The Test Bench Factory: Building Verification Environments Faster, Better, Smarter

Thorsten Dworzak, Infineon Technologies AG, Neubiberg, Germany
(thorsten.dworzak@infineon.com)

Dr. Johannes Grinschgl, Infineon Technologies AG, Neubiberg, Germany
(johannes.grinschgl@infineon.com)

***Abstract*— Code generation is a common solution for improving the productivity and adaptability of software. In the functional verification domain, for example, generators that model the hardware/software interface, or "configuration registers," of a design under verification are widely used in the industry. These generators help improve productivity, reduce repetitive tasks, ensure specification compliance, or simply overcome the "blank page syndrome". In this paper, we present a test bench code generation approach that combines several technologies. This approach enables the creation of a fully functional SystemVerilog test bench using the Universal Verification Methodology (UVM). The generated test bench covers the three main verification aspects: stimulus generation, checking and functional coverage. This approach significantly accelerates the development process, reduces test bench errors, and facilitates the reuse of verification components. By providing a standard architecture, users can avoid a steep learning curve when starting a new project.**

***Keywords*—functional verification; code generation; verification components; verification IP; bus topology; bus fabric; reuse**

I. Introduction

In the functional verification domain, code generation with in-house and commercial tools is standard practice. Besides code generators for driving/monitoring the hardware/software interface (HSI) of a design-under-verification (DUV), some open-source [2] and commercial [3] approaches exist for complete test benches (TBs). In our company we had been using a Specman/*e*-based verification component [1] and other SystemVerilog (SV) based generators. We wanted to exploit the benefits of all approaches while eliminating their shortcomings and incorporating modern concepts such as harnesses [5]. Furthermore, we also needed to support plug-and-play re-use of verification components (VCs, aka Verification IP – VIP) and automation of bus fabric verification (e.g. bus bridges/crossbars, network-on-chip). This forced us to find an alternative. In this paper we present a UVM TB code generation approach called TBGEN. It is based on the combination/integration of several enabling technologies, listed in the following section.

II. Enabling Technologies

A. *Python Data Model + API Generator*

An in-house tool [8] exists that takes a generic data-model described as UML diagram and generates the Python API to populate + query this model. In general, this is a three-stage process:

1. Define the UML diagram: defines how the data-model should look like (classes with attributes and their relationship).
2. Write an input reader: reads a configuration file (e.g. XML, JSON, …) and populates the defined data-model using the API generated from the UML
3. Write an output generator: use the data-model and its API to generate the output files.

The advantage of this approach is that you can easily replace the input format or the generated code without rewriting the whole tool. The TBGen tool is implemented based on this framework.

B. UVM-HSI Generator Using XML Register Model

An in-house tool exists – RegGen - that takes configuration+status register data described in XML format and creates a uvm_reg-based register abstraction layer (RAL). This format is proprietary and contains more information than IP-XACT. The tool follows the same approach as described in Section II.A.

The tool generates:

- UVM register block with all the corresponding registers, fields, reset values, …
- Functional coverage for reset, access types, …
- UVM sequences for testing the functionality of the registers
- Extensions of the standard uvm_reg_* base classes, adding useful API functions and small bug fixes

C. UVM-VIP Generator Using VC Meta-data

An in-house tool that takes VC Meta-data (for example, UVC name, agent types and instances, etc.) and the composition thereof described in JSON format and creates structural UVM SV while applying standard UVM concepts (environments, agents, sequencers, configuration objects, etc.). It generates code using Mako templates [6]. Basically, this is used by TBGEN as SystemVerilog renderer. This tool is intended to bring-up a test bench in an efficient way. Therefore, following aspects were the focus of this tool:

- Same look and feel of all verification components for the whole company
- Reuse of existing verification components horizontal and vertical
- Ease of use: following basic UVM concepts; quick ramp-up time

D. UVM-SystemVerilog Base class and Framework Libraries

To support the verification use-cases for IP and (sub) system level verification, we created

a) a base class library which adds only a thin layer on top of the UVM library – giving us the ability to roll-out improvements to the UVM library and adding a few utility classes and macros

b) a comprehensive framework library [4] supporting the three main aspects for the use-case of bus fabric verification: stimuli generation, checking, and coverage. It contains a protocol agnostic sequence framework, scoreboards for different transaction types (protocol-specific and *uvm_tlm_generic_payload*), a database for querying bus topology information at simulation-time, and coverage collection classes. This library is mainly intended for (sub) system level verification.

