To appear in the proc. of the 35th Intern. IEEE Workshop on Rapid System Prototyping, 3rd Oct. 2024, Raleigh, NC, USA

Transaction Level Hierarchy Guided and Functional Coverage Driven Deductive Formal Verification

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

***Abstract*—We demonstrate how dynamic verification (e.g. simulation) can be replaced by deductive formal verification and how to benefit from the advantages of symbolic verification and the reuse of verification proofs. To do this, we swap the well-known module-hierarchy based concept with a transaction-level (TL) based alternative, which still allows us to describe the design as precisely as on RTL. We enhance the aspect-oriented and TL oriented language PDVL to support the definition of functional coverage (FC) and assertions at all levels of a TL-hierarchy.**

We then show how to use a deductive formal verification (DFV) flow which compiles PDVL code into Gallina code to be used by the Coq theorem prover. It can be argued that FC can be converted into proof obligations and that proving them is equivalent to 100% coverage. We also demonstrate how lower-level proofs can be reused when verifying aspects at higher-levels of a TL-hierarchy. We argue that the traditional assertion-based verification (ABV) methodology is still supported and SVA can be proven using DFV.

***Index Terms*—Transaction level design and verification, deductive formal verification, coverage driven verification, ABV**

I. Introduction

Verification can generally be divided into dynamic and static verification methods. Dynamic methods such as simulation are usually associated with a testbench (TB) that stimulates and monitors the design behavior. If the guidelines of the Portable Stimulus Specification (PSS) and the Universal Verification Methodology (UVM) are followed, then the functional coverage (FC) is defined and checked within testbenches. SystemVerilog Assertions (SVA) and their automatically derived coverage are typically defined within the Design Under Verification (DUV) and can be (dynamically) verified during simulation. Alternatively, static verification methods such as an SMT solver are used to statically verify SVA.

We exploit the aspect-oriented and transaction-level (TL) language PDVL [1], which allows the definition of TL-hierarchies. We associate FC and assertions with virtual transactions on individual hierarchy levels. Then we convert the DUV and verification code into Gallina [2] code to be used for deductive formal verification (DFV). We argue that FC can be converted into theorems and that proving them is equivalent to 100% FC.

In Section II FC as well as assertions and their coverage are discussed. Our work is described in Section III. Then PDVL is reintroduced briefly and the conversion to Gallina code is outlined in Section V. TL-hierarchy guided and functional coverage driven DFV is discussed in Section VI, followed by the presentation that the traditional SVA based ABV approach is also supported. The paper finishes with related work, results, and the concluding Section IX.

II. Functional coverage and assertions

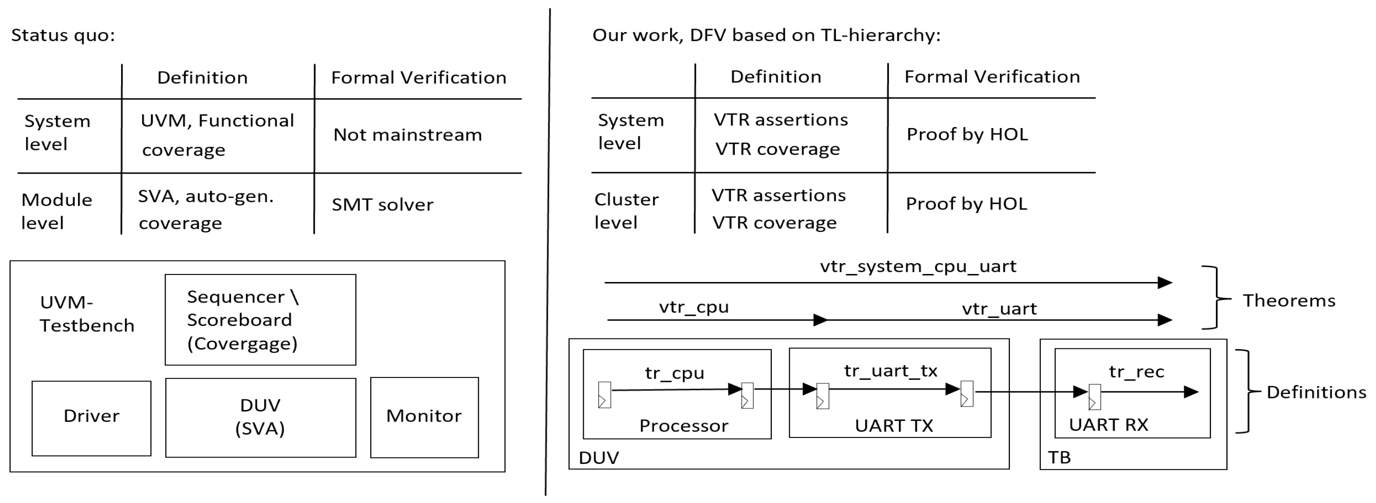
Today, FC comes in two flavors in SystemVerilog. One type of FC is sample-based coverage provided by a covergroup. Covergroups record the number of occurrences of various values specified as coverpoints. These coverpoints can be hierarchically referenced by testcases and testbenches so that it can be verified whether certain values or scenarios have occurred. They also provide a means for creating cross coverage. Unlike assertion-based cover properties, covergroups may be used in both class-based objects or structural code.

