Dynamic Inside-Out Verification Using Inverse Transactions in TLM

Tobias Strauch
EDAptix, R&D
Munich, Germany
Email: tobias@edaptix.com

***Abstract*—With growing design complexity, the reuse of module and subsystem level verification knowledge on electronic system level (ESL) becomes more and more challenging. The ”Portable Stimulus Specification Working Group” intends to offer solutions such as stimuli reuse for today's verification challenges. This paper proposes a novel Inside-Out Verification (IOV) methodology, which makes module level dynamic verification knowledge highly reusable on system level by using transactions and inverse transactions. IOV can be combined with SystemVerilog based UVM. The examples in this paper are based on PDVL (a super-sub-set of SystemVerilog) and SystemC.**

***Index Terms*—Circuits and systems, system modeling language, system verification**

I. Introduction

An ESL verificationist needs to understand the design functionality, how to drive and to monitor the design behavior and how to pass information/data through a complex electronic system. Another challenge is, that the defined (cross-)coverage points must be hit during the dynamic verification runs. These tasks get increasingly challenging with rising design complexity. The research field to automate this process is called automatic functional pattern generation (AFPG). From the industry's perspective, the universal verification methodology (UVM, [1]) has become the standard testbench environment, according to Foster in [2].

SystemVerilog (SV) [1] and SystemC (SC) [1] support UVM. Nevertheless, it becomes evident, that what has been a good verification strategy on module level is not very applicable on system level. The ”Portable Stimulus Specification Working Group” [1] intends to offer solutions to this problem. Another problem in this verification domain is the missing solution for the integration of software sequences (e.g. drivers) as an integral part of the dynamic or static verification flow.

This paper proposes a novel inside-out verification (IOV) methodology. Verification knowledge is related to individual cores or functional units (FU). This knowledge is reused on ESL. It is based on transactions (Tr) and inverse transactions (InvTr). Trs describe the activity flow of data and information within an electronic design and are synthesized. InvTrs pass information through a system in the inverse direction of Tr and are not synthesized. The combination of Trs and InvTrs defines an executable graph, which can be utilized to find solutions on how the electronic design needs to be stimulated in order to activate and to propagate a defined data or information flow within the design. We use SystemC and an aspect and transaction oriented programming, design and verification language (PDVL), which has been introduced by Strauch [3] to demonstrate the feasibility of the IOV methodology.

In section II the problem is defined and related work is listed in section III. Inside-out verification is outlined in section IV. A short overview over the relevant PDVL aspects is given in section V. Section VI describes the building blocks of the proposed methodology and more details are given based on an example in section VII. The paper finishes with a comparison to related work and a results section.

II. Problem definition

In this section we define dynamic verification as the process of passing information through a testbench (TB) and the design under test (DUT), while checking the correctness of that process at the same time. This is similar in dynamic verification on module level as well as on system level. In this context, information can range from a simple event to a complex data payload etc..

We also define, that a given set of coverage points must be reached. These coverage points can define for instance corner cases of a specific design behavior (e.g. an overflow of a FIFO buried within a DUT subsystem).

$$I_{\text{exp}} = Tr_{\text{mon}}(Tr_{\text{prop,n}}(Tr_{\text{cover}}(Tr_{\text{sen,n}}(Tr_{\text{driver}}(I_{\text{submit}}))))) \tag{1}$$

The information passing through the system is transferred based on transactions (Tr). Equation (1) shows how information is modified during that process. The testbench driver $\text{Tr}_{\text{driver}}$ converts the submitted information I_{submit} and passes it to the DUT. The sensitizing $\text{Tr}_{\text{sen,n}}$ transfers the information to the relevant transaction representing the coverage point (Tr_{cover}). Propagation transactions ($\text{Tr}_{\text{prop,n}}$) transfer the activity further to the monitor Tr_{mon} in the TB. It is defined, that if Tr_{mon} returns the expected information I_{exp}, then the coverage point under test is marked as covered in the scoreboard.

III. Related work

A. Portable stimulus standard (PSS)

A key purpose of the PSS [1] is to automate the generation of test cases and test suites. An activity graph can define the scheduling of critical actions. The execution of a concrete verification scenario essentially consists of invoking its actions' implementations, if any, in their respective scheduling order.

B. Related work on auto refinement

Abstraction refinement techniques for simulation-centered verification is proposed in the literature. For example, Chatterjee et al. [4] propose an activity-based refinement for abstraction-guided simulation. Their main contribution is a refinement engine, which uses individual design modules as the building blocks of the abstraction.

A top-down methodology for the generation of RTL tests from SystemC TLM specifications is presented by Chen et al. in [5]. Their paper makes two important contributions: automatic test generation from TLM specification using a transition-based coverage metric and automatic translation of TLM tests into RTL tests using a set of transformation rules.

Related work on auto refinement limited to specific designs such as processors or interfaces (protocols) is not considered.

