Semi-formal specifications and formal verification improving the digital design: some statistics

D.Torres*[1], J.Cortéz[2], R.E. González[1]

[1]Research Center and Advanced Studies of IPN,
Av. Científica 1145, C.P. 44019, Zapopan, Jalisco, México
*dtorres@gdl.cinvestav.mx
[2]ITSON, Antonio Caso S/N, C.P. 85130, Cd. Obregón, Sonora, México

ABSTRACT
In this work, an improvement of the traditional digital design methodology is proposed. The major change is the use of a semi-formal specification for the code implementation, the use of a verification tool and the establishment of properties for the formal verification of Finite State Machines (FSM). From semi-formal specifications, assertions were written using Property Specification Language (PSL) for an alignment circuit. Finally, a set of properties for the verification of this module were established and proved using a model checking tool. Our statistics proved that the whole design process was improved and considerable design time was saved.

RESUMEN
En el presente trabajo se propone una mejora a la metodología del ciclo de diseño digital tradicional. La contribución principal es la generación de un conjunto de propiedades a partir de una *especificación semi-formal* de requerimientos, que permiten la verificación formal automática de una *máquina de estados finitos* (FSM). Estas propiedades se escriben en el lenguaje PSL. Se muestra cómo, a partir de las propiedades, se puede obtener código VHDL que implementa la máquina de estados. Nuestros resultados muestran que la metodología de diseño propuesta resulta en una disminución del tiempo requerido para realizar la verificación.

Keywords: Formal verification, assertion based verification, finite state machines, semi-formal specification, model checking tool.

1. Introduction

Ensuring functional correctness on RTL designs continues being one of the greatest challenges for ASIC's design teams. With ever increasing design sizes, verification becomes the bottleneck in modern design flows. Up to 80% of the overall costs are due to the verification task [1].

Generally, the verification engineers use the simulation to demonstrate that the design's implementation satisfies its specification using a *black-box* testing approach. They create a model of the design written in a Hardware Description Language (HDL) like Verilog or VHSIC Hardware Description Language (VHDL). After they create a testbench, which includes the model for *Device Under Verification* (DUV), then input patterns, so-called simulation stimuli, are created to represent typical or critical execution traces and are applied to the DUV.

The functional verification process will never guarantee that a design is error-free [2], i.e., it is, until now, a hard time consuming task, which can only demonstrate the presence of errors but not their absence. One way to improve significantly the performance and saving time of this traditional verification is to increase the *observability* and the

[1] Corresponding Autor

controllability of the design during the verification process. We can do this using *assertions and formal verification* techniques in a process called *Assertion Based Verification (ABV)*.

A problem with this technique is that the designer and verification engineers are reluctant to adopt these design and verification methodologies without a clear proof that these methodologies improve both quality and efficiency of the whole design process. Up to this point, it should be added that *the majority of engineers do not have deep knowledge about Logic.* This fact does not allow them to define a set of properties, from the requirements, describing the intended behavior of the design [3][4].

Some design engineers create a gap between the specification of requirements and their implementations; this has a remarkable impact on verification efforts. For example, let us consider a requirement: "The design must perform the initialization of the circuit". In this case, there is too much missing information: What signals are involved and initialized? How is the reset asserted (active high or active low, positive, or negative edge)? These ambiguities, in these situations, can lead to misinterpretations.

In [5] Winkelmann presents the advantages of using a tool as property checker for verifying the block-level functional correctness of a large ASIC; and, he says: "the two biggest advantages are

- Coding and Verification can be done in parallel.
- The whole state space of a test case will be verified in a single run."

But, he does not show a detailed methodology for this purpose. For this reason, we propose the use of a new form to express the set of requirements of the design, which we will call *semi-formal specification* and whose objectives are

- To fill the gap existing between the specification of requirements of the design and the implementation in HDL realized by a design engineer.
- To avoid ambiguity and misinterpretations of the requirements specifications of the design when the implementation and the verification process is carried out, i.e. semi-formal specification is a common "well-defined" point for the design and verification engineers.
- To serve as help to obtain the set of properties necessary to carry out the formal verification of the design, without the need that the verification engineers have deep knowledge of techniques associated with the formal hardware verification.

Definition 1. Let ψ_{IMP} *and* ρ_{SPEC} be the representation of the *implemented code* in some language and the representation of the set of *the requirement specifications,* respectively, of a design. Then, with the verification process, we will prove that the following general logical relationship is true:

$$\psi_{IMP} \rightarrow \rho_{SPEC} \qquad (1)$$

A formal verification tool should be able, *employing logical properties*, to prove mathematically an equation equivalent to this one.

The main objectives of this paper are

- To introduce the use of the semi-formal specifications inside the design cycle
- To establish a set of properties written as assertions using PSL, which allows the development of a formal specification for FSMs
- To use a design cycle, which involves the use of a formal verification tool (*Safelogic Verifier*) for FSM validation
- To apply the methodology ABV for the verification of FSM showing major **statistics.**

2. Theoretical background

2.1 Formal Verification

An *implementation,* $\psi_{IMP,}$ consists of a description of the actual hardware design that is to be verified. A *specification,* ρ_{SPEC}, is a description of the intended/required (properties) behavior of a hardware design*. Formal Verification (FV)* is a systematic process of ensuring, through exhaustive algorithmic techniques, that a design implementation ψ_{IMP} satisfies the properties established in its specification ρ_{SPEC}. With formal verification, we are able to overcome both of simulation's challenges: the observability and the controllability, since formal verification algorithmically and exhaustively explores all possible input values over time. The simulation is empirical - meaning you use trial and error to discover bugs. It would take an intractable amount of time to test all possible combinations. Therefore, it is never complete. However the verification engineers are focusing their effort on *how* to break the design, not on *what* the design is intended to do. Formal verification, on the other hand, is mathematical and exhaustive and allows the engineer to focus solely on intent, or "*what* is the design's correct behavior". The main disadvantage of the formal verification is its limited capacity when the explosion state [6] is present.

There are two types of formal verification:

- *EQUIVALENCE CHECKING*: It is concerned with the verification that two representations of the same design are equivalent. Typically, it is gate level versus another gate level or RTL
- *MODEL CHECKING*: Specification is in the form of a temporal logic formula, the truth of which is determined with respect to a semantic model provided by an implementation

It does not matter what form of proof method is employed, several criteria need to be evaluated in order to make meaningful comparisons between various approaches, and these are

- Nature of the relationship explored between the *requirements specification* of the design and their *implementation*

$$\psi_{IMP} \cong \rho_{SPEC}$$
$$\psi_{IMP} \Rightarrow \rho_{SPEC}$$

- Soundness and completeness of the proof method
- Degree of automation
- Computational complexity of the proof method

gister

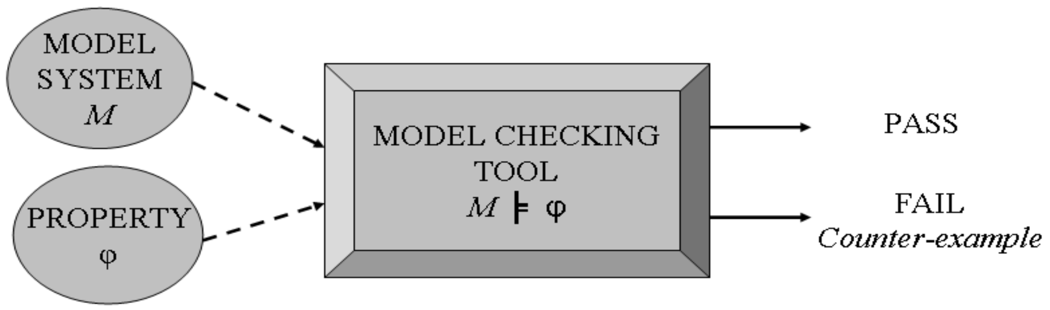

Fig 1. Formal Verification

In Figure 1, it is shown the formal verification process. If the model satisfies the behavior, a message of *PASS* is given by the tool. If the proof fails, a *counter-example* is presented.

2.2 Definition of Assertion-Based Verification (ABV)

The purpose of the ABV [7] is to convert functional features of a specification into explicit logical properties. The intended behavior of a design can be defined as a set of logical and timing relationships called *properties* which describes *logical behaviors of the system over the time* while an *assertion can be considered as an implementation of a property*.

Assertions can be used as inputs to both simulation and static verification tools. In the market, there are some languages and tools that allow the implementation of assertions, among them SystemVerilog [8], but only a few of them are accepted as standards.