The second type of FC is assertion based. Assertions became very popular at a time when individual IP modules had to be connected to an SoC. Despite the introduction of an SoC module interconnect specification (e.g. AMBA from ARM), IP providers wanted to ensure that the bus interface of their encrypted IP block was functioning properly and was stimulated correctly. Since then, assertion-based verification (ABV) has gained greater acceptance and is now used for FC and the definition of lower-level assertions of various design aspects.

Temporal logic is widely used in property checking based formal verification as well as ABV. To express different types of properties, a variety of temporal logics have been proposed. For example, Linear-time Temporal Logic (LTL) is able to describe a property along with a single execution. However, it lacks the ability to express other possible executions and Computation Tree Logic (CTL) is proposed to solve the problem. The assertion definitions are used to (auto-)generate assertion-based FC points.

The relevant type of coverage comes from a cover property, which uses the same temporal syntax as defined by SVA. Since cover properties use the same properties as asserts, the same work in creating the properties can be reused in both checking and coverage gathering. Cover properties are typically used for protocol coverage since the temporal syntax is ideal for describing sequences of events over time, such as those required for bus interfaces.

However, cover properties can only be placed in structural code (i.e., modules, programs, or interfaces) and cannot be used in class-based objects. Likewise, their coverage information is not easily accessible in SystemVerilog (SV) for use in a testbench (e.g. steering stimulus generation).



Status quo:

	Definition	Formal Verification
System level	UVM, Functional coverage	Not mainstream
Module level	SVA, auto-gen. coverage	SMT solver

Our work, DFV based on TL-hierarchy:

	Definition	Formal Verification
System level	VTR assertions VTR coverage	Proof by HOL
Cluster level	VTR assertions VTR coverage	Proof by HOL

Fig. 1. Comparing status quo and transaction-level-hierarchy guided and functional coverage driven deductive formal verification.

Fig. 1 shows that SVAs are defined at the module level according to the current state of the art. Dynamic ABV can be based on simulation, emulation or FPGA based prototyping. All of them face the challenge of activating an assertion. When dynamic ABV is combined with constraint random or directed verification, the runtime can be very high to activate coverage points of corner cases. HW-based dynamic ABV solutions have the problem of collecting relevant assertion coverage data. Static ABV uses predominantly SMT solver to verify the correctness of assertions.

III. Our work

Our work enables the definition of FC and assertions from the lower-level up to the system-level and supports their coverage-driven DFV. It is set in contrast to the status quo in Fig. 1.

1) Adding relevant language constructs to PDVL: PDVL is an aspect-oriented and transaction-level Programming Design and Verification Language, which itself adds language constructs on top of SystemVerilog (SV).

In our work we demonstrate further improvements to PDVL to enable DFV from lower-level up to the system-level. We refer to code that asserts expected behavior as virtual transactions (VTRs) in PDVL. VTRs are not meant to be synthesizable unless explicitly specified.

2) Defining FC and assertions from lower-level up to system-level: When using SV, system-level coverage points can be defined to capture complex system behavior. To make optimal use of this possibility and to support a certain level of reuse, the UVM and the PSS were defined. Using SVA at the system level to assert complex system behavior remains challenging.

In our work we demonstrate the use of VTRs that can bundle a set of less complex VTRs. This allows the definition of a hierarchy of VTRs which asserts behavior from system-level all the way down to lower-level. Higher-level VTRs can therefore also assert system behavior that spans over DUV and TB behavior alike. This reusable TL-hierarchy based on VTRs seamlessly fills the gap which exists between lower-level SVA and the expected behavior defined by the UVM.

3) Verifying FC and assertions from lower-level up to system-level: System-level coverage is primarily captured using simulation techniques. As symbolic simulation is still limited, many simulation runs must be executed by directed or constraint random tests. Additionally, the possibility to reuse verification throughout the module hierarchy is limited. Verifying system-level coverage using static methods such as DFV continues to pose significant challenges, and its usage cannot be considered mainstream today.

In our work we enable VTRs to define coverage. VTRs can be used to assert behavior seamlessly throughout the complete hierarchy, from lower-level up to the system-level. Therefore, coverage can be defined and verified throughout the TL-hierarchy as well. We demonstrate how these VTRs and their associated coverage can be converted into Gallina code and how the individual coverage points can be proven. We argue that individual coverage points of VTRs can be associated with individual proofs. Subsequently, VTRs which are built on other VTRs can define coverage that can be proven by reusing proofs of their lower-level VTRs.