C. Related work on auto abstraction

The opposite way of using auto-refinement is auto-abstraction. In [6] Liu et al. present HYBRO, an automatic methodology to generate high coverage input vectors for RTL designs based on branch-coverage directed approach. HYBRO uses dynamic simulation data and static analysis of RTL control flow graphs (CFG). The main novelty of this technique lies in the hybrid analysis between concrete simulation data and static analysis of the RTL code and the heuristical branch guided path exploration.

Acharya et al. propose in [7] a functional test generation method for RTL circuits. They state that a popular metric for measuring the effectiveness of an RTL test suite is branch coverage. The challenge in exercising a hard-to reach branch is in the understanding of the semantics of the design. Without a good guidance, hard branches might require unnecessarily long test sequences or are missed altogether. With their method, they extract such semantics from the circuit using a lightweight static analysis of the code in order to guide the search.

In [8] Guzey et al. propose a novel automatic constraint extraction method for increasing the efficiency and the eventual success of simulation-based verification. Input constraints used to increase controllability are automatically extracted from simulation data without requiring knowledge about the structure of the design. Extracted constraints can be utilized to increase verification coverage by specifically targeting parts of the design according to selected coverage metrics.

Pierre et al. [9] borrow concepts from automatic test pattern generation (ATPG) such as backward traversal to improve dynamic, assertion based verification through the generation of proper test sequences. Their work focuses on the assertion based verification of PSL properties for already synthesized designs (thus given as netlists of gates and memory elements). The algorithms that are developed rely on the structural circuit representation.

D. Related work on AFPG using a dedicated language

Emek et al. present in [10] a methodology for scheduling system-level transactions using a hierarchical transaction scheduling language. Test template writers describe precisely the transactions they wish to generate, as well as their relative scheduling.

In [11], Piccolboni et al. show a stimuli specification language and a corresponding stimuli generation engine targeting the reproduction of specific conditions by scenario and assertion based verification. The language allows to intuitively write directives for the engine to generate constraint-based stimuli sequences that respect the desired conditions.

E. Related work on testbench reuse

Bombieri et al. [12] show a theoretically-based methodology to evaluate the quality of transactor based verification (TBV) with respect to a more traditional RTL verification flow. The evaluation relies on comparing both fault coverage and assertion/property coverage by using and not using TBV to verify the correctness of an RTL design. In this way, they showed that TBV is effective for reusing testbenches as well as assertions when a transaction level description is refined into an RTL implementation.

F. Related work on coverage

Verma et al. [13] propose a method which allows generation of a functional coverage model based on a provided set of properties derived from design specification. They use computational tree logic (CTL) expressions to represent properties of the specification. CTL uses atomic propositions to describe the states of a system and combines them into expressions using logical and temporal operators.

A methodology to define and to compute code coverage of an assertion is presented by Athavale et al. in [14]. Their method is based on static and dynamic analysis of the RTL source code.

Ramineni et al. [15] present a technique to estimate the correlation between a coverage metric and design error detection. Test sequences are generated for a set of benchmark examples and coverage is computed for each test sequence using the coverage metric which is under evaluation.

Yang et al. [16] consider the generation of dedicated stimuli for simulation-based verification which covers scenarios to be triggered in all possible fashions. For this purpose, three approaches have been proposed including a naive approach based on minimal stimuli generation, an advanced approach with less complexity, and an approach employing a partitioning scheme. The general idea is to partition a global problem into several smaller ones and, then, solve them separately.

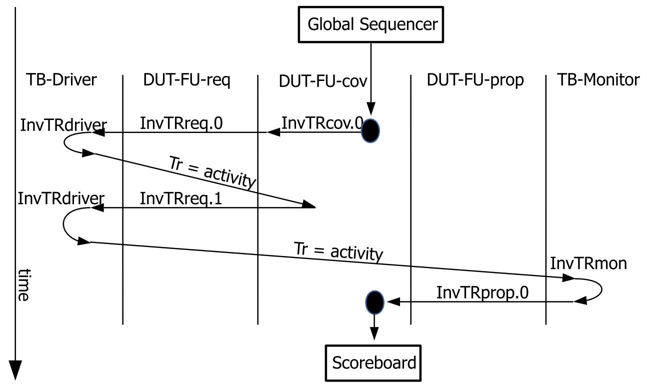

Fig. 1. Scenario for Inside-Out Verification.

IV. Introducing inverse transactions and inside-out verification

A. Inverse transactions

The usage of inverse transactions (InvTr) in the context of inside-out verification is discussed in this section. Transactions (Tr) describe the activity flow of data and information within an electronic design. Tr are usually synthesized and describe the desired design behavior.

It can be said, that InvTrs pass information through a system in the inverse direction of Trs. They are not (necessarily) synthesized and are only used for verification. A collection of Trs and InvTrs define one possible scenario, how elements of the electronic design need to be stimulated in order to activate a defined data or information flow within the given design.