The current available static verification solutions are capable of verifying functional equivalence of different implementations (e.g. Register Transfer Level and Gate Level Descriptions). However, these tools have not been designed to verify an implementation against their specification. Each assertion verifies a certain functional feature of the design. Assertions can be embedded into HDL source code or they can be located in a separate property file. In both cases, they are compiled together with HDL code. Therefore, the target is to improve both verification quality and efficiency.

Assertions can be used in methodologies based on simulation, as hardware monitors to complement the existing verification environment. In static methods, assertions provide also an alternative verification path to verify all computational paths of the design.

2.3 Logical Temporal Linear (LTL)

The LTL is an extension of classical propositional logic devised to formulate statements about events that change over the time. In the LTL semantics the time is discrete and it is seen as an infinite succession of a linearly ordered set of states, usually represented by the set of the natural numbers N . This line has an initial state

(state zero) and it extends infinitely towards the future. Typical modalities of this logic are

"***Sometimes*** P": this proposition is true if P is completed in some future moment (or now)

"***Always*** P": this proposition is true if P is completed in all the future moments (or now)

The collection of temporal formulas Φ that are constructed on a set of boolean variables $\Omega = p_0, p_1, \ldots, p_N$ is defined inductively in the following way:

- Every $p \in \Omega$ is a temporal formula.
- If α is a temporal formula, then so are $\mathrm{O}\alpha$ (in following state α), and $\neg\alpha$ (not α).
- If α and β are temporal formulas, then so are $\alpha\ \mathcal{U}\ \beta$ (α until β), and $\alpha \vee \beta$ (α or β).
- Other operators can be defined in terms of the above operators:

eventually α : (2)

$$\Diamond\alpha = \mathbf{T}\ \mathcal{U}\ \alpha$$

always α **:** (3)

$$\Box\alpha = \neg\Diamond\neg\alpha$$

However, the properties to describe logical behaviors of the hardware's design can be specified using LTL or PSL. All the operators of the propositional logic have been defined for both languages.

2.4 Property Specification Language (PSL)

PSL, developed by Accellera [9], is a language for the formal specification of hardware. A detailed description of PSL can be found in [9]. It is used to describe properties that are required to hold on the design under verification and it can be used to capture requirements regarding the overall behavior of design as well as assumptions about the environment in which the design is expected to operate. A PSL specification consists of a set of *assertions*. PSL provides a standard means for hardware designers and verification engineers to document the design specification rigorously. It is easy to learn, write, read and it has a concise syntax and rigorously well-defined formal semantics. Both VHDL and Verilog languages are provided.

PSL has been divided into *four different layers*: *Boolean, temporal, verification*, and *modeling layers*. The *boolean Layer* is where *PSL* specifies the conditions that define a behavior of interest, referring to signal, variables, and values within an HDL description of a design. *Boolean expressions* are evaluated in a single evaluation cycle. The *temporal layer* is used to describe the behavior of the design over time and represents the heart of the language. It can also describe properties involving complex temporal relation between signals. The v*erification Layer p*rovides directives to the verification tool. It is used to indicate the verification tools what to do with the properties described by the others layers.

PSL can express the following kind of behaviors:

- A behavior may always or never hold true
- A behavior may always or never hold true only within some cycles
- A behavior may express some specific sequences of events
- A behavior may express an eventuality

- A behavior may <u>have exceptions to the rule</u>

The next classification of property specifications was given in [6]:

- *Safety*: It denotes that certain condition crucial for a proper functioning must not be violated at any time instance: *Something bad will never happen*
- *Liveness*: It comprises properties where a desired or necessary system condition will be reached: *Something good will eventually happen*
- *Fairness*: It is used if certain properties must hold again and again: *Something good will happen infinitely often*

2.5 Structure of properties using PSL

PSL allows the writing of design properties. A *PSL property* defines a behavior that can be checked (asserted) or assumed as a constraint. A property can be thought as a sentence with a containing *enabling* condition, a *fulfilling* condition (which is the behaviour to be checked), an optional *discharging* condition, and an optional *clocking* condition.

The most general form of a property is shown below:

<u>property</u> <name> <u>is</u>
[occurrence operator] [enabling_condition(s)]
[implication_operator(s)] (fulfilling conditions)
[discharging_condition] [@(clock_expression)];

The key terms of a property expression are:

- **<u>property : It</u>** is a keyword specifying that what follows is a behavior
- A **<u>property <name></u>** is a unique identifier. HDL naming conventions apply
- The **<u>ocurrence operator</u>** is one of ***<u>always or never and eventually!</u>*** and indicates whether the behavior should always occur, should never occur or should eventually occur; omitting the occurrence operator causes the check to occur at time zero only
- The **<u>enabling condition</u>** can be any Boolean expression or sequence expression
- The **<u>implication operator</u>** specifies the logical relationship between the **<u>enabling condition</u>** and the **<u>fulfilling condition</u>.** The operator is one of the following symbols:
  - $\Rightarrow$ **Logical IF implication:** The behavior can be read as **<u>“if</u> <u>(enabling condition) right hand side(RHS) is true, then the (fulfilling condition) left hand side(LHS) must also be true</u>**
  - $\Rightarrow$ **next[n]:** The behavior can be read as **<u>“if</u> <u>(enabling condition) right hand side(RHS) is true, then the (fulfilling condition) left hand side(LHS) must be true in the next nth verification cycle. The LHS must be Boolean</u>**
- The **<u>fulfilling condition</u>** is the behavior to be tested. The **<u>fulfilling condition</u>** can be any boolean expression or sequence
- The **<u>discharging condition</u>** is a condition that indicates to stop the checking of the behavior. Discharging conditions include the following:
- ***Until*** **(*Boolean expression*):** When all of the **<u>enabling conditions</u>** have evaluated to true, the **<u>fulfilling condition</u>** must be true until the *Boolean expression* becomes true.

abort* (*Boolean expression): The **abort** operator cancels the checking of an assertion

2.6 Safelogic Verifier

Safelogic Verifier is a model checking tool developed by Safelogic [10] and it has already been used for circuit verification. It performs an automatic formal verification, without user intervention, of hardware designs implemented with VHDL language, using PSL assertions.

3. The Design Cycle

3.1 Design cycle

Figure 2 shows our proposal of the whole design cycle, where the highlighting points are the inclusions of the semi-formal specification and the use of a formal verification tool. The semi-formal specification is a starting point for the designer, which uses it for writing the code in some HDL, as also for verification engineers, who write the properties using LTL or PSL for *Safelogic Verifier*. The most important aspects of the whole design cycle are described.

A. Stage of specification

Requirements Specification (or informal problem description). It defines a set of general characteristics or requirements $\rho = \{\rho_1, \rho_2, \ldots, \rho_n\}$ for the design of the FSM or a circuit. Each requirement must capture only a particular behavior of the FSM and each behavior will be expressed by one or more properties. PSL assertions can be used to implement these properties.

Architecture. State diagrams are constructed to define the behavior of an FSM, which must satisfy the requirements specification. State diagrams for FSMs are particularly useful to define PSL assertions, because they show their expected behaviors.

Semi-Formal Specification. Making a refinement of the original requirements ρ jointly with the architecture document, timing diagrams, signals description and entity, a *semi-formal specification* $\varphi = \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$ is obtained, where a *refinement* is defined as the process of transforming a requirement of the circuit or FSM into another one, equivalent and expressed in semi-formal form.

Definition 2. A *semi-formal specification* is a form to express a *requirement* using key words that

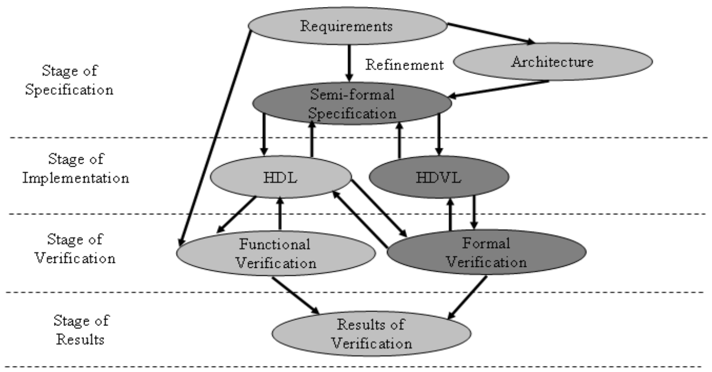

Fig 2. Design Cycle

represent the functionality of *logical, temporal operators* and *specific statements expressed as well-formed formulas* associated with a property or expected behavior of the design.

