Observable Time Windows: Verifying the Results of High-Level Synthesis

R. A. Bergamaschi § S. Raje †
§ IBM T. J. Watson Research Center, NY, USA
† Northwestern University, IL, USA

Abstract

One of the main problems in high-level synthesis has been the lack of verification techniques for checking the equivalence between the behavioral specification and the scheduled implementation. Due to scheduling it may not be possible to compare simulation results before and after high-level synthesis using the same simulation drivers. Given that simulation is the most time consuming step in the design process, this severely reduces the advantages of high-level synthesis. This paper presents techniques and algorithms for comparing simulation results using the same simulation drivers. The approach is based on creating special hardware structures in the implementation and comparing the simulations only at synchronization points called observable time windows.

1 Introduction

Verification is an integral part of any design methodology. It can assume various forms from manual design checks, to simulation-based checks, to formal verification. Despite recent advances in formal verification [1], simulation remains the dominant verification mechanism in all designs. The task of verifying an implementation against a specification depends primarily on the abstraction levels of both, as well as on the notion of equivalence.

In the finite-state machine, register-transfer and gate-level domains there are well-known verification techniques. The concept of equivalence is well defined and it can be informally stated as: the values of the outputs in the specification and implementation machines must match at every clock cycle when subject to the same set of inputs (of finite length, after an initialization period). Simulation-based verification can be done by simply comparing the simulation results of the specification and the implementation, using the same test vectors. In a synthesis-based environment, this is equivalent to simulating the descriptions before (specification) and after synthesis (implementation). Several formal verification techniques exist which can symbolically prove the equivalence of two descriptions at these abstraction levels [1].

In the behavioral or high-level domain there are no well established and practical verification techniques. One of the main problems to be overcome is the change in the cycle-by-cycle behavior allowed by high-level synthesis [2], specifically the scheduling task. If scheduling is performed as part of generating an implementation, it may change the behavior such that sequences of operations that execute in one clock cycle in the specification, may need several clock cycles to be completed in the implementation. At this abstraction level, the issue is how to define equivalence between two machines that may not have the same cycle-by-cycle behavior.

A vague definition that has been adopted in high-level synthesis is that: “specification and implementation are equivalent if both eventually (independently of number of cycles) produce the same result”. This may be acceptable in simple digital-signal processing applications where computations may be finished any number of cycles (within bounds) after the inputs are sampled, as long as throughput is preserved. However, this definition is unacceptable in more complex applications, where inputs and outputs are sampled and produced at intermediate cycles – a change in the cycle-by-cycle behavior in this case may render the implementation functionally incorrect.

A main problem of practical importance is the reuse of simulation drivers and vectors. A great deal of design time is spent generating simulation vectors and simulating the design specification prior to synthesis. Normally, the design implementation is re-simulated (after synthesis) using the same vectors. If the cycle-by-cycle behavior of the design is changed by high-level synthesis, it may be necessary to rewrite the simulation vectors. This would make it impossible to compare automatically the simulation results before and after synthesis, which would be unacceptable in an efficient high-level design methodology.

The Satya System [3] was one of the first to face this problem. Its approach is based on comparing the simulation of the design after synthesis with the simulation of the specification annotated with scheduling information. The limitation of this approach is that verification is at the level of the scheduled specification and not at the original behavioral level. The HIS System [4] proposed a methodology and algorithms to give the user control over the scheduling with the goal of easing the verification burden. Some of these ideas have recently been incorporated into a commercial tool [5]. However, all previous approaches may require rewriting the simulation vectors.

This paper presents an approach for comparing simulation results of design specification against design implementation using the same simulation vectors,
where the implementation is generated by a high-level synthesis system performing scheduling (thus, possibly changing the cycle-by-cycle behavior). The approach consists first of defining a notion of equivalence which is compatible with the simulation behavior of the specification. Secondly, the synthesis system has to generate the necessary hardware in the implementation and a special synchronization signal. Thirdly, the two simulations (before and after synthesis) are compared automatically at synchronization points.

These synchronization points are called observable time windows (OTWs). The concept of OTWs was briefly introduced in [4]. This paper presents a complete formulation of the approach including the necessary hardware support that must be automatically generated by high-level synthesis. This work has been implemented in the HIS system [6].