The purpose of InvTrs is to add, to extract and to purify design knowledge of a given design under test (DUT) to be used in the design verification process. In this context, InvTrs establish a link between the design behavior and the testbench (TB) environment by making the functionality inside a DUT visible to members of the TB outside the DUT.

B. Inside-out verification

A functional unit (FU) can be a CPU, a peripheral, a systembus, a memory controller, etc.. Each FU is enhanced to either "request" verification related information, or to "propagate" it. The proposed inside-out verification (IOV) methodology combines this verification knowledge of individual FUs inside the DUT and utilizes it for the system wide dynamic verification task.

$$I_{\text{req}} = InvTr_{\text{driver}}(InvTr_{\text{req,n}}(I_{\text{stim}})) \tag{2}$$

$$I_{\text{exp}} = InvTr_{\text{prop,n}}(InvTr_{\text{mon}}(I_{\text{prop}})) \tag{3}$$

The definition of dynamic verification as a process of passing information through a TB and the DUT reflected in (1) is therefore modified to a process of requesting information by a FU from the TB (2) (inverse request path) and by propagation information from the FU to the TB (3) (inverse propagation path), whereas the FU resides inside the DUT. When this process is correct, the relevant coverage point in the scoreboard is achieved.

The requested information I_{req} (2) from the TB is derived by $\text{InvTr}_{\text{driver}}$ and various $\text{InvTr}_{\text{req,n}}$ transformations initialized by the information to stimulate a specific behavior inside the FU (I_{stim}). The expected information (I_{exp}) (3) from the FU's point of view must be equal to the $\text{InvTr}_{\text{mon}}$ and $\text{InvTr}_{\text{prop,n}}$ transformation of the propagated information (I_{prop}) within the TB monitor.

Fig. 1 gives an overview of such a testcase scenario. We can derive the following rules for the individual members used in the IOV.

The FU inside the DUT, which has the testcase relevant coverage point implemented (Fig. 1, DUT-FU-cov), controls the testcase execution by initializing the local test sequences $\text{InvTR}_{\text{cov.n}}$ and by forking (multiple) "request" and "propagate" actions to the adjacent FUs. All FUs on the activity request path must be capable of handling request actions (Fig. 1, DUT-FU-req). An information request action at the output of a FU uses $\text{InvTR}_{\text{req.0}}$ and $\text{InvTr}_{\text{req.1}}$ to request the relevant information at its inputs. A TB driver converts an information request into the relevant stimuli at the input of the FU, which requested a defined information. The stimulated activity on the request path is responsible for sensitizing the pre-defined coverage point.

The FU scheduling the testcase (Fig. 1, DUT-FU-cov) is also responsible, that the result from the activated coverage point is propagated to the TB monitor by sensitizing the FU on the propagation path (Fig. 1, DUT-FU-prop). The TB monitor collects the activity leaving the DUT. The received information is then acknowledged by passing its confirmation backward into the DUT again using $\text{InvTR}_{\text{prop.n}}$ (here $\text{InvTR}_{\text{prop.0}}$). All FUs on the propagation path must therefore also be capable to confirm the successful propagation of information. The status is confirmed backward along the propagation path using individual InvTrs.

The DUT-FU-cov can react to the receive of that acknowledge of a successful information propagation, depending on the given testcase. One way is to define the relevant coverage point in the scoreboard as covered.

Each active FU must be enhanced to support the rules which were just outlined. This local verification related code can interact with the behavior of the given FU. The interaction can guide the execution of a testcase towards a given corner case.

V. Guided, coverage driven inside-out verification using PDVL

We will demonstrate in the next section the benefits of IOV in an example, for which we are using the aspect and transaction oriented programming, design and verification language PDVL, which was introduced by Strauch in [3]. An overview of the relevant PDVL aspects is given here.

A. Overview

First of all, PDVL can be seen as an additional set of language constructs on top of SystemVerilog. This allows the

