2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Using an Enhanced Verification Methodology for Back-to-Back RTL/TLM Simulation

Frank Poppen and Ralph Görgen, OFFIS Institute for Information Technology, Germany
Kai Schulz, Andreas Mauderer and Jan-Hendrik Oetjens, Robert Bosch GmbH, Germany
Joachim Gerlach, Hochschule Albstadt-Sigmaringen, Germany

***Abstract*—Critical electronic systems as in the automotive domain have to comply with the functional safety norm ISO 26262 and make extensive verification in the development process mandatory. Still, their success as a product and the return on investment are at risk if the time to market window should be missed. With test and verification consuming main effort in the development process, hardware/software co-design through application of virtual prototypes is an option to parallelize development tasks and shorten time to market significantly. Virtual prototypes can contain heterogeneous components potentially implemented at different levels of abstraction to support earliest concept evaluation and software development without the target hardware platform available. Achieving virtual prototype integration across mixed levels of abstraction is a challenge though, which is addressed in this work. We introduce a register-transfer abstraction level verification method and how it is enhanced by a technique for the simulation of transaction-level models in order to enable the mixed-level simulation of hardware and software systems and evaluate it in a Back-to-Back verification scenario.**

Keywords—design methodology; virtual prototyping; digital signal processors; automotive electronics.

I. INTRODUCTION

High-end technology applications such as internet of things or automated driving cars have a raising demand for performance, functionality and consequently complexity in the design of semiconductor products. We are past the time when silicon chips contained a simple microcontroller and a few IO peripheral components. Today's challenge is to handle complete and complex Systems-on-Chips. A single die of a few square millimeters gives room for several subsystems containing processor cores with memories, peripherals and analog components connected by hierarchies of on-chip buses. Guaranteeing properties of safety, sustainability and comfort is mandatory, which requires consistent verification of quality along every stage of the development process. Literature speculates that this combination of demands leads to a 70% verification effort of overall design costs, a number that is not really proven, but for sure must be avoided for future products.

An established approach to address verification is the application of virtual prototypes (VP) to parallelize development tasks and shorten time to market significantly. A VP is an integration of executable models of any relevant component under development and enables the early simulation and verification of a product's functionality. Unfortunately, there is no such thing as one size fits all. Verification engineers need to choose the abstraction level to combine what fits best for the verification task at a given development state. Along the design and verification process the VP and its components are prone to constant change and reorganization of abstraction levels, for the reason that a component may not be implemented in all details yet or that it is not relevant for a certain test scenario and should be abstracted to increase simulation performance. Of course, a component needs to show a consistent behavior independent of, but with respect to, the chosen abstraction level. Back-to-Back (BtB) comparison is a verification setup to guarantee this and which also requires mixed-level abstraction simulation techniques. In this paper, we introduce the enhancements of our "Integrated Functional verification Script environment" (IFS) verification method [1-4] to be able to stimulate and verify mixed-level VPs. In the context of this work, we used an automotive Digital Signal Processor (DSP) for sensor data processing as a reference for applicability.

The remainder of this document is structured as follows. In Section II we give an outline on the state-of-the-art for multi-abstraction-level verification methodologies followed by Section III describing our mixed-abstraction-level communication concept. Section IV describes our practical results for which we applied our new

concept to an automotive DSP. This paper finishes with our conclusions in V followed by acknowledgements and the list of references.

II. STATE OF THE ART

Our verification methodology named IFS was continuously enhanced from VHDL with VHDL-AMS [1] to SystemC [2], MATLAB/Simulink [3] and further on to SystemVerilog and the Universal Verification Methodology (UVM) [4]. Major aspects of the test bench architecture as defined by IFS can also be found in the established standard UVM that has its roots in the Open Verification Methodology (OVM) and the Verification Methodology Manual (VMM). The idea is to conceptually separate the Design Under Verification (DUV) from a test setup, which again is separated into test bench and test cases. The methodologies support automated directed or constrained random test pattern generation. This enables reuse of design IP, test IP and test patterns and dramatically improves design and verification efficiency between projects.

