From Assertion-based Verification to Assertion-based Synthesis

Y. Oddos, Katell Morin-Allory, D. Borrione

From Assertion-based Verification to Assertion-based Synthesis

Yann Oddos, Katell Morin-Allory and Dominique Borrione

TIMA Laboratory (CNRS/Grenoble-INP/UJF),
46 Av. Félix Viallet, 38031 Grenoble CEDEX, France
{yann.oddos,katell.morin-allory,dominique.borrione}@imag.fr
http://tima.imag.fr

Abstract. We propose a linear complexity approach to achieve automatic synthesis of designs from temporal specifications. It uses concepts from the Assertion-Based Verification. Each property is turned into a component combining classical monitor and generator features: the extended-generator. We connect them with specific components to obtain a design that is correct by construction. It shortens the design flow by removing implementation and functional verification steps. Our approach synthesizes circuits specified by hundreds of temporal properties in a few seconds. Complex examples (*i.e.* conmax-ip and GenBuf) show the efficiency of the approach.

Key words: Assertion Based Verification, Assertion Based Synthesis, PSL, LTL, High-level Automatic Synthesis, Monitors, Generators

1 Introduction

To guarantee the correct functionality of a System On Chip is a daunting challenge. Over the past few years, design methods have started from higher description levels, and have been supported by more efficient CAD tools. In contrast, validation methods suffer from a growing lag, costing an increasing part of the overall design time.

Among all the proposed methods, the ABV [FKL03] (Assertion Based Verification) is widely used for its efficiency, both for static or dynamic verification, and is applicable all along the design flow. Within this framework, two kinds of temporal properties are of special interest:

- **Assertions** describe the correct behaviors of the design. They can be turned into monitors which check if the corresponding property is violated.
- **Assumptions** constrain the inputs produced by the environment to comply with the expected communication protocol of the Design Under Verification (DUV). They can be turned into generators which provide an executable model for this environment.

Two IEEE standards are mainly used to write temporal properties : PSL (Property Specification Language) [FWMG05] and SVA (SystemVerilog Assertions) [SMB+05]. In the following, all the properties are written in PSL, but our method applies to SVA as well.

In our approach, we use the simple subset of PSL (denoted PSL_{simple}). It conforms to the notion of monotonic advancement of time and ensures that formulas within this subset can be used easily in dynamic verification: simulation, emulation etc...

As an example, consider the following PSL property **P1**:

$$\text{Property } \mathbf{P1} : \mathsf{always}(\mathit{Start} \rightarrow \mathit{Req} \ \mathsf{until} \ \mathit{Ack})$$

Property **P1** states: for each cycle where Start is '1', a request *Req* should be produced and maintained active (*Req*='1') as long as the acknowledge signal *Ack* is not active (*Ack*='0').

The trace on Figure 1 satisfies property **P1**. Signal Start is active at cycle ♯1 and *Req* is fixed to '1' during the same cycle. Then *Req* remains active up to ♯7. At ♯8, signal *Ack* takes value '1'. After this cycle, all the constraints have been satisfied and the trace complies with **P1**.

In the case of an unbounded trace, the property should be permanently valid due to the presence of the always operator. Each cycle when Start is active should initiate a time sequence where *Req* is active at least up to the cycle when *Ack* is '1'.

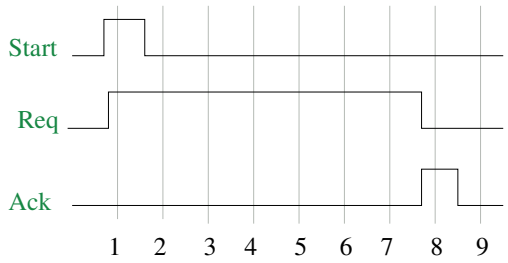

Fig. 1. A trace satisfying property P1

Figure 2 shows this retriggering of the property evaluation due to successive activations of Start. Although not necessarily what one would expect, the trace of Figure 2 also satisfies property P1.

We have developed the Horus project which aims at providing methodologies and tools for efficiently supporting property-based design all along the design flow. With respect to previous works and existing CAD software, Horus exhibits new and competitive advantages:

- The monitors and generators are built in a modular way, so that modules for sub-properties can be reused in more complex ones.
- The modular construction is very efficient: it takes a fraction of a second for dozens of complex properties.
- The method for building the monitors and the generators has been proven correct with the PVS theorem prover [MAB06].

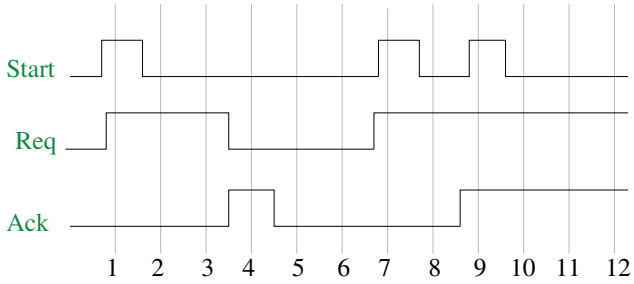

Fig. 2. Trace with several activations of $Start$

Another promising research domain is the automatic synthesis from temporal specifications. It consists in automatically transforming a temporal specification into an HDL description which is correct by construction. Figure 3 depicts a typical design flow where the hand-coded design has to be verified to guarantee that the realized functionality complies with the specification.

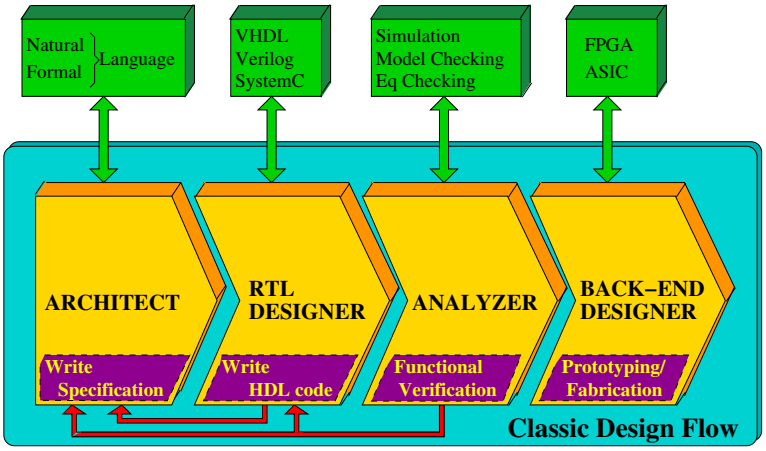

Fig. 3. Design flow including the functional verification of a hand-coded RTL design

As shown on Figure 4, the main advantage of automatic synthesis is the elimination of the implementation and functional verification steps from the design flow.

The problem of the automatic synthesis was first formulated by Church in 1982 by the following question: “Given a specification, is there a realization satisfying it?”. Unfortunately, this problem has a solution triply exponential, resulting in three major limitations:

– explosion of the memory size

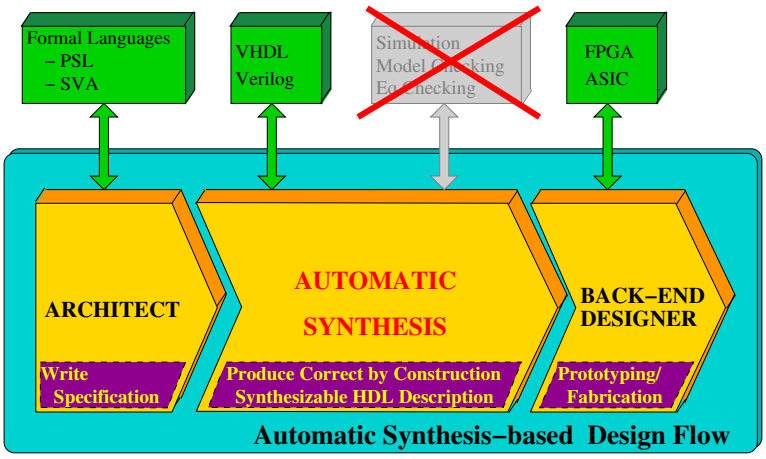

Fig. 4. Design flow with automatic synthesis of the RTL from formal properties

– explosion of the synthesis time.
– the resulting design is too complex and not efficient (huge area and low frequency).