4) Using lower-level assertion definition and associated coverage: Lower-level assertion definitions (such as SVA) are very well established for lower-level logic such as finite state machines (FSMs) and interface protocols. Usually, assertion coverage definition is automatically derived from these lower-level assertions and can be verified statically by SMT solvers or during dynamic verification (e.g. simulation).

Our work demonstrates how to reuse SVA in PDVL and discusses the conversion of the design and the lower-level assertion definitions into Gallina code. Combined with the compiled design behavior, the assertions can then be verified by proving using a DFV tool like the Coq proof assistant [3].

5) Proof by symbolic simulation: In our work we focus on DFV based on symbolic simulation, as demonstrated in [4]. Theorems can be proven that for a given initial state and a given stimulation sequence, where state and sequence values can have a symbolic type, all possible resulting states and state transitions match the expected behavior.

IV. PDVL

PDVL was introduced in 2017 [5]. We therefore only give a very brief overview. PDVL is based on SV and encapsulates conditions and assignments. The latter are then called datapaths. Transactions (TRs) determine the conditions under which individual datapaths are valid. PDVL is not limited to CPUs, but an example of a RISC-V instruction is given in Alg. 1.

Aforementioned elements are stored in clusters. The aspect-oriented paradigm of PDVL becomes obvious, when looking at the hardware generation process (Alg. 2). A hardware module hierarchy is built, and clusters are then joined into the individual modules. The merging and signal routing must be handled by the compiler. More information can be found in the PDVL specification [1].

In this paper, we introduce the concept of VTRs which allows us to define a TL-hierarchy (Fig. 2). We also show by examples, which SV constructs are reused in PDVL for FC and assertion definition.

V. PDVL to Gallina compiler

The tool “MRPHS” compiles PDVL code into synthesizable SV code. MRPHS’ PDVL to Gallina compiler extension was introduced in [6]. We therefore only give a very brief overview.

The design “DUV” must be generated by using the relevant PDVL build commands (Alg. 2) before PDVL code is compiled into a Gallina representation (Alg. 3). Once the logic is joined, all sequential and combinatorial signals of the DUV are identified.

We mention the standard Boolean type ”bool” for signal bits in this paper, but any user-defined type can be used instead. Alg. 3 shows that all signals are added to an inductively defined type t_item and that a dynamic list t_state can be generated based on t_item members.

Only signals with a defined value are added to the t_state list. A signal can be removed from the list once its value becomes undefined. This applies to both sequential and combinatorial signals. The t_state list therefore contains all sequential elements and combinatorial signals that are defined at a given point of time.

Alg. 3, line 12 shows that design-specific functions are defined, which are subsequently used in the generated Gallina code. MRPHS also generates a set of more general functions which, for instance, assign a given value to an item in the t_state list (e.g. f_set) or extract the value of a specific item in the t_state list (e.g. f_get). The generated Gallina code mentioned so far is stored as a DUV specific library.

For each condition in the PDVL source code, a Boolean signal is generated and added to the item list as well. The condition body is converted into a Gallina definition.

All condition, datapath, and transaction definitions in PDVL are compiled into Gallina code and stored as a second DUV-specific library, which is used by the final proof scripts. Alg. 4 gives an example related to the aforementioned RISC-V ADDI instruction. We will see in the following section, that theorems can be proven based on the automatically generated Gallina code mentioned so far.

Algorithm 1 PDVL: RISC-V instruction example (ADDI)

```
cl_instr_addi {                                   (* cluster *)
c_instr_i_addi { if (opcode_i == 7’h13        (* condition *)
  & funct3i == 3’h0) this; }
d_addi { dp_out = rs1_dato+instr[31:20]; }  (* datapath *)
tr_rv32i_addi {                               (* transaction *)
  unique @c_instr_i_addi { d_rs1i_addr; d_addi;
    d_rd_dp_out; d_rd_addr; c_rf_write; d_pc4; } } }
```

Algorithm 2 PDVL: Generating a module hierarchy, joining a cluster into a submodule and defining clock input and edge sensitivity for registers.

```
build TB {
  build i_duv DUV;                       (* build hierarchy *)
  join cl_rv32i cl_rv32imc;                       (* joining *)
  join { tr_reg { @e_clk { tr_rv32i_addi; }}} cl_rv32imc;
  join cl_rv32imc i_duv; }
```

MRPHS generates Gallina definitions to support symbolic simulation techniques. The definition sim_update (Alg. 5, line 1) walks through the complete design and updates all possible non-sequential signals. Assuming the DUV has only one single clock, then the definition sim_cycle (Alg. 5, line 1) simulates a complete cycle, which includes an update of all sequential elements, followed by an update of all non-sequential signals. For more complex clock domain structures, the sim_cycle definition is adapted accordingly. Alg. 5, line 3 demonstrates, how two cycles of the design can be simulated.

