A Classification of Design Steps and their Verification

Wolfgang Ecker
Siemens AG
Corporate Research and Development
Munich, Germany, 81730

Abstract

Hardware design using the hardware description language VHDL has to consider three independent property scales that influence the design process from an abstract level to the gate level, namely the design view, the timing aspect, and the value representation.

Considering this classification, a systematic way for design steps and their verification with special emphasis on VHDL is presented in this paper.

1 Introduction

VHDL[1] is a world-wide accepted and applied standard hardware description language. It was originally developed as design and as description language in the VHSIC[2] project of the United States DoD[3]. This gave VHDL its original name VHD^{2}L[4].

Two reasons forced VHDL to be a very powerful however complex language: Its target which are very complex systems and its goal to be design and description language. The first aspect requires a wide range of different description capabilities to allow for design-oriented descriptions during the design process. Examples are a large variety of different types, support of reactive as well as imperative descriptions, and different levels of abstraction. The second aspect requires deterministic and portable descriptions to allow for design data exchange and deterministic models. This was achieved by the explicit definition of the VHDL simulation algorithm together with elaboration and execution regulations based on the simulation algorithm.

In 1987, VHDL was standardized by IEEE, and shortly after, a lot of different VHDL CAD-tools were available commercially. A formalized methodology, however, for the use and application of the complex language did not exist. This paper presents a classification for design steps and their verification based on the design cube presented in [5]. It serves as frame for a VHDL-based design methodology but can be used for other languages also.

In the first sections, aspects of design and verification[5] are discussed. Verification strategies, the classification of description styles and design steps as well as a systematic approach for verification of design steps are presented in the subsequent three sections. A summary and an outlook to VHDL-based requirement representation conclude the paper.

2 Design and Verification

2.1 Design and Design Step

During design process, information is added to already known features. An existing description or design representation, which is also called *specification* is refined. Defining S as a representation of a specification, C, the constraints, as the explicit or implicit design process degree of freedom, and D as the design decisions taken during the design process then the representation of the result of the design process R can be understood as a function $\mathcal{D}$:

$$R = \mathcal{D}\left(\left(S, C\right), D\right)$$

For complexity reasons, the design process mostly is divided into a set of design steps. The design result or (pre-)implementation is refined by each design step piece wise. This design strategy is called top-down design.

$$R^i = \mathcal{D}^i\left(\left(S^i, C^i\right), D^i\right) \quad (1)$$

$$\left(S^{i+1}, C^{i+1}\right) = R^i \quad (2)$$

The result of the i-th design step also represents the specification and the degree of freedom of the $i+1$-th design step. The design process as a whole consists of a concatenation of the functions $\mathcal{D}^i$ representing the transformations of the i-th design step, ie:

$$\mathcal{D} = \mathcal{D}^1 \circ \mathcal{D}^2 \circ \ldots \circ \mathcal{D}^n \quad (3)$$

Due to the fact that the representation of a specification should allow for refinement, the representation may not carry all information of the final design result. This can be achieved in three different ways:

[1] VHSIC Hardware Description Language, see [7]

[2] very high speed integrated circuits

[3] Department of Defense

[4] VHSIC Hardware Design and Description Language

[5] Verification is seen in this paper as full or partial verification by formal methods (see [3]), worst case estimations as used in timing verification or as complete or incomplete correctness check via simulation. Main focus however lies on the simulation aspect.

Verification can be assigned to the question "are we building the product right?". Validation in comparison answers the question "are we building the right product?" (see [2])

1. Specification consists of requirements only, ie. as little implementation details as possible are coded explicitly. The degrees of freedom are implicit in this case. An example for this kind of description are the VHDL built-in assertion statements or the VHDL annotation language VAL [1].

2. Specification describes degrees of freedom explicitly. An example is the specification of sets of values instead of one value. This can be performed using the *don't care value* from the VHDL standard logic [6] or nondeterministic statements like included in UDL/I [11, 12].

