Automated Verification of Temporal Properties Specified as State Machines in VHDL*

Yatin V. Hoskote  Jacob A. Abraham  Donald S. Fussell
Computer Engineering Research Center
University of Texas at Austin
Austin, TX 78712

Abstract
This paper presents a new verification methodology to prove that a high level HDL description of a synchronous sequential circuit satisfies certain desired behavior or that it is free of certain malicious behavior. The correctness specifications are modeled as state machines with some transitions having unspecified inputs. We show that this suffices for specification of a large class of properties, including both safety and liveness properties. The properties are described as VHDL programs to enable the designer to simulate them for sample inputs and gain some measure of confidence in their correctness. Experimental results are presented for the Viper microprocessor.

1 Introduction
The complexity of today’s designs makes it imperative that some tools or techniques be developed to aid designers in proving the correctness of their designs and in debugging them. Formal verification has received heightened interest of late because of the perceived inadequacy of simulation to detect bugs in even moderately sized designs. Formal verification can be applied at two stages in the design cycle: in verifying that an abstract, high level design possesses the intended behavior or satisfies certain properties, and secondly, in verifying that a low level implementation (e.g., gate level or switch level) correctly implements the high level design. In some circles, these techniques are referred to as design verification and implementation verification, respectively.

This paper deals with the first problem, the application of formal techniques to gain some measure of confidence in the correctness of a design. The design is viewed as an incompletely specified, deterministic Mealy finite state machine (FSM) and the specification properties are modeled as a type of non-deterministic state machine, with inputs being unspecified for some transitions. This enables the verification of certain liveness properties not expressible using conventional FSMs. The verification program computes all states in the design which satisfy this behavior, i.e., all states with the same input-output behavior as the starting state of the specification machine. The intent is to provide designers with an easy and intuitive specification format, along with the power to implicitly search the entire state space of the design. Frequently, the designer has already identified instances of “bad” behavior in the form of a sequence of inputs and consequent outputs. He or she can then check for the existence of such behavior in the design by modeling the sequence as a state machine.

There are a few characteristics of this problem which can be exploited to make the verification of real circuits tractable. The model to be verified is usually at a sufficiently high level of abstraction to be understandable by the user, as opposed to a netlist which can be extremely difficult to decipher. This makes it easier to specify the intended behavior in terms of the observable signals of the design and to debug the design. Also, since the design is usually described in some hardware description language (HDL), it is fairly easy to modify it in order to add or remove some behavior to aid in the verification. There has been extensive research on using abstraction techniques to reduce the size of the model to be verified such that the proof of correctness holds on the actual design if it holds on the abstract model [1] [2]. One simple technique is to reduce the size of the data path in the design whenever possible. More sophisticated techniques introduce varying degrees of non-determinism in the model.

1.1 Related Work
Almost all approaches to design verification express the properties to be checked either as formulas in temporal logic or as automata. Symbolic trajectory evalu-
ation [3] uses a restricted form of linear temporal logic to express properties as bounded length sequences of circuit states. Verification is done using an extension of symbolic simulation and has been applied to large designs. Limited expressiveness of the temporal logic is a hindrance in verifying certain types of liveness properties. The DDL verifier [4] automatically translates formulas in linear temporal logic to the form of state transitions, extended to express eventuality conditions. Satisfiability of the formulas is checked by looking for infinite paths in the transition graph.

CTL model checking [5] verifies that a finite state system satisfies properties expressed as formulas in branching-time temporal logic CTL and has been successfully applied to industrial systems. The drawback is that the temporal logic formulas can become exceedingly difficult to understand and there is a danger of the designer misunderstanding what property has actually been verified.

The language containment approach [2] describes a property as an \( \omega \)-automaton and verifies correctness by checking that the language of the system is contained in the language of the property. This approach has been applied to the verification of large systems using a succession of verification and refinement steps. However, certain types of properties involving existentiality are not expressible by \( \omega \)-automata. For example, the property that there exists a path to a reset state from any design state cannot be so expressed. A unified approach to verification, incorporating both language containment and CTL model checking to complement each other, has been presented in [6].