From the semi-formal specification it is possible:

- To obtain a set of properties $\xi = \{\xi_1, \xi_2, \ldots, \xi_p\}$
- To implement the code in HDL

This way can ensure that each requirement ρ_i, $\rho = \{\rho_1, \rho_2, \ldots, \rho_n\}$ is described by its semi-formal specification φ_i.

eer xe

B. Stage of implementation

VHDL Implementation & Formal specification**.** From the semi-formal specification, the VHDL implementation ψ_{VHDL} of the FSM and a set of PSL assertions $\xi = \{\xi_1, \xi_2, \ldots, \xi_p\}$ are obtained.

C. Stage of verification

Then, the FSM or circuit is verified using the model checking tool *Safelogic Verifier*. It allows to assure that an implementation in VHDL satisfies the properties of its specification, i.e. :

$$\psi_{VHDL} \rightarrow \xi \quad (4)$$

When an error is found within of ψ_{VHDL}, a counterexample is generated and used to correct ψ_{VHDL}. Also, to validate the design of the circuit, a functional verification is executed to ensure the global correctness of the implementation. The functional verification is necessary to detect errors in the *semi-formal specification* φ generated during the refinement of ρ.

D. Stage of results

In this stage, reports containing statistics of metrics about formal and functional verification are generated. Therefore, conclusions about the complete verification process over the properties defined are obtained.

3.2 Generation of PSL assertions and semi-formal specification from requirements specification for the design

An *assertion* is built from *Boolean Expressions* (which describe behavior over one cycle), *sequential expressions* (which describe multi-cycle behavior), and *temporal operators* (which describe relations over the time between Boolean expressions and sequences). Assertions are able to capture complex intended behaviors of the design in a formal way. We use PSL as an extension of the Linear Temporal Logics (LTL) to express the assertions. The following notation was used for the elaboration of the requirements and semi-formal specification of our designs.

R_SM:i => i-*th* requirement

SF_SM:I => i-*th* semi-formal specification

We used the following words as reserved words for the description of the semi-formal specifications; they represent the functionality of the logical and temporal operators needed to elaborate the specification:

Word	LTL operator	PSL operator
Always happens that	$\square$	**always**
Never happens that	$\neg\square$	**never**
If α **then** β	$\alpha -> \beta$	$\alpha \rightarrow \beta$
In the following clock edge α	$O\alpha$	**next**(α)
α **and** β	$\alpha \wedge \beta$	α **and** β

An example at low-level of how a PSL assertion can be extracted from a requirement is explained:

Reset requirement (requirement specification)

[R_SM:1] **The design must perform the initialization of the circuit**

Refinement

[R_SM:1.1] The module must have a synchronous reset mode to load FPG_FPind_out port and internal states with initial values.

[R_SM:1.2] Synchronous reset mode must be reached when the corresponding signal is active with ‘1’ logical

Semi-formal specification

[P_SM:1] ResetOp

Always **happens that**

If **an operation of reset is required**, then in the following clock edge **the port FPG_FPind_out will be loaded with the initial value (LOW)**

PSL assertion

```
property ResetOp is
always(FPG_Reset_in→next(FPG_FPind_out
= LOW); assert ResetOp;
```

3.3 Finite State Machine (FSM)

One of the major elements in the hardware designs is the FSM. Due to its practical importance, the problem of FSM verification has motivated research in this area to ensure its correct functioning and to discover aspects of its behavior [11]. The problem of FSM becomes more complex when the number of states and their transitions grows. Therefore, it is difficult to construct checking sequences to assure that the implementation of the FSM satisfies its specification. An FSM contains a finite number of states and produces outputs on state transitions after receiving data. FSMs are widely used to model systems in diverse areas. A general FSM is shown in Figure 3.

Definition 3. A Finite State Machine (FSM) is a sextuple:

$$M = (I, O, S, \delta, \lambda, S_0) \quad (5)$$

- *I, O, and S* are *finite and nonempty sets of input symbols, output symbols, and states*, respectively

$\delta : S \times I \rightarrow S$ *is the State Transition Function* (6)

$\lambda : S \times I \rightarrow O$ *is the Output Function* (7)

- When the machine is in the current state $s \in S$ and receives an input *a* from *I*, it goes to the

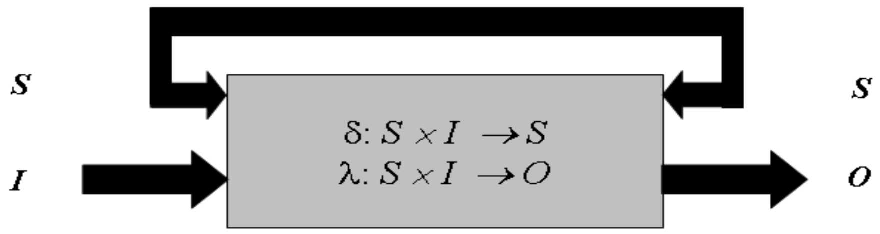

Fig 3. Finite State Machine

- next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$
- $S_0 \subseteq S$, is the set of initial states

3.3.1 Implementation in VHDL of FSM

The functionality of an FSM includes the following signals:

- Clock Input (FSM_Reset_in)
- Reset Input (FSM_Clk_in)
- Symbols Input (FSM_Data_in[n_I,0])
- Symbols Output (FSM_Data_out[n_O,0])
- Set of States (S)

Figure 4 shows the input and output signals of the entity for a general FSM.

The implementation in VHDL for an FSM can be done using two processes; while Figure 5 shows the diagram, in Table 1 a description in VHDL for this architecture was given.

```
architecture fsm_arch of fsm is
type TState is (FSMSt-0,FSMSt-1, … , FSMSt-n); -- States
declaration
signal FSM_State : TState;   -- State signal
begin
State_Process : process (FSM_clk_in & FSM_Data_in &
FSM_Data_out)
...
Store_Process : process ( clk, rst )
...
end fsm_arch;
```

Table 1
VHDL description of an FSM using two processes

The architecture in VHDL for an FSM that uses one process is shown in Figure 6 and a description in VHDL for this architecture is shown in Table 2.

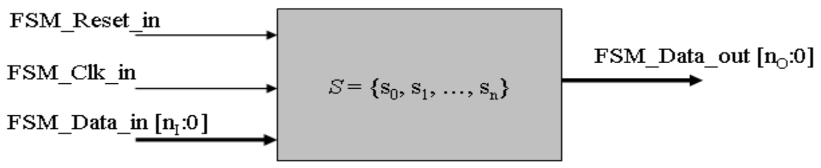

Fig 4. Finite State Machine

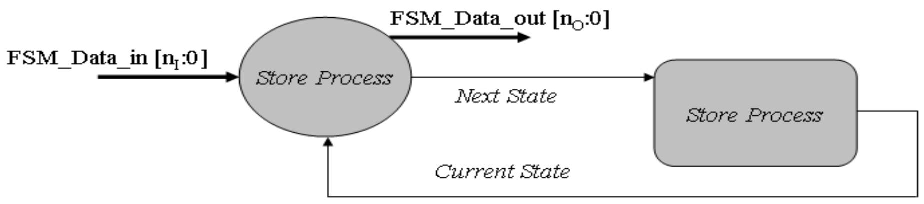

Fig 5. Architecture for an FSM using two processes

```
architecture fsm_arch of fsm is

   type TState is (FSMSt-0,FSMSt-1, … , FSMSt-n); --
States declaration
   signal FSM_State : TState;   -- State signal

begin

State_Process : process (FSM_clk_in & FSM_Data_in &
FSM_Data_out)
...

end fsm_arch;
```

Table 2
VHDL description of an FSM using one process

This implementation is difficult to simulate and to synthesize but a description can be done using natural language.

3.4 Specification of requirements for the FSM

A specification of the set of necessary requirements of the FSM is

1) *Reset Operation.* It must have a "Reset" input port to re-establish in a way [synchronous | asynchronous] that initializes its operation and to load default values for signals and variables used from an active signal in [high | low]
2) *Number of states for the FSM*. The number of states must be finite. In each state the behavior of the outputs must be clearly defined.
3) *Transition between states*. All conditions for the transitions between states must be defined.
4) *Loops*. It must have defined all conditions of loop in each one of the states of the FSM.