Algorithm 3 Gallina: Signal definition examples, the design state list and a boolean function example

```
 1: Inductive t_item : Type :=
 2:   | i_nil
 3:   | instr ( l : t_bus32 )
 4:   | ...
 5:   | pc ( l : t_bus20 )
 6:   | reg_file ( l : t_arr32x32 ) .
 7:
 8: Inductive t_state : Type :=
 9:   | st_nil
10:   | st_cons ( s : t_item ) ( l : t_state ) .
11:
12: Definition f_equal32 ( a b : t_bus32 ) : bool := ...
```

Algorithm 4 Gallina: Compiled condition, datapath and transaction definitions (ADDI example)

```
Definition c_instr_i_addi ( st : t_state ) : t_state := ...
Definition d_addi ( st : t_state ) : t_state := ...
Definition tr_rv32i_addi ( st : t_state ) : t_state := ...
```

Algorithm 5 Gallina: Helper definitions for symbolic simulation

```
Definition sim_update ( st : t_state ) : t_state := ...
Definition sim_cycle ( st : t_state ) : t_state := ...
Definition two_cycles ( st : t_state ) : t_state :=
  sim_cycle (sim_cycle st).
```

VI. TL-HIERARCHY GUIDED COVERAGE DRIVEN DFV

A. *Introducing virtual transactions*

In this section we introduce virtual transactions (VTRs) which allow us to define a TL-hierarchy and ultimately the reuse of lower-level verification results on higher-level. VTRs can define testbench related functionality such as sequences, randomness, coverage points, etc.. A VTR can bundle a set of TRs and VTRs. This generates a TL-hierarchy that can have multiple top-level VTRs, whereas each of the top-level VTRs groups a specific set of TRs and VTRs for a specific verification goal. VTRs are not meant to be synthesizable unless explicitly specified.

One of the key ideas behind PDVL is that it extends the SV language by only a limited number of new constructs. It is therefore intended that VTRs in PDVL reuse many of the testbench related language constructs known from SV. We mention some keywords by providing examples.

B. *VTRs and testbench related language constructs*

Fig. 2 gives an overview of the design example we use to demonstrate our methodology. The synthesizable UART transmit (TX) module in the DUV provides standard UART TX capabilities. The outgoing datastream is captured by an UART monitor. We now show how a VTR (Alg. 6) can be defined, which creates a verification environment for these two entities.

A VTR can define time consuming behavior by using the keyword sequence (Alg. 6, line 5). A sequence always starts at the “init” state, can traverse a finite number of user defined states (here “finish”) and becomes redundant when reaching the keyword “exit” (Alg. 6, line 13). A sequence can therefore also be considered as an FSM.

In our example (Alg. 6, line 6), a random “symbolic” value is assigned to the byte “axi_tx_data” during the “init” state and the condition “c_axi_trans” is set valid. The sequence then transitions to the finish state. Once the condition “c_uart_rx_valid” is true, it is checked whether the coverage point “cp_tx_rx_eq” is covered and the sequence is exited.

If the VTR “vtr_tx_rx_transfer” is called within a VTR that defines the clock for the registers involved, then the sequence is clocked by the specified clock (e.g. UART clock). In this case we are talking about a cycle-timed VTR.

A cycle-timed VTR checks signal values and condition states in the respective cycle. It also assigns condition states and signal values to the relevant signal at the given clock edge.

C. *VTRs as instruction code generator*

A VTR can call other VTRs. The calling VTR can then be considered as an untimed VTR, which has more the character of a function written in a sequential programming language. The called VTR starts at the “init” state of the sequence. The called VTR finishes when the “exit” keyword is reached. At this timepoint, the calling VTR continues processing its sequence.

Algorithm 6 PDVL: VTR UART TX-RX Transfer

```
 1: cluster tb_axi_uart {
 2:   c_tx_rx_data_eq {
 3:     if (uart_rx_data == axi_tx_data) this; }
 4:   vtr_tx_rx_transfer {
 5:     sequence tx_rx_transfer {
 6:       init: {
 7:         random axi_tx_data;
 8:         c_axi_trans;
 9:         finish; }
10:       finish: {
11:         @c_uart_rx_valid {
12:             cover cp_tx_rx_eq { c_rx_tx_data_eq; }
13:             exit; } } } } }
```

The execution of the resulting VTR tree becomes time consuming when a VTR calls a cycle-timed VTR. Our enhanced PDVL version supports the fork-join and other known mechanisms to run multiple time-consuming sequences in parallel.