3. Degrees of freedom are based on abstract descriptions. Freedom results from different implementation possibilities, which satisfy the specification respectively the abstract description.

The first two possibilities are not considered for functional specification in this paper. The third possibility will be discussed in more detail.

The major problem which arises from using descriptions with different levels of abstraction is the definition of the abstraction levels. This is required to allow for systematic definition of design steps and their verification.

2.2 Verification of a Design Step

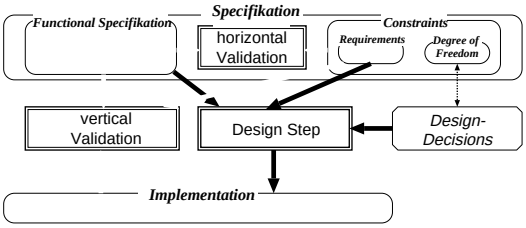

Figure 1: Design and Verification

Due to [13] verification of one design step can be split into two tasks (see figure1):

1. *Horizontal verification* which checks consistency between functional specification and its requirement.

2. *Vertical verification*, which checks correctness of the design steps. We focus in this paper mainly on verification of design steps.

Vertical verification can in addition be refined in checking consistency between implementation and its functional specification and testing satisfaction of requirements by the implementation. We call the first part of vertical verification *functional verification* and the second part *requirement verification.*

3 Verification Principles

According to [1], two different approaches for verification are reasonable:

1. *Constraint verification.*

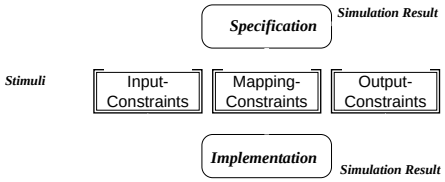

Figure 2: Verification of Constraints

2. *Equivalence verification.*

Verification of constraints checks, as shown in Figure 2, the correctness of input-, output- and mapping constraints for both, the specification and the implementation. This verification step, however, must not be performed in one run. Either specification or implementation can be checked. Due to the fact, that this check is implementation-independent, the check is obviously assigned to the interface. Concurrent assertion statements, passive processes or (passive) concurrent procedure calls are proper VHDL constructs for this task. VAL (see [1]) allows to describe requirements in a more comfortable way and supports verification in an VHDL environment.

Figure 3 shows an approach for the verification of equivalence. Here, both, specification as well as implementation must be simulated concurrently. The test consists in comparing the results of both models considering their abstraction levels.

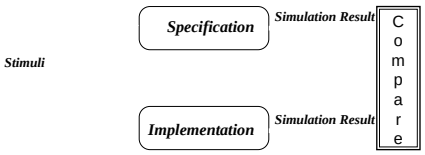

Figure 3: Verification of Equivalence

Constraint verification strongly relates but is not assigned to *horizontal* and *vertical requirement verification* and *equivalence verification* to *vertical functional verification.* This will be discussed in detail in section 5. Before, a classification of design steps will be proposed in the next section.

4 Classification of Design Steps

This section classifies design steps accordingly to the design cube, which is shortly described in the next subsection.

4.1 The Design Cube

As mentioned in section 2.1, a systematic way is required to allow to classify design steps. We selected the design cube [5] due to the fact, that it seems well suited for hardware description languages, especially VHDL [9].

The design cube realizes three independent properties: the view of the model, timing property, and abstraction of values. In this sense, a model is a single point in the three-dimensional space spanned by

the axes called *view*, *timing*, and *values*. This space is called the *design space*.