3.5 Properties for the formal verification of FSMs

To carry out the formal verification of the design of some FSMs, a set of properties• was given in [12]. To these ones, we added the properties for non-connectivity between states and output FSM.

1. Reset Operation: (**RstOpFSM**)
   It always **happens that** if **a reset operation is required,** then the **FSM takes its initial value**
   Property ***RstOpFSM*** is always **(reset_in $\rightarrow$ (FSM $\Leftarrow$ Initial value))**
2. States Reachability: (**StateReach**)
   Each one of the states [$s_i \in S$ $i = 1, 2, \ldots, n$] of the **FSM** will eventually be reached
   Property ***StateReach*** is eventually **(FSM_state $\Leftarrow$ [$s_i \in S$ $i = 1, 2, \ldots, n$])**
3. Connectivity of states: (**StateConnectivity**)
   It always happens that **if the** FSM reaches a state different to the current, **then the FSM** will **eventually** return to current state again
   Property *StateConnectivity* **is always** ((FSM_state /= s_k) $\rightarrow$ **eventually** (FSM_state = s_k)
4. Transitionº between states: (**TransStates**)
   It always happens that **if** the FSM is in state s_k **and** completed the conditions of transition to go to another state **then** in the **following clock edge** the FSM moves to that state s_j
   Property *TransStates* **is always** ((FSM_state = s_k and $\delta_{TRANSITION}$ (s_k *to* s_j)) $\rightarrow$ **next** (FSM_state = s_j));
5. **Permanency in a state: (LoopState)**
   It always happens that **if** the FSM always remains in state s_k **then** the transition conditions were **never satisfied**
   Property *LoopState* **is always** (**always**(FSM_state = s_k) $\rightarrow$ **never** ($\delta_{TRANSITION}$));
6. **Non-connectivity between states(Nonconnectivity)**

• Where symbol $\rightarrow$ is the logical operator for the implication
Symbol $\Leftarrow$ is the assignation operator

It always happens that **if** state s_k is reached, **then** state s_j is never reached
Property *Nonconnectivity* **is always** (FSM_state = s_k $\rightarrow$ **never** (FSM_state = s_j));

7. **Valid Output(ValOutput)**
   It always happens that **if** the FSM reaches every state [$s_i \in S$ $i = 1, 2, \ldots, n$], **then** the FSM_Data_out takes a value of the set $O_i \in O$ *from* $i = 1, 2, \ldots, n$

   Property *ValOutput* **is always** (**FSM_state** $\Leftarrow$ [$s_i \in S$ $i = 1, 2, \ldots, n$] $\rightarrow$ (FSM_Data_out $\Leftarrow$ [$O_i \in O$ *from* $i = 1, 2, \ldots, n$]));

This set of properties was proposed since it covers the principal functionalities for FSMs. The particular objectives for each property are described next:

Property **RstOpFSM** tests that the FSM begins in a specific state a clock cycle later than a reset operation is required.

Property **StateReach** tests that it exists at least an input sequence that allows reaching each one of the states defined in the FSM.

Property **StateConnectivity** tests that it does not exist any sink state in the FSM.

Properties **TransStates** and **LoopState** test that only the transitions and loop conditions defined in the specification of the FSM occur when their correspondent valid events in the input are present.

Property **Nonconnectivity** tests transitions non-specified in the requirements of the FSM that can be occurring in unexpected conditions.
Property **ValOutput** tests only valid outputs appearing in the response of the FSM, also defined in the specification of the FSM.

3.6 Relationship between requirements (ρ) and assertions (ξ)

There is a relationship between the set of general requirements ρ of the FSM and the set of assertions ξ obtained from the semi-formal specification of the FSM. This relationship can be expressed by the formula (8).

$$\rho_i \leftrightarrow \xi_1 \wedge \xi_2 \wedge, \ldots \wedge \xi_k \tag{8}$$

That means: to each requirement ρ_i a set of assertions $\xi_i \ (i \in 1, \cdots, k)$ is associated, where k is finite but not equal for all requirements. Therefore, a requirement ρ_i is verified using own corresponding set of assertions, an implementation in a HDL language, and a formal verification tool.

3.7 Properties classification

From the set of all properties defined to carry out the formal verification of FSMs, we identify a subset of them in order to write code in VHDL. This classification is shown in Table 3.

Classification	Subset of properties
Used only to formal verification	1. Connectivity of states: (StateConnectivity) 2. Non-connectivity between states(Nonconnectivity)
Used to write code in VHDL and to formal verification	1. States Reachability: (StateReach) 2. Reset Operation: (RstOpFSM) 3. Transition between states: (TransStates) 4. Permanency in a state: (LoopState)

Table 3 Properties Classification

The following scheme was proposed to write VHDL code, using this property classification, for VHDL architectures with one and two processes:

One process	Two processes
architecture *FSM_arch* ***of*** *arch* ***is*** *--* ***Values defined in States Reachability:*** ***type state is*** *(FSMSt-0,FSMSt-1, … , FSMSt-n);* ***-- State signal*** ***signal*** *FSM_state: state;* ***begin*** *State process:* ***process****(FSM_clk_in & FSM_Data_in & FSM_Data_out)* *begin* ***-- Reset Operation: (RstOpFSM)*** *if FSM_rstSignal = ACTIVE then* *-- FSM* $\Leftarrow$ *Initial value* *elsif clkSignal'event and clkSignal=ACTIVE then* *-- State Machine Scheme* *case FSM_state is* ***--*** *FSMSt-i Scheme* *when FSMSt-i => --* $0 \le i \le n$ ***-- Transition between states: (TransStates)*** *if "State Transition" then* *-- Next State Assignment* *elsif Other "State Transition" then* *...* ***-- Permanency in a state: (LoopState)*** *else* *-- State Loop Assignment* *end if;* *when others => null;* *end case;* *end if;* ***end process;*** ***end*** *FSM_arch;*	***architecture*** *fsm_arch* ***of*** *fsm* ***is*** *--* ***Values defined in States Reachability:*** ***type TState is*** *(FSMSt-0, FSMSt-1,…., FSMSt-n);* ***-- State signal*** ***signal*** *FSM_State : TState; -- State signals* ***begin*** *State_Process :* ***process*** *(FSM_clk_in & FSM_Data_in & FSM_Data_out)* ***-- Reset Operation: (RstOpFSM)*** *if FSM_Reset_in = ACTIVE then* *-- FSM* $\Leftarrow$ *Initial value* *elsif clkSignal'event and clkSignal=ACTIVE then* *-- State Machine Scheme* *case FSM_state is* ***--*** *FSMSt-i Scheme* *when FSMSt-i => --* $0 \le i \le n$ ***-- Transition between states: (TransStates)*** *if "State Transition" then* *-- Next State Assignment* *elsif Other "State Transition" then* *...* ***-- Permanency in a state: (LoopState)*** *else* *-- State Loop Assignment* *end if;* *when others => null;* *end case;* *end if;* ***end process;*** *Store_Process :* ***process*** *(clk, rst)* ***-- Reset Operation: (RstOpFSM)*** *if FSM_Reset_in = ACTIVE then* *-- FSM_State* $\Leftarrow$ *Initial value* *elsif clkSignal'event and clkSignal=ACTIVE then* *-- Current State Assignment* *end if;* ***end process;*** ***end*** *FSM_arch;*

4. Alignment process – A Study Case

In order to apply the above-mentioned concepts, an example should be given. In this paper, a circuit for the alignment process was selected because it has more than two states and characteristic temporal behaviors. We will begin by defining the concepts *framing pattern, header*, and *alignment* in our context. Other definitions, close to these ones, can be read in [13].

Definition 4. A *framing pattern* <A> is a small piece of information that allows the receiver to identify the beginning of the frame for data processing and, therefore, it provides the synchronization between the transmission and reception parts.

Definition 5. A *header* is a finite sequence of bits containing a pattern called framing *pattern* <A>.

Definition 6. The *alignment* is a process to identify the alignment pattern, or a part of it, contained in each one of the input frames in order to establish the synchronization between the receiver and the transmitter.

Definition 7. An *information flow* can be seen as a sequence of frames.

While circuits that build and process the frames are called *framers*, aligner circuits are or may be parts of the framers. The *framer* is the first module in which the input data stream is processed [14]. The function of this circuit is very important because delimits bytes and input frames, providing synchronization for the correct operation of the remaining modules such as processing as generation in SONET/SDH [15] devices.