III. TAXONOMY OF VIPS

To facilitate reuse of verification IP for TB generation, each VC is associated with meta-data containing the information needed to generate the VIP using the UVM VIP generator and for integration into a UVM environment. Existing VIPs can be reused by creating the meta-data for them. This includes e.g. name of the VC, required packages for compilation, type and instance of sequencers and interfaces, etc. The meta-data is captured in JSON format which can be conveniently read and written by various programming language libraries.

VCs are categorized into 5 different types, because each requires a different set of meta-data (see Table 1).

Table 1 VC Taxonomy

Type of VC	Description
iVC	an interface VC drives and monitors connection to a DUV via SV interfaces; the iVC needs minimal information from the user to control the UVM-VIP-generator, because most information is implicitly inferred from the generator (standard UVM architecture)
iVCg	same as iVC but the "g" denotes a 3rd party iVC that does not follow the structure of generated VCs; it contains explicit information about its sequencers, analysis ports etc.
mVC	typically containing an agent with a scoreboard; does not connect to the DUV (no SV interfaces) but receives items from iVCs on TLM1 analysis ports
sysVC	the integration layer environment; typically containing instances of iVC(g)s, mVCs, other sysVCs, and register components
TB	this is similar to a sysVC but also contains tests, testbench modules etc.

IV. Test Bench Structure Definition

The TBGEN tool requires user input for the test bench composition. For this testbench topology definition (TTD) we decided to use the HOCON format [7] which is a superset of JSON. HOCON is more user-friendly because it requires less quoting and supports comments, substitution, and file inclusion. The TTD contains several top-level sections which can have nested sections. These sections specify:

- Global information about the test bench, e.g. DUV name and instance, DUV parameters, name of the generated top-level environment
- Paths to imported VC meta-data (see previous section); used to build a data model of the TB
- Instantiation of VCs; used for simple drop-in of VCs with basic configuration (like interface parameters, number of agents, bind targets, etc.). This allows VCs to be added to the test bench in a plug-and-play style and connected to one or more scoreboard VCs.
- (optional) Topology of the bus fabric of the DUV which consists of components, buses and bus-interfaces. This is used for instantiation and full configuration of VC bus protocol agents from the supported set, and automatic inference of scoreboards and coverage collection. When this section is present, the aspects of stimulus generation, checking and coverage are part of the generated test bench.
- Register model generation information; this specifies the location of the XML single source and instructions whether to run RegGen to create the RAL model.
- Register address maps; the connection to analysis ports of bus agents of the VCs is made here. Additional attributes include types of predictor/adaptor.

V. Generated Test Bench

The generated UVM top-level environment is based on the SV class libraries from section II.D. It instantiates the VCs (which are also UVM environments) and the register abstraction layer. Users may choose to group the VCs into a container environment. The generated test bench is a structural UVM hierarchy, with base classes for tests and sequences that are already hooked up to the available sequence drivers and register models. Otherwise the structure is fixed. One advantage of using a given architecture is that users new to a project can quickly become productive because of its familiar design. The tool provides different hooks for customizations, e.g. to automatically incorporate platform requirements or standard configurations of VCs.

If the bus fabric verification use-case is targeted, other sub-environments will be added. Code for stimuli generation, checking and coverage is added that is tailored to the described bus topology.

Two Verilog modules are generated for the connection between the high-level verification language domain (HVL) and the hardware description language domain (HDL). Separating these two aspects is necessary to support emulation. See Figure 1 for a high-level overview and Figure 2 for the relationship between the two domains.