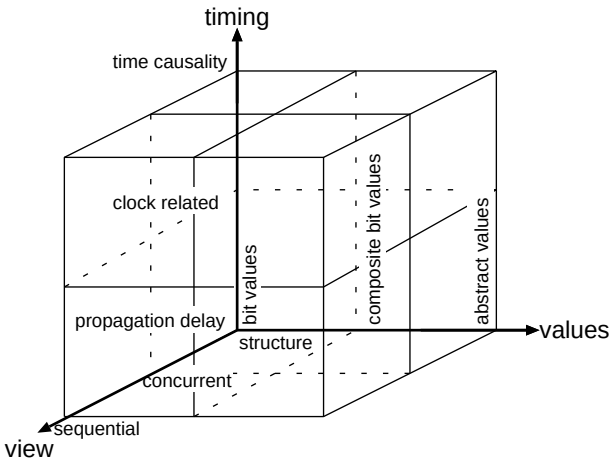

Figure 4: The Design Cube

A set of discrete coordinates classifies differences in view, timing and values. Geometrically, these coordinates describe a three-dimensional cube, which is called the *design cube*. Graph theoretically, the design cube is a lattice graph G which is described as follows:

1. The vertex set consists of all triples $(view\_spec, timing\_spec, value\_spec)$, where

   - $view\_spec \in \{structure, concurrent, sequential\}$;
   - $timing\_spec \in \{propagation\_delay, clock\_related, time\_causality\}$;
   - $value\_spec \in \{bit\_values, composite\_bit\_values, abstract\_values\}$.

   These sets are assumed to be ordered.

2. There is an edge between the vertices (x_1, x_2, x_3) and (y_1, y_2, y_3) if and only if the vertices differ in exactly one component, say i, such that either x_i covers y_i or y_i covers x_i with respect to the corresponding set ordering.

3. An edge is directed from vertex $Y = (y_1, y_2, y_3)$ to vertex $X = (x_1, x_2, x_3)$, if the distance of X from the coordinate origin

   $$(structure, propagation\_delay, bit\_values)$$

   equals the distance of Y from the origin minus 1. (Here, "distance" means the ordinary graph theoretic distance function, which measures shortest paths between vertices. The length of a path is its number of edges.)

   An edge in the cube represents one (primitive) design step.

4. We extend the design cube of [5] with hyper-edges. They describe composed design steps except hyper-edges starting and ending in the same point. These hyper-edges classify optimizations.

   Hyper-edges representing optimizations are not directed in the design due to the fact, that they start and end in the same point. We assign these hyper-edges a direction to distinguish between view-, value- and timing optimization.

Design steps and optimizations are discussed in detail. We assume, for simplification reason, that a composite design step can be split into its primitive design steps and that its verification can be performed by combination of verification approaches described below.

4.2 Optimization

Optimization is a special design step, which improves measured or estimated costs of a design but keeps the abstraction due to the design cube unchanged. As mentioned above, different optimization classes, namely view optimization, value optimization, and timing optimization exist.

4.3 View Transformation

The view transformations partitioning and structuring are in the design cube transformations parallel to the view axis. Ie. all coordinates representing the modeling style of the transformed units with exception of the coordinate *view_spec* remain unchanged.

Partitioning

Consider the problem of partitioning a design. Partitioning[6] means to divide one sequential specification into a set of concurrent interacting units. This process can be described by the design cube in terms of transformations from the points

$$(sequential, timing\_spec, value\_spec)$$

to the points

$$(concurrent, timing\_spec, value\_spec),$$

where *timing_spec* and *value_spec* are constant during one transformation (see Section 4.1 Point 1).

It is important that such a partitioning opens new design cubes associated with each component. The coordinates of the modeling style in each sub-cube are

$$(sequential, timing\_spec, value\_spec),$$

[6] In this case concurrent partitioning only is considered as design step. Sequential partitioning is either performed for software engineering and description reasons or for space optimization reasons.

ie., each sub-unit is described sequentially.

Moreover it is essential to note, that partitioning may affect the description of timing. For example partial time orderings that may differ strongly from the total orderings proper to a sequential description are introduced by partitioning. The abstraction level of timing, however, remains unchanged.

Structuring

Structuring a unit transforms the coordinates representing the modeling style from the points

$$(concurrent, timing\_spec, value\_spec)$$

to the points

$$(structure, timing\_spec, value\_spec),$$

where *timing_spec* and *value_spec* remain stable during the transformation step. Moreover, structuring is represented by creating new and independent design cubes with the abstraction level of the encapsulated units.

