# Functional Validation of Hardware Components

Rabéa Ameur-Boulifa, Sophie Coudert, Renaud Pacalet
System-on-Chip laboratory (LabSoC), GET/Telecom-Paris
2229, routes des Crêtes
BP 193
F-06904 Sophia-antipolis Cedex
Email:first.last@telecom-paristech.fr

*Abstract*— **Functional validation of digital hardware components is an important problem. Actually detecting bugs early in the design cycle is crucial for both economic and methodological reasons. Even though formal methods have emerged as a successful approach to ensure the correctness of hardware their use is still quite limited due to scalability problems. This paper discusses a partial verification technique based on computation of data dependency.**

## I. Introduction

Nowadays data abstraction technique is widely used to manage the complexity of verification and improve the comprehensiveness of simulation. Abstraction of complex calculations in data-path of hardware component should greatly simplify the validation process. Moreover it can provide the means to find suitable components satisfying given requirements and so improving reuse of off-the-shelf components.

A digital hardware module has a set of input and output signals. It receives data values via input signals and gives out results via output signals. Some functional properties of modules do not require data values to be verified, since they deal with the intentional presence and absence of data at a specified instant. A data is said intentional or on purpose when its value is considered relevant to the computation. It is then called *significant*. Otherwise, it is ignored and called *non-significant*

Based on this abstract representation of data, we aim at proposing a verification framework which would enable an engineer to verify that the functionalities of a given module correspond to its specifications (without considering actual concrete value of data). In particular, we are interested in verifying properties of the hardware modules which state that the desired outputs arrive well at the time when they are expected to arrive. By means of an approximation based on significance the verification of such properties comes down to showing that, *an output is significant iff all inputs on which it depends were significant*. The dependency relationship between significance of an output and fraction of inputs corresponds to the data dependency semantics. This notion will be discussed in section IV. In this paper, we propose a framework which allows us to demonstrate functional properties for a hardware module of large size. This framework integrates automatic computation and injection of significance and it provides a convenient way for specifying properties by using observers as in [3] : the assumptions adopted to produce inputs; and the oracle which compares the outputs with expected ones.

## II. Functional verification

Functional verification of hardware module by formal method attempts to prove mathematically that certain requirements are met, or that certain undesired behaviors cannot occur. The requirement is a property which expresses a condition on the hardware module that should always be true in reachable states. For example we wish to show that : *whatever the values of input data of a given module, the corresponding output data value must be correct.* Correct means that, "the data-path does what is intended". This property may be specified formally by the formula :
$G((input = v \wedge validin) \Rightarrow X^n(output = f(v) \wedge validout))$
where $G$ and $X$ represents the temporal operators (always and next), $input$ (and $output$) the input (and output) vector, $v$ an input value, $validin$ (and $validout$) the control signal indicating that the input (and output) data is valid, and $f$ represents the function of data-path that the module performs. The exponent $n$ represents the time between the input sampling date and the availability of the corresponding output date. For example, in a module, a particular output will be available after 8 clock cycles after sampling the inputs.

Verifying whether the function $f$ performs really as intended could require a lot of effort. In fact, exhaustive verifying and developing tests for all possible values of input signals can be prohibitively expensive not to say impossible in some cases. Thus, abstracting away concrete value of data can be necessary for such verification. By using the significance concept, it is possible to produce an abstract version of the previous property in such a way that it can be specified as :
$G((input = sig \wedge validin) \Rightarrow X^n(output = sig \wedge validout))$
where $sig$ denotes the significant value and, $input = sig$ and $output = sig$ indicate that the input and the output are assigned with *significant* value (without giving there concrete values). Note that the function $f$ is no longer mentioned in the property. It is reduced to a data dependency computation, and translated into a block of boolean equations.

## III. Verification Framework

The proposed framework is composed of 3 components: the module given for verification **Module Under Validation (MUV)**, and two observers : **Assume** and **Property**. Each component is provided in the form of a VHDL entity/architecture with distinct control and data inputs. Actually, we apply the significance issue only on the data-path. So have to identify distinctly data inputs from control inputs. All components are composed in parallel, in a top module called "validation module" (see Fig. 1) also provided in VHDL form.