For connecting the DUV to the UVM test bench we support the harness approach [5] and recommend it as the main method because it allows vertical re-use of the connectivity. The role of the two Verilog modules is

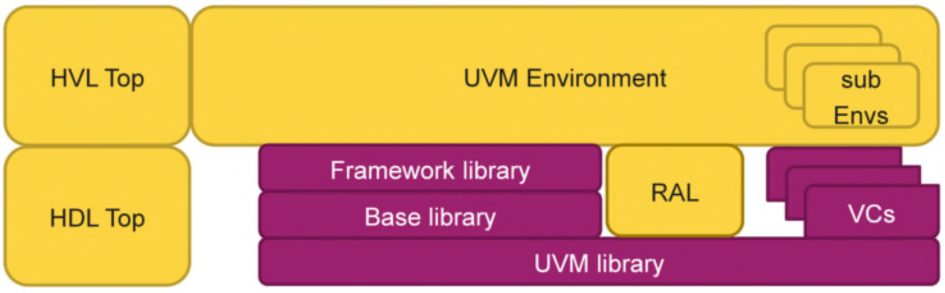

Figure 1 This shows the layered relationship between the generated parts of the test bench (yellow) with the re-used parts (magenta). VCs themselves can also be based on the SV libraries (not shown).

- The HDL module contains the instantiation of the DUV, and the bind statements that connect the harnesses to the DUV or its submodules. The harnesses instantiate the physical SystemVerilog interfaces. The connectivity between the DUV and the SV interfaces is encapsulated in the harness modules and thus reusable. Additional glue logic can be added to this module to connect a generated clock to the interfaces of other VCs, for example .

- The HVL module is the connection to the HVL class world. It contains the *run_test()* statement for the UVM and registers interfaces with the UVM config DB.

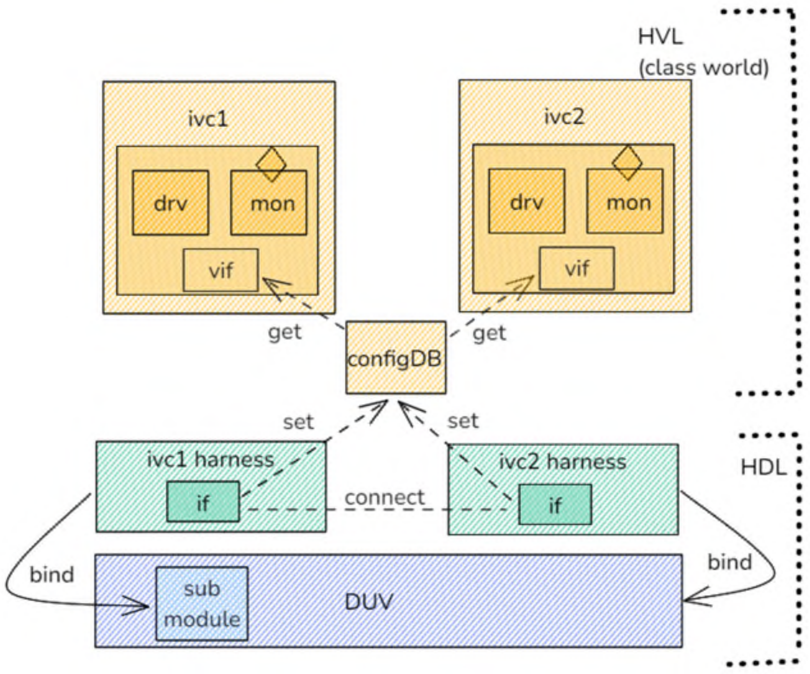

Figure 2: This shows the relationship between the HDL and HVL aspects and how the SV interfaces are encapsulated in harness modules that are bound to the DUV. Interfaces are set in the UVM config DB and can be retrieved by the VCs. Connections between interfaces allow for example sharing clock/reset signals.