Up to now, all the proposed methods that have been explored reduce the complexity by constraining the type of properties or designs that can be processed. It results in exponential or polynomial complexities that still limit the approach to simple case studies.

Whereas linear complexity approaches are available for Boolean specifications, no such methods exist for complex temporal specifications.

We use concepts of Assertion-Based Verification and the property synthesis approach developed for Horus to define the method SyntHorus. It has a linear complexity approach regarding the specification and automatically synthesizes a temporal specification written in PSL_{simple} into a correct by construction HDL description.

We turn assertions and assumptions into a new kind of hardware components mixing monitors and generators: the extended-generators. They are the basic components of the synthesized design and have been proved correct with respect to the PSL semantics. Combined with specific components called Solvers, the resulting circuit provides the final design corresponding to a given specification.

The SyntHorus tool can process complex specifications, composed with hundreds of properties, and produces the final circuit in a few seconds. The size of the circuit is proportional to the size of the specification.

2 State of the Art

2.1 Assertion-Based Verification

Many static verifiers, and the main commercial RTL simulators support PSL [CVK04]. The European funded project "PROSYD" has published methodologies for the use of PSL, and developed tools around PSL [PRO]. Notably, the RAT prototype, based on SAT and bounded model checking, helps verify property consistency [BCE+04].

The first industrial implementation of monitors is IBM FoCs [IBM]. The result of FoCs is a cycle accurate simulation process that may be turned into a synthesizable component with minor edition.

In the context of model checking, the usual technique translates a LTL formula into a non-deterministic automaton that recognizes all the acceptable sequences of values [DGV99, GO01]. The transformation into a deterministic automaton, needed for a hardware implementation, is exponential in the number of non-deterministic decision states [ST03]. A syntactically simple PSL formula can easily expand into a large LTL formula, so the direct automata theoretic approaches are too inefficient.

Moreover, due to the `always` operator, the monitors generated for on-line observation must be retriggerable, and may be concurrently evaluating several instances of the property for several starting points in the sequence of signal values. When a property fails, for debug purposes, it is essential to identify the starting point of the subsequence that caused the failure. The implementation principles for this feature have been described either in terms of control graphs with colored tokens [MAB07], or as a multiplication of monitors [BCZ06]. A similar automata-based method [CRST06] is used in model checking.

Another aspect is the use of properties to specify constraints on the design environment. Most industrial simulators provide software test vector generators based on the generation of constrained pseudo-random numbers [ABC+].

Among the static methods, Calamé [Cal05] builds the product of the design automaton and the property automaton to extract test vectors that may lead to a wrong execution. Another approach is based on slicing [YJC04]: the design is cut between registers to extract constraints and produce test vectors for each slice, which are then composed to get the test vector for the whole component. All these techniques are difficult to use due to the complexity of their algorithms.

More efficient approaches rely on the property formulas, not the design under verification, to generate test vectors. The concept of "cando objects" [SNBE07] is technically very different, and oriented towards model checking rather than on-line execution; in particular, they do not support arbitrary repetition ('+' and '*') operators.

2.2 Automatic Synthesis From Specifications

A large body of research was devoted to the synthesis of combinatorial operators from untimed mathematical relations. We shall not discuss them, since the target of our work is the synthesis of control-type sequential circuits, where the successive occurrence of events is an essential aspect of the circuit behavior.

The specification of such sequences with regular expressions is not new [FU82, SB94]. These pioneer works use different kinds of BDDs (Free-BDDs and Reduced-Ordered BDDs respectively) to support the synthesis process. The use of BDD-based algorithms leads to the so called "state explosion problem" which confines the application to simple designs.

Other specification methods have been proposed, focusing on communication protocols. From BNF grammars, Öberg synthesizes controllers using a directed acyclic graph representation [Öbe99]. Alternatively, Müller [SM02] defined a dedicated SystemC library to describe the specifications and uses automata-based methods to build the final design.

Aziz et al. use logical S1S formulas (second order properties on naturals) to perform automatic synthesis of sequential designs [ABBSV00]. Despite the application of a variety of optimization techniques to reduce the complexity during the creation of the design, the automata-based approach cannot apply to complex designs.

Taking as input a specification in a standard assertion language is a more recent concern. Bloem et al. [BGJ$^+$07] defined a "Generalized Reactivity(1)" subset of PSL from which properties are translated to automata; game theory algorithms are applied to compute all the correct behaviors of the design under all admissible interactions with the environment. The method is more powerful than the preceding ones. It is polynomial in N^3, where N is the sequential complexity of the specification.

Eveking et al. [SOSE08] aim at generating verification-friendly circuits. This method takes as input ITL, a proprietary dialect of interval temporal logic defined by OneSpin Solutions. It is similar to the PSL simple subset, limited to finite traces. In contrast to our approach, non deterministic behaviors assume additional inputs fed by an external random source, and consistency is checked statically during the construction.

Our main objective is to quickly generate efficient circuits from complex sets of properties. Instead of using an automata based approach, or restricting the application to a specific type of designs, we use a modular construction. The complexity is then encapsulated in different levels of basic components. The overall synthesis method was proved to correctly generate circuits that comply with the PSL semantics.

3 Assertion-based Verification

3.1 Monitors

A monitor is a synchronous design detecting dynamically all the violations of a given temporal property. We detail here the last release of our approach used to synthesize properties into hardware monitors. It is based on the principles described in [MAB06].

The monitor synthesis is based on a library of primitive components, and an interconnection scheme directed by the syntax tree of the property. In particular, there is one primitive monitor for each FL operator of PSL. We have defined two types of primitive monitors: connectors and watchers. The first one is used to start the verification of a sub-property. The watcher is used to raise any violation of the property.

The sets of connectors and watchers are given Table 1. The watcher mnt_Signal is used to observe a simple signal.

Watchers	mnt_Signal, ↔, eventually!,never, next_e, next_event_e, before
Connectors	→, and, or, always, next!, next_a, next_event, next_event_a, until

Table 1. Primitive PSL monitors

Primitive monitors have a generic interface depicted Figure 5. It takes as input two synchronization signals *Clk* and *Reset_n*, a *Start* activation signal, and the ports *expr* and *cond* for the observed operands. The output ports are: *trigger* and *pending* for a connector; *pending* and *valid* for a watcher.

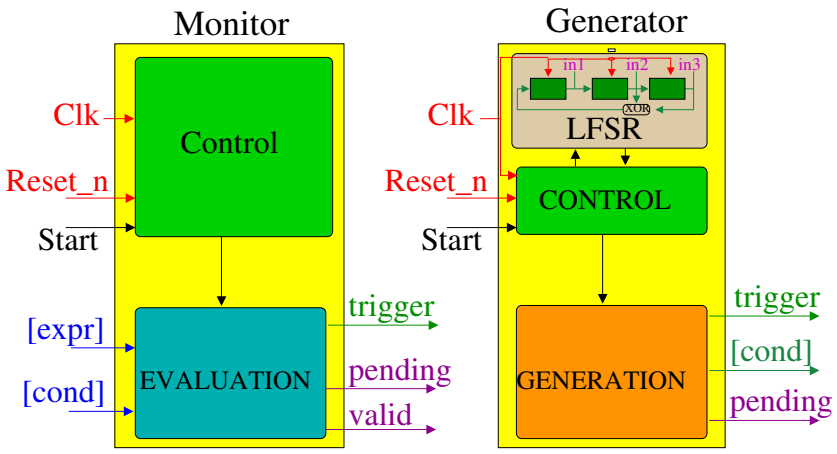

Fig. 5. Architectures and Interfaces for primitives monitors and generators

The overall monitor is built by post-fixed left to right recursive descent of the property syntax tree. For each node of type connector, its Boolean operand, if any, is connected to input *cond*. The output *trigger* is connected to input *Start* of its FL operand. For the watcher type node, its Boolean operands are directly connected to the inputs *expr* and *cond* of the current monitor. Its output *valid* is the *valid* output of the global monitor.