The objectives of these circuits are

- To *align the input data stream* by means of the alignment pattern.
- To *detect the main defects* or possible failure when processing the input stream.

Problem 1. *An ALIGNER is a circuit that carries out the search and validation of the framing patterns in the input data stream; this function will be represented by* ψ_{ALIGN}.

An aligner can be seen as a circuit, see Figure 7, composed of two major blocks: the *processor*, where the data are processed; and the *controller*, where a state machine controls the operations performed by the processor. It can defined as: <*I*, ψ_{ALIGN} ,*O*>, where

1. $I = \{s_{indata}, s_{CLK}, s_{reset}, s_{CONFIG}\}$.
   Where s_{indata} is a variable of *n* bits that represents the input data stream, s_{CLK} is the clock system variable, s_{reset} is the reset variable, and s_{CONFIG} configures the component.
2. $O = \{s_{outdata}, s_{SYNC}, s_{OOF}, s_{LOA}, s_{NF}\}$. Where $s_{outdata}$ is the aligned data output, s_{SYNC} indicates the presence or absence of the frame synchronization or alignment state, s_{OOF} indicates the hunt or out of frame state, s_{LOA} indicates lost of alignment state, and s_{NF} indicates a new frame beginning.
3. $\psi_{ALIGN} = \{\varphi_{reset}, \varphi_{ALIGN}\}$. These functions are temporal logical functions that satisfy generally one specification, where:
   - φ_{reset} is invoked when the reset occurs. The state machine will be in the initial state and the registers will be loaded with default values.
   - φ_{ALIGN} is the frame alignment function and it carries out *the bytes delimitation (if necessary)*, the *framing pattern detection*, the *new frame signal generation*, and it indicates the following states, hunt, and lost of alignment.

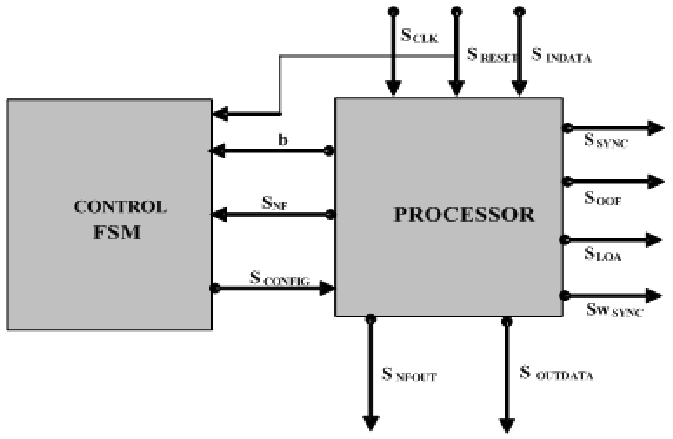

Fig 7. Principle of an Alignment

The processor performs function ψ_{ALIGN}, which is controlled by the state machine running in the block controller. When a *framing pattern* is found, signal ***b*** will be true. If *n* consecutive frames are taken from the information flow, see Figure 8, then the relationship given by Equation (9) is hold:

$$in = (b)_{r_1} \wedge \mathrm{O}^{r_1}(\neg b)_k \wedge \mathrm{O}^{r_1+k}(\neg b)_{r_2} \qquad (9)$$

where

in is the sequence of boolean values given by ***b***;

b is true when the framing pattern **<A>** is found;

O is the logical operator *next*, and in this context means *next frame*;

$(b)_n = b \wedge \mathrm{O}b \wedge \mathrm{O}^2 b \wedge \cdots \wedge \mathrm{O}^{n-1} b$,

where r_1, r_2 *and* k are nonnegative integers such that $r_1 + r_2 + k = n$.

Equation (9) permits us to explain the most diverse situations, for example:

1. *If* r_1 = 0, *then* the circuit evolves from the initial state to another without synchronism, or it stays in the initial state.
2. From the initial state, *if* r_1 > 0, *then* the circuit evolves to another with *some type of synchronism*.
3. From the initial state, *if* $r_1 > C_{SYNC}$, *then* the circuit evolves to another with a *strong synchronism*.
4. *If* $k > C_{LOA}$, *then* the circuit will go to the state *of loss of alignment*.
5. *If* $k > C_{HUNT}$, *then* the circuit will reach the state *of hunt*.

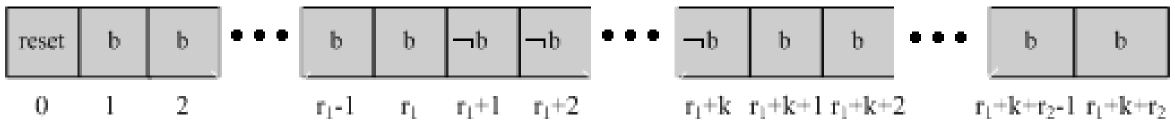

Fig 8. Flow information processed

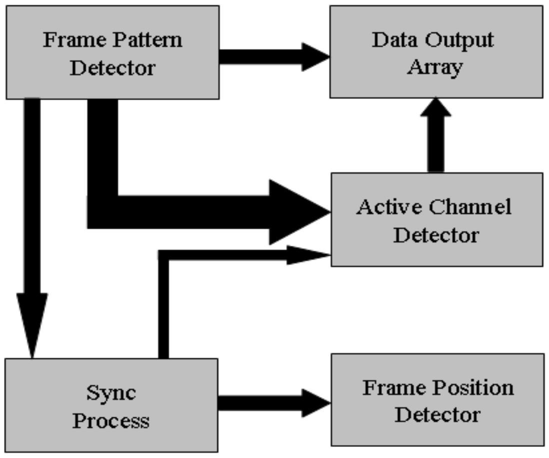

Fig 9. Aligner Architecture

In statements 3-5, thresholds C_i are positive integers, and $C_{HUNT} < C_{LOA}$.

4.1 Aligner Module for SONET/SDH- Sync Process

The aligner blocks for SONET/SDH [16], see Figure 9, are (A) Frame Pattern Detector, (B) Controller, (C) Data Output Array, (D) Active Channel Selector and (E) Frame Position Detector.

The major task, during the alignment process, is to find this pattern within each frame and to keep the data transmission at all times. This process is controlled by a Finite State Machine located in the Controller Block, called “Sync Process”.

The verification of the aligner described above must be done by validating each one of their blocks. The verification process of the components of the aligner can be carried out generating a set of PSL assertions for each component and applying formal verification using the model checking tool *Safelogic Verifier*.

A. Requirement Specification

The next seven requirements for the aligner circuit $\rho = \{\rho_0, \rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6\}$ are defined.

ρ_0 *Reset Operation.* A “Reset” input port allows the initialization of the circuit.

ρ_1 *Transition to synchrony state*. The module will be declared in “synchrony state”, activating a “synchronous system” signal after the “detected frames counter” reaches a specific value.

ρ_2 *Permanence in synchrony state.* If the module reaches the “synchrony state”, it must verify that the "frame pattern detect" signal appears each 125µs; otherwise, a “loss frames counter” must be incremented. If the "frame pattern detect" signal appears correctly, then the "loss frames counter" must be set to zero.

ρ_3 *Transition to out of frame state*. If the module reaches the “synchrony state” and the “loss frames counter” reaches a “four” value, the module will change to the “non-synchronous state”, deactivating the "synchronous system" signal.

ρ_4 *Out of frame state indication.* If four consecutive times the "frame pattern detect" signal does not appear each 125µs, the module will change to state “out of frame” and this state will be indicated with an active signal.

ρ_5 *Maximum time to detect an out of frame state.* The maximum detection time of an “out of frame state” must be 625µs, or 5 frames, for a random SONET/SDH input frame.

ρ_6 *Loss of the "out of frame" state*. The module will leave state "out of frame" (OOF) when signal "frame pattern detect" appears twice each 125 µs.

B. Architecture of Sync Process

Figure 10 shows the input and output signals of the aligner module. After a reset operation, the circuit starts in state "out of frame" indicated by "offOut" output. Whenever the alignment pattern is detected, some internal counters are incremented. When "syncCounter" reaches a specific value fixed by the input "syncConfig", the "synchrony state" is declared, and it is indicated by the "syncOut" and "syncAux" outputs. Once the synchrony state is declared, the module will return to state "out of frame" when four consecutive patterns of alignment are lost, which is controlled by "lossCounter". Also, whenever frame last byte is detected, it will be indicated by port "lastByte". See table 4 for the port description.