UVM though comes with mentionable overhead as stated by [6]. A typical UVM training class has an average length of four days. This assumes a working knowledge of the SystemVerilog language as their starting point. According to the authors of [6], a typical formal training program for engineers already familiar with Verilog would consist of a four day training class to teach the verification features of the SystemVerilog language, followed by four days to teach UVM. Adhering to the authors' experience it takes about another half a year of practical work with UVM to master it completely. As a solution they introduced Easier UVM "…a comprehensive set of coding guidelines for the use of UVM together with an open-source UVM code generation tool. The code generator creates project-specific boilerplate UVM code according to the Easier UVM guidelines."[6] The idea behind this is the support of a best practice, and to avoid the most common pitfalls.

As we state in [4], the IFS approach is even less complex than Easier UVM and simpler to apply. After an afternoon introduction (analog) designers and system integrators are able to use test benches and create command files for own test cases. All stakeholders in the development process, digital designer, analog designer, verification engineer and system engineer make use of the same, simple IFS command language to create self-checking test cases. Reuse is crucial for the exponentially growing verification task and requires a structured test bench design. IFS and UVM are both methodologies that specify such a structure well suited for test bench reuse. Computational behavior is separated from communication so that both aspects can be modelled individually according to verification needs (compare with Figure 1). Simultaneously, separation of communication and computation allows for an effortless change of protocols without touching functional behavior of the computational core. This is an important feature for generic IP reuse.

UVM in this matter makes use of communication abstraction by means of Transaction-Level Modeling (TLM) [7] and Register-Transfer Abstraction Level (RTL). Our IFS methodology bases on SystemC which also includes TLM as a standard. Nevertheless, our previous work did not focus on TLM for IFS. This work demonstrates in a practical application how TLM is also applicable for IFS. In the automotive domain we use TLM together with RTL for co-verification in a way that our applied method is conceptually equivalent to the BtB comparison testing between model and code according to the standard ISO 26262 as named in [8]. The authors refer to automatic software code generation from executable models and their parallel simulation with simultaneous comparison. Nevertheless, the concept also holds for hardware development across different abstraction levels where models need to be verified against each other after each refinement step. This includes manual reimplementation of algorithmic C/C++ golden models in a RTL

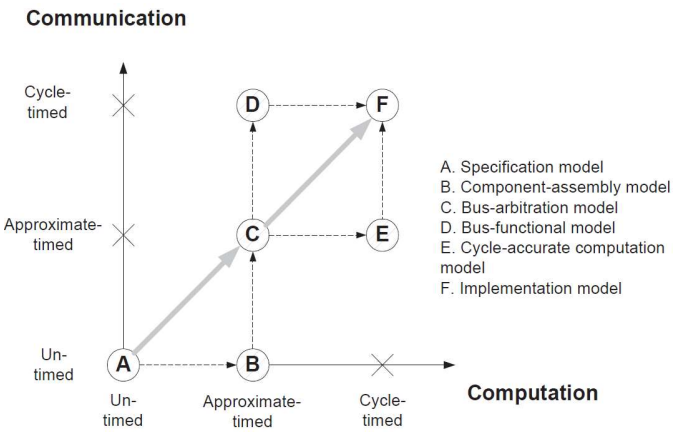

Figure 1. System modeling graph by L. Cai and D. Gajski [7].

Hardware Description Language (HDL), or might also be applied for automatically synthesized Gate-Level representations.

The authors of [9] describe a similar methodology for the verification of SystemC models in RTL test benches. They verify a TLM module by comparing the output with a "golden RTL module" and its RTL test bench. Initially, the RTL test bench is simulated with stimulation vectors. The sequence of activities and results is traced and later transformed into TLM transactions by use of proprietary scripts. The TLM module is then stimulated by these transactions and the responses are compared to the previously recorded. MathWorks is following the same approach with their tools MATLAB/Simulink and the HDL-Verifier toolbox [10]. The tool is capable of automated SystemC/TLM code generation from a computational model specified in the block-based graphical Simulink language. The tools also generate a test environment that firstly executes and traces the Simulink model and secondly generates, compiles and executes the SystemC/TLM model to apply the same inputs as before and check for same results on the outputs.

In this matter, neither [9] nor [10] provide BtB comparison as stated in [8]. In the work presented here, we simulate both instances of the DUV at RTL and TLM simultaneously in a real BtB comparison and verify the results at runtime without the need to previously record any activity.