VI. Bus-Fabric Verification

Consider a simple DUV with a CPU that has two initiator interfaces (e.g., instruction and data fetch) and two peripherals with target interfaces that are connected to the CPU via an arbitration unit (see Figure 3). Since the arbitration unit is transparent for bus transactions, we can ignore it and treat it as if it were not present. Thus, there is only one logical bus. In addition this structure, the DUV has an address map, and certain bus attributes (e.g., “peripheral SPI_0 is read-only”). This information is captured in the TTD file and will be used to generate and configure the test bench.

The test bench entities defined in the topology are categorized as

- Components: they represent DUV elements and can be of type *peripheral* (end-point of the bus fabric), *bridge* (connects two buses) or *crossbar* (provides multiple routes between bus interfaces of different protocols). Each component contains bus interfaces.
- Buses contain more than one bus interface and have a dedicated bus protocol. A bus is mapped to an interface VC
- Bus interfaces belong to a component and a bus. They are mapped to agents of interface VCs. An initiator bus interface has a static routing table attached which specifies the targets it can reach. A target bus interface has an address map attached that is decoded by the corresponding DUV element. A default target can be inserted to model bus termination (it does not have to be an actual agent, depending on the termination rules).

Buses and bus interfaces can have limitations defined, i.e. non-supported attributes of the protocol, that are considered in stimulus generation and coverage.

The assumption is that this taxonomy allows to model the architecture of many bus-based DUVs. The tool also supports reconfiguring the test bench agents with respect to active/passive at compile-time. For example, the CPU in the example could be black-boxed. In this case the agents attached to its initiator interfaces must be active. It can be controlled by editing a text file (“working topology”) containing the default configuration.

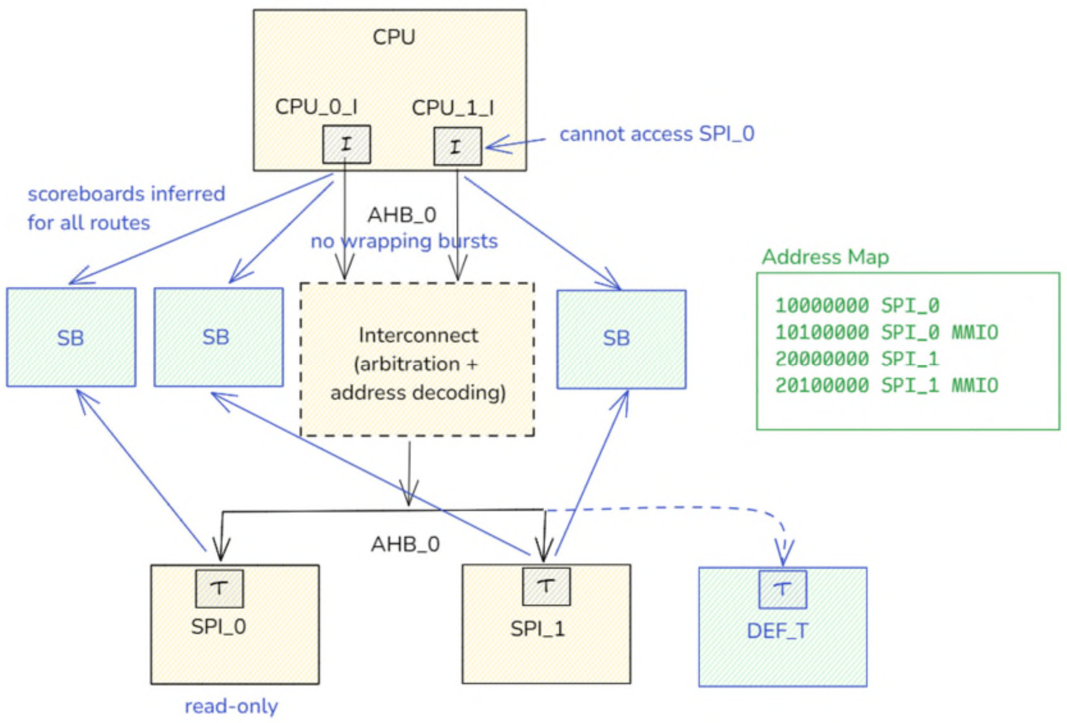

Figure 3: Bus fabric with CPU and two peripherals and their initiator and target interfaces (yellow). The system has inherent properties (blue), like “only incremental bursts on bus AHB_0”. In green, test bench elements added by TBGEN: scoreboards and a default target (for bus termination).