2 Equivalence in High-Level Synthesis

Verifying the equivalence between two designs usually means that there is a golden model of the design (specification) which is simulated and analyzed for functional correctness. In most ASIC design methodologies, this golden model is a hardware description which can be simulated. The hardware-description language (HDL) possesses implied semantics which must be enforced by the simulator. The simulation of the synthesized design (implementation) is compared with the golden model. In practice, functional correctness of a design is based on its simulation behavior which must conform to the simulation semantics of the HDL. This is especially true for current design languages, VHDL and Verilog, which lack formal semantics. Therefore, equivalence can be defined in terms of the simulation semantics of the HDL. This work will concentrate on the semantics of VHDL [7], however, it can be easily extended to Verilog.

2.1 Specification States vs. Implementation States

The abstraction level that is of interest in this work is the synchronous, sequential, behavioral level, which in VHDL corresponds to Processes with Wait statements (as opposed to sensitivity lists). VHDL simulation semantics state that a sequential process will execute until it encounters a condition suspending its execution, upon which it will wait until the condition for suspension becomes false and then resume execution.

Given that this work deals with synchronous systems, specifications must obey the following restrictions: 1) signals are sampled on the clock edge (assuming an edge-triggered system); 2) signals are updated on the clock edge (possibly delayed by transition times); and 3) conditions for suspension and resumption are evaluated on the clock edge. Note that signals and variables have different semantics in VHDL. These restrictions fall into the class of designs described using VHDL processes with wait until statements, where the wait condition includes a clock-edge-triggered expression. Such a wait statement assumes the general form: \textit{Wait Until Not clock Stable And clock=1/0 And <expression>;} and it suspends the process execution until the following clock transition where <expression> is true. If <expression> is not present, the execution resumes on the next clock transition edge. Such a wait statement will be hereafter referred to as “wait until clock” for conciseness. Note that although these restrictions limit the classes of finite-state machines (FSM) that can be modeled with one process, general FSMs can still be modeled using multiple processes plus combinational logic.

A Specification State (SSt) is defined as the sequence of operations (statements) between points of suspension and resumption in the specification process. Transitions between specification states happen upon resumption of execution (of simulation) which is always on the clock transition. Therefore, signals are sampled and updated also on transitions between specification states. Hence, when simulating the golden specification model, correctness is based on the signal values observed on the transition points between specification states.

An Implementation State (ISt) is defined as the sequence of operations executed in one clock cycle in the implementation machine. The scheduling task in high-level synthesis maps operations in the specification to control steps (cycles) in the implementation. Hence, for each specification state there may be one or more implementation states, which may change the cycle-by-cycle behavior of the design.

A State Transition (Tr) between two states Sti and Stj under condition Ci occurs when the machine (implementation or specification) is in state Sti and condition Ci is true on the triggering edge of the clock. Such a transition is represented as \(\{Tr_i, Ci\}\) and its Boolean function can be written as: \(Tr_i = \{State \equiv St_i\} & Ci\).

For each transition STr between two specification states SSti and SSTj there is at least one corresponding transition \(\{ITr_i, Ci\}\) between two implementation states ISTi and ISTj, under condition Ci. STr and ITr denote transitions in the specification and in the implementation respectively.

The following mapping functions need to be defined between specification and implementation:

1) \(SSt_{to}_{IST}(SSt_i) = \{IST_{t1}, IST_{t2}, ..., IST_{tn}\}\) – this function returns the set of implementation states within one specification state.

2) \(first_{IST\ in\ SSt}(SSt_i) = IST_{t1}\) – this function returns the first implementation state scheduled in a specification state.

3) \(last_{IST\ in\ SSt}(SSt_i) = \{IST_{tp}, Cip\}, IST_{tp1}, Cip1, IST_{tp2}, Cip2, ..., IST_{tpn}, Cipn\}\) – this function returns the set of implementation state and condition pairs which represent implementation transitions that have a corresponding specification transition from state SSti.

4) \(STr_{to}_{ITr}(STr_i) = \{ITr_{t1}, ITr_{t2}, ..., ITr_{tn}\}\) – this function returns a set of transitions in the implementation corresponding to a given transition in the specification.

By construction, the scheduling algorithm guarantees that a transition between SST's (due to “wait until clock” statements) always has at least one corresponding transition between IST's. Scheduling can also be seen as a task that inserts extra “wait
until clock” statements in between the original “wait” statements in the specification.