Figure 6 gives the syntax tree of Property P1 defined in section 1:

Property **P1** : always($Start \rightarrow Req$ until Ack)

The corresponding monitor for P1 is given Figure 7. The couple of signals (*valid*, *pending*) gives the current state of the property at any cycle: failed, holds, holds strongly or pending.

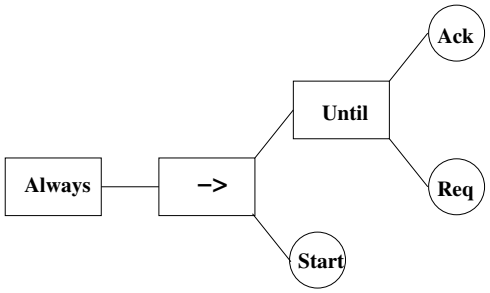

Fig. 6. Tree Structure of PSL Property P1

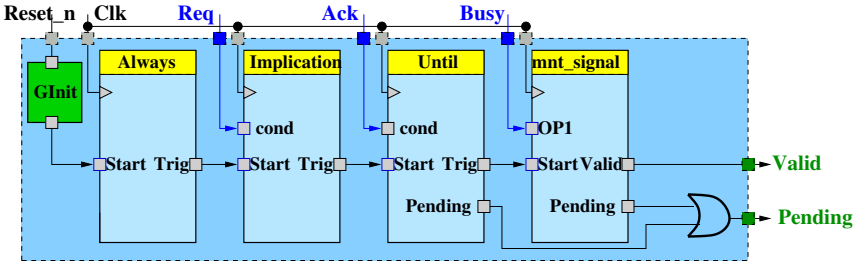

Fig. 7. Monitor for the PSL property P1

3.2 Generators

A generator is a synchronous design producing sequences of signals complying with a given temporal property. Their synthesis follows the same global principle as for the monitors: the overall generator is built as an interconnection of primitive generators, based on the syntax tree of the property [OMAB06].

Primitive generators are divided into all the connectors (associated to all PSL operators) and the single type of producer to generate signal values: gnt_Signal.

The interface of primitive generators (Fig 5) includes:

- the inputs Clk, $Reset\_n$, $Start$: same meaning as for monitors.
- the outputs $trigger$ and $cond$ used to launch the left and right operand (for connectors).
- the output $pending$, to indicate if the current value on $trigger$ and $cond$ are constrained or may be randomly assigned.

Since many sequences of signals can comply with the same property, we need the generators to be able to cover the space of correct traces. To achieve this goal, the gnt_Signal embeds a random number generator (based on a Linear Feedback Shift Register or a Cellular Automaton). By default, the outputs of an inactive complex generator are fixed to '0'. It is possible to produce random values by switching the generic parameter RANDOM to 1. If $pending$ is inactive, the values on $trigger$ and $cond$ are not constrained and produced by the random block.

4 From Assertion-based Verification to Assertion-based Synthesis

A design specification involves signals to be monitored (inputs of the design), and to be generated (outputs of the design). It is necessary to make a clear distinction between these two kinds of signals, directly into the specification. PSL has been designed for functional verification and not for the synthesis of temporal specifications. It does not provide this distinction at the specification level. We have to adapt PSL for specification synthesis.

Moreover, as a generated signal can be driven by several sources (*i.e* a signal generated in different properties), a resolution mechanism has to be designed to solve the value of multi-source signals.

4.1 Example: the CDT design

The controller CDT is used to illustrate our synthesis approach of temporal specifications. It is a simple communication interface enabling to send data. If the controller is idle and a data request is received on *Req*, it transfers the value present on *Cmd* to an external component via the output port *Data*.

The transfer ends when the signal *Ack* is received from the external component. The controller needs 4 cycles to initiate a new transfer.

4.2 Annotations of PSL Properties

We have chosen to annotate each signal *sig* using the following convention:

- sig_m: signal *sig* is monitored
- sig_g: signal *sig* is generated

Consider the following specification Spec_cdt for the CDT design:

- inputs={*Init*,*Req*,*Ack*,*Cmd*}, outputs={*Data*,*Busy*, *Send*}
- **F0** : always($Init_m \rightarrow$ (!$Send_g$ && !$Busy_g$ && $Data_g$="0000"));
- **F1** : always(!$Init_m$ && Req_m)$\rightarrow$ (($Send_g$ && $Data_g$=Cmd_m)until Ack_m);
- **F2** : always((!$Init_m$ && $Send_m$)$\rightarrow$ next_a[1:4]($Busy_g$));

Property F0 states that during the initialization (signal *Init* active), all the output signals are fixed to '0' (*Send*, *Busy* and *Data*).

The two following properties are used when the design is not being initialized.

Property F1: if a request is submitted, the value *Cmd* is passed on output *Data* until the transfer is ended by receiving '1' on input *Ack*. Meanwhile the signal *Send* is maintained to '1' until the reception of *Ack*. It notifies the external component that the transfer is currently being processed.

Property F2: for each request, the interface is busy during 4 consecutive cycles and cannot receive any other request.

In our examples, properties only deal with input and output ports of the design. This is not a limitation. In other cases, internal signals may be used as well.

A signal that is annotated "g" in several properties, or several times in a single property, is called *duplicated*. For instance, signals *Send*, *Busy* and *Data* are duplicated in Spec_cdt.

In the current status of our development, the annotation of signals is partially automated and may have to be complemented by the designer.

4.3 Consistency of the specification

Before engaging into the process of producing an implementation, it is a pre-requisite that the specification be proven to be consistent. Two concepts must be distinguished:

Inconsistency A set of properties is inconsistent if the set of value traces for which they are satisfied is empty. As an example, the following two properties are inconsistent:

- assert always $(a_m \wedge c_g)$
- assert always $(a_m \wedge !c_g)$

Tools such as RAT (Requirement Analysis Tool) [BCE+04] or the method on which Cando objects are built [SNBE07] detect inconsistencies statically.

We shall call consistent a set of properties such that, whatever the input combination, all the outputs always can be given a valid value.

Realizability It may happen that one, two or more properties are satisfied under some constraints on the input signals, and exhibit inconsistencies otherwise. Take the following example:

- assert always $(a_m \Rightarrow c_g)$
- assert always $(b_m \Rightarrow !c_g)$

If *a* and *b* have different values, the specification is realizable. But if signals *a* and *b* are both active at a given cycle, *c* is constrained to contradictory values.

Overcoming this problem is done by adding assumptions to the specification. In the above example, the following assumption is added: assume never (*a* and *b*).

Synthesizing the specification In our approach, the specification is analyzed using RAT. *Assume* properties may be added in the process to produce realizable specifications.

Methods described in pre-existing works statically enumerate all the possible responses of the design under all the possible actions of the environment, and hard-code all the responses directly in the design. In contrast, our technique relies on the use of special purpose "solver" components that compute the output values on the fly.

5 Assertion-based Synthesis

5.1 The Extended-Generator

An extended-generator component is a combination of a monitor and a generator. The basic principle is to produce a predefined sequence of signals (generator part) when a special sequence is recognized (monitor part).

Primitive extended-generator Only binary operators OPb can have both a monitored and a generated operand. Such operator has two corresponding primitive extended-generators: OPb_{mg} and OPb_{gm} which respectively (monitor, generate) and (generate, monitor) the left and right operands.