TBGEN automatically infers one-to-one scoreboards for all possible routes between initiators and targets, which can also be configured by users. For network-on-chip (NoC) verification, an n-to-1 scoreboard is also available. Checked scoreboard items are sent to a central coverage collector. This allows to collect functional coverage for the different interfaces and all possible routes through the bus-fabric. Figure 4 depicts the UVM hierarchy generated

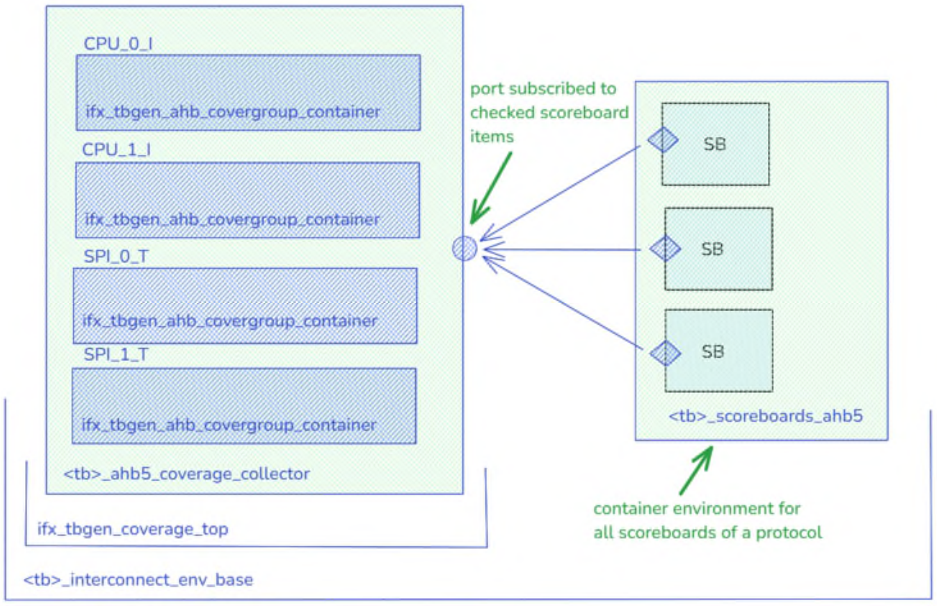
CPU_0_I
ifx_tbgen_ahb_covergroup_container
CPU_1_I
ifx_tbgen_ahb_covergroup_container
SPI_0_T
ifx_tbgen_ahb_covergroup_container
SPI_1_T
ifx_tbgen_ahb_covergroup_container
<tb>_ahb5_coverage_collector
ifx_tbgen_coverage_top
<tb>_interconnect_env_base
port subscribed to checked scoreboard items
SB
SB
SB
<tb>_scoreboards_ahb5
container environment for all scoreboards of a protocol

Figure 4: UVM hierarchy of scoreboards and central coverage collection. All items, regardless of protocol, have the same base class and can thus be received by the coverage collector through a single analysis port. The *<tb>_interconnect_env_base* is the environment instantiated inside the top-level environment.

for the coverage collection.

VII. Test Development

The bus topology information is stored in a database class, the Topology DB, provided by the Framework library and can be accessed via the Topology API. The API offers methods to retrieve handles to DB objects (e.g., components and buses) and UVM objects (e.g., sequencers and scoreboards) that participate in the bus fabric. The database can be searched using object names and filters. For example, it is easy to search for all active bus interfaces of the AXI protocol or all bridge components.