While all the above approaches are powerful and have been proved successful in their respective domains, the approach presented in this paper can be seen as a bridge for the designer between current methodology and the more esoteric verification techniques. It seeks to draw on the verification and debugging experience of the designer and provides a simple tool to make the task easier. This work, in conjunction with the techniques presented in [7], is aimed at empowering the designer with the capability for both design verification and implementation verification of sequential circuits within the same framework.

2 'Design Verification' in our Framework

We have previously verified gate level netlists of synchronous sequential circuits against high level specifications in VHDL by checking for containment of the specification in the implementation [7]. This proves that for every state of the specification, there exists a corresponding state in the implementation such that the two circuits exhibit identical input-output behavior when starting from those states. The specification state is then said to be compatible with the corresponding implementation state. The specification may be partially specified, so compatibility is checked only for input sequences applicable to the specification.

The specification is a behavioral VHDL description with operations scheduled into clock periods, but possibly without any structural information. It contains abstract data types like integers and high level operations (e.g., arithmetic operations) on these data types. This description is compiled into a finite state machine by using a library of Boolean templates for the high level operations. Each template in the library has been proved correct with respect to first principles using the Boyer-Moore theorem prover [8]. The netlist is then verified to contain this compiled finite state machine. This approach has to assume correctness of the VHDL specification. The techniques presented here are targeted at helping the designer gain some confidence in the correctness of this specification.

Since the operations in the data path of the specification have been proved correct using theorem proving techniques, the subsequent verification ensures the correct implementation of these operations in the netlist. Thus, design verification does not need to perform this correctness check. The properties to be checked deal more with control and synchronization aspects of the design. In many cases, this enables us to reduce the size of the data path of the design without affecting the validity of the verification. Boundary conditions verified on the scaled down model are then also true of the actual design. In some instances, it is possible to specify the properties to be verified on the control part of the design only, eliminating the data path altogether [9]. Since the input format is VHDL, it is relatively easy for the designer to modify the design. Also, the design does not need to be translated into another language solely for the purpose of verification as in most verifiers. This reduced model is then verified against properties which are specified in the form of state machines described below.

2.1 Specification state machine

The property or behavior to be verified is viewed as a sequence of inputs and consequent outputs and is encoded as a finite state machine with a designated start state. It may have don't cares in the outputs or next states. This state machine differs from a conventional FSM in that some of the transitions in this
machine may also have unspecified inputs. This implies that the machine undertakes that transition for some value of the inputs. This enables checking for the existence of transitions satisfying the specification without requiring prior knowledge of the inputs that would induce them. There must be only one such transition specified for a state and that must be the only transition from that state. A sink state is added to this specification to absorb all transitions with undefined next states.

As an example, consider the state machine shown in Figure 1(a) with one input and one output. We wish to check that whenever the input goes high, the output will go high within two clock cycles. We perform this verification by checking for occurrence of “bad” behavior which conflicts with the specified behavior: is there any sequence of states such that the output stays low for two clock cycles after the input has gone high? The specification machine is shown in Figure 1(b). The dashed arrows denote the transitions with unspecified inputs. The verification program indicates that design state D is compatible with specification state start and points to the existence of a “bad” sequence in the design starting at state D. In the general case when we are required to check for desirable behavior, the verification program computes all states that satisfy the specified behavior. It is then a simple matter to check whether a given design state, usually the most general starting state, belongs to this set.

Safety properties are of the form “nothing bad will ever happen”, whereas liveness properties take the form “something good has to happen”. Both are equally important in proving the correctness of a design, and properties of both types can be specified in our framework. A safety property may be verified by checking for the existence of “bad” behavior. Liveness properties can be verified by looking for the existence of states from which there is no path to a “good” state. The property in Figure 1 is a simple example of a liveness property. Of course, not all liveness properties can be handled in this manner. We are evaluating the full expressiveness of this specification framework.