We impose the following restriction for the PSL properties: *a monitored operand of an extended-generator must be Boolean*. This limitation is just the adjustment of the PSL simple subset in the context of the extended-generators. To clarify the problem met in the monitoring of an FL operand, consider the following example:

$$\text{Property P2: } (sig_m \rightarrow (\mathsf{next[4]}A_m) \text{ until } B_g)$$

Assume that sig is '1' at cycle 0 and the production of signal B='1' is planned at cycle *t*. For all cycles j<t, the sub-property $\mathsf{next[4]}A_m$ must be verified. The verification will complete at cycle *t+3*. It is then impossible to ensure at cycle *t* that the extended-generator always fulfills the corresponding property.

The monitoring of an FL property that requires a knowledge of the future to produce signals at the present cycle is contrary to the principles of the PSL simple subset. Our restriction applies these principles in the new context introduced by the extended-generators, which combines the observation and generation concepts.

It follows that all the binary operators have at least one Boolean operand. This avoids all the potential ambiguities concerning the choice of a primitive extended-generator (OPb_{mg} or OPb_{gm}). As an example, it is possible to use untilgm since the right operand is Boolean, but not untilmg as the left operand can be a FL property.

Primitive extended-generators have the same interface as the generators (*cf.* Figure 8). The only difference lies in the port *cond* which is an input for the extended-generators. All the extended-generators are connectors. Their architecture is based on the monitors.

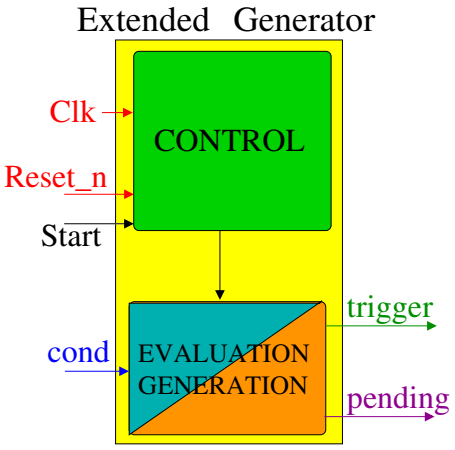

Fig. 8. Primitive Extended Generator Interface

Complex extended-generator While a primitive extended-generator matches a *binary* PSL *operator*, the complex extended-generator corresponds to a PSL *property*. It is composed of three kinds of primitive components: monitors, generators and extended-generators.

This feature raises the issue of using the appropriate primitive, and justifies the need for property annotation discussed in section 4. The property (A until B) can be turned into a monitor (A and B observed), a generator (A and B produced) or an extended-generator(A produced, B observed). All these ambiguities are solved by annotating the property before building the complex extended-generator.

A complex extended-generator has a generic interface taking as inputs the observed and synchronization signals, and providing on its outputs the generated and associated *pending* signals.

Building the complex extended-generator is performed in two steps, based on the syntax tree.

Step1 - Selection of primitive components A node of the syntax tree that has generators or extended-generators as operands is defined as a generator. If it has one operand of type generator and one of type monitor, it is an extended-generator. Two monitored operands produce a primitive monitor.

Step2 - Interconnection The interconnection scheme is the same as the one used for the monitors. All the basic elements are producers gnt_Signal. The extended-generator for property F1 is given Figure 9.

F1 : always(!$Init_m$ && Req_m)→ (($Send_g$ && $Data_g$=Cmd_m)until Ack_m);

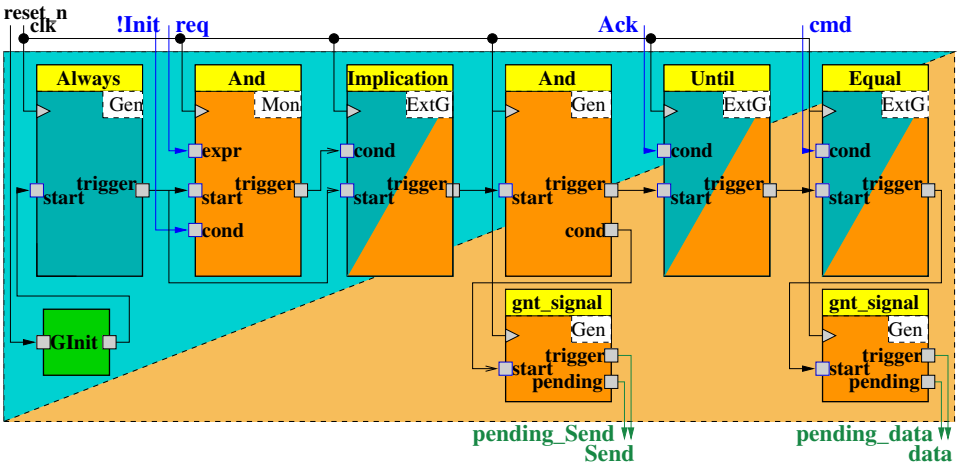

Fig. 9. Complex Extended Generator Architecture for F1

5.2 Building the Final Design

One extended-generator is produced for each property of the specification. The elaboration of the final design is achieved by connecting the duplicated signals to *Solvers*, one for each duplicated signal. As shown on Figure 10, properties F0 and F1 have one duplication of signal *Send*, which is input to component Solver$_{send}$. The correct final value for *Send* is produced as output.

A solver has two input ports *SIG* and *PENDING* (N bits each). *SIG* takes the N duplications sig(i) of sig. *PENDING* takes the corresponding N duplications $pending$(i) indicating if sig(i) is constrained. The resolved value for sig is provided on the output port *Val*.

Consider a consistent specification. If *PENDING* is all zeros, *SIG* is not constrained, and *Val* is set to 0 (or to a random value if mode RANDOM is active). Otherwise, *SIG* is constrained and all the constrained values are known to be identical. The first one is assigned to *Val*.

If the consistency of the specification has not been verified, an enhanced release of the Solvers is required. In this situation, it may happen that two duplications sig(i) and sig(j) take different values. The Solver has an extra output port *Err* used to dynamically notify this inconsistency.

Once the extended-generators have been interconnected to the Solvers, the resulting design is encapsulated into a top-level entity that takes as inputs signals Clk and $Reset\_n$ and all the observed external signals. The outputs are the generated external signals. Figure 10 gives the structure of the final design obtained for the specification Spec_cdt defined in section 4.

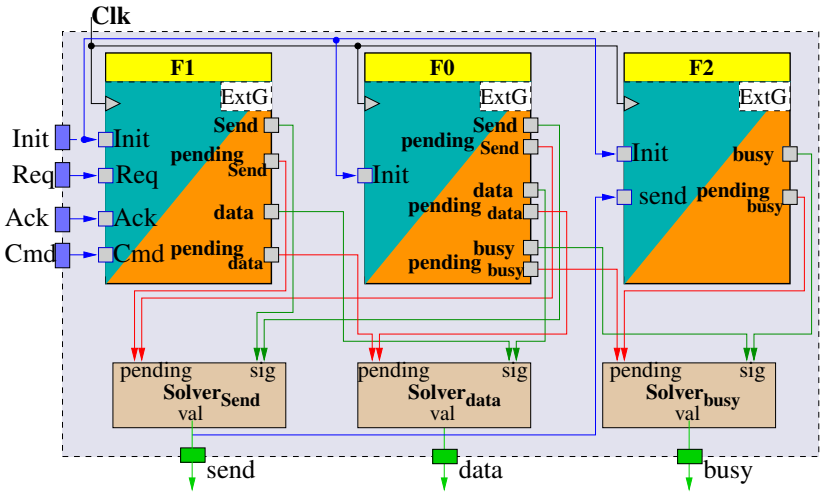

Fig. 10. Synthesized Spec_cdt

5.3 Verification of the Approach

The initial verification of the synthesized circuits has been performed by model-checking on a set of test cases: we used RuleBase Parallel Edition [HIL04] to check that all the properties listed in the specification hold on the circuit synthesized from them.