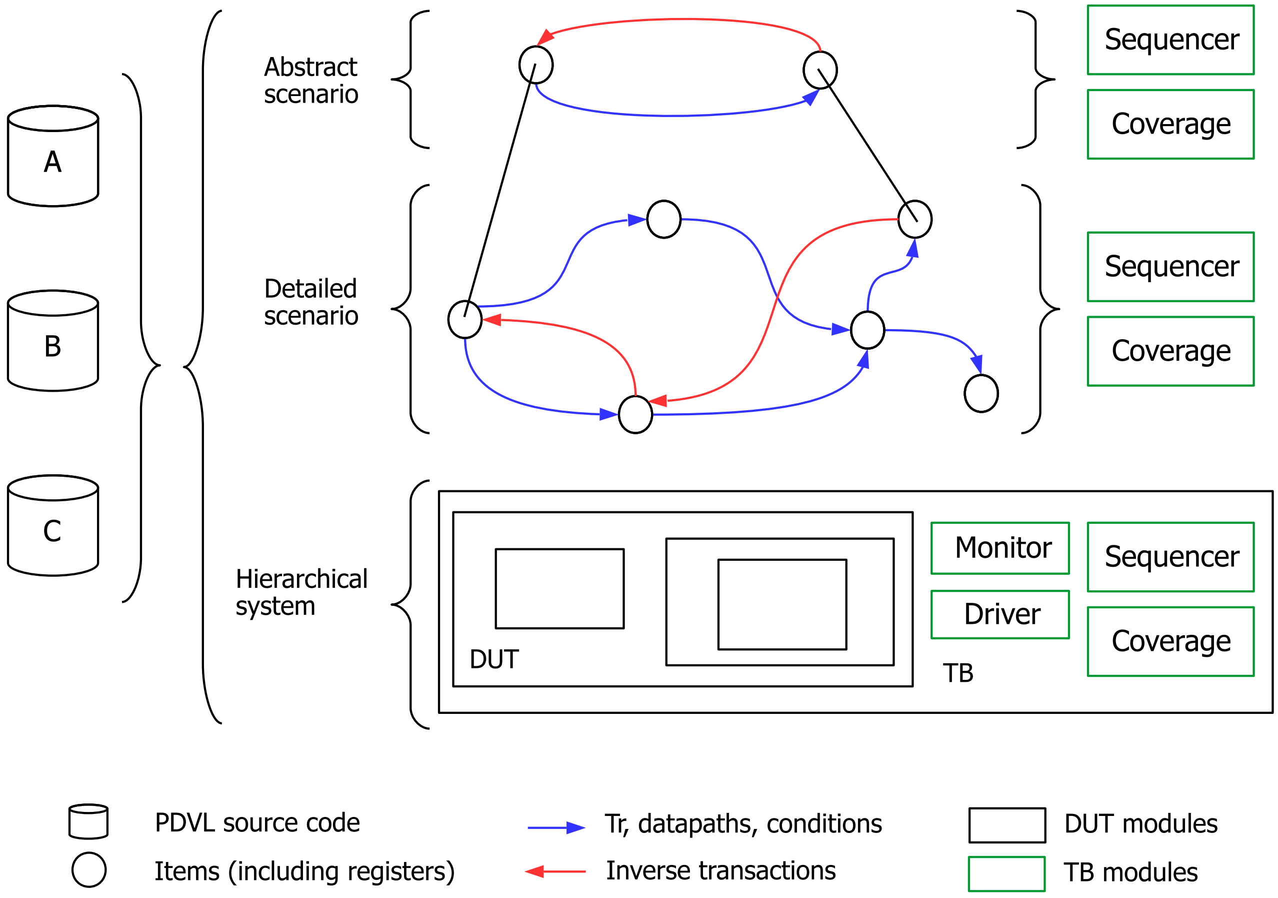

Fig. 2. Applying the UVM at different abstraction levels using the aspect and transaction oriented PDVL.

reuse of the rich set of language constructs in SystemVerilog, in particular the reuse of the one needed for implementing a verification strategy guided by the universal verification methodology (UVM [1]).

Most importantly, PDVL is aspect oriented (see Fig. 2). This allows the elaboration of scenarios at different abstraction levels as well as the generation of a classical, synthesizable hierarchical system.

Fig. 2 shows, that the DUT can be tailored towards a specific test by using a dedicated design scenario for it. Specific DUT sections can be utilized on the relevant abstraction level.

Trs and InvTrs defined in PDVL can be placed within the DUT as synthesizable code. The same code can be reused and can be placed within the TB to serve as driver or monitor element instead. Coverage can be defined locally close to functional behavior within the design itself.

B. Inverse transactions

PDVL offers InvTrs to generate guided sequences for the dynamic verification of a DUT. InvTrs can trigger each other. They also trigger Trs in order to execute a desired behavior of the design. InvTrs do not need to have an inverse match of a forward oriented Tr. They overwrite activity which is initiated by Trs. InvTrs are not meant to be synthesized and have a limited lifetime during the verification process.

Once the PDVL simulator reaches an "execute" command ("!"), it traverses through the given layered graph of the given test scenario. InvTrs are executed without modifying the simulation time pointer, until Trs are found which are emitted and which are time consuming. InvTrs can fork multiple activity trails to be joined again at a later point of time.

C. Example

In this example, we want to stimulate a behavior within the peripheral of an SoC by forcing the simulator to generate a test case executing the "it_peripheralWrite" InvTr:

```
!it_peripheralWrite(AAh);

 it_peripheralWrite (unsigned data) {
     !it_systemBusWrite(CNTRL, ENBIT);
     !it_systemBusWrite(DATAREG, data); }

 it_systemBusWrite(unsigned addr, unsigned data) {
     !it_cpuStoreData(addr, data); }

 it_cpuStoreData(unsigned addr, unsigned data) {
     load(r1, addr);
     load(r2, data);
```

store(r1, r2); }

The InvTr "it_peripheralWrite" is executed and triggers the "it_systemBusWrite" InvTr twice (for configuration and data transfer), which then triggers "it_cpuStoreData" each time. The time consuming "load" and "store" Trs are then emitted throughout the next cycles, while the simulator starts to increase the simulation time pointer.