5) \( ITr \cup \overline{ST} \equiv ST \) or 0 – this function returns a transition in the specification corresponding to a given transition in the implementation, if any, or 0 if \( ITr \not\equiv ST \).

The concepts introduced in this section are summarized in Figure 1. Figure 1a shows a pseudo-VHDL specification with the corresponding specification states and transitions; Figure 1b gives one possible schedule and the corresponding implementation states and transitions. Figure 1c presents the mapping relationships between specification and implementation. Note that in the implementation there are extra cycles (states) introduced by scheduling between statements (operations) \( stmt1/2, stmt5/6, \) and \( stmt8/9 \). In this context, equivalence can be defined as:

**Definition 1:**

An implementation (after scheduling) is equivalent to a specification iff the values of all signals in both descriptions are the same when observed at clock edges corresponding to transition points between specification states. That is, the values of specification signals at transition \( ST \), must match the values of corresponding implementation signals at transitions \( ITr \cup \overline{ST} \equiv ST \), for all specification transitions and all signals.

### 2.2 Handshaking Schemes for Equivalence

Equivalence, as defined above, is only possible if the design interfaces with the environment under certain handshaking schemes. Clearly if inputs change at a rate faster than the implementation can finish processing one set of inputs then input values may be discarded which invalidates the equivalence definition. Therefore, certain restrictions must be imposed on the way in which the design interfaces with the environment; the basic premise is that inputs and/or outputs have to be updated according to request and acknowledge signals. The environment can represent the test bench or other designs. Three basic types of handshaking, supporting the definition of equivalence, are described below. Several variations of these types also work.

**Type 1:** Two-way handshake with request / acknowledge signals. The environment provides an input ready signal upon which the design starts processing. When the processing is finished, the design provides an output ready signal which prompts the environment to provide a set of inputs and acknowledge by resetting an output request signal. Only then can input ready be asserted again. Figure 2a shows a VHDL example with this type of handshake.

**Type 2:** One-way handshake controlled by the design. The design reads the inputs from the environment and starts processing. When finished, the design provides an input request signal which prompts the environment to provide a new set of inputs. Figure 3a shows a VHDL example with this type of handshake.

**Type 3:** One-way handshake controlled by the environment. The environment provides an input ready signal upon which the design starts processing; and

---

**Figure 2:** Divider Circuit: a) VHDL (Type 1), b) implementation fsm, c) OTW function.

The environment provides an output request signal upon which the design makes the outputs available. In this scheme, the environment must guarantee that the minimum number of cycles between an input ready and an output request is greater than or equal to the maximum number of extra states introduced by the scheduler between any two consecutive specification states.

Figure 2a contains an example of a VHDL description of Type 1 which implements a simple integer division algorithm. It has two “wait until clock” statements, hence two specification states. Figure 2b shows the implementation state machine generated by HIS (under no resource constraints).

### 3 Observable Time Windows

Observable Time Windows (OTWs) correspond to the instants when the simulation of the implementation can be directly compared with the simulation of the specification. According to Definition 1 of equivalence, these instants correspond to the transition points between specification states.

**Definition 2:**

An Observable Time Windows is a Boolean function in the implementation which describes the union of all transition functions in the implementation which have a corresponding transition in the specification:

\[
OTW = \bigvee ITr \cup \forall ITr \in \{ ITr : \exists ST \land \exists ST \rightarrow ITr(ST) \}
\]

\[
ITr \in ITr \cup \overline{ST} \equiv ST \land ST \rightarrow ITr(ST)
\]

\[
\forall (ISt, Cond) \in \{ (ISt, Cond) : \exists ITr = ISt \land Cond \land \exists SSST \in SSST (ISt) \}
\]
3.1 Generation of OTW Function in HIS

The OTW function is generated as an internal signal in the implementation during high-level synthesis (called OTW signal). This signal is used during simulation to synchronize the simulation outputs of implementation and specification. Details on how the simulation outputs are compared are given in Section 5. The OTW signal has no functional purpose in the implementation and is deleted during final logic synthesis.