Thus, the test writer does not have to make too many assumptions about the topology; consequently, it is possible to write tests that insensitive to changes to the topology. For example, a test can retrieve all targets connected to the initiator CPU_0_I and run sequences that automatically retrieve the targets' address ranges to direct transactions to SPI_0 and SPI_1. Bus or bus interface limitations (e.g. "read-only for SPI_0") are automatically adhered to. The virtual sequence framework implements also protocol-independent sequences allowing tests to cover more than one protocol type. This is especially advantageous for NoC verification where the bus topology specification can often be subject to change and the test environment must adapt quickly.

VIII. Vertical Reuse

TBGEN creates the meta-data files from chapter III of type *TB* also for the generated test bench, as well as for chapter III. This allows it to be integrated into a higher-level test bench. For example, one or more subsystem TBs can be instantiated into an SoC TB. The SoC's TTD file just has to point to the JSON file's location and contain the desired instantiations. Because the active/passive roles of agents in the subsystem TBs will most likely change in the context of the SoC, the user can control this at compile-time by supplying a different working topology. The standard UVM method of controlling this at build-time is still available. Furthermore, build-time controls can disable creation of UVM components that are unnecessary at the SoC level.

IX. RESULTS

The TBGEN tool has been used in many projects to date and has proven its ability to reduce bring-up time for functional verification. The following table gives examples for the complexity of the test benches for three projects:

DUV	No of different iVC/mVC types/instances	No of different sysVC types/instances	Lines of code for top-level test bench (generated)
SPI memory controller (IP)	8/9	-	18900
CPU with interconnect, DMA, memory controllers	7/22	6/6	144k (120k of which for registers)
System Resources Sub-System (clock, reset, power etc.)	49/529	18/18	220k (146k of which for registers)

TBGen does not have the short-comings of earlier approaches that were either too simplistic (e.g., replacing only VC names in a set of templates), tied to a particular DUV architecture, or using a proprietary language (Specman/*e*). By providing a standard TB architecture, users moving between projects can quickly ramp up to a new environment. The creation of a test bench, from writing the TTD file based on a pre-generated template to compiling the generated test bench, takes only a few hours as opposed to days for the manual process. The library of VIP metadata has been extended to allow more and more generally useful VCs - in-house and 3rd-party - to be automatically integrated into the generated TBs. We are currently rolling out the bus fabric verification aspect, which provides a fully functional, scalable test bench where the user only needs to add tests.

REFERENCES

[1] D. Robinson, "Shorten and simplify SoC verification using a generic eVC," Verilab, Whitepaper, Nov. 2005.

[2] Siemens Verification Academy, "UVMF: Universal Verification Methodology Framework," [Online]. Available. [Accessed: 25-MAR-2025].

[3] Cadence, "System VIP: Transforming SoC verification through system-level testbenches," [Online]. Available: https://www.cadence.com/en_US/home/resources/technical-briefs/system-vip-tb.html. [Accessed: 25-MAR-2025].

[4] K. Cwalina and B. Abrams, *Framework Design Guidelines*. 2nd ed., Boston, MA, USA: Addison-Wesley, 2009.

[5] J. Vance, J. Montesano, and K. Johnston, "Verification prowess with the UVM harness," presented at the Synopsys Users Group (SNUG) Conference, Austin, TX, USA, 2017

[6] Mako Templates, "Mako Templates Documentation," [Online]. Available: https://www.makotemplates.org/. [Accessed: 25-MAR-2025].

[7] J. Roper et al, "HOCON: Human-Optimized Config Object Notation," [Online]. Available: https://github.com/lightbend/config/blob/main/HOCON.md. [Accessed: 25-MAR-2025].

[8] W. Ecker and J. Schreiner, "Metamodeling and code generation in the hardware/software interface domain," in Handbook of Hardware/Software Codesign, Springer, 2017, doi: 10.1007/978-94-017-7267-9_32.