D. Programming sequences

The simplified example shows, how a program and an instruction sequence can become an integral part of the overall testcase sequence. The alternative flow of compiling C code and feeding the assembler code through the ecosystem to an on-chip CPU becomes obsolete. It is obvious, that assembler code can be directly converted into InvTrs.

In the given example, the CPU is modeled on a higher abstract level. Its purpose is to generate a program memory footprint for a specific peripheral related testcase. Nevertheless, InvTrs can be used to simulate dedicated program sequences, while stimulating certain timing scenarios within a CPU such as exceptions or cache misses at predefined timeslots at the same time.

E. Coverage driven

State-of-the-art dynamic verification environments use functional coverage to measure the efficiency and the progress of the verification process. PDVL enables the definition of coverage points on Tr level. The Tr can reflect the design behavior at the lowest possible level (which can be a simple assignment for instance) inside the DUT as well as on higher system level. PDVL can be seen as an additional set of language constructs on top of SystemVerilog. The verificationist can therefore generate a verification environment, which follows the guidance defined by the universal verification methodology (UVM, [1]).

For the purpose of verification, further abstract Tr (InvTr) graphs can be defined as well. They can build sequences, which span multiple hierarchical and functional levels throughout the complete system. Coverage points can also be defined for these higher level Trs and InvTrs as well.

During an individual test-sequence, the status of a coverage point can be read. This information can be used for the unfolding of the remaining test-sequence to concentrate on areas which are less covered.

VI. IOV BUILDING BLOCKS

In this section we list the building blocks of an IOV based design and verification ecosystem example (Fig. 3). The TB contains at least a global sequencer (GS) and a scoreboard (SB). The DUT is a collection of FUs, which represent the design to be verified. Each FU is enhanced by a set of IOV specific Tr- and InvTr-containers (blue boxes in Fig. 3).

The FU communicate with each other using virtual channels (VC, blue lines in Fig. 3). This link also exists for the GS and the SB to the FUs, which is not shown in Fig. 3. The VCs use InvTrs to exchange testcase relevant data, which is stored in the individual FU container. The VCs are built using a binding process in parallel to the DUT module hierarchy building process.

The sequencer generates specific scenarios by scheduling individual Trs inside the FU using VCs. Once the testsequence has finished and a coverage point is verified, then the FU communicates this information to the SB using VC.

IOV related Trs inside a FU can be activated by the GS in the TB. These Trs are then executing local scheduling tasks in order to execute a given test sequence. The goal is to cover defined coverage points. IOV related InvTrs inside FUs can also be activated by VCs. The goal is to react on an information request (IR) or to react on a propagation acknowledge (PA). When an IR occurs, the FU checks if this information can be provided by the FU or the IR needs to be requested from a connected VC. When a PA occurs, the FU checks whether the PA can be used for further sequencing or if that PA needs to be propagated to a connected VC.

A driver is defined as an element inside the TB which is associated with an individual FU. It provides the information (stimuli) to the DUT upon IR through a connected VC. A monitor is defined as an element in the TB which is associated with an individual FU. It receives the information (pattern) from the DUT and converts it into a PA through a connected VC.

VII. BENEFITS OF IOV BY EXAMPLE

In this section, we outline the benefits of the IOV methodology based on an example using SystemC and PDVL. Fig. 3 shows the relevant FU. A Camera interface (CAM) sends data through a DMA core towards an Ethernet UDP core. A CPU is used for configuration. A CAM FIFO overflow during transmission is defined as a coverage point for the testcase.

A. Verification code reuse

The FUs have been verified stand alone using IOV. They contain therefore already all necessary IOV related Trs and InvTrs, which can be re-used for system level tests. We also re-use the monitors and the drivers of the SDRAMC, Ethernet and CAM FU in the system level TB.

B. ESL binding in SystemC

The first step is to generate the DUT and the TB. This is done by instantiating the individual modules of the FU and by connecting the building blocks.

For IOV we also need to generate and bind VC. The verification section of each FU in SystemC has already sockets implemented for interfacing. The sockets binding process is offered by SystemC. The payload of such an interface depends on the VC usage. When it is in parallel to the connection of two FUs, then the payload is defined by the design specific functionality. When a VC is used to communicate between FUs and the global sequencer (or the scoreboard), then the payload must be adapted.