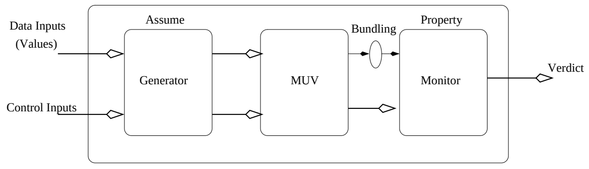

Fig. 1. Validation Framework

*a) Module Under Validation (MUV):* is a hardware module given in VHDL form. For illustration purpose, let us suppose that the module under validation is a multiplication module SPM which multiplies two numbers provided simultaneously at the data inputs A and B and produces the result on its output S, 8 cycles later (Fig. 2).

The above property applied to this module will give :

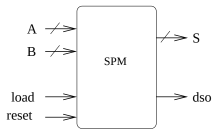

Fig. 2. SPM Module

$G((A = sig \wedge B = sig \wedge load) \Rightarrow X^8(S = sig \wedge dso)\ U\ reset)$

In the formula $U$ is a temporal operator (until). It specifies that it is always the case that a significant input of A and B will result in a significant output S unless the RESET signal is activated. LOAD and DSO are control signals that indicate when the input is loaded and the output is valid respectively.

*b) Assume:* is the input generator which feeds the SPM module with a sequence of inputs satisfying the requirements in this case by a flow of significant values on inputs A and B.

The following VHDL process is an example of an input generator that produces significant inputs according to the value of control signal LOAD. An implementation that ensures that, when LOAD is

```
process(load)
begin
      A <= (others => to_sig(load,'0'));
      B <= (others => to_sig(load,'0'));
end process;
```

active, only significant inputs are loaded to module ; the function *to_sig(load,'0')* converts bit vectors A and B into significant value when LOAD is high, into non-significant value otherwise. Note that this function has two parameters the second one corresponding to the input value. Since the value inputs are ignored we assign them in this case inconsequentially to the value 0.

Once the computation is completed. Since the property is only concerned about the significance of the output S this former is bundled to a single bit S_SIG representing either significant or non-significant result. With the aim of bundling vectors we apply the rule that states: "if any bit in the bit vector is non-significant the whole bit vector is non-significant". This operation contributes to the minimization of the module.

*c) Property:* is the monitor, the component which takes as input all the outputs coming from the module under verification and computes a single boolean output which is true as long as the observed satisfy the property.

For instance, the following process encodes the observer that checks at each clock cycle whether the output that

```
process(clk)
begin
  if (clk'event and clk = '1') then
    if ((dso = '1' and S_sig = '0') and reset = '0') then
          d_error <= '1';
    else
          d_error <= '0';
    end if;
  end if;
end process;
```

observed conforms to expected one or not, i.e, "if the output is significant when DSO is active while RESET is non-active".

## IV. Data dependency

This work is based on a formal characterization of data dependency among data values. By dependence of an output on an input, we mean that the value on this input has an impact of the value of the output. In other words, this input is determinant of the output (notion similar to prime implicants of the Boolean functions [4]). The dependency is formulated by the property on significance by : *an output is significant iff it is computed from significant inputs*. Among different dependencies we are interested in *static data dependency* and *dynamic data dependency*. – The static semantic is conservative, the significance of any input is propagated towards the output no matter it is being used under the corresponding context or not. For example, the output of an AND logical function will be non-significant if at least one input is non-significant even if the value of the other is 0. – Conversely the dynamic semantic is contextual, the impact of an input value on the result can depend on the values of other inputs. The impact of an input on the result can depend on both the significance and the value of other inputs.

Our implementation supports the different semantics and provides an easy way to set up our verification framework according to the constraints that we use for computation.

## V. Conclusion

We build a framework for automating functional verification of hardware modules. It provides the user with programmatic way of the property setting. We have used Candence compiler [2] for logic synthesis and VIS model checker [1] for our experimentation.

## References

[1] The VIS Home Page : http://vlsi.colorado.edu/ vis/.
[2] Cadence Design Systems Inc., 2008. Encounter RTL Compiler Synthesis.
[3] Gordon Pace, Nicolas Halbwachs, and Pascal Raymond. Counter-example generation in symbolic abstract model-checking. *Int. J. Softw. Tools Technol. Transf.*, 5(2):158–164, 2004.
[4] W. Quine. The problem of simplifying the truth functions. *Amer.Math.Monthly*, 59:521 – 531, 1952.