Then, the overall construction scheme has been proved correct using the PVS theorem prover following the method described in [MAB06].

6 Case Study: Wishbone crossbar controller

6.1 A Crossbar bus and the Wishbone specification

A crossbar switch is a bus-based architecture allowing to connect M masters to N slaves simultaneously. It enables parallel communications and enhances the whole system performances.

In this chapter, we illustrate the application of Horus on a Wishbone compliant crossbar [Her02]. The Wishbone specification defines a generic interface and a communication protocol for the insertion of predefined IPs in a design. This context appears particularly appropriate for demonstrating the use of Horus.

Figure 11 illustrates the generic interfaces of master and slave components. For each signal sig that connects a master to a slave (directly or through a switch), we denote M_sig_o the output port of the master and S_sig_i the input port of the slave connected to it, S_sig_o and M_sig_i in the other direction. For readability M^j (S^j) denotes the j-th master (or slave). In the actual PSL properties, the indices are macro-generated as constant values, and part of standard identifiers.

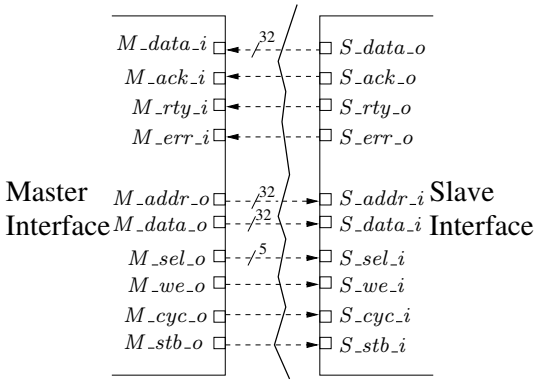

Fig. 11. Interfaces for Wishbone Master and Slave components

The communication between a master and a slave is based on a hand-shake protocol. To initiate the communication and hold the bus during the transfer, the master asserts a

signal *cyc* (cycle valid). To transfer data, it asserts the signal *stb* (strobe) and indicates the write/read operation with signal *we* (write enable). Both signals remain '1' until the slave sets either *ack*, *busy*, or *err* (transfer error).

In the last two cases, the master has to retry (*rty*). Signals *data* and *addr* represent the data being sent and the address where it must be read or written. The four most significant bits of *addr* represent the address of the slave. Since data is transferred through a 32-bit bus, signal *sel* indicates where the valid data is expected. This protocol is clock (*clk*) and reset (*reset*) synchonized.

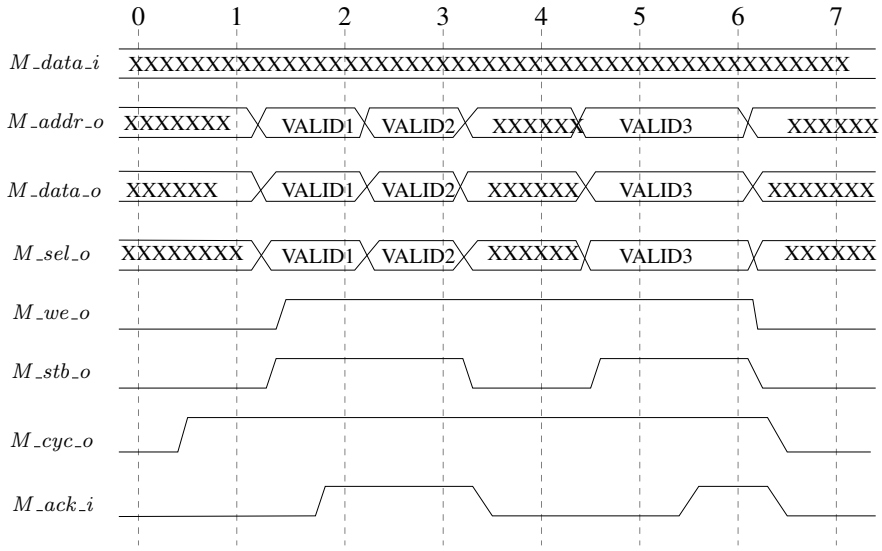

Fig. 12. Write Burst with the Wishbone Communication Protocol

Figure 12 illustrates a write burst. At cycle ♯1, the master asks for a transfer by asserting M_cyc_o that remains asserted until the end of the communication (at ♯7). At cycle ♯2, data is ready to be written (M_stb_o is asserted and M_we_o is set to '1'), ports M_data_o and M_addr_o are valid. The request is acknowledged at cycle ♯2 allowing a new request at cycle ♯3 which is acknowledged too. At cycle ♯4, the bus communication is maintained (M_cyc_o is '1') but no data is transferred: a wait state has been inserted by the master. At cycle ♯5, the master makes a new request, but the slave does not acknowledge it immediately: it has inserted a wait cycle. The request is acknowledged on the next cycle. At cycle ♯7, all control signals are negated.

For a read burst, the principle is the same, except that M_we_o keeps value '0' and M_data_i receives the read data.

Two more remarks will help understand the communication protocol. When a slave is being used, its S_cyc_i port is asserted: this signal is observed to test if it is busy. Furthermore, a controller ensures the exclusive interconnection between a master and a slave on a Wishbone crossbar switch.

The conmax-ip controller The conmax-ip controller [Uss02] allows the communications between up to 8 masters and up to 16 slaves on a crossbar switch with up to 4 levels of priority. The four most significant bits on M_addr_o address the slave. The selection of the master that will own a slave is based on two rules:

- **Priorities:** Each master has a priority that is stored in an internal register `CONF` of the controller. The master i priority is given by `CONF[2i..2i-1]`. At each cycle, the master with the greatest priority gets the slave.
- **Round-Robin:** Among masters of equal priority, a round-robin policy is applied.

6.2 Assertion-based Verification with Horus

The Horus flow The Horus software is based on two parts: the core implemented in C and the graphic interface implemented in Java. The core implements a PSL parser, generates the VHDL description of the property. It is approximately 10000 lines long.

The Horus environment helps the user build an instrumented design to ease debugging: it can synthesize monitors and generators, connect them to the design under test (DUT) and adds a device to snoop the signals of interest. It comes with the VHDL flavor. The Horus system has a friendly graphical user interface for the generation of the instrumented design in 4 steps.
Step 1 - Design selection: The DUT, with its hierarchy, is retrieved.
Step 2 - Generators and Monitors synthesis: Select properties or property files, define new properties, select target HDL language, synthesize monitor or generator (verification IP).
Step 3 - Signal interconnection: The user connects the monitors and the generators to the design. All the signals and variables involved in the DUT are accessible in a hierarchical way. The user needs only select the signals to be connected to each verification IP.
Step 4 - Generation: The design instrumented with the verification IPs is generated. When internal signals are monitored, the initial design is slightly modified to make these signals accessible to the monitors.

The outputs of the verification IPs are fed to an instance of a generic analyzer; this component stores the monitors outputs and sends a global status report on its serial outputs. It also incorporates counters for performance analysis.

The instrumented design has a generic interface defined for an Avalon or a Wishbone bus. If the FPGA platform is based on such a bus, the user can directly synthesize and prototype the instrumented design on it.

Monitors Many aspects of the conmax-ip have been verified with the Horus platform. A brief overview is given.