The *state diagram* to model the behavior of "*sync process*" is shown in Figure 10. In the diagram, state **OOF** represents the "non-synchronous state" and state **SYNC** the "synchrony state". Each one of the arrows represents the defined *transitions* and *loops* of the FSM; they are based on requirements (ρ_0,…, ρ_6) established in the specification.

For example, requirement (ρ_1), see Section 4.1.A, describes when the FSM reaches state **SYNC**, and the corresponding transition is **OOF$\rightarrow$ SYNC,** see Fig.10**.** A first refinement of ρ_1 involves input and outputs signals, i.e:

State* SYNC *is established when "syncCounter" reaches a value fixed by input "syncConfig", input "framePulse"(not activated) indicates the frame beginning, and in the following clock edge a "syncOut" output signal indicates the new state.

In this requirement, all signals are specified completely, but the values of some signals are not clear enough. Then, these values must be specified, and the requirement can be written in a semi-formal way, i.e.:

-- [P_SM: 6] OOFTrSync
It **always** happens that
if the FSM is in state **OOF and** at least one of the next conditions is completed

- ***φ_1: syncCounter = '1'* and *syncConfig = '0 '* and *cyclecounter = '9719'* and *framepulse = '0 logical'***
- ***φ_2: syncCounter = '3'* and *syncConfig = '1'* and *cyclecounter = '9719'* and *framepulse = '0 logical'***

then in the **following clock edge** the FSM moves to state **SYNC and** the ***"syncOut"*** output signal will be loaded with '*1 logical*'

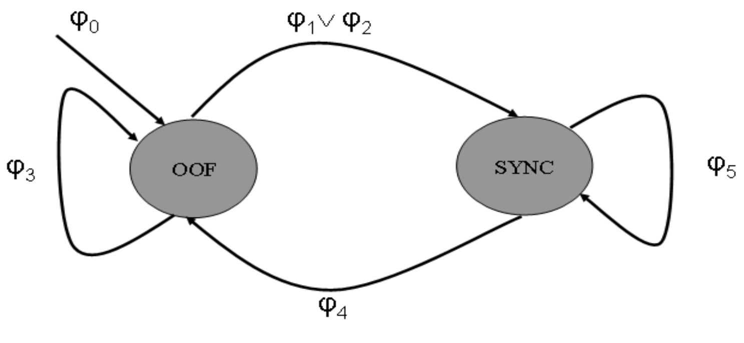

Fig 10. State diagram for *Sync Process*

The specification in PSL for this requirement is

-- [PSL_SM: 6] OOFTrSync

Property *OFFtoSYNC* **is always** ((FSM_state = **OFF and** ($\varphi_1 \vee \varphi_2$)) $\rightarrow$ **next** (FSM_state = **SYNC and** syncOut = HIGH);

The VHDL code for this requirement is

-- [P_SM: 6] OOFTrSync

if **(FSM_State = OOF) then**

if **(((sync_counter = 1** and **sync_config = 0** and **cyclecounter = 9719** and **frame_pulse = LOW)** or **(sync_counter = 3** and **sync_config = 1** and **cyclecounter = 9719** and **frame_pulse = LOW)**

then

FSM_State <= SYNC

syncOut = HIGH;

Where LOW = '*0 logical*' and HIGH = '*1 logical*'.

It can be seen that the writing of the assertion in PSL language and the corresponding code in VHDL from the *semi-formal specification* is almost direct and not complex. For each transition or loop of the state machine, the same procedure to obtain the *semi-formal*, formal *specification* and the code in VHDL can be applied. Some properties do not generate a code, but they verify certain behaviors. An example is property Connectivity of states which tests that the circuit does not remain forever in a specific state (sink state), see requirement ρ_6.-- [P_SM: 5] OOFConnectivity

It **always** happens that

if the **FSM** reaches the state of **SYNC then, eventually,** the **FSM** will return to **OFF** again

-- [PSL_SM: 6] OOFConnectivity

Property *OOFConnectivity* **is always** ((FSM_state /= **OFF**) $\rightarrow$ **eventually** (FSM_state = **OFF**)

C. A set of assertions for Sync Process

Making a refinement of the original requirements ρ jointly with the architecture document, a semi-formal specification $\varphi = \{\varphi_1, \varphi_2, \ldots, \varphi_{33}\}$ for the module Sync Process is obtained. From the Semi-formal specification, it is possible to obtain a set of PSL assertions $\xi = \{\xi_1, \xi_2, \ldots, \xi_{33}\}$ and to implement the code in VHDL. Let $\psi_{Sync\ Process}$ be the implementation obtained in VHDL from the semi-formal specification, and let $\xi_{Sync\ Process}$ be the set of PSL assertions representing a formal description, as shown in Table 5; then, the verification process must prove that

$$\psi_{Sync\,Process} \rightarrow \xi_{Sync\,Process} \tag{10}$$

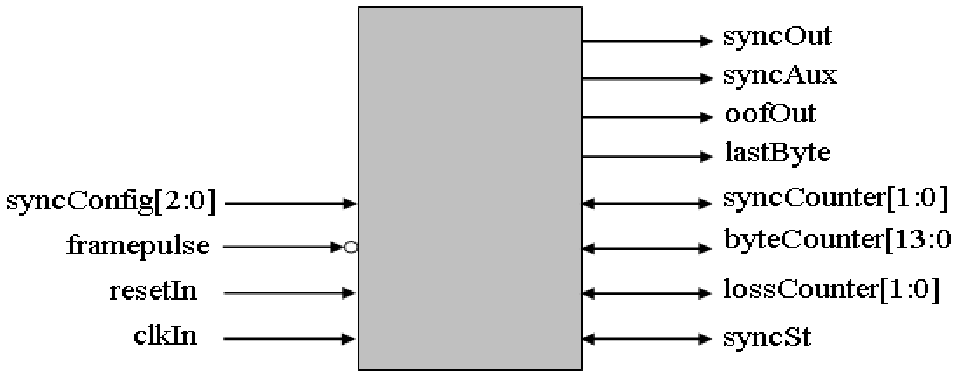

Fig 11. *Sync Process* Architecture

Port	Description
syncConfig[2::0]	Configuration port
framePulse	When activated, it indicates the frame beginning
ResetIn	Aligner's reset
ClkIn	Clock of the system. 77.76 MHz for STS-12 and 622.08 MHz for STS-48
SyncOut	It indicates the frame synchronization
SyncAux	It indicates the synchronization to internal modules
OofOut	It indicates the loss of synchronism.
LastByte	It indicates the last byte of frame
syncCounter[1::0]	Frame pulse counter.
byteCounter[13::0]	Frame byte counter.
lossCounter[[1::0]	Loss frame counter.
SyncSt	It denotes the aligner's state

Table 4 Sync Process Port Description

Category	PSL Assertions
Init State Reset	$\{\xi_0\}$
State Reachability	$\{\xi_1, \xi_2\}$
State Connectivity	$\{\xi_3, \xi_4\}$
State Transitions	$\{\xi_8, \xi_9, \xi_{10}\}$
State Loops	$\{\xi_5, \xi_6, \xi_7\}$
State non-connectivity	$\{\xi_{11}, \xi_{12}\}$
Counters Range	$\{\xi_{13}, \xi_{14}, \xi_{15}, \xi_{16}, \xi_{17}, \xi_{18}, \xi_{19}, \xi_{20}, \xi_{21}, \xi_{22}, \xi_{23}, \xi_{24}, \xi_{25}, \xi_{26}, \xi_{27}, \xi_{28}, \xi_{29}, \xi_{30}\}$
Outputs Behavior	$\{\xi_{31}, \xi_{32}, \xi_{33}\}$

Table 5 PSL Assertions for the verification of *Sync Process*

Besides, for the control of the verification process of the mentioned module, we used the following set of metrics: Writing time of $\xi_{Sync\ Process}$; number of syntax errors for each version of $\xi_{Sync\ Process}$; verification time used by *Safelogic Verifier*; number of assertions in PSL against the number of lines of code in VHDL; errors detected in the semi-formal specification and $\psi_{Sync\ Process}$ and number of developed versions for $\xi_{Sync\ Process}$.