In order to generate the OTW signal, high-level synthesis has to implement the expression in Definition 2. The OTWs correspond to cycles when the specification process moves from suspension to resumption, which happens on the “wait until clock” statements. Therefore by keeping track of the states in which these statements are scheduled and the conditions leading to them, it is possible to generate a logic network representing the expression in Definition 2.

Figure 2c gives the OTW function for the implementation FSM in Figure 2b.

4 Hardware Support for OTWs and Equivalence

Definition 1 of equivalence is applicable to specifications which follow the restrictions in Section 2.1. The specification can be made to follow such restrictions by coding the VHDL using processes with “wait until clock” statements. In the implementation, however, it is the task of the high-level synthesis system to generate hardware which obeys these restrictions.

In the specification, restrictions 1) and 2) require signals to be sampled and updated on the triggering edge of the clock, which corresponds to the transition points between specification states. When applied to the implementation, these restrictions imply that signals should be sampled and updated on the state transitions in the implementation which have a corresponding transition in the specification. In order to enforce these restrictions on the implementation, extra hardware structures may have to be created (by high-level synthesis) for storing intermediate values of inputs and outputs.

Consider the VHDL example in Figure 3a (of type 2 handshake) which computes the averages of two and four numbers and suppose that high-level synthesis schedules it under the constraint of using a single adder. The corresponding schedule requires three states as shown in Figure 3b. According to this schedule, outputs avg2 and avg4 are assigned in states 0 and 2 respectively, while inputs \( \{in1, in2\} \) and \( \{in3, in4\} \) are used in states 0 and 1 respectively.

The specification in Figure 3a contains a single state \( SS_{lo} \) and a single state transition \( ST_{r0} \) from and to \( SS_{lo} \). The implementation contains three states and three transitions, such that:

\[
SS_{lo} \rightarrow SS_{lo} = \{ISt_0, ISt_1, ISt_2\} \\
first_{ISt_{lo}} \rightarrow SS_{lo} = ISt_0 \\
last_{ISt_{lo}} \rightarrow SS_{lo} = \{ISt_1, 1\} \\
ST_{r0} \rightarrow ST_{r0} = \{ITr_k\}
\]

The input signals in the specification are all sampled...
in state \textit{SSt}_0, however, in the implementation they are sampled (used) in states \textit{IS}t_0 and \textit{IS}t_1. Since there is no guarantee that the inputs should remain constant for more than one cycle, it may be necessary to store the values of all inputs which are used in implementation states that are not the \textit{first} states scheduled in a specification state. Hence, the values of inputs \textit{in}_3 and \textit{in}_4 during state \textit{IS}t_0 are stored in registers (at the end of this state) and these stored values are used as \textit{in}_3 and \textit{in}_4 in state \textit{IS}t_1.

The output signals in the specification are all updated at the end of state \textit{SS}t_0; however, in the implementation output \textit{avg}_2 is assigned in state \textit{IS}t_0 while output \textit{avg}_4 is assigned in state \textit{IS}t_2. In the implementation, \textit{avg}_4 is updated at the end of state \textit{IS}t_2 which does correspond to a transition in the specification. However, output \textit{avg}_2 is assigned in state \textit{IS}t_0 which does not correspond to a specification transition. Hence, to enforce the restrictions (of Section 2.1) on the implementation, \textit{avg}_2 should be updated only at the end of \textit{IS}t_2. This requires the value being assigned to \textit{avg}_2 in state \textit{IS}t_0 to be stored in a temporary register (at the end of \textit{IS}t_0) and then transferred to output \textit{avg}_2 at the end of \textit{IS}t_2.

Two techniques have been developed to generate the necessary hardware for OTW support within the HIS system. One is based on Boolean analysis and the other on graph manipulations.

\subsection{4.1 Boolean Analysis Method}

The extra storage needed for inputs and outputs can be fully determined based on the Boolean conditions under which inputs and outputs are used and assigned in the implementation. Four types of Boolean conditions need to be defined:

1. \textit{State}\textsubscript{Read} = \bigvee (\textit{IS}t \equiv \textit{IS}t_0), \forall \textit{IS}t \in \{\textit{IS}t_0 \cup \textit{SS}t \cap \textit{IS}t \cup \textit{IS}t_2\}

2. \textit{State}\textsubscript{Write} = \textit{OTW} (see Definition 2)

3. \textit{Signal}\textsubscript{Read} = \bigvee (\textit{IS}t \equiv \textit{IS}t_0) \& \textit{Cond}, where \textit{IS}t is a state containing an operation using the signal under condition \textit{Cond}.

