obtained through intense consulting with UVM experts and whiteboard discussions that go far beyond a typical UVM training class, online material etc.
- Verification Leads will need to involve Design Architects early in the verification brainstorming sessions to arrive at optimal choices/partitions for reusing verification – this is quite a significant revelation for us as we have heard the term “DFV – Design For Verification”, but this experience shows something at a higher level of “architecting IP partitions for easier reuse at SoC level verification”.
- RAL Randomization, Reusable score boarding, UVM-ML and UVM-AMS is the focus for our future work.
- We have found limitations in RAL randomization which further limits the reusability of sequence from IP level to SoC level. These ranges from turning on and off randomization, effectively using incremental constraints etc, hampered by the inherit limitations of the UVM Library source code.
- Multi Language (ML) in Design is very common these days since the sources of IPs are so diverse hence UVM-ML support is crucial.
- Analog constitute major portion of our Mixed Signal SoC and UVM infrastructure needs to be reusable and work seamlessly for Digital simulations, Digital Mixed Simulation (DMS) and Analog Mixed Simulations (AMS)
- Finally, we stand by our title – Reuse isn’t free!

## REFERENCES

[1] Accellera UVM standard

[2] SystemVerilog IEEE 1800-2012 LRM.

[3] DVC_UVM template generator - http://www.verifworks.com.

[4] I2C specification - http://www.nxp.com/documents/user_manual/UM10204.pdf.

[5] CoReUse guidelines, internal NXP.