Property Reset_Mj For all masters, signals M_cyc_o and M_stb_o must be negated as long as *Reset* is asserted (*c.f.* [Her02], rule 3.20):

Property Reset_Mj:assert always$(Reset \rightarrow (\text{not } M^j_cyc_o \text{ and not } M^j_stb_o) \text{ until not } Reset)$;

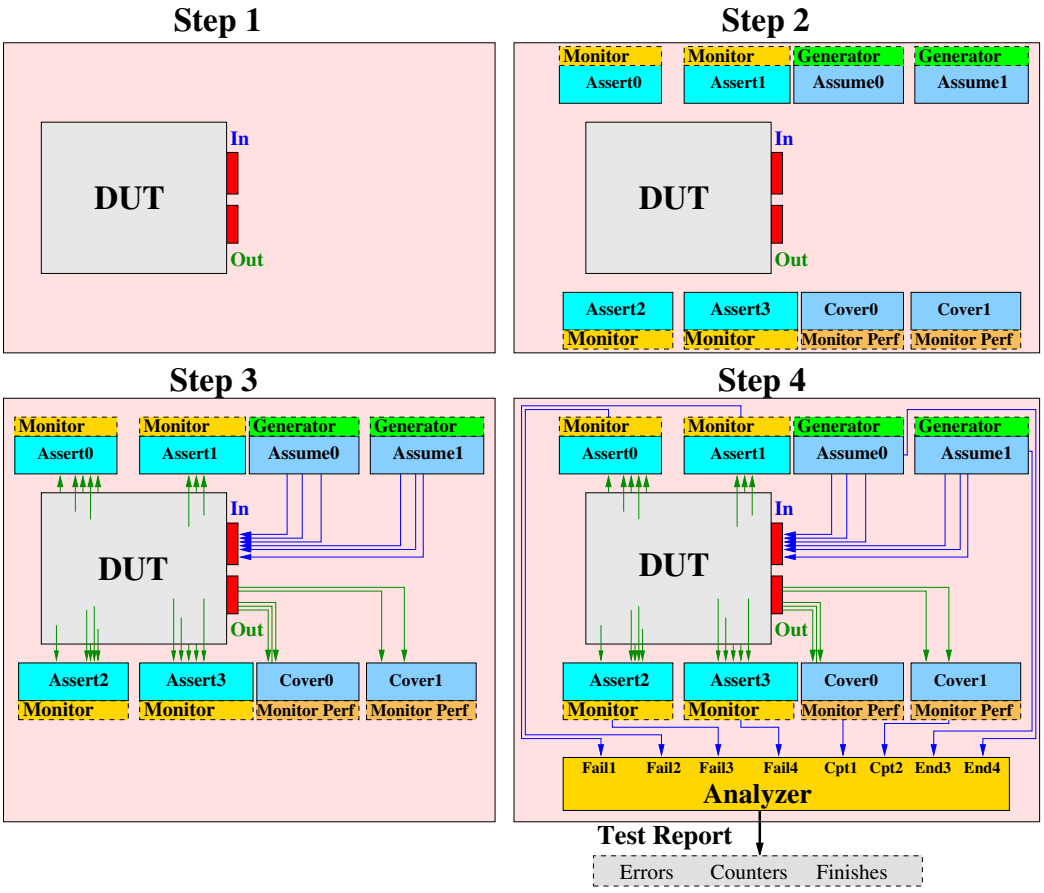

Fig. 13. Instrumentation of a design with Horus

Property PrioMj_Mk Assume two masters M^j and M^k have priorities p_j and p_k such that $p_k > p_j$. If M^j and M^k request the same slave simultaneously, M^k will own it first.

Property PrioMj_Mk:assert always($M^j\_cyc\_o$ and $M^k\_cyc\_o$
and $CONF[2k..2k-1] > CONF[2j..2j-1]$
and $M^j\_addr\_o[0..3] = M^k\_addr\_o[0..3]$)
$\rightarrow$ ($M^k\_ack\_i$ before $M^j\_ack\_i$);

Property LinkMj_Sk It checks the connection between the j-th master and k-th slave by analyzing that each port is well connected.

Property LinkMj_Sk:assert always($M^j\_cyc\_o$ and $S^k\_cyc\_i$ and $M^j\_addr\_o = S^k\_addr\_i$)
$\rightarrow$ ($M^j\_data\_o = S^k\_data\_i$ and $M^j\_data\_i = S^k\_data\_o$
and $M^j\_sel\_o = S^k\_sel\_i$ and $M^j\_stb\_o = S^k\_stb\_i$
and $M^j\_we\_o = S^k\_we\_i$ and $M^j\_ack\_i = S^k\_ack\_o$
and $M^j\_err\_i = S^k\_err\_o$ and $M^j\_rty\_i = S^k\_rty\_o$);

Modelling the environment of the conmax-ip To test the correctness of the conmax-ip controller in isolation, without the overhead of simulating a complete set of masters and slaves, we need to embed the controller in an environment that provides correct test signals. To this aim, we model masters and slaves with generators that must comply with the hand-shake protocol.

Property WriteMj_Sk A write request from the j-th master to the k-th slave is specified by the following property, to which a generator is associated:

Property WriteMj_Sk:assume(
($M^j\_cyc\_o$ and $M^j\_we\_o$ and $M^j\_sel\_o$ and $M^j_stb_o$and
$M^j\_data\_o =$ VAL_DATA and $M^j\_addr\_o =$ VAL_ADDR_k)
until_ $M^j\_ack\_i$)

Since we are interested in the communication action, but not in the particular data value being written, the value VAL_DATA that is displayed on the port $M^i\_data\_o$ is a randomly computed constant. The four most significant bits of VAL_ADDR are fixed to select the j-th slave. This property is a simplified model of a master, it does not take into account signals $M^i\_rty\_i$ and $M^i\_err\_i$ (they are not mandatory). These signals would be present in a more realistic model. This property involves the acknowledgment input signal $M^i\_ack\_i$ that stops the constrained generation.

Property GenLaunch The scenario GenLaunch illustrates the request of three masters numbered 0, 1, 2 to the same slave numbered 1. Master 0 first makes a request; then between 16 to 32 cycles later, masters 1 and 2 simultaneously make their request. This scenario is modeled using a property that generates the start signals for three instances

of master generators (according to the previously discussed property), and one slave. These different start signals are denoted $start\_WriteM0\_S1$, $start\_WriteM1\_S1$, $start\_WriteM2\_S1$.

assume eventually! ($start\_WriteM0\_S1$
$\rightarrow$ next_e[16..32]($start\_WriteM1\_S2$ and $start\_WriteM2\_S2$));

A large number of scenarios of various complexity have been written, and implemented with generators, in order to have a realistic self-directed test environment. Modeling test scenarios for requests (i.e. read, burst, ...) is also performed with assumed properties, from which generators are produced.

For a slave, the most elaborate action is the response to a read request: signal S_ack_o is raised and the data is displayed on S_data_o. The following property expresses this behavior, at some initial (triggering) time.

assume next_e[1..8]($S^j\_ack\_o$ and $S^j\_data\_o = DATA$));

The generator for property `Read_Sj` must be triggered each time the slave receives a read request: its start signal is connected to the VHDL expression not $S^j\_we\_i$ and $S^j\_cyc\_i$.

Performance analysis Monitors can be used to perform measurements on the behavior of the system. To this aim, the Horus platform is instrumented to analyze the monitor outputs, and count the number of times when a monitor has been triggered, and the number of times when a failure has been found.

On the wishbone switch, and assuming that it is embedded in a real environment, it may be useful to test on line the number of times the signal M_err_i of a slave is asserted, or how often a slave is requested simultaneously by several masters.

Property CountError The following property is used to count the number of transfers ending with an error:

Property CountError: assert never ($M^0\_err\_i$ or ... or $M^7\_err\_i$);

Property ColliMj_Sk The property allows to know the number of times when more than one master asks for the same slave:

Property ColliMj_Sk: assert never ($S^j\_cyc\_i$ and $S^k\_cyc\_i$ and $S^j\_addr\_i = S^k\_addr\_i$);

6.3 Assertion-based Synthesis with SyntHorus

SYNTHORUS The SyntHorus tool aims at providing an environment to support automatic synthesis of HDL descriptions from PSL specifications, within the framework described here. It represents 5000 lines of C source code and has been tested on a laptop equipped with a dual core processor and 2Go of RAM. The SyntHorus process flow is depicted Figure 14.