4. \textit{Signal}\textsubscript{Write} = \bigvee (\textit{IS}t \equiv \textit{IS}t_0) \& \textit{Cond}, where \textit{IS}t is a state containing an operation assigning to the signal under condition \textit{Cond}.

\textit{State}\textsubscript{Read} represents the conditions under which signals should be sampled. \textit{State}\textsubscript{Write} represents the conditions under which signals should be updated. \textit{Signal}\textsubscript{Read} represents the condition under which a signal is used (as input to an operation) in the implementation. \textit{Signal}\textsubscript{Write} represents the condition under which a signal is assigned in the implementation.

The additional hardware required for an input or output signal is called an \textit{I/O stage} and consists of a multiplexer and a register as shown in Figure 4. The function of the \textit{I/O} stage, in general, is to allow the signal to pass through under certain conditions, and to allow the stored value to pass through under other conditions. It is also possible to use a transparent (level-sensitive) latch as \textit{I/O} stage in place of the \textit{mux/regist} structure, however, since certain design methodologies discourage the use of transparent latches, this work will focus on the \textit{mux/regist} structure. This \textit{I/O} stage can be simplified to a simple register or eliminated completely depending on the relationships between \textit{State}\textsubscript{Read} and \textit{Signal}\textsubscript{Read} (for input signals) and \textit{State}\textsubscript{Write} and \textit{Signal}\textsubscript{Write} (for output signals).

A \textit{mux/regist} input stage can be simplified to a register if the signal is never used when \textit{State}\textsubscript{Read} is true, that is, there is never the need for the signal to pass through the \textit{I/O} stage. This happens if \textit{Signal}\textsubscript{Read} \& \textit{State}\textsubscript{Read} = \textit{NULL} and it applies to \textit{in}_3 and \textit{in}_4 in Figure 3. A \textit{mux/regist} input stage can be completely eliminated if the signal is always used when \textit{State}\textsubscript{Read} is true, that is, the signal should always pass through the \textit{I/O} stage. This happens if \textit{Signal}\textsubscript{Read} \& \textit{State}\textsubscript{Read} = \textit{Signal}\textsubscript{Read} and it applies to \textit{in}_1 and \textit{in}_2 in Figure 3.

A \textit{mux/regist} output stage can be simplified to a register if the signal is never assigned when \textit{State}\textsubscript{Write} is true, that is, \textit{Signal}\textsubscript{Write} \& \textit{State}\textsubscript{Write} = \textit{NULL}. This is the case of \textit{avg}_2 in Figure 3. A \textit{mux/regist} output stage can be completely eliminated if the signal is always assigned when \textit{State}\textsubscript{Write} is true, that is, \textit{Signal}\textsubscript{Write} \& \textit{State}\textsubscript{Write} = \textit{Signal}\textsubscript{Write}. This applies to \textit{avg}_4 and \textit{inp}\textsubscript{req} in Figure 3. In this case, the \textit{enable} signal to the inferred register becomes \textit{Signal}\textsubscript{Write}.

Using this technique, the extra hardware is generated at the end of HIS, after control and datapath have been created. HIS computes the four Boolean conditions above and determines the extra hardware structures needed for each signal based on the relationships among these conditions.
4.2 Control and Data-Flow Graph Manipulation Method

The extra hardware for OTW support can be created by inserting temporary variables and assignments directly into the control and data-flow graphs representing the specification. Consider, for example, the specification in Figure 3a and a possible schedule as shown in Figure 3b. Since inputs $in3$ and $in4$ are not being used in a first state they will need to be stored. Similarly, output $avg2$ is not being assigned in a last state and will also need to be stored.

This technique creates (in this case) three temporary variables - $in3_{tmp}$, $in4_{tmp}$ and $avg2_{tmp}$ which are used in place of $in3$, $in4$ and $avg2$ respectively. Assignments from $in3$ to $in3_{tmp}$ are inserted in the first implementation states for which the corresponding specification state uses $in3$ (similarly for $in4$); and assignments from $avg2_{tmp}$ to $avg2$ are inserted in the last implementation states for which the corresponding specification state assigns to $avg2$. Figure 3c shows these extra assignments inserted in the implementation FSM.