3 Verification Algorithm

The verification algorithm must compute all starting states in the design such that the input-output behavior is identical to that of the specification from its starting state. In other words, it must find all design states compatible with the specification starting state. If there exists a transition with unspecified inputs in the specification, the algorithm must look for the existence of at least one transition in the design such that the ensuing behavior of the design satisfies that of the specification.

The verification algorithm computes all pairs of compatible states in the two circuits by finding successive sets of pairs of k-compatible states. Two states are k-compatible if every input sequence of length k, applicable to the specification, produces the same output sequence when applied to the circuits in those initial states. Two states are compatible if and only if they are k-compatible for all k. To account for the transitions with unspecified inputs in the specification machines, the notion of compatibility is slightly modified. The input sequences applicable to the specification in any state can be divided into mutually disjoint sets by enumerating all possible values of the inputs for transitions with unspecified inputs. For example, these sets for the start state of the specification in Figure 1(b) for an input sequence of length three are \{100\}, \{101\}, \{110\}, \{111\}. We now consider a specification state to be compatible with a design state if the sequence of outputs produced by the design is identical to that produced by the specification, starting in their respective states, for all input sequences in at least one of the sequence sets.

3.1 Computing pairs of compatible states

We first find the set of all pairs of 1-compatible states \(P_1\) in the two circuits, then the set of all pairs of 2-compatible states \(P_2\) as those 1-compatible state pairs with all or some successor states also 1-compatible, depending on whether the outgoing transition from the specification state is fully specified, and so on till we reach a fixed point \(P_{k+1} = P_k\). The set of all pairs of compatible states is then given by \(\mathcal{P} = P_k\) (this is a modification of the algorithm for equivalent states given in [10]). For this algorithm to be efficient, sets of states must be represented by their characteristic functions [11], which are predicates on Boolean
state variables, or by vectors of Boolean functions, and stored as ordered Binary Decision Diagrams (OBDDs) [12]. Next-state functions are represented as Boolean functions that map sets of states into sets of states, and are also stored as OBDDs.

Consider a Mealy machine defined by the 5-tuple $(I, O, S, \delta, \lambda)$ where $I = \{0, 1\}^m$ is the input space, $O = \{0, 1\}^l$ is the output space, $S = \{0, 1\}^n$ is the state space, $\delta : I \times S \rightarrow S$ is the next-state functional vector, and $\lambda : I \times S \rightarrow O$ is the output functional vector. Let $\vec{q}$ represent the vector of present-state variables, $\vec{Q}$ the vector of next-state variables, $\vec{x}$ the input vector and $\vec{y}$ the output vector. Thus, we have $Q_i = \delta_i(\vec{x}, \vec{q})$, $i \in \{1..n\}$ and $y_j = \lambda_j(\vec{x}, \vec{q})$, $j \in \{1..l\}$. Let $U(\vec{q}_a)$ be the characteristic function representing the set $U$ of specification states having an outgoing transition with unspecified inputs. Let $C_a$ and $C_b$ be the partially specified machines representing the specification and the design respectively.

We first compute the set of pairs of transitions in the two machines with identical outputs for the same inputs. This set must exclude transitions in the design with undefined outputs, given by the negation of the predicate $\eta_b$, for the ith output. On the other hand, transitions in the specification with undefined outputs, $\neg \eta_a$, for output i, match all corresponding transitions in the design. The predicate $T$ given below encodes this set of transition pairs,

$$T(\vec{x}, \vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^l \left[ \left( (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b)) \bigwedge \eta_{b_i}(\vec{x}, \vec{q}_b) \right) \bigvee \neg \eta_{a_i}(\vec{x}, \vec{q}_a) \right]$$

(1)

where $\odot$ is the Boolean equivalence (XNOR) operator.