The independent new design cubes resulting from structuring represent both, the possible mapping of a sub-unit onto a unit, which was already or is currently designed, and the in-dependency of the subunits (which was achieved by structuring and which allows for independent as well as concurrent design of the subunits).

4.4 Value Transformation

Value transformation is either value coding or value flattening. Coding maps abstract values on a vector of bits. In most cases the ordinal number or the `TYPE'POS` representation of the value is 2's-complement coded.

Two goals are important for coding:

1. To allow for easier description of the model.
2. Optimization of the implementation of a circuit in area and time. This can be done with relation to gate level implementation, only.

The effect of coding for the quality of the design result is indisputable. Today, however exist approaches for state as well as input and output coding of finite state machines only. No approach exists, which attacks the coding problem for design architectures in general.

Value flattening divides the vector of bits in single bits. This simple task is performed by both, layout tools and synthesis tools.

The coordinates of the descriptions change during value coding and flattening from

$$(view\_spec, time\_spec, abstract\_values)$$

over the point

$$(view\_spec, rime\_spec, composite\_bit\_values)$$

to the point

$$(sequential, clock\_related, bit\_values).$$

4.5 Timing Transformation

Synthesis is defined as the transformation of a description of one design level to a description on the next lower design level. Due to the the design cube the definition of design levels can be based on time abstraction. Thus synthesis would only be a transformation of the coordinates along the time axis from the points

$$(view\_spec, time\_causality, value\_spec)$$

to the points

$$(view\_spec, clock\_related, value\_spec),$$

or a transformation of the coordinates from the points

$$(view\_spec, clock\_related, value\_spec),$$

to the points

$$(view\_spec, propagation\_delay, value\_spec)$$

where *view_spec* and *value_spec* keep their values during one transformation. The first and second transformation relates, but meets not exactly, tasks of system level and RT-level synthesis, respectively.

However, to be more general it is better to define synthesis as a transformation of descriptions, where at least one coordinate is changed.

5 A Systematic Approach to the Verification of Design Steps

This section finally presents a systematic approach for *vertical functional* and *requirement* verification for each class of design steps as defined in section 4. Timing and coding constraints are considered only.

5.1 Verification of View Transformation

View transformations serve primarily for the task of parallelization, which is necessary due to the high concurrent nature of hardware. They do not change coding and timing[7] of the description. Thus, requirement verification must not be performed for this design step.

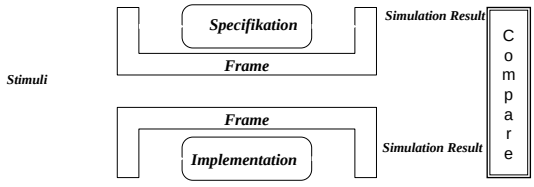

Figure 5: Detailed Verification of Equivalence

Functional verification can be done by applying stimuli to both, functional specification and implementation and by comparing the output values. This

[7] It is important to note, that view transformation often goes in hand with timing optimization. In this case, verification strategies for both design steps must be combined.

stimuli must not have identical signal value behavior. To allow for verification, the method shown in figure 3 must be refined as depicted in figure 5. A timing dependent frame takes the stimuli and applies them separately to each unit. Similarly, the frame takes results and generates simulation results for comparison. Tasks of the frame can be described for view transformation verification as follows:

- If causality is specified only, subroutines, which preserve the abstract, causality preserving protocols, are used in the frame to apply stimuli and to take the result. Signal values may differ especially in this case due to the fact, that causality specifies order of operations (at the interface) only.
- Pure clock related descriptions are determined by clock related storage devices and delay less[8] combinational operations. This implies, that input stimuli are equivalent but the time of output values may differ in delta cycles. The best way focusing simulation results for verification is by synchronizing with the clock. Postponed processes, which are included in VHDL'93, can be used for comparison of not synchronized simulation results.
- Both, stimuli and simulation result must be equivalent, if propagation delay is specified.