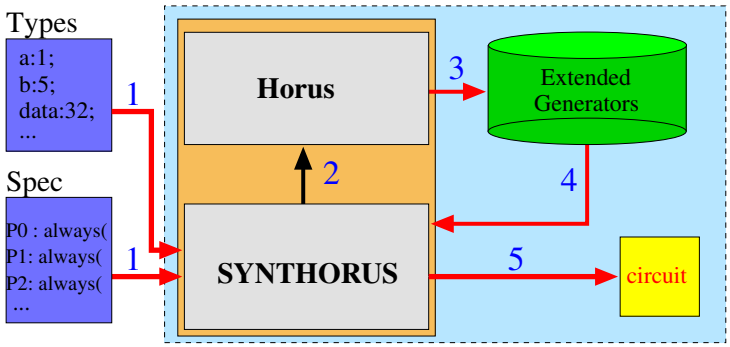

Fig. 14. The SyntHorus Tool

The idea consists in specifying control parts with PSL and synthesizing them with SyntHorus to guarantee their functionnality, while operative parts must be validated with other suitable methods.

SyntHorus takes as inputs the annotated specification file and a description for all the signals involved in the specification (input, output, internal and type). It is interfaced with HORUS, which produces the extended-generators. Afterwards SyntHorus interconnects all these components, adding Solvers if necessary. Finally it encapsulates the result in a global enclosing component that constitutes synthesized design.

We fed SyntHorus with various complex specifications containing hundreds of properties. For 700 properties, it produces an HDL description of 2,7 Mb within 16 seconds.

Synthesis Results All syntheses have been done with Quartus7.0 for a CycloneII EP2C35F672C6 FPGA-based platform. The synthesis optimization option "balanced" has been used.

CDT Synthesis The annotation was complete for the Spec_cdt specification.

Table 2 gives the synthesis results for the CDT built by SyntHorus. The original hand-coded design has 18 LCs, 6 FFs and a maximum frequency of 420Mhz. The design obtained from SyntHorus is slightly bigger than the original one.

In the following examples, we shall see that the efficiency of a SyntHorus design is greatly affected by the number of embedded solvers.

GenBuf Controller Our first complex case study is the GenBuf controller described in [BGJ+07]. GenBuf can connect N data senders to 2 data receivers.

Communications between senders and receivers use the handshake protocol. A round-robin algorithm schedules the data transfers through the controller. The annotated specification is not given here for space reasons, but can be found on our web page [Odd09].

Figure 15 shows the area used by different instantiations of GenBuf containing a number of slaves ranging from 1 up to 10, for the method of [BGJ+07] (higher fast

Table 2. CDT produced by SyntHorus

SyntHorus	LCs	FFs	Freq. (Mhz)
$F0_{cdt}$	1	0	-
$F1_{cdt}$	3	2	-
$F2_{cdt}$	6	6	-
Solvers	11	0	-
CDT	**21**	**8**	**420,17**

growing curve) and our tool SyntHorus (bottom two curves). The area of the circuit produced by our approach grows linearly with the number of senders (actually with the number of properties in the specification). In contrast, the best automata-based method, that hardcode the enumeration of the correct behaviors of the circuits, has a polynomial complexity on $0(N^3)$ [BGJ+07].

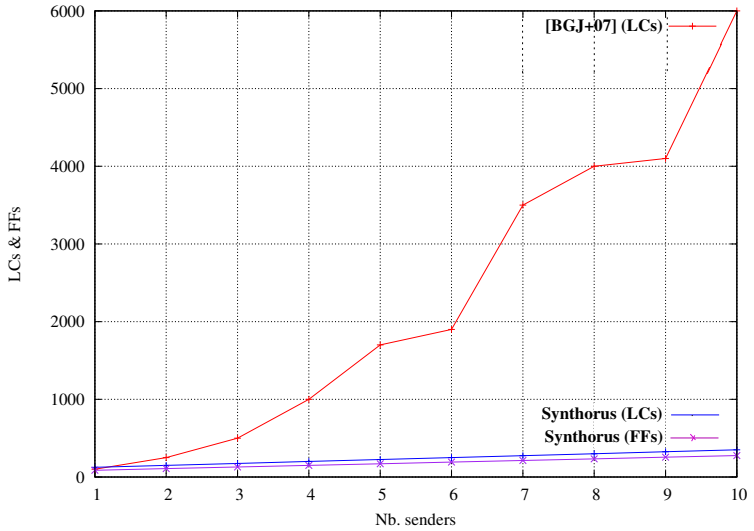

Fig. 15. GenBuf Synthesis Results : Method [BGJ+07]/SYNTHORUS Method

In addition to producing efficient circuits, the construction time is also very small: a dozen seconds to build GenBuf for 60 senders, compared to hours reported in [BGJ+07]. Again, this is explained by the fact that no enumerative state space traversal is involved in our construction. For SyntHorus, frequency results are good.

CONMAX-IP The conmax-ip annotated specification represents dozens of complex temporal properties. It can be found on the following web page [Odd09].

The following property Master1ToSlave2 is an example of the used properties:

Master1ToSlave2: assert always((
($M^1\_cyc\_i_m$ and $M^1\_addr\_i_m$="0010") and
(not($M^0\_cyc\_i_m$) or $M^0\_addr\_i_m$/="0010") and
not($S^2\_cyc\_o_m$))⇒ next[1](($S^2\_addr\_o_g$=$M^1\_addr\_i_m$ and
($S^2\_data\_o_g$=$M^1_data_i_m$and $S^2\_cyc\_o_g$=$M^1\_cyc\_i_m$)and
($S^2\_sel\_o_g$=$M^1\_sel\_i_m$ and $S^2\_we\_o_g$=$M^1\_we\_i_m$) and
($S^2\_stb\_o_g$=$M^1\_stb\_i_m$ and $M^1\_data\_o_g$=$S^2\_data\_i_m$) and
($M^1\_ack\_o_g$=$S^2\_ack\_i_m$ and $M^1\_rty\_o_g$=$S^2\_rty\_i_m$ and $M^1\_err\_o_g$=$S^2\_err\_i_m$)
until ($M^1\_ack\_o_m$ or $M^1\_rty\_o_m$ or $M^1\_err\_o_c$)));

This property models the connection between the first master and the second slave when no other masters are trying to access the second slave. The connection is performed by connecting each port of the master to the corresponding port of the slave interface. The connection is maintained until the transfer is ended by the reception of one of the three following signals: $M^1\_ack\_o_m$ (transfer succeeded) or $M^1\_rty\_o_m$ (transfer aborted because slave was busy) or $M^1\_err\_o_c$ (transfer error).

Figure 16 compares the synthesis results obtained for the conmax-ip originally designed by hand and by SyntHorus. Results are given for 4 masters using 2 priority levels, connected to a number of slaves varying between 1 and 16.