An example of an untimed VTR is an instruction generator for an UART TX driver shown in Alg. 7 called “vtr_cpu_uart_tx_driver”. It is also outlined in Fig. 2 in the sequencer block, indicating that it calls the CPU multiple times to force the CPU to issue individual bus master writes.

After assigning a symbolic value to “axi_tx_data”, the VTR successively calls a list of VTRs. The goal is that the UART TX Enable Bit is set (“vtr_cpu_uart_tx_enable”) and that the “axi_tx_data” value is written into the UART TX register (“vtr_cpu_uart_tx_data”) which starts the transmission. Finally, the CPU should constantly loop over a NOP instruction (“vtr_cpu_loop_nop”). The called VTRs force the CPU in the given system to execute individual instructions, which results in individual AXI bus-writes.

The VTR “vtr_cpu_uart_tx_data” (Alg. 7, line 10) shows how the CPU can be forced to execute a sequence of instructions to issue the relevant AXI master data write command.

VTRs can be untimed, timed and cycle-timed. When a VTR is called by a VTR outside an edge sensitive block, the VTR is untimed and the VTRs interact by an calling-init-exit-continue process. Multiple VTRs can also interact using a fork-join or alternative mechanisms.

D. *VTRs and coverage related constructs*

PDVL supports the concept of SV to assign a random value to a signal. This is the case, for example, if the payload of a data transmission is generic. PDVL uses the keyword “random” to indicate, that any valid value within the given range of a signal must be considered during verification (Alg. 6, line 7 and Alg. 7, line 5). In the remainder of the paper we use the term ”symbolic” value for such an assignment.

We also introduce a coverage definition for individual VTRs. A coverage point is defined by the keyword “cover”, followed

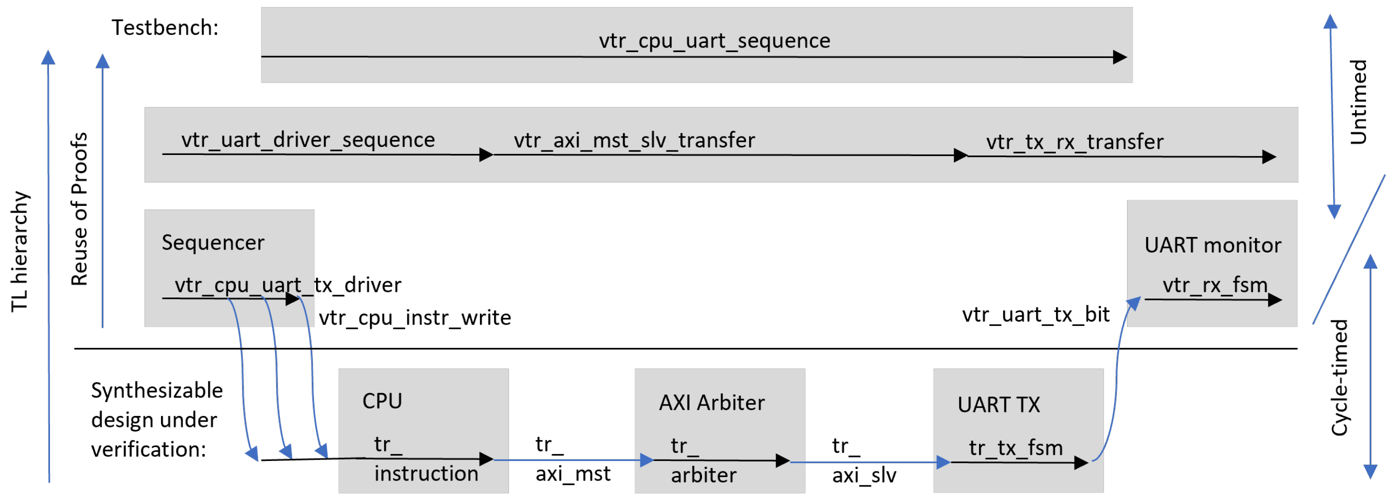


Fig. 2. Transaction level hierarchy example, showing the design under verification (DUV) and transactions as well as the testbench (TB) and virtual transactions.

Algorithm 7 PDVL: CPU UART Driver Instruction Generator

```
cluster tb_cpu_uart_driver {
  vtr_cpu_uart_tx_driver {
    sequence cpu_uart_driver {
      init: {
        random axi_tx_data;
        vtr_cpu_uart_tx_enable;
        vtr_cpu_uart_tx_data;
        vtr_cpu_loop_nop;
        exit; } } }
  vtr_cpu_uart_tx_enable { ... }
  vtr_cpu_uart_tx_data {
    sequence cpu_tx_data {
      init: { instr2;
        ... /* code executing ”lui s0, 0x80030;” */ }
      instr2: { instr3;
        ... /* code executing ”li a5, axi_tx_data;” */ }
      instr3: {
        ... /* code executing ”sw a5, 4(s0);” */ }
        exit; } } }
  vtr_cpu_loop_nop { ... }}
```

by a property name and a body that, for example, checks whether a certain condition is true at a given cycle (Alg. 6, line 12).