The VC building process can be done locally when two

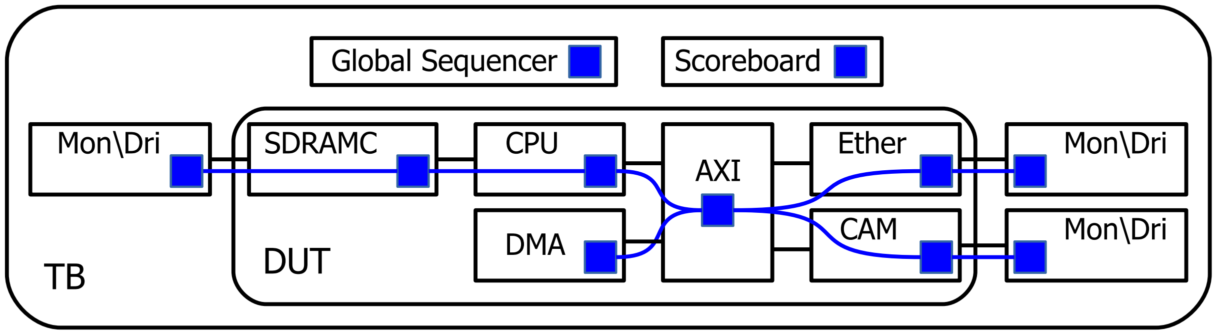

Fig. 3. Building blocks of an IOV example (SoC).

FUs interchange data. The process of building complex action graphs in UVM can be limited to communication channels from the global sequencer to FUs (or from FUs to the scoreboard).

C. Testcase guiding in PDVL

PDVL is aspect oriented. The target scenario is generated during the building process based on the given source code (see Fig. 2). This process can be done testcase specific. In PDVL, VCs are built throughout the system by using an explicit language construct for InvTrs. Here are two examples:

Assuming the CAM requests configuration data. The following PDVL code can be used in this specific testcase. It can be considered as an InfoReq:

```
cluster testcase {
    subcluster cam_section {
        it_systemBusWrite(unsigned addr, unsigned data) {
            !it_cpuStoreData(addr, data); } } }

place testcase.cam_section dut.cam;
```

By using the PDVL ”place” construct, individual aspects can be added to the system. The information flow through the system can therefore be guided for each specific testcase.

The next example can be considered as a PropAck. Assuming the Ethernet core wants to acknowledge to the DMA core, that the last payload has been successfully transferred. This can be done in PDVL by using the following code snippet:

```
cluster testcase {
    subcluster ether_section {
        it_etherPayloadAck(payloadType payload) {
            !it_dmaPayloadAck(payloadType payload); } } }

place testcase.ether_section dut.ether;
```

D. Test sequencing

Each testcase requires a specific test sequencing. We implement a global sequencer (GS) within the TB, which triggers individual subsequences within the FU using VC. These subsequences can be for instance:

1) Turn on power of CPU and peripherals
2) Configure peripherals for the specific testcase needs
3) Execute target sequence for specific coverage points

In SystemC, these sequences can be defined using processes, which interact with the payload definition, the interface sockets and the design which is to be verified. The FU related sequences designed during module level verification can be reused for system level verification as well.

E. Test execution

The GS activates the Tr inside the FU, which then initiates information requests (InfoReq). These InfoReqs travel through the VCs and adjacent FUs until the drivers in the TB convert them into signal stimuli.

In the given example, peripherals issue InfoReqs through the CPU related VC to initiate the power-up sequence, peripheral configuration etc.. The CPU uses InvTrs to evaluate, which code sequence is needed to serve the InfoReqs. The relevant code is then requested from the SDRAM controller which itself requests it from the TB driver.

After the initialization phase, the GS then initiates the Tr in the CAM FU to execute the targeted CAM transmissions. It requests stimuli data from the CAM, which then passes through the design and through the DMA channel towards the Ethernet UDP core and its monitor in the TB. The monitor converts these pattern into a PropAck, which is sent back via VCs and FUs to the CAM FU.

The test sequence is set up to generate a FIFO overflow during execution. The event is considered as a coverage point and it is checked if the CAM design signals an interrupt to the CPU. In the given example it is efficient to reported the interrupt appearance to the scoreboard via a VC.

IOV reuses the FU related Trs and InvTrs to execute a specific testcase. The DUT behavior does not need to be rebuilt in the TB in any way. Input stimuli is generated based on an InfoReq from inside the DUT. Output patterns are monitored and a PropAck is returned to the DUT for cross verification.

The example shows that processors can be integrated efficiently into the verification process. A requested bus access results in an ”on-the-fly” code generation to provide this access. The processor specific code is then requested from the SDRAMC in the given example. No program code is compiled.

F. Dynamic shortcuts

IOV supports the usage of InfoReq and PropAck shortcuts. Information does not always has to be requested from a driver or acknowledged by a monitor. InfoReq and PropAck can also be looped back internally. In the given example, the CPU can alternatively serve as a bus functional model (BFM). When a data bus access is requested by an InfoReq, the service can be provided by a BFM like behavior. No executable code is then requested by the CPU from the SDRAMC.

IOV related shortcuts can also be used when a certain activity is passed through a network of FUs. This activity propagation is acknowledged by a monitor in the TB, which itself activates the inverse PropAck transaction path. It is possible to limit the activity propagation to a predefined FU and to loop back that information by sending back a PropAck already at the FU, without transmitting the activity to a monitor in the TB.