III. Communicating Across Levels of Abstraction

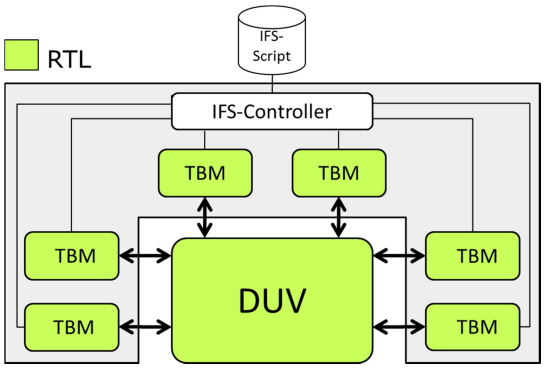

Figure 2. IFS test bench environment for RTL simulation.

As stated in Section II, our IFS methodology has its roots in VHDL and the RTL. When IFS was introduced, UVM and even TLM weren't standardized and verification at RTL was state-of-the-art and feasible. Figure 2 depicts the original concept which separates the DUV, test bench and test cases as it is state-of-the-art today. Test cases are defined separately in script files by means of a proprietary IFS scripting language. The script controls the actions of Test Bench Modules (TBM) which stimulate and monitor the in- and output port signals of a DUV. A descriptive example of the IFS scripting language is part of [4] and [5] for further reference. We note that a DUV can be as simple as a single component, but could also be a completely integrated VP. The abbreviations DUV and VP are therefore replaceable for the following discussion. Communication protocols are modelled at signal level. Increased verification complexities and reuse of third party verification IP require an abstraction of communication to TLM.

We describe how we enhanced our TBM concept to be easily configurable for different levels of communication abstractions. This has been done in two aspects: With new synchronization features and a stricter separation of communication and computation.

The first is required because of different timing accuracy in RTL and TLM components that may lead to incomparable simulation results or even incorrect execution orders. Interrupts have been available already before to ensure the execution of a command at a specified point in time or with a specific period. In addition to that, we introduced events in the test case description language (IFS-Script). This allows to notify events in the script and to force TBMs to wait for the

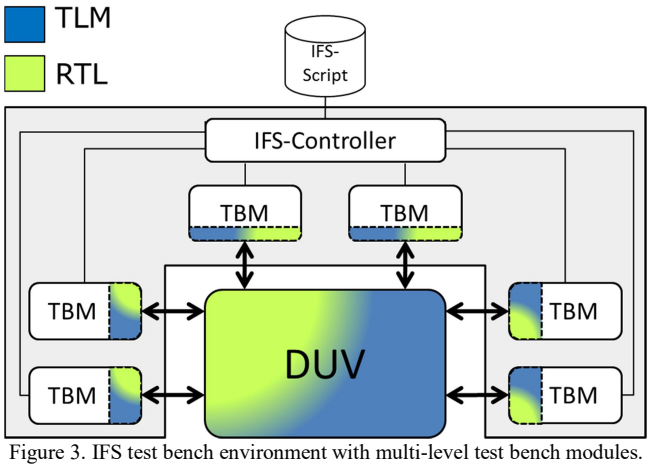

Figure 3. IFS test bench environment with multi-level test bench modules.

notification of an event. Together, the two features allow ensuring causality and the correct temporal behavior of the entire test bench even if individual components are modeled at different abstraction and timing accuracy levels.

```
struct master_if_base
{
  // IF methods
  virtual
  void if_write( unsigned int, unsigned int ) = 0;
  virtual
  unsigned int if_read( unsigned int ) = 0;
  // Module methods
  virtual
  void ack_write_msg( int, unsigned int ) = 0;
  virtual
  void ifs_error( const char * ) = 0;
};
```

Figure 4. IF base class.

The second, strict separation of communication and computation, is required to reuse TBMs at different abstraction levels. We separated the TBM's command implementation from its interface implementation. This allows a quick and easy exchange of the communication interface and abstraction levels depending on the development state of different components of the DUV or the needed simulation detail for chosen verification scenarios. While components of interest are verified in detail using RTL abstraction, other components may remain to be more abstract at TLM abstraction to increase simulation performance as shown in Figure 3.