Requirements	Assertions
ρ_0	ξ_1, ξ_{10}, ξ_{12}, ξ_{13}, ξ_{14}, ξ_{15}, ξ_{19}, ξ_{21}
ρ_1	ξ_2, ξ_2, ξ_4, ξ_5, ξ_6, ξ_7, ξ_8, ξ_9, ξ_{10}, ξ_{12} ξ_{15}, ξ_{17}, ξ_{23}, ξ_{27}, ξ_{32}
ρ_2	ξ_3, ξ_5, ξ_{13}, ξ_{14}, ξ_{15}
ρ_3	ξ_2, ξ_3, ξ_4, ξ_5, ξ_{11}, ξ_{20}, ξ_{28}, ξ_{32}
ρ_4	ξ_2, ξ_3, ξ_4, ξ_5, ξ_9, ξ_8, ξ_{14}, ξ_{15}, ξ_{28}, ξ_{32}
ρ_5	ξ_2, ξ_4, ξ_{13}
ρ_6	ξ_2, ξ_4

Table 6 Relationship between ρ and ξ for the module *Sync Process*

D. Relationship between ρ and ξ for module Sync Process

This relationship is shown in Table 6. It can be seen that one property can have a relationship with various requirements of the specification, for example, property ξ_3 appears in requirements ρ_2, ρ_3 and ρ_4. In these cases, testing their behaviors using simulation and test vectors is a hard task while using techniques of formal verification is straightforward.

E. Implementation of Sync Process

In order to study the impact of our proposal in the whole design cycle compared with the traditional methodology, three implementations of the circuit were developed and verified. We will use the following notation to make reference to these three implementations in the rest of the document.

1. ψ_{imp}^{O} Original or reference Implementation.
2. ψ_{imp}^{PSL} Assertion-Based Implementation.
3. ψ_{imp}^{F} Free Implementation of the Designer.

The first implementation ψ_{imp}^{O} was done by a non-experienced engineer [16] and previously verified functionally. It was developed from the requirements specification, i.e. this implementation was not intended for formal verification.

The second and third implementations were written by an experienced engineer. The second one ψ_{imp}^{PSL} was elaborated directly from our semi-formal specification and it was verified with functional tests additionally.

Thus, we call it Assertion-Based Implementation. And, the last one, ψ_{imp}^{F}, written by the same engineer, was elaborated from our semi-formal specification, but the code was optimized based on the experience of this engineer, obtaining a version with minimum lines of code. We call it Free Implementation of the Designer.

5. Results

During the code implementation of the circuit represented by $\psi_{Sync\ Process}$ and its verification, some statistics were controlled, particularly design errors and the number of design cycles, see Figure 12.

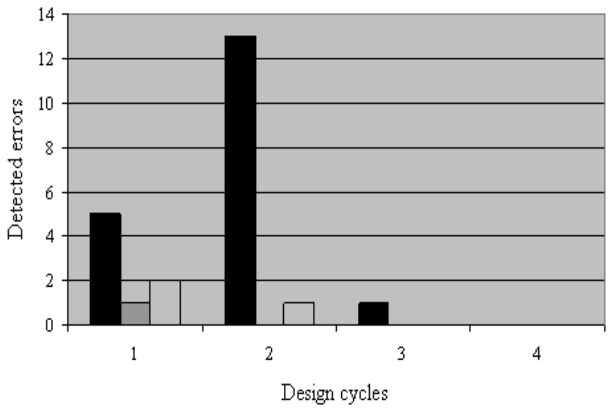

Fig 12. Error Distribution vs Design Cycles for $\psi_{Sync\ Process}$

From our analysis, we concluded that

- In ψ_{imp}^{O}, more errors were detected and it requires more time than the other two versions to obtain a final version for $\psi_{Sync\ Process}$ with zero errors. Also, the designer's effort to obtain the final code in VHDL for ψ_{imp}^{O} requires more analysis and error probability due to a wrong interpretation of the requirements specification is greater than error probability due to a wrong interpretation of a semi-formal specification.
- With ψ_{imp}^{F}, some errors were detected, but it required more time than ψ_{imp}^{PSL} to obtain a zero errors implementation. Nevertheless, this implementation required less designer's effort and analysis than ψ_{imp}^{O} because it was obtained from a semi-formal specification which defines an exact description of the expected behavior, and the structured and formal approach allows an abstract view of the described system, so the error probability is reduced significantly. We saved time in the verification process, but we required time to analyze the semi-formal specification and to obtain an optimized code.
- ψ_{imp}^{PSL} had fewer errors and was corrected in a very short time because it was written directly from the semi-formal specification, i.e. the comparisons and/or decisions written in the code correspond to the PSL assertions. This implementation requires more lines than ψ_{imp}^{F}, see Table 7, but less effort and analysis. It was only necessary to determine the order in which the PSL assertions would have to be written in VHDL code. Therefore, in ψ_{imp}^{PSL} the use of a formal methodology improves communication within the design team reducing the error probability.

Also, it should be mentioned that these implementations were synthesized with Synopsis^{TM}; the corresponding results for verification time, circuit area in cells, delay, and the total lines of effective VHDL code are shown in Table 7. As we can see, while the synthesis of implementations ψ_{imp}^{PSL} and ψ_{imp}^{F} have the same area and delay, ψ_{imp}^{O} requires more cells and has a greater delay.

Implementation	Verification Time (Sec)	Area (Cells)	I/O Delay (nSec)	VHDL Lines
ψ_{imp}^{O}	319	72	7.09	183
ψ_{imp}^{PSL}	16	69	7.08	145
ψ_{imp}^{F}	11	69	7.08	128

Table 7 Synthesis Results

From the learning process for writing specification $\xi_{Sync\ Process}$ for this circuit (see Figure 12 and 13) we concluded that

- $\xi_{Sync\ Process}$ had few errors and was corrected in a very short time because it was written directly from the semi-formal specification. The types of errors detected were syntax and semi-formal specification errors.
- We found errors in properties ξ_{32} and ξ_{33} (see Table 5). These errors were due to the lack of precision in the semi-formal specification regarding the current state of the FSM. Therefore, ambiguities in the semi-formal specification lead to an interpretation error.

Finally, the total time employed to obtain the final versions of $\xi_{Sync\ Process}$ is shown in Figure 13. Components of this amount of time are *writing time* of all the assertions; *correction time of syntax errors in* $\xi_{Sync\ Process}$; *correction time of errors detected in the semi-formal specification*; and *correction time of errors detected in the* $\psi_{Sync\ Process}$.

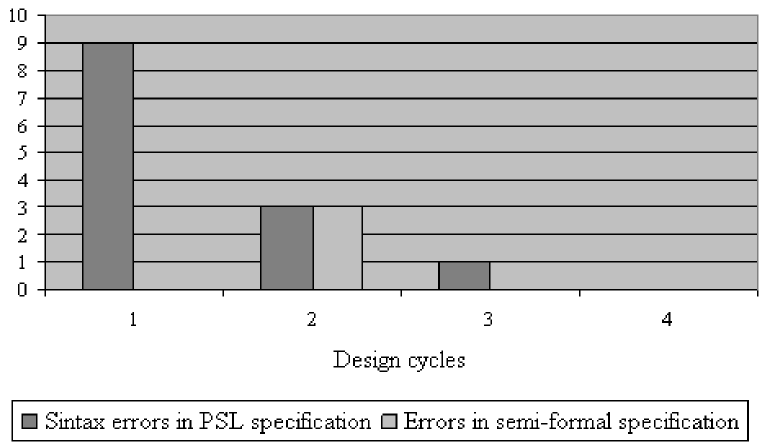

Fig 13. Learning process for writing $\xi_{Sync\ Process}$

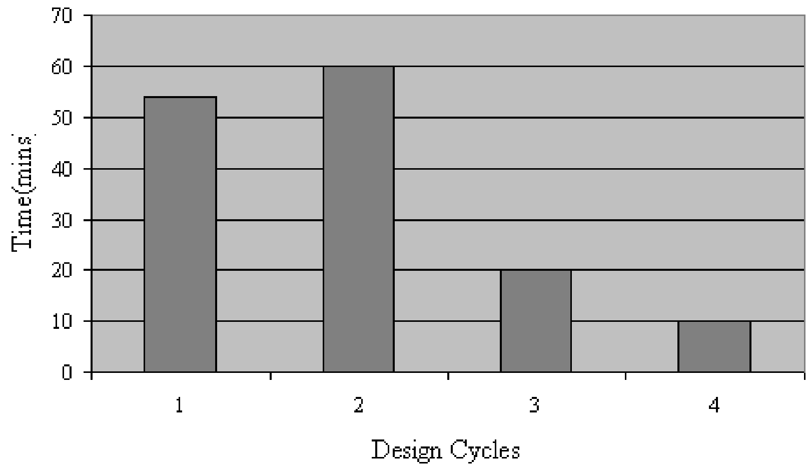

Fig 14. The total time employed to elaborate $\xi_{Sync\ Process}$

Let $\xi^{(i)}_{Sync\ Process}$ be the PSL specification in version *i*. The set of assertions related with $\xi^{(i)}_{Sync\ Process}$ does not generate counterexamples, for *i* = 1,2,3. Therefore, some assertions for generation of counterexamples were included in $\xi^{(4)}_{Sync\ Process}$ in order to find all possible errors in $\psi_{Sync\ Process}$.