E. TL-hierarchy

So far we have discussed the CPU and the UART section of our example design. We need to mention the AXI arbiter block, which basically converts AXI master writes to the relevant AXI slave writes. A VTR can also be defined for an AXI master-slave transfer, such as “”vtr_axi_mst_slv_transfer” in Fig. 2.

There are additional TRs and VTRs which connect the design modules and testbench elements. Fig. 2 illustrates that all TRs are part of the synthesizable design. We have also discussed VTRs which give an abstract view of local behavior such as a CPU sequence and a UART TX-RX data transfer.

We now broaden the perspective and introduce a VTR which produces a system-wide sequence, shown in Fig. 2. In our example, the top-level sequence “vtr_cpu_uart_sequence” forces the CPU to generate a set of instructions which enable the UART to transmit data. The CPU then writes a symbolic value into the UART, which is then transmitted and captured by the UART monitor. Fig. 2 illustrates that it can be seen as a top-level VTR covering previously mentioned VTRs so far. We will see how our approach to use DFV will benefit from such a TL-hierarchy.

F. VTRs and deductive formal verification

We now show how VTRs and the resulting TL-hierarchy can be used for DFV. We improved the PDVL to Gallina compiler (“MRPHS”, Section V) to support VTRs.

1) Proving theorems based on combined transactions: Alg. 8 shows a simplified code of the example VTR (Alg. 6) compiled into Gallina code. The theorem lists a symbolic value as a hypothesis. Coverage properties are compiled into definitions, which check that the defined conditions are true (“cp_uart_tx_rx”). The sequence “vtr_tx_rx_transfer” is also compiled into a definition, which modifies the list of the given design states. The compiled and provable theorem “th_tx_rx_transfer” executes the “vtr_tx_rx_transfer” and checks whether the property “cp_uart_tx_rx” is covered.

2) Proving theorems based on sequences: In this subsection, we refer to the CPU UART driver code generator described in Alg. 7. To prove that each of the code generation sequences (vtr_cpu_uart_tx_enable and vtr_cpu_uart_tx_data) initiates correct AXI master writes, individual cover properties can be defined similar to the coverage property in Alg. 6.

Alg. 9 lists the compiled and provable theorem for the “cpu_uart_driver” sequence listed in Alg. 7. After the individual sequences have been executed, it is checked whether the relevant properties are covered (“true”).

Algorithm 8 Gallina: Coverage theorem (simplified)

```
Theorem th_tx_rx_transfer :
    foreach axi_tx_data : t_bus,
    cp_uart_tx_rx (
    vtr_tx_rx_transfer ( reset_st )).
```

Algorithm 9 Gallina: CPU UART driver sequence

```
Theorem th_cpu_uart_driver :
    foreach axi_tx_data : t_bus,
    cp_cpu_tx_data (
    cp_cpu_tx_enable (
    vtr_cpu_loop_nop (
    vtr_cpu_uart_tx_data ( axi_tx_data
    vtr_cpu_uart_tx_enable ( reset_st ))))).
```

G. *Reuse of proofs*

The ability to define more abstract VTRs to build a TL-hierarchy can be particularly useful when proofs of lower-level theorems derived from lower-level TRs or VTRs can be reused. We start with a top-level VTR.

1) Top-level VTR reuses lower level proofs: We can see in Fig. 2 how the top-level VTR "vtr_cpu_uart_sequence" functionally extends over all VTRs mentioned so far. We want to prove that the symbolic value which is programmed by the CPU is transmitted from the UART TX module to the UART monitor. This can be asserted within a VTR similar to the "cp_uart_tx_rx" cover property in Alg. 6, line 12. The top-level VTR "vtr_cpu_uart_sequence" needs to call the same sequence as defined in Alg. 7, lines 6-8.

After compiling the VTR "vtr_cpu_uart_sequence" to the relevant theorem, the proof can reuse proven theorems of lower-level theorems such as the one of "th_tx_rx_transfer" etc..

2) Mid-level VTR reuses lower level proofs: When proving mid-level theorems such as "th_tx_rx_transfer", reusing lower-level theorems (not previously mentioned) is also advantageous.

In the given UART example, it can be argued that a finite state machine (FSM) controlling the UART TX has its counterpart in the FSM of the UART receiver (RX) in the testbench monitor. For simplicity, let's assume that the transmission is based on a synchronous single-bit protocol.