To support the easy switching of abstraction levels in TBMs, we introduced a further level of separation between communication and computation. In general, computation is already implemented in an abstract way in TBMs because they do not need to be synthesizable or deployable on the target platform. The communication however has to match the abstraction level of the DUV to be connected correctly. The additional interface level provides a number of methods to the computation part. These methods realize all required communication features so that the computation part does not need to access the actual interface as ports, TLM sockets etc. We can then provide different implementations of these methods for the different abstraction levels. In the following, we will describe this concept by using a bus master TBM as an example that provides various commands to trigger sequences of read and write accesses to a bus.

```
struct master_rt_if : public master_if_base
{
  // Ports
  sc_in< bool >  clk;   // clock
  sc_in< bool >  rst;   // reset

  sc_out< bool > req;   // master request
  sc_in < bool > gnt;   // grant from arbiter
  sc_out< bool > rreq;  // read request
  sc_out< bool > wreq;  // write request
  sc_out< sc_dt::sc_bv<16> >  addr;  // address
  sc_out< sc_dt::sc_bv<32> >  wdata; // write data
  sc_in < sc_dt::sc_bv<32> >  rdata; // read data
  sc_in < bool >  ack;  // acknowledge

  void if_write( unsigned int , unsigned int );
  unsigned int if_read( unsigned int );

  master_rt_if();
};
```

Figure 5. RT interface implementation.

The general technical idea is to have a class encapsulating the communication. The TBM is derived from that class to be able to use the communication facilities. Firstly, we extract the communication patterns from the TBM's behavior. In our case this is writing a given value to a specific address and reading the content from a specific address. Then, we define an abstract base class `master_if_base` for the interface (Figure 4). It contains method prototypes supposed to implement the communication pattern, in our case `if_write` and `if_read` to perform the write or read access. In addition to the communication methods, there might be more functionality that should be available in the communication interface. But this functionality has to be implemented in the context of the TBM. Prototypes for such methods are declared here as well.

```
struct master_tl_if : public master_if_base
{
  sc_in< bool >  clk;   // clock
  // TLM-2 socket, defaults to 32-bits wide, base
          protocol
  tlm_utils::simple_initiator_socket<master_tl_if>
          socket;

  void if_write( unsigned int , unsigned int );
  unsigned int if_read( unsigned int );

  master_tl_if();
};
```

Figure 6. TL interface implementation.

Next, we define two derived classes to implement the RTL interface (Figure 5) and the TL interface (Figure 6). The declaration of the actual interface resides here, signal ports in the RTL version and a TLM socket in the TL version. In addition, the actual communication protocol is hidden behind in the definitions of the `if_write` and `if_read` methods.

```
unsigned int
master_rt_if::if_read( unsigned int address )
{
    // set request
    req.write(true);

    // wait for grant
    wait(gnt.posedge_event());
    wait(clk.posedge_event());

    // set read request and address
    rreq.write(true);
    addr.write(address);

    // wait for acknowledge
    wait(ack.posedge_event());
    wait(clk.posedge_event());

    // deassert request and read request
    req.write(false);
    rreq.write(false);

    return rdata.read().to_int();

}
```

Figure 7. RT implementation of read method in RT interface.

```
unsigned int
master_tl_if::if_read( unsigned int address )
{
  /// transaction pointer
  tlm::tlm_generic_payload* trans = new
  tlm::tlm_generic_payload;

  sc_time delay = sc_time(30, SC_NS);
  unsigned int data = 0;

  // Initialize 8 out of the 10 attributes,
  trans->set_command( tlm::TLM_READ_COMMAND );
  trans->set_address( address );
  trans->set_data_ptr( reinterpret_cast
            <unsigned char*> (&data) );
  //…
  // Blocking transport call
  socket->b_transport( *trans, delay );

  // obliged to check response status
  if ( trans->is_response_error() )
      ifs_error("TLM-2.0: Response  error from
            b_transport");

   //…

  return data;
}
```

Figure 8. TLM implementation of read method in TLM interface.

In Figure 7, the RTL implementation of the method `if_read` is shown. It executes the signal-level protocol for the bus to read a value from the address given as argument. The according TL implementation (Figure 8) does the same by creating and initializing a transaction object and calling the transport method to initiate the transaction.

Here, we can also see the reason for methods used in the interface but implemented in the TBM. If an error occurs while reading is performed, this error should be reported correctly. This cannot be done entirely in the TBM because the cause and circumstances of the error are not available here. They are stored in the transaction object and handing the transaction object to the TBM would break the separation of computation and communication aspects. Likewise, it cannot be realized in the interface. Here, the reporting and error handling facilities are not available as they reside in the TBM. To solve this, we declare the abstract method `ifs_error` in the interface base class. The interface implementation is then able to call this method and give it the error circumstances as arguments, whereas the method implementation is done in the TBM.