5.2 Verification of Value Transformation

Coding and value flattening do not change timing or view. Thus, stimuli, constraint checks as well as simulation results can be applied or tested at the same simulation time. The representation of the values, however, differ.

Coding transforms abstract values into a composite bit representations. The mapping can be specified as algorithm or table which can be implemented in VHDL by using array constants or functions. Functions can be directly used as type conversion functions in the port map to perform required transformations of the frame.

Value flattening increases the number of objects. The specification of flattening can also be done by using subroutines, but procedures are required in this case because a set of objects must be returned. A separate frame unit must be described additionally in VHDL because VHDL does not allow to combine several port objects by one type conversion function.

Functional and constraint verification can be performed in one run by comparing simulation results of specification and implementation using equivalence verification. Requirement verification can accordingly be performed by replacing the abstract values with composite bit values or composite bit values with a set of single bit values in an equivalence verification run.

5.3 Verification of Time Transformation

Verification of time transformation can not be achieved by one verification principle only. Functional verification, one verification task, is performed by equivalence verification. For this, stimuli must be related to the more abstract timing.

- Stimuli are applied in relation to a clock, if timing changes from clock related to propagation delay. In addition, simulation results must be synchronized to allow for comparison of simulation results.
- Stimuli must be processed by both, an abstract protocol, which communicates with the causal description, and a concrete, clock related protocol, which communicates with the clock related description. Results are processed in the same way. Simulation results can be compared, each time one protocol operation was finished successfully for both descriptions.

Requirement verification checks, whether either propagation delay constraints or clock cycle constraints have been violated.

- Propagation delay constraints are mostly specified in relation to clock cycles ie. setup and hold time violation. Verification checks the time difference between the last change of a value before the clock edge and the first change after clock. Useful VHDL constructs to do so are all signal related attributes and the function now. Examples can be found eg. in [4] or in [8].
- No technique, however, is known to verify whether clock cycle constraints are met. We propose to use a similar approach as applied for the verification of propagation delay requirements. Timing constraints are however not bound to a clock edge but to the successful execution of a protocol. So, cycle constraints are checked between different protocols. An example, which flexibly checks clock cycle constraints between two handshake protocols is shown in listing 1.

```
use SynchronousLib.HandShake.all;
use UtilityLib.IntegerBuffer.all;

procedure CheckCycle(
   signal clk                    : in bit;
   signal ack_in, valid_in       : in bit;
   signal ack_out, valid_out     : in bit;
   constant pipeline_stage       : in integer;
   constant min_cycle, max_cycle : in integer ) is
-- types and objects
   variable cycle, old_cycle : integer;
   variable cycle_buffer     :
            integer_vector( pipeline_stage downto 0 );
   variable cycle_index: integer;
begin
   loop
      wait until clk = '1';
      cycle := cycle + 1;
      if ProtocolReady(ack_in,valid_in) then
         PutBuffer(cycle,cycle_buffer,cycle_index);
      end if;
      if ProtocolReady(ack_out,valid_out) then
         old_cycle := GetBuffer(cycle_buffer,cycle_index);
         assert ( (cycle-old_cycle) >  min_cycle ) and
                ( (cycle-old_cycle) <= max_cycle )
```

[8] Delay less means in this case delta or zero delay.

```
        report "cycle constraint violation detected"
        severity error;
      end if;
    end loop;
end CheckCycle;
```

Listing 1: Clock Cycle Verification

5.4 Verification of Optimization

The description style of the specification and result model remains unchanged through optimization as mentioned in subsection 4.2.

This suggests, that equivalent stimuli, constraint checks and comparators can be used. This assumption, however, is only partially true.

5.4.1 Verification of View Optimization

View optimization changes behavior similarly as view transformations. So, the same verification strategy as described in 5.1 can be used.