The TR in the TX-FSM that defines the transmission of a single bit is combined with the TR in the RX-FSM to form a new VTR that defines the entire transmission process of a single bit value. This abstract VTR in PDVL is compiled into Gallina code which can be used for proving. A theorem is proven stating that the transmitted data bit value is correct.

The same mechanism is applied for the transfer of a byte and a full packet, while using the previously formally verified theorem of transmitting a bit (or byte respectively).

H. *SVA in PDVL and intermediate representation*

The work in [7] shows how SVAs can be converted into synthesizable code by extracting FSMs, datapath and checker logic. We follow these guidelines and generate an IR that is in-line with PDVL representations. Local SVA variables are converted into PDVL items. The Boolean layer is compiled into condition and datapath logic as defined by PDVL. SVA sequences and properties are converted into VTR sequences as defined in this paper. Here multiple VTR sequences can result from this conversion process and the possibility to use fork-join methods becomes relevant. SVA local variables can also be compiled into registers within the IR. Assumptions and coverage goals are converted into coverage properties.

At the current state, our work does not support the full range of SVAs. There are limitations in aspects like liveliness, overlapping transactions, etc.. However, it is not our goal to fully support this specific methodology. Let's assume there are SVAs in a sender and a receiver of an interface. We argue that our approach of a TL-hierarchy covers both sides with VTRs spanning over sender and receiver functionality, so that local SVAs become redundant.

VII. Related work

BSV: Bluespec SystemVerilog (BSV) [8] is a high-level hardware description language of guarded atomic actions. The BSV concept is based on BSV rules and a term rewriting system, whereas each rule can be viewed as a declarative assertion expressing a potential atomic state transition. In [9] it is discussed, how the modular concept of BSV generates new challenges for predicting the compiler output. The sequential behavior of the hardware defined by PDVL is exact and the cycle behavior of the compiler's SV and Gallina code is therefore predictable since no scheduling is involved.

KAMI: According to [10], KAMI is a framework to support implementing, specifying, formally verifying, and compiling hardware designs based on BSV and the Coq theorem prover. It emphasizes modular verification of digital hardware. In contrast, using PDVL provides a TL design paradigm and a TL formal verification paradigm. TRs can span multiple modules, creating a TL-hierarchy that can range from low-level cycle-accurate TRs to approximately-timed or untimed VTRs. The concept of a module becomes only relevant when the final SV code is generated.

Rules rewriting: The work in [11] describes the proving and disproving of assertion rewrite rules with automated theorem provers. The work is based on the assertion language PSL and concentrates on rule rewriting. We outlined our flow to convert SVA into an IR of FSMs, datapath and checker logic, which is then compiled into Gallina code and used for DFV. We do not use rewriting as such, but benefit greatly from reusing proofs through applying proven theorems.

RTL to TLM: The main intent of the work in [12] is to automatically build a dynamic ABV environment for a TLM model, with no restrictions on the abstraction level, by starting from a set of properties initially defined for a corresponding RTL implementation. First, cycle-accurate RTL properties are automatically rewritten into a set of properties suited to be checked on an event-based TLM model. Secondly, an approach is defined to synthesize TLM properties into checkers to be adopted for dynamic ABV of the TLM model. In our work, we use a TL language to define TL assertions throughout a user defined TL-hierarchy.

TL assertion language: In [13] an assertion specification language is presented which is based on formal definitions that allows the specification of TL properties and their execution in simulation. It derives the language from known ABV languages and extends these by the required TL functionality. It also explains how simulation traces of finite length can be checked against properties. Our work, on the other hand, is optimized for DFV. Nevertheless, the work in [13] was a great inspiration for our work, especially because it is based on industry experience.

VIII. Results

The SoC design was written in PDVL. The number of synthesizable TRs and VTRs are listed in Tab I for each individual core and the complete SoC. MRPHS was used to compile the design into synthesizable SV code. The design and the VTRs are compiled into Gallina code. The compilation process of the SoC is completed within less than 10 seconds for each of the two outputs. For compilation and runtime evaluation, we use an i7, 2.6GHz CPU.

The number of resulting theorems is listed in Tab. I. It also shows the complete consecutive execution time (CCET) of all proofs on a single thread for each individual core and the entire SoC. We introduce a maximal incremental runtime (MIRT), which is the worst case runtime to prove all relevant proofs based on an incremental change in the source code.

We developed a testbench in SV with the same coverage we achieve by the DFV flow. We use Verilator to run the regression suite. Tab I lists the simulation runtime (SRT) for each individual core and the entire SoC.

The execution time of a simulation based regression suite (SRT) is faster than the complete execution time of the DFV flow (CCET). This is mainly due to Coq theorem prover runtime issues. Nevertheless, DFV-based tests can be run in parallel to reduce this disadvantage, just like simulation-based regression suites. Alternatively, a faster theorem prover might eventually be used as our DFV is not limited to the Coq theorem prover.