Figure 9 shows the declaration of the bus master TBM. We see no declarations of ports or sockets here. Instead, the module is derived from the interface implementation given as template parameter T. In addition to the derivation, we can check statically if T itself is derived from `master_if_base` by using `std::is_base_of` or similar type traits. This is omitted here for readability. With this, we are sure that the methods defined in the abstract interface base class are available and we can use them in the implementation of

```
template < typename T >
IFS_MODULE(master) , public T
{
  typedef T if_type;

  /* Commands */
  void Write(std::list<std::string>
            parameters);

  void Read(std::list<std::string>
            parameters);

  // IF Module method impl
 void ack_write_msg(int,unsigned int);
 void ifs_error(const char *);
   //Constructor
  master(sc_module_name);
};
```

Figure 9. Bus master TBM.

```
template < typename T >
void master<T>::Read(list<string> parameters)
{
  //…
  unsigned int address = IFS::StringToUInt(parameters.front());
  //…
  parameters.pop_front();

  unsigned int exp_value =
            IFS::StringToUInt(parameters.front());

  unsigned int value = if_type::if_read(address);

  //…
}
```

Figure 10. Implementation of read command.

the TBM's commands. The `Read` command is shown as an example in Figure 10. Instead of accessing ports or sockets directly, it uses the `if_type::if_read` method to perform the actual bus access.

Finally, we can instantiate the TBM in the version we need by simply using another template argument (Figure 11). Module `m1` is a bus master TBM with a TLM interface. Module `m2` is a TBM with the same behavior but with an RTL interface. Beyond this flexibility, we save effort when other interface implementations are needed, e.g., a TLM interface facilitating the non-blocking transport calls. Then, we only need to define another interface class implementing the `if_read` and `if_write` methods and use it in the TBM instantiation. The behavioral and computational part of the TBM remains untouched and the same test bench can be easily reused for RTL or TLM verification.

```
master<master_tl_if> *m1;
m1 = new master<master_tl_if>("MS1");

master<master_rt_if> *m2;
m2 = new master<master_rt_if>("MS2");
```

Figure 11. Instantiation of bus master TBM.

IV. APPLICATION TO AN AUTOMOTIVE DSP

Following the long tradition of the V-Model systems engineering process [11], we are challenged with the verification need for system components across several levels of abstraction along the refinement process down the "V". BtB comparison is one approach to handle this verification task.

Figure 12 depicts an IFS test bench environment configuration setup for BtB verification of two instances of the same DUV at RTL and TLM communication abstraction. The shown setup is not different to the one depicted in Figure 3 if one considers the two instances of the DUV as one larger DUV with two components, one at RTL and one at TLM. The refinement of TLM components towards RTL might seem to be the typical case at first glance. But the inverted direction of development is legit when legacy RTL components of projects prior to the VP era are reused and become part of a higher abstraction level verification. In our scenario, we abstracted an in-house DSP from RTL to an Instruction Set Simulator (ISS) implementation with a TLM interface and verified the abstraction using BtB comparison.

Whenever timing implementation details are removed, DUVs at different levels of abstraction imply different timing behavior. On one hand, this affects the time it takes to execute the model on a host machine. We call this time Model Execution Time (MET). On the other hand, we refer to the simulated time period as Simulated Time (SIT). Such abstraction in timing can be part of communication when bus handshaking signals across several clock cycles are abstracted away by TLM, or can be part of the functional behavior of the DUV, if it is specified at behavioral/algorithmic level executing instantly instead of spreading state machine behavior and algorithmic operations across many clock cycles. It is to be expected and the intended idea of abstraction that high-level modeled DUVs will execute faster than such at lower levels.

For this reason, BtB requires synchronization of paired DUVs. They differ in MET, as well as SIT. Not only timing behavior but also the sequence of results from a DUV at lower abstraction levels is not guaranteed to be identical. While a TLM/behavioral model might jump right to the correct result, a refined model can produce intermediate, faulty results at its outputs before reaching the correct response. This makes synchronization mandatory for BtB, which is in general not a trivial task to accomplish. The target application of our work though is signal processing in the automotive domain. In this specialized context, we look at sensor data processing triggered periodically through external interrupt signals, a given source for simulation synchronization events.