We now wish to compute pairs of compatible states in the specification and the design using the predicate $T$. The 1-compatible state pairs which include those specification states in $U$ are those pairs which satisfy $T$ for some input vector. The 1-compatible pairs which include those states not in $U$ are those pairs which satisfy $T$ for all values of the input. The set of 1-compatible pairs of states is then the union of these disjoint sets, with characteristic function,

$$P_1(\vec{q}_a, \vec{q}_b) =$$

$$\left( U(\vec{q}_a) \bigwedge \exists \vec{x} : T \right) \bigvee \left( \neg U(\vec{q}_a) \bigwedge \forall \vec{x} : T \right)$$

(2)

To find the set of $k+1$-compatible state pairs $P_{k+1}$ from the set of $k$-compatible pairs $P_k$, we find the inverse image of the set $P_k$, i.e., the pairs of states whose corresponding next-states are $k$-compatible. The inverse image of $P_k$ such that all successor states are $k$-compatible is given by $\forall \vec{x} : P_k(\vec{Q}_a, \vec{Q}_b)$. The inverse image such that some successor states are $k$-compatible is given by $\exists \vec{x} : P_k(\vec{Q}_a, \vec{Q}_b)$. However, we must ensure that the transitions to such successor states have identical outputs, i.e., satisfy the predicate $T$. The set of $k+1$-compatible state pairs is then given by the union of these two inverse images, restricted to the set of $k$-compatible pairs.

$$P_{k+1}(\vec{q}_a, \vec{q}_b) = P_k(\vec{q}_a, \vec{q}_b) \bigcup$$

$$\left[ \left( U(\vec{q}_a) \bigwedge \exists \vec{x} : (T \bigwedge P_k(\vec{Q}_a, \vec{Q}_b)) \right) \bigvee \left( \neg U(\vec{q}_a) \bigwedge \forall \vec{x} : P_k(\vec{Q}_a, \vec{Q}_b) \right) \right]$$

(3)

The set of compatible state pairs $P$ is obtained when the fixed-point is reached.

$$P(\vec{q}_a, \vec{q}_b) = P_k(\vec{q}_a, \vec{q}_b)$$

if $P_{k+1} = P_k$

The design of state machines compatible with the start state of the specification has characteristic function $P(\vec{q}_a, \vec{q}_b)^{\vec{q}_a=\vec{q}_{a_1}}$. Design states not covering any specification state are given by $\neg P(\vec{q}_a, \vec{q}_b)$.

The inverse image operation and fixed point computation using next state functions has also been used to implement a CTL model checking algorithm [13]. It has been observed that while using next state functions overcomes the problem of trying to build a transition relation for complex circuits, the inverse image operation has inherently exponential complexity. Hence, it is important to be able to reduce the size of the design to be verified to make the verification of real circuits tractable.

### 3.2 Error Diagnosis

Providing the designer with information pointing to the cause of errors in the design is just as important as the verification itself. Generating an error trace is one way of aiding debugging. Given a design state not compatible with the specification starting state, in the absence of input-unspecified transitions in the specification, a distinguishing sequence starting from those respective states can be found by traversing the product state space [14]. For input-unspecified transitions, the procedure has to be modified to consider all satisfying design transitions. Note that in case of properties involving existence of “bad” behavior, we may wish to find an input sequence of specified length that contributes to compatibility rather than distinguishes between the starting states. In this case, the specified sequence length is the stopping criterion for the forward traversal rather than differing output values. Generation of such traces is currently being automated.
architecture BEHAVIOR of SPEC is
  type ST is (START, S1, S2, SINK);
begin
  process
  begin
    wait until CLK = '1' and not(CLK'stable);
    case ST is
      when START =>
        if (INP = '1') then
          ST <= S1; OUTP <= '0'; UTRN <= '0';
        else
          ST <= SINK; OUTP <= 'X'; UTRN <= '0';
        end if;
      when S1 =>
        ST <= S2; OUTP <= '0'; UTRN <= '1';
      when S2 =>
        ST <= SINK; OUTP <= '0'; UTRN <= '1';
      when SINK =>
        ST <= SINK; OUTP <= 'X'; UTRN <= '0';
    end case;
  end process;
end;
end BEHAVIOR;

Figure 2: Example specification machine in VHDL.