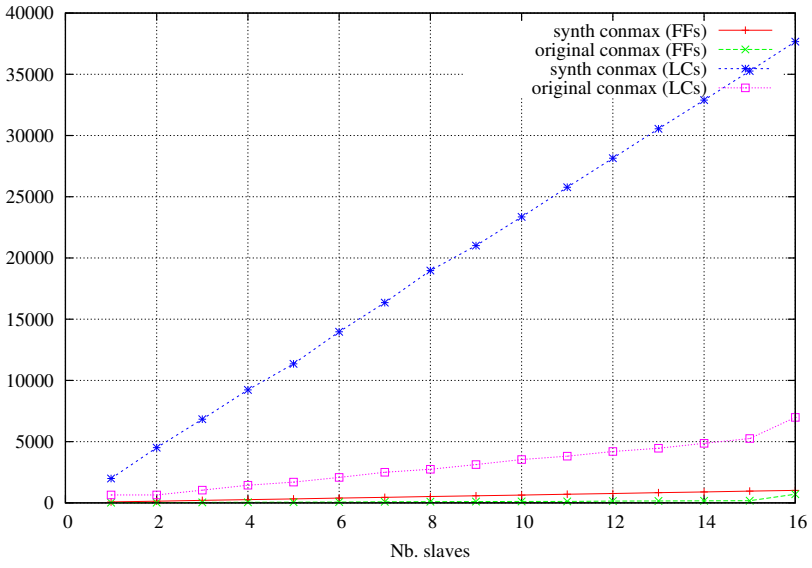

Fig. 16. Synthesis Results: Original/SyntHorus Conmax-ip

LCs overhead is clearly visible and remains linear when the number of slaves increases. The main part of this overhead is due to the use of the Solvers. Notice that the FFs overhead is noticeably less important.

If the specification contains just a few duplicated signals, then the resulting design is efficient and its complexity is slightly higher than the same hand-coded design. If there are lots of duplicated signals a significant overhead is introduced by SyntHorus. The designer should consider rewriting the specification to minimize the number of duplications.

7 Conclusion

A method to efficiently synthesize a test-bench from temporal properties has been presented. On one hand, the stimuli generation is carried out by hardware components synthesized from assumptions: the generators. On the other hand, monitors support the verification of the DUT behavior by on-line verifying that it complies with the corresponding assertions.

By annotating PSL signals and developing a new kind of hardware component called extended-generator, we went beyond Assertion-Based Verification to produce a correct-by-construction module (at RTL) from its temporal specification. This "Assertion-Based Synthesis" starts from a more abstract, formal and declarative specification than the conventional "High-level Synthesis" that takes as inputs an algorithmic specification. Assertion-Based Synthesis is best suited for control circuits.

The modularity of the approach encapsulates the high complexity of the specification into a hierarchy of verified components. The formal proof of the whole approach guarantees that the final circuit is correct by construction.

SyntHorus is a prototype tool that implements our Assertion-Based Synthesis method. It takes as input PSL specifications and produces a RTL design in VHDL. To the best of our knowledge, ours is the first approach with a linear complexity, able to process the full simple subset of PSL. Compared to the state of the art previous approaches, the resulting designs are very efficient (*i.e* small and fast). Despite the fact that they are in general less efficient than the same hand coded designs, the automatically-synthesized designs can still be used as a golden model. Our method has a distinct advantage: it can be used to produce a reference model, correct by construction, from the very first logic and temporal specifications. More efficient hand-coded designs can then be proved correct, by conventional equivalence checking with the reference produced by SyntHorus. Otherwise, the design produced by SyntHorus can be directly taped-out.

Currently, annotating the specification is not fully automated. While it is obvious to annotate the input ports with "m", output ports may occur as "m" and "g" in different properties (e.g. signal *Send* in Spec_cdt). More work remains to be done to complement the set of annotation rules we developed so far.

References

[ABBSV00] A. Aziz, F. Balarin, R-K. Brayton, and A-L. Sangiovanni-Vincentelli. Sequential synthesis using S1S. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(10):1149–1162, 2000.

[ABC+] T. Anderson, J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. Systemverilog reference verification methodology: Introduction. *EE Times*, 27/3/2006.

[BCE+04] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and assurance tool (deliverable 1.2/4-5). Technical report, PROSYD Project, Jan. 2004.

[BCZ06] M. Boulé, J-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proceedings of the 24th International Conference on Computer Design: ICCD'06*, Oct 2006.

[BGJ+07] R. Bloem, S. Galler, B. Jobstman, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run : Hardware from PSL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190, 2007.

[Cal05] J.R. Calamé. Specification-based test generation with TGV. Technical Report R0508, Centrum voor Wiskunde en Informatica, May 2005.

[CRST06] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From PSL to NBA: a Modular Symbolic Encoding. In *IEEE Formal Methods for Computer Aided Design FM-CAD'06, November 11-12, 2006, Proceedings*, pages 125–133, 2006.

[CVK04] B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, 2004.

[DGV99] M. Daniele, F. Giunchiglia, and M. Vardi. Improved Automata Generation for Linear Temporal Logic. In *Proc. CAV'99*, volume 1633. LNCS, Springer, July 1999.

[FKL03] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, Jun. 2003.

[FU82] R-W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29(3):603–622, 1982.

[FWMG05] H. Foster, Y. Wolfshal, E. Marschner, and IEEE 1850 Work Group. *IEEE standard for property specification language PSL*. pub-IEEE-STD, pub-IEEE-STD:adr, Oct 2005.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. CAV 2001*, volume 2102. LNCS, Springer, July 2001.

[Her02] R. Herveille. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores. Technical report, http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf, Sept 2002.

[HIL04] Haifa-IBM-Laboratories. *RuleBase Parallel Edition*. IBM, Nov. 2004.

[IBM] IBM. PSL/Sugar-based Verification Tools. Web page. http://www.haifa.il.ibm.com/projects/verification/sugar/tools.html.

[MAB06] K. Morin-Allory and D. Borrione. Proven correct monitors from PSL specifications. In *DATE 2006*, Jan. 2006.

[MAB07] K. Morin-Allory and D. Borrione. On-line monitoring of properties built on regular expressions sequences. In *Applications of Specification and Design Languages for SoCs*. A. Vachoux editor, Springer, 2007.

[Öbe99] J. Öberg. *ProGram : A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols*. PhD thesis, Royal Institue of Technologoy - Department of Electronics, Eletronic System Design, Sweden, 1999.

[Odd09] Y. Oddos. PSL Specification for the WISHBONE Interconnect Matrix IP Core: http://tima.imag.fr/vds/horus/synthorus_specs/, 2009.

[OMAB06] Y. Oddos, K. Morin-Allory, and D. Borrione. On-line test vector generation from temporal constraints written in PSL. In *Proc. VLSI SoC'06*, 2006.

[PRO] PROSYD. Tools and techniques for property verification. Web page. http://www.prosyd.org/twiki/view/Public/DeliverablePageWP3.

[SB94] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE Trans. on VLSI*, pages 172–185, Jun 1994.

[SM02] R. Siegmund and D. Müller. Automatic synthesis of communication controller hardware from protocol specifications. *IEEE Design & Test of Computers*, 19(4):84–95, 2002.

[SMB+05] J. Srouji, S. Mehta, D. Brophy, K. Pieper, S. Sutherland, and IEEE 1800 Work Group. *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language*. pub-IEEE-STD, pub-IEEE-STD:adr, Nov 2005.

[SNBE07] M. Schickel, V. Nimbler, M. Braun, and H. Eveking. *Advances in Design and Specification Languages for Embedded Systems*, chapter An Efficient Synthesis Method for Property-Based Design in Formal Verification: On Consistency and Completeness of Property-Sets, pages 179–196. Number 978-1-4020-6149-3 (Online). Springer Netherlands, Jul. 2007.

[SOSE08] M. Schickel, M. Oberkönig, M. Schweikert, and H. Eveking. A case-study in property-based synthesis: Generating a cache controller from property-set. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, pages 271–275. Springer Netherlands, 2008.

[ST03] R. Sebastiani and S. Tonetta. More Deterministic vs Smaller Büchi Automata for Efficient LTL Model Checking. In *Proc. CHARME 2003*, pages 126–140. Springer, Oct 2003.

[Uss02] R. Usselman. WISHBONE Interconnect Matrix IP Core, 2002. http://www.opencores.org/projects.cgi/web/wb_conmax/overview.

[YJC04] C. Yen, J. Jou, and K. Chen. A divide-and-conquer-based algorithm for automatic simulation vector generation. *IEEE Design & Test of Computers*, 21(2):111–120, 2004.