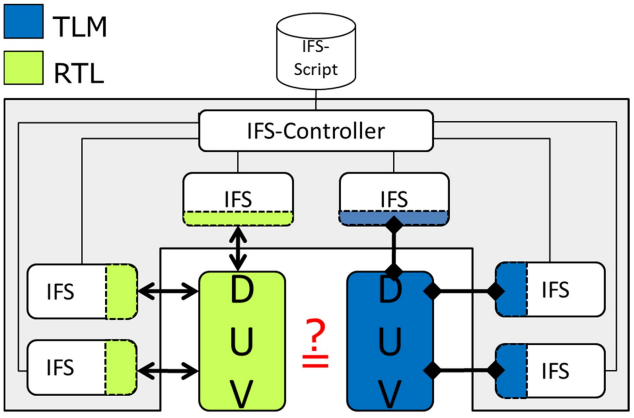

Figure 12. IFS test bench environment for BtB verification.

The use case of this work is the VP of a DSP. Any model representation of the DSP always needs

to finish computations correctly within the SIT timeframe of two succeeding interrupts with a defined period to achieve real-time response requirements. The availability of such a signal is used as our solution to the BtB synchronization problem for signal processing in the automotive domain. For a functionally correct DUV it is valid to trace the previously computed response in synchronization with the next triggering interrupt as shown in Figure 13. We used BtB on the above named instruction accurate ISS plus TLM communication in combination with the cycle-accurate model plus RTL communication. The details of our setup are depicted in Figure 14. Referencing Figure 1, it can be said that we BtB-simulated a model "B" against a model "F." We also show a zero delay computational model in Figure 13 (model "A") to visualize how e.g. a MATLAB/Simulink model would behave in the same BtB scenario.

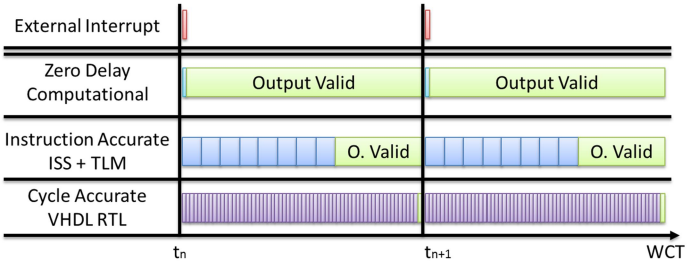

Figure 13. Synchronization of BtB comparison for each IRQ event.

The three input TBMs in Figure 14 named IRQ, CLK and STIM use SystemC ports and signals to connect to both DUTs simultaneously. This is an efficient decision as both models process the same data in a BtB simulation. The TBMs are controlled by commands of the IFS-script. This script defines what and when stimuli are applied to the DUTs. It also defines interrupt triggered test routines in the IFS scripting language.

The CTRL IRQ has a special role in this scenario and serves as a feedback loop into the IFS-script execution which is not linearly scripted IRQ event by IRQ event. The TBM IRQ is rather programmed to autonomously generate events at the required period. The script environment does not need to take track of the periods as they evolve. Instead, CTRL IRQ is sensitive to the same IRQ event as the two DUVs and synchronizes the test bench's IFS-script with the simulated DSPs' interrupt service routine execution. With each event on the IRQ, the IFS script executes its own interrupt routines to collect results from the Master TBMs and applies new stimuli at STIM, while in parallel the simulated DSPs are triggered to proceed the next cycle of data processing. The interrupt event is used to synchronize two DUVs and the test script execution at the same time.

The outputs of the two DUVs are monitored by a TBM master that was implemented according our new concept as introduced in the previous section. The great achievement in this setup is that the DUTs' outputs are compared automatically by a simple compare component but can also be examined manually by the user by plotting the signals of the models within the same simulation waveform viewer. This increases the efficiency for error detection and debugging of the ISS/TLM abstraction model. We were able to comfortably analyze deviations in the behavior of the models of different abstraction levels. This work helped to reduce our verification effort significantly. Our concept processes human readable tests specified in an IFS scripting language and BtB simulation automatically generate assertion values for comparison. These are easier to handle, read and understand than generated, cryptic test vectors. The time synchronization makes debugging in one waveform window an easy task. This was achieved by the synchronization of test, test bench and DUVs.