5.4.2 Verification of Value Optimization

Value optimization describes the mapping of one value implementation into another value implementation however without changing abstraction of values.

For verification of this kind of optimization, the same method can be applied as described in 5.2 for value transformation but with one difference: The functions represent mapping of values of the same abstraction level. Thus, the functions parameters are abstract or bit composite values and the return values are abstract or bit level values also. Similarly, the procedures responsible for the mapping of bit values have a set of bit type inputs and outputs.

5.4.3 Verification of Timing Optimization

Verifying timing optimizations must be split into two tasks as validating timing transformations. Optimizing causal descriptions requires equivalence verification only, due to the fact that these descriptions contain neither clock cycle nor propagation delay.

Same techniques as described in 5.2 can be applied for requirement verification. Functional equivalence must consider, that timing may be changed by optimization.

- For causal descriptions, subroutines, which hide abstract protocols are required for application of stimuli and taking stimuli to the specification as well as to the implementation.
- Synchronous protocols, which orientate according to description causality, are used in the frame to process stimuli and result, if clock related descriptions are optimized.
- Similarly, stimuli are applied and result is taken clock related, if propagation delay is optimized.

We would like to point out, that the next higher time abstraction is used to guide the functional verification of timing optimization.

6 Summary and Outlook

This paper presented a classification for design steps and their verification. The relation to hardware description languages, especially VHDL, restricted the methods to design steps changing view, value and timing of a functional specification. Considered constraints were also focused on view, value and timing for the same reason.

Extensions of the technique shall also consider in hardware description languages not directly included constraints like area, power or testability. The first step in this direction is the hardware-description-language-based specification of this design constraints. The second step is the extension of verification steps in this direction. VHDL constructs, which seem to be from interest for this topic, are physical type declaration and global variables.

Acknowledgments

I would like to thank Michael Hofmeister, Sabine Rössel and all reviewers for their support, their suggestions and for helpful discussions.

References

[1] L.M. Augustin, B.A. Gennart, Y. Huh, D.C. Luckham, and A.G. Stanculescu, *Verification of VHDL Designs using VAL*, Proceedings of the 25th Design Automation Conference (DAC), Las Vegas 1991, pp. 48-53.

[2] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[3] D.D. Borrione, L.V. Pierre, A.M. Salem, *Formal Verification of VHDL Descriptions in the Prevail Environment*, IEEE Design and Test of Computer, June 1992, pp 42-56.

[4] D. R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

[5] W. Ecker and M. Hofmeister, *The Design Cube - A new Model for VHDL Designflow Representation*, Proceedings of the EURODAC/EUROVHDL'92, Hamburg 1992, pp. 752-757.

[6] *IEEE Standard Logic Package*, IEEE Std 1164-3992.

[7] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-3987.

[8] *IEEE Timing Working Group*, IEEE Par 1076.4.

[9] F. Rammig, *System Level Design* in J. Mermet ed., *Fundamentals and Standards in Hardware Description Languages*, Kluwer Academic Publishers, Dornbrecht 1993, pp 109-351.

[10] F.J. Rammig, *A Multilevel Cybernetic Model of the Design Process*, Proceedings of the IFIP WG 10.1 Working Conference on Methodologies for Computer System Design, 1985.

[11] Osamu Karatsu, *VLSI Design Language Standardization Effort in Japan*, Proceedings of the 26th Design Automation Conference (DAC) 1985, pp. 50-55.

[12] Hiroto Yasuuma, Nagisa Ishiura, *Semantics of a Hardware Design Language for Japanese Standardization*, Proceedings of the 26th Design Automation Conference (DAC) 1985, pp. 836-839.

[13] A. Schmitt, *Entwurf der Hardware-Beschreibungssprache REGLAN im Rahmen von CONLAN: Spezifikation der dynamischen Semantik und deren Implementierung*, Dissertation, Technische Hochschule Darmstadt, Institut für Datentechnik, 1992.