The effect of these changes is to move all uses of inputs to first states and all assignments to outputs to last states. Once these extra variables and assignments are inserted, high-level synthesis will create any necessary registers according to the normal register inferring rules. For example, a register will be created for $in3_{tmp}$ because it is assigned in one state ($IST_0$) and used in another ($TST_1$).

To minimize the amount of extra hardware needed, the scheduling algorithm can move operations using input signals to first states and operations assigning to signals to last states. These moves have to take into account data dependencies and other constraints and may not always be possible.

Both methods presented may incur some hardware overhead for I/O stages. The graph manipulation method has the advantage that the I/O stage registers are treated as temporary variables which may be shared by resource sharing according to their lifetimes (which minimizes the overhead). Moreover, the user has full control over the extra hardware structures created. For example, if a given input remains constant for a number of cycles, he/she can pass that information to HIS, via attributes, so that HIS will not create extra storage for this input. Similar controls apply to the output signals.

5 Comparing Simulations Before and After High-Level Synthesis

The synthesis and simulation environment used in this work is the following. The specification VHDL is first simulated and checked for correctness. Then, it is submitted to high-level synthesis (HIS) which produces an RT/gate-level netlist from which the implementation VHDL is generated. This VHDL is then simulated using the same simulation driver and the results of both simulations are compared using the algorithm given below.

According to definitions 1 and 2, the instants of interest in the implementation simulation are those corresponding to transitions in the specification, which happen on triggering-edges of the clock when the OTW signal is ‘1’. The signal values in the following cycle can be compared with the specification simulation.

Let $sV_{sig}$ be a vector of simulation values for signal $sig$ in the specification, such that $sV_{sig}[p]$ gives the stable value of the signal in cycle $p$ (that is, after all delays following the triggering-edge of the clock). Let $iV_{sig}$ be a similar vector of simulation values in the implementation. The comparison algorithm is as follows:

```
if (OTW is '1' immediately before the triggering-edge of clock in cycle $i_{cmt}$) then 
  /* simulations can be compared */
  if ($iV_{sig}[i_{cmt}] = sV_{sig}[s_{cmt}]$) then 
    Implementation in cycle $i_{cmt}$ matches specification in cycle $s_{cmt}$.
  else 
    Error: Implementation in cycle $i_{cmt}$ mismatches specification in cycle $s_{cmt}$.
  end if
else 
  /* simulations cannot be compared in this cycle, increment $i_{cmt}$ only */
  $i_{cmt}++;$ $s_{cmt}++;$
end if
```

Figure 5 shows waveform comparisons for specification and implementation for the divider circuit in Figure 2. For space reasons it is not possible to show more extensive waveforms, however, the methods described apply to any design that satisfies the criteria specified in Section 2.

The implementation generated by HIS is then submitted to logic synthesis for optimization and mapping. A formal verification tool is then used to check that the HIS implementation is equivalent to the final optimized and mapped design after logic synthesis.
6 Conclusions

One of the main obstacles in the use of high-level synthesis has been the lack of verification mechanisms that allows an implementation, after scheduling, to be compared with an unscheduled specification. The change in the cycle-by-cycle behavior due to scheduling makes it impossible, in general, to compare simulation results before and after synthesis using the same simulation drivers/vectors. All approaches to date have required the user to rewrite the simulation vectors if scheduling is performed.

This paper presented a novel approach for verifying, through simulation, the results of high-level synthesis in a manner that allows the use of the same simulation drivers used for simulating the specification (prior to synthesis). The approach consists in generating in the implementation a special synchronization signal – called observable time window – and additional hardware support structures for inputs and outputs. The two simulations (before and after synthesis) can be automatically compared when the OTW signal is true just before the clock edge. This paper has laid the theoretical foundation for equivalence in high-level synthesis as well as given algorithms for generating an implementation that can be checked in a practical manner against the specification. The approach presented in this paper has been implemented in the HIS system. Several examples have been synthesized, simulated and compared successfully. The hardware overhead for storing intermediate values can be significantly reduced by resource sharing and user directives.

As future work, it may be possible to combine the concept of OTWs with formal methods by formally checking the equivalence between specification and implementation only when the OTW signal is true.

References