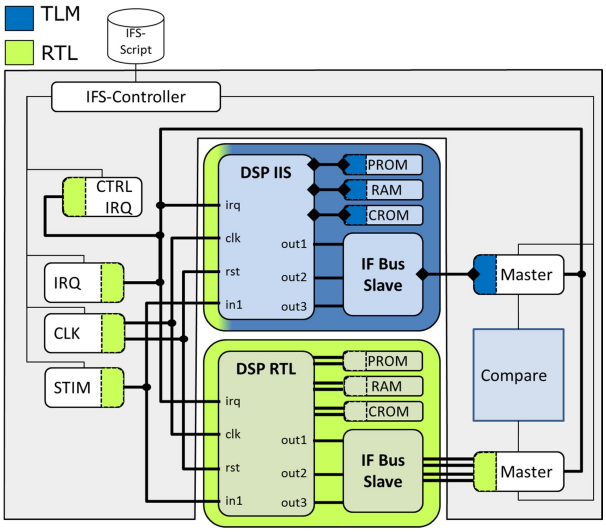

Figure 14. BtB verification setup for DSP at RTL VHDL and ISS SystemC.

V. CONCLUSION

We argued that mixed-level VPs are a valid approach to handle the exponentially growing verification task which requires structured test bench design to enable reuse of design IP, test IP and test patterns. This methodology helps preventing that verification to implementation share increases above tolerable limits. In this context, we introduced our established RTL verification method IFS which is even easier to use than "easier UVM" of [5]. We explained how it was enhanced by a technique for the simulation of transaction-level models in order to enable the mixed-level simulation of hardware and software systems. We achieved this by enhancing our TBM concept with C++ template class definitions to be easily configurable for different levels of communication abstractions. An important requirement for this approach was not to interfere with the established and operational design flow environment of a global player in automotive microelectronics. We needed to create added value without interfering with the compliance to the applied IFS methodology. We proved the validity of our approach by its application to an automotive DSP sensor example in a BtB setup of two VPs of the application at two different abstraction levels. The synchronization challenge for signal processing in the automotive domain is solved by using an external synchronization signal. We observed that our work helped to reduce significantly our verification effort. Newly designed tests automatically create comparable outputs without the need for the verification engineer to design fitting assertion, too. In case of simulated differences, debugging is convenient in the synchronized waveforms of a DUV and its reference.

In our future work we will enhance our IFS methodology even further towards concepts for fault-effect-simulation. The idea behind this is the instantiation of fault injectors similar to our TBM that deliberately cause erroneous functional behavior. The application's robustness for error handling becomes verifiable by means of the further enhanced mixed-level VP simulation possibilities.

ACKNOWLEDGMENT

This work has been funded by the German Federal Ministry for Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under the grant 01IS13022 (project EffektiV). The content of this publication lies within the responsibility of the authors.

REFERENCES

[1] P. Jores, P. Borthen, R. Dölling, H.-W. Groth, T. Halfmann, S. Kern, M. Lampp, M. Olbrich, M. Pfost, R. Popp, D. Pronath, P. Rotter, S. Steinhorst, G. Wachutka, Y. Wang and S. Weber, "Verifikation analoger Schaltungen (Kurztitel: VeronA)", Schlussbericht zur BMBF-Förderinitiative IKT2020, 2009.

[2] R. Lissel and J. Gerlach, „Introducing New Verification Methods into a Company's Design Flow: An Industrial User's Point of View," Design, Automation and Test in Europe, DATE, 2007.

[3] K. Hylla, J.-H. Oetjens and W. Nebel, "Using SystemC for an Extended MATLAB/Simulink Verification Flow", FDL 2008.

[4] F. Poppen, M. Trunzer and J.-H. Oetjens, "Connecting a Company's Verification Methodology to Standard Concepts of UVM," DVCon Europe, 2014.

[5] F. Poppen, M. Trunzer and J.-H. Oetjens, "Using Synopsys VCS to connect a Company's SystemC Verifiacation Methodology to Standard Concepts of UVM," SNUG Germany 2015.

[6] J. Aynsley, C. Sühnel and D. Long, "Easier UVM: Guidelines and Automatic Code Generation to Accelerate UVM Adoption," SNUG Germany, 2014.

[7] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," Hardware/Software Codesign and System Synthesis, CODESS, 2003.

[8] M. Conrad, "Verification and Validation According to ISO 26262: A Workflow to Facilitate the Development of High-Integrity Software," Real-Time Software and Systems Congress, ERTS, 2012.

[9] R. Jindal, K. Jain, "Verification of Transaction-Level SystemC models using RTL Testbenches," ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE, 2003.

[10] The MathWorks, Inc., "HDL Verifier User's Guide," Tool Documentation R2016a, 2016.

[11] M. Hoffman and T. Beaumont, "Application Development: Managing a Project's Life Cycle," Mc Press, 1997.