6. Verification Time Used For *Safelogic Verifier* for $\psi_{Sync\ Process}$

From the developed PSL assertions for $\psi_{Sync\ Process}$, see Table 5, we selected all the assertions in order to do a comparison of the verification time versus the number of assertions. The tests were executed in a computer with a Pentium IV processor at 2.8 GHz, 256MB of RAM Memory, and Windows XP Professional.

Furthermore, we specifically considered the obtained results for all assertions with frame size of 10 and 100 bytes. The obtained results are shown in Table 8. The verification process was carried out using implementation ψ^{PSL}_{imp} for Sync Process.

Frame Size	Verification Time without counter-examples	Verification Time with counter-examples	Number of counter-examples
10	0.3 sec	2.4 sec	6
100	5.3 sec	1661.2 sec	6

Table 8 Total verification time for ψ^{PSL}_{imp}

Now, from Table 8 we can see that the total verification time for $\psi_{Sync\ Process}$ with frames of 100 bytes length is considerably greater than $\psi_{Sync\ Process}$ for a 10 bytes frame length. These results show that for a longer frame length and assertions that generate counterexamples, the verification time increases exponentially or according to a power law.

If the VHDL codes use constants as circuit parameters, we can do formal verifications for small values, which reduce significantly the amount of used Boolean variables and, therefore, reduce the amount of time used by the formal verification tool. In this way, a fast feedback in the error correction process is attained.

In [17], a work was presented; it includes the verification of a second circuit where the same procedure was followed and where design time was saved too.

6.1 Rules of thumb

We propose to use the following rules in the design cycle; these have been proven in several different designs obtaining excellent results:

1. Obtain the *semi-formal specification* from the general specification of requirements.
2. *The Semi-formal specification is very useful, not only for the verification engineers, but also for the designers*, since it is the input document for both types of engineers.
3. *The design engineers should write the code following the semi-formal specification* and document it according to the referred requirement.
4. Before simulation, the *verification engineers should write assertions from the semi-formal specification*, not only to prove that the implementation satisfies the expected behaviors, but also to check non-expected behaviors that generate counterexamples.
5. The *assertions from the semi-formal specification and the implemented code* should be used in a *verification tool*, e.g. *Safelogic Verifier,* during the verification process.
6. Then, only a few simulations are needed by the designers to check expected behaviors.
7. The global functional verification should be reduced to some general cases.

7. Conclusions

A set of PSL properties was established to carry out the formal specification and verification of a circuit.

The structure of a basic set of properties for verification of FSMs is presented. Therefore, it can be used, in general, for different FSMs. The use of LTL and/or PSL is very important, not only for the verification, but also for the design process. By means of this methodology, not only the design of each block and its verification can be done in parallel, but also this parallelism leads to shorten the whole design process.

We give statistics in order to show the improvement that the use of assertions in the design cycle of a class of digital circuits provides, specifically in the implementation of the code in VHDL.

Our statistics in the verification process of the circuit represented by $\psi_{Sync\ Process}$ show that the number of versions for $\xi_{Sync\ Process}$ was relatively low; the writing time decreased from version to version, except in the second version of the study case. The cause was the discovery of some limitations in the tool. *These metrics should be used, joint to classical ones, to control the whole design process*.

A set of rules of thumb, for the engineers, is given in order to help to improve the whole design process. Particularly, we pointed out the following:

- The writing of the *semi-formal specification* from the general specification of requirements.
- *The semi-formal specification is very useful, not only for the verification engineers but also for the designers.*
- The design engineers should *write the code following the semi-formal specification*, and document it according to the referred requirement.

As the formal or semi-formal specification cannot discover possible errors at the refinement phase of the requirements, then for the discovery of these errors *is useful to maintain the functional verification*. Therefore, ABV using PSL is useful not only for the verification process but for the implementation design.

References

[1] H. Foster, A. Krolnik, D. Lacey, "Assertion-Based Design", Kluwer Academic Publishers, Second Edition, (2004).

[2] J. Bergeron, "Writing Testbenches", Kluwer Academia Publishers, (2000).

[3] H. Foster, R. Drechsler, T. Kropf, "Formal Verification: Current Use and Future Perspectives", IEEE Design and Test, (2002).

[4] Kausik Datta , P. Das, "Assertion Based Verification Using HDVL " , Proceedings of the 17th International Conference on VLSI Design (VLSID'04), pp. 319, January 2004.

[5] K. Winkelmann, Cost-efficient Block Verification for a UMTS Up-link Chip-rate Coprocessor, IEEE, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04), (2004), Vol I, pp: 162-167.

[6] T. Kropf, "Introduction to Formal Hardware Verification", Ed. Springer, First Edition, (1999).

[7] H. Foster, A. Krolnik, D. Lacey, "Assertion-Based Design", Kluwer Academic Publishers, Second Edition, (2004).

[8] http://www.systemverilog.org/

[9] http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[10] http://www.jasper-da.com/safelogic/

[11] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite States Machines- A Survey" Proceedings of the IEEE, vol. 84, no. 8, pp. 1090–1123, August (1996).

[12] A. Aguilar, D. Torres, R. E. González, "Verificación Formal de un Alineador de Tramas utilizando Lógica Temporal Lineal", IEEE ROC&C, Acapulco, Guerrero, México, CP-20. (2003).

[13] UIT-T Rec. G.780 Vocabulary of terms for SDH networks and equipment (UIT 06/99).

[14] D. Torres, "Alignment Problem for the Synchronization in Digital Circuits", Proceedings of the IASTED International Conference Circuits, Signals, and Systems, Cancun México, pp. 339-343, May (2003).

[15] D.Torres, A. Redondo, M. Guzman, " MSOH Processor for STM-0/STS-1 to STM-4/STS-12: component of a SONET/SDH Library ", Microelectronics Reliability 43 (2), 217-223, (2003).

[16] J. Verdin, D. Torres and E. García, " SONET/SDH Alignment Problem ", IEEE Latin America CAS Tour (2002).

[17] J. A. Moreno, D. Torres, S. Robles, "Design with formal verification of a ADM switch module", IEEE CIINDET, September 28, (2005), Cuernavaca, Morelos, Mexico.

Acknowledgment

The authors gratefully acknowledge the contribution of J. Moreno, Research Assistant at CINVESTAV-IPN, Guadalajara. Likewise, they would like to acknowledge *Safelogic* for allowing them to use the *safelogic verifier* for the formal verification process of their code

Authors Biography

Deni TORRES-ROMAN

He received a Ph.D. degree in telecommunication from the Technical University in Dresden, Germany in 1986. He was professor at the University of Oriente, Cuba. Dr. Torres is co-author of a book on Data Transmission. He was awarded the Telecommunication Research Prize in 1993 from the AHCIET Association and was the recipient of the 1995 Best Paper Award from the AHCIET Review, Spain. Since 1996, he has been associate professor at the Research and Advanced Studies Center of IPN (CINVESTAV- Mexico). His research interests include hardware and software designs for applications in the telecommunication area. He is member of the IEEE.

Joaquin CORTEZ-G

He received the Master's degree in electric engineering from CINVESTAV-Guadalajara, in 2001 and the Ph. D. degree in electric engineering from CINVESTAV-Guadalajara, in 2008. He is currently an electrical and electronics engineering professor at the Technologic Institute of Sonora,. His teaching and research interests include digital communications and digital signal processing.

Raúl Ernesto GONZÁLEZ-TORRES

He has a PhD in mathematics from The University of Houston, USA. He is currently professor at CINVESTAV-Guadalajara and is doing research on applications of Logic to Engineering and Computer Science; more specifically, he is working on the formal verification of reactive systems using temporal logics, the development of efficient machine learning algorithms in order to improve the actual capabilities of verification techniques and tools, and on automated reasoning methods.