4 Specification Input Format

A drawback of expressing specifications in some temporal logic is the complexity of writing them as formulas in that logic. It can sometimes be very difficult to understand the formulas and thus easy to misunderstand what properties have actually been verified.

In keeping with our stated intention of providing a framework more intuitive to designers, we accept descriptions of the specification machines in VHDL. This also enables designers to simulate the description for sample input vectors and gain some measure of confidence in the correctness of the specification itself. The specification is a single VHDL process with a single wait statement depending on the clock and a single case statement depending on the state. The specification can contain signals which are bits or bit vectors, with associated arithmetic operations. The only values signals can assume are '0', '1' and 'X' (don’t care).

As an example, Figure 2 shows the state machine for the specification in Figure 1(b). UTRN is high for transitions with unspecified inputs.

5 Experimental results

The Viper is a 32-bit microprocessor designed for safety-critical applications [15]. In the Viper, any illegal operation, overflow or computation of an illegal address halts execution and raises the ‘stop’ signal. The instruction set provides several arithmetic, logical and comparison operations. The VHDL design is a behavioral description with operations scheduled into states. The netlist synthesized from this description has 251 flip-flops and approximately 5000 gates. In our experiment, each register in the register file was reduced from 32 bits to 2 bits, and the data offset in the instruction register was reduced from 20 bits to 1 bit. This reduction is valid because the properties we wish to check do not deal with function implementation. Suitably scaling down data values appearing in specifications correctly exercises boundary conditions, such as overflow. The scaled down design has 94 flip-flops (approximately 10⁸ reachable states) and is of manageable size. We verified several properties of this design, including part of the instruction set, some of which are described below. The program runs on an IBM RS 6000 with 128Mb RAM and uses the OBDD package written by David Long of Bell Labs.

Property (1): Verifying that the reset state is not reachable from all states took 4.3 seconds. This was done by adding a dummy output to the design, setting it high only in the reset state and modeling the specification machine as a single state with a self loop and output always low. The reset state is then unreachable from states compatible with the specification state.

Property (2): An instruction requires a maximum of five clock cycles. We checked that execution always returns to the fetch instruction state within six clocks by adding an output to the design which was high only in the fetch state and using the specification machine in Figure 3. The result told us that execution will not return to the fetch state if ‘stop’ goes high. We then checked only those execution sequences where ‘stop’ stays low. Adding a second output to the specification which was always low, corresponding to the ‘stop’ output of the design, did not show any design state compatible with the specification start state, which proved that the desired behavior was satisfied by the design. Checking this property took 8.3 seconds.

Property (3): We also verified a few instructions. Since the data path of the model is 2 bits wide, the intent was to verify the decoding logic and proper data transfer rather than correct function implementation. The specification machine for the simple XOR instruction is shown in Figure 4. The fetch state in the design was shown to be compatible with the specification start state. Verification took 4.7 minutes.

Note that verification is always over all possible data values, all possible initial states and for all possible input sequences. This can prove indispensable when the execution flow is more complex, as in pipelined processors and processors with interrupts.
6 Conclusions

This paper has presented a verification methodology used to gain some measure of confidence in the correctness of a HDL design before using it as the specification against which a netlist can be verified. The correctness specifications are modeled as finite state machines with some transitions possibly having unspecified inputs. We have shown that this specification framework is expressive enough to specify types of both safety and liveness properties, including instruction set specifications. Properties can be specified in VHDL which enables the designer to simulate them as a sanity check. Some properties have been verified for the Viper microprocessor using these techniques.

Though this framework may be less sophisticated than temporal logic model checkers, it is adequate and closer to the domain familiar to designers. Future work will deal with exploring the limits of the expressiveness of this specification framework. We intend to incorporate non-determinism in the model and improve the efficiency of the inverse image computation by drawing on related work in this area.

References