G. Switching abstraction levels on the fly

The synthesizable design of FUs is enhanced by elements which are used for verification when using IOV. These two sections can interact with each other during verification. This local interaction can be used to optimize the verification process during execution.

This enables the possibility to vary the abstraction level of the FUs during execution. In the given example, the IOV related code of the CPU can turn the RTL CPU into a BFM or vice versa during execution ”on the fly”.

H. Scoreboard

Classical approaches in dynamic verification stimulate a DUT internal coverage point by applying pattern at the input of the DUT and by monitoring its output pattern. This is done for a given list of coverage or cross-coverage points.

IOV adds the methodology to drive test sequences internally and to detect predefined coverage points already inside the DUT within the relevant FU. This coverage information can then be transferred to the scoreboard in the TB via VCs. The definition of local coverage points can be reused when integrating the FUs in a system.

VIII. Comparison to related work

A. Portable stimulus standard (PSS)

IOV and PSS have the common goal to reuse design specific verification knowledge for system verification tasks on electronic system level.

PSS supports the verificationist to reuse a stimuli intent. Additional knowledge of the DUT must be collected to create scenarios for specific corner cases inside the DUT. The verificationist also has to collect further knowledge on how the resulting behavior can be checked for correctness.

IOV emphasizes on two main challenges. Firstly, how does the DUT must be stimulated to achieve a certain corner case? Secondly, is the result of a behavior propagated correctly? IOV introduces InvTrs to accomplish these goals. In IOV the test scheduling starts with the corner case definition within a FU. Stimuli generation and result propagation are automatically derived from the already provided verification knowledge of connected FU.

B. Comparison to related work

One motivation for proposing IOV was said to be the increasingly complex task for verificationists, to drive and to monitor design behavior and to pass the relevant information through the system in order to achieve the desired coverage.

Auto-refinement has been proposed in [4], [5] and auto-abstraction in [6]–[9]. In contrast to that, the proposed IOV methodology offers a user defined, layered, directed graph approach. IOV fills the gap between the guided activity on lowest level and an abstract command on high level to execute a given testcase. Trs and InvTrs can be reused throughout the individual abstraction layers and design sections.

Dedicated languages for system-level transactions scheduling have been proposed in [10] and [11]. Challenges for testbench reuse is discussed in [12]. SystemC (although limited) and in particular PDVL offer language frameworks which can be used for design and verification alike and allow the reuse of intellectual property as design or testbench elements when the proposed IOV is used.

It has been proposed to automatically derive coverage points (CP) from specifications [13] or assertions [14] whereas the importance of the accuracy of such a coverage metric is also elaborated [15]. The wide range of possible simulation fashions is discussed in [16].

The aspect oriented programming paradigm provided by PDVL enables an efficient reuse of CP when IOV is used. They can be defined close to the design source (DUT) for each individual Tr or InvTr on any relevant level. The DUT related CP can be reused and combined to cross-coverage products on TB level.

IX. Results

To demonstrate the efficiency of our proposed methodology we implemented the system outlined in Fig. 3 in SystemVerilog and PDVL. For each core (such as SDRAM-controller, CPU, etc.) we generated a module level stand-alone verification environment. One relevant aspect of this feasibility study is, how many lines of code (LOC) generated on module level can be reused for system level tests.

Table I (and Table II) lists all relevant SystemVerilog (PDVL) LOC numbers for each individual core. The ”DUT-LOC” numbers show the complexity of the individual cores. The ”TB-LOC (module)” are listed for the testbench on module level. This number includes code related to test sequencing and scoreboard code as well as InvTrs for PDVL. Table I (and Table II) also shows in column 4 (”TB-LOC (reuse system)”)

TABLE I
SYSTEMVERILOG LOC SOC MODULE LEVEL

System-Verilog	DUT-LOC	TB-LOC (module)	TB-LOC (reuse system)
SDRAMC	741	2384	962
CPU	10014	4680	0
DMA	9021	3857	0
AXI	961	2855	0
Ether	12220	31681	1839
CAM	657	1489	968
SUM	33614	46946	3769

TABLE II
PDVL LOC ON SOC MODULE LEVEL

PDVL	DUT-LOC	TB-LOC (module)	TB-LOC (reuse system)
SDRAMC	633	4268	2931
CPU	9213	4714	2658
DMA	3870	2893	1311
AXI	923	2638	1214
Ether	8469	18990	10725
CAM	764	2026	1466
SUM	23871	31730	17935

TABLE III
SYSTEMVERILOG AND PDVL LOC ON SYSTEM LEVEL

Level	aspect	SV	PDVL
Module	TB-LOC reuse	3769	17935
System	TB-LOC new	53217	24596
System	TB-LOC sum	56986	42531
System	TB-LOC reuse factor	6,62%	42,17%

how many LOC of the module specific testbenches can be reused on system level. The last row lists the sum of the individual numbers for the used cores.