The runtime of the DFV flow when only an incremental update needs to be verified (MIRT), looks promising compared to the individual simulation runtime (SRT) of individual core related tests.

We see one of the most important advantages at the system level. While it is becoming increasingly difficult to write efficient system-level tests in simulation, we have found that it is very inviting to write top-level tests for our DFV flow. The possibility to reuse lower level proof is particularly helpful when solving software driver related issues. In this respect, we see a clear advantage of our demonstrated DFV flow over a simulation-based verification approach.

IX. Conclusion

We have demonstrated the use of deductive formal verification (DFV) to prove functional coverage and assertions as an alternative to simulation-based verification. One key contribution of DFV is the ability to symbolically verify complete coverage areas in a deductive manner. In order to benefit from verification reuse, we introduce a transaction level hierarchy, which enables the definition and verification of functional coverage and assertions from lower-level to system-level.

This hierarchical verification approach allows us to focus on level specific verification challenges and enables incremental updates when behavior only changes locally. It fills the gap that exists today between low-level SVA-based verification and the state-of-the-art coverage-based verification defined in UVM and PSS. At the same time, lower-level SW routines (e.g. peripheral drivers) become an integral part of the system verification at the same time.

TABLE I
Results for reference SoC design.

	TR	VTR	Theorems	Coq	Coq	Verilator
				CCET	MIRT	SRT
				[sec]	[sec]	[sec]
RV32IMC	107	144	219	52.4	5.01	27.8
SDRAM	93	160	197	55.9	3.49	27.0
Ethernet	72	89	106	18.3	2.48	14.0
AES	64	76	111	13.7	2.92	8.03
SoC	445	813	1048	186	6.26	125

References

[1] T. Strauch. PDVL Specification v0.1. [Online]. Available: https://github.com/cloudxcc/PDVL

[2] Gallina Development Team. The Gallina specification language. [Online]. Available: https://coq.github.io/doc/v8.9/refman/language/gallina-specification-language.html

[3] Coq Development Team. The coq proof assistant. [Online]. Available: http://coq.inria.fr/

[4] Y. Morihiro, and T. Toneda, "Formal verification of Data-path Circuits Based on Symbolic Simulation", Proc. of the Ninth Asian Test Symposium, 2000, 6th Dec, Taipei, Taiwan, pp. 1-8.

[5] T. Strauch, "An Aspect and Transaction Oriented Programming, Design and Verification Language", IEEE Euromicro DSD 2017, 30 Aug. - 1 Sep., Vienna, Austria, pp. 30 - 39.

[6] T. Strauch, "Deductive Formal Verification of Synthesizable, Transaction-level Hardware Designs Using Coq", Design, Automation & Test in Europe Conference & Exhibition (DATE), 25-27 March 2024, Valencia, Spain, pp. 1-6.

[7] J. Long, and A. Seawright, "Synthesizing SVA Local Variables for Formal Verification", 44th ACM/IEEE Design Automation Conf (DAC), 4-8 June 2007, San Diego, CA, USA. pp. 75-80.

[8] R. Nikhil, "Bluespec SystemVerilog: Efficient, correct RTL from high level specifications," Proc. 2nd ACM and IEEE Int. Conf. Formal Methods and Models for Co-Design, MEMOCODE'04, 23-25 June 2004, San Diego, CA, USA, pp. 69–70.

[9] M. Vijayaraghavan, N. Dave, and Arvind, "Modular Compilation of Guarded Atomic Actions", ACM/IEEE Intern. Conf. on Formal Methods and Models for Codesign, MEMOCODE 2013, 18-20 Oct. 2013, Portland, OR, USA, pp. 177-188.

[10] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. "Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification", Proceedings of the ACM on Programming Languages, Volume 1, Issue ICFP, Article No.: 24, pp 1–30.

[11] K. Morin-Allory, M. Boulé, D. Borrione, and Zeljko Zilic, "Proving and Disproving Assertion Rewrite Rules with Automated Theorem Provers", IEEE Intern. High Level Design Validation and Test Workshop, 19-21 November 2008, Incline Village, NV, USA, pp. 56-63.

[12] Nicola Bombieri, Riccardo Filippozzi, Graziano Pravadelli and Francesco Stefanni, "RTL property abstraction for TLM assertion-based verification", Design, Automation & Test in Europe Conference & Exhibition (DATE), 9-13 March 2015, Grenoble, France, pp. 85-90.

[13] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Specification Language for Transaction Level Assertions", IEEE Intern. High Level Design Validation and Test Workshop, 8-10 November 2006, Monterey, CA, USA, pp. 77-84.