Table III shows the relevant SystemVerilog and PDVL LOC numbers on system level, which are only related to system level integration tests. When SystemVerilog (PDVL) is used, 3769 (17935) TB-LOC can be reused from the individual module level verification environments. 52217 (24596) new TB-LOC are added. The complete system level TB-LOC are 56986 (42531). We can therefore say, that in our SystemVerilog based testcase, only 6,62% of LOC related to the system level testbench can be reused from module level testbenches. When a language is used, such as PDVL, which supports the proposed IOV methodology, 42,17% of the module level TB-code can be reused for system level tests on the complete SoC.

X. CONCLUSION

This paper proposes a guided, dynamic verification method called inside-out verification and demonstrates, how it can be combined with the existing, dominant universal verification methodology (UVM) when written in SystemC or PDVL. Based on an example we showed, that the testbench related code, which was developed on module level, can be efficiently reused when the proposed IOV is applied. An SoC example developed in the state-of-the-art language SystemVerilog allows only 6,62% of the module level testbench code to be reused on system level, whereas the same example, developed in a language supporting the IOV methodology (PDVL) results in a much higher reuse factor of 42,17% for system level verification tasks.

REFERENCES

[1] Accellera Systems Initiative. http://accellera.org, 2016.
[2] H. Foster, ”Trends in functional verification: A 2014 industry study”, DAC 2015, 52nd Design Automation Conference, June 8-12, San Francisco, CA, USA, pp. 1-6.
[3] T. Strauch, ”An Aspect and Transaction Oriented Programming, Design and Verification Language”, Euromicro 2017, 20th Euromicro Conference on Digital System Design, 30th Aug. - 1st Sep., Vienna, Austria, pp. 30 - 39.
[4] D. Chatterjee and V. Bertacco, ”Activity-based Refinement for Abstraction-guided Simulation”, HLDVT 2009, Intern. High Level Design and Test Workshop, Nov. 4-6, San Francisco, CA, USA, pp. 146-153.
[5] M. Chen, P. Mishra, and D. Kalita, ”Towards RTL Test Generation from SystemC TLM Specifications”, HLDVT 2007, Intern. High Level Design and Test Workshop, Nov. 7-9, Irvine, CA, USA, pp. 91-96.
[6] L. Liu and S, Vasudevan, ”Efficient Validation Input Generation in RTL by Hybridized Source Code Analysis”, DATE 2011, Design, Automation & Test in Europe Conference & Exhibition, March 14-18, Grenoble, France, pp. 1-6.
[7] V. Acharya, S. Bagri, and M. Hsiao, ”Branch Guided Functional Test Generation at the RTL”, ETS 2015, European Test Symposium, May 25-29, Cluj-Napoca, Romania, pp. 1-6.
[8] O. Guzey, and L. Wang, ”Coverage-directed test generation through automatic constraint extraction”, HLDVT 2007, Intern. High Level Design and Test Workshop, Nov. 7-9, Irvine, CA, USA, pp. 151-158.
[9] L. Pierre, and L. Damri, ”Improvement of Assertion-Based Verification through the Generation of Proper Test Sequences”, FDL 2011, Forum on Specification and Design Languages”, Sep. 13-15, Oldenburg, Germany, pp. 1-8.
[10] R. Emek, and Y. Naveh, ”Scheduling of Transactions for System-Level Test-Case Generation”, HLDVT 2003, Intern. High Level Design and Test Workshop, Nov. 12-14, San Francisco, CA, USA, pp. 149-154.
[11] L. Piccolboni, and G. Pravadelli, ”Simplified stimuli generation for scenario and assertion based verification”, LATW 2014, 15th Latin American Test Workshop, March 12-15, Fortaleza, Brasil, pp. 1-6.
[12] N. Bombieri, F. Fummi, and G. Pravadelli, ”On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL”, DATE 2006, Design, Automation & Test in Europe Conference & Exhibition, March 6-10, Munich, Germany, pp. 1-6.
[13] S. Verma, I. Harris, and K. Ramineni, ”Automatic Generation of Functional Coverage Models from CTL”, HLDVT 2007, Intern. High Level Design and Test Workshop, Nov. 7-9, Irvine, CA, USA, pp. 159-164.
[14] V. Athavale, S. Ma, S. Hertz, and S. Vasudevan, ”Code Coverage of Assertions Using RTL Source Code Analysis”, DAC 2014, 51nd Design Automation Conference, June 1-5, San Francisco, CA, USA, pp. 1-6.
[15] K. Ramineni, S. Verma, I. Harris, ”Evaluation of an Efficient Control-Oriented Coverage Metric”, HLDVT 2008, Intern. High Level Design and Test Workshop, Nov. 19-21, Incline Village, NV, USA, pp. 153-157.
[16] S. Yang, R. Wille, and R. Drechsler, ”Improving Coverage of Simulation-based Verification by Dedicated Stimuli Generation”, Euromicro 2014, 17th Euromicro Conference on Digital System Design, Aug. 27-29, Verona, Italy, pp. 599-606.