A VHDL Subset for model-checking

Edmund M. Clarke
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
e-mail: Edmund.Clarke@cs.cmu.edu

February 11, 1994

Abstract

VHDL is a hardware description language that is widely used in digital circuit design and is likely to be used even more in the future. Ensuring the correctness of descriptions written in VHDL is therefore very important. One of the most powerful techniques for formal verification of finite-state systems is temporal logic model checking. We propose to use this method for verifying VHDL designs. In this paper we describe in detail a subset of VHDL which accomodates the most commonly used description styles and allows for efficient verification using model checking techniques. Based on this subset, we propose to build a model checker that can handle industrial designs of realistic complexity.

1 Introduction

Logical errors in hardware designs are an important problem for circuit designers. They can delay getting a new product on the market or cause the failure of some critical device that is already in use. My research group has developed a verification method called *temporal logic model checking* in which specifications are expressed in a propositional temporal logic, and devices are modeled as state–transition systems. An efficient search procedure is used to determine automatically if the specifications are satisfied by the transition systems. The technique has been used in the past to find subtle errors in a number of non-trivial hardware designs.

During the last few years, the size of the transition systems that can be verified by model checking techniques has increased dramatically. By representing transition relations implicitly using Binary Decision Diagrams (BDDs), we have been able to check some examples that would have required 10^{20} states with the original algorithm. Various refinements of the BDD-based techniques have pushed the state count up to 10^{100}. By combining model checking with various abstraction techniques, we have been able to handle even more complex systems like the cache coherence protocol in the IEEE Futurebus+ Standard.

Because of these advances, we believe model checking techniques are already sufficiently powerful to be useful in verifying real industrial designs. Since these techniques avoid the construction of complicated proofs and provide a counterexample trace when some specification is not satisfied, we think that engineers will find them much easier to learn and use than hardware verification techniques based on automated theorem proving or proof checkers.

In order for these techniques to be accepted in industry, we believe that it is essential to provide an interface between the verification tools that we have developed and some widely used hardware description language. VHDL is the obvious choice for such a language: It is used as the input language for many CAD systems, it provides a wide variety of descriptive styles, and it is an IEEE standard. However, VHDL is a complex language, and some features of the language cannot be handled by model checking techniques. Consequently, it is necessary to restrict the language so that it can be used in verifying hardware designs. In this paper we describe in detail a subset of VHDL which accomodates the most commonly used description styles and has a well-defined formal semantics that allows for efficient translation and verification.

This paper is organized as follows: Section 2 contains a description of the subset of VHDL that we propose to use as the basis for our model checking system. In section 3 we present an example VHDL program which illustrates the expressive power of the subset and how it can be used for verification. Section 4 gives milestones for the various phases of the project. A detailed BNF description for the language subset can be found in the appendix.

2 Features of the VHDL subset

The top-level design unit in a VHDL description system is an *entity declaration* together with its corresponding *architecture body*. An entity declaration defines the interface between the modelled digital system and the environment in which it is used, by describing the ports that provide the external communication. The architecture body describes a possible implementation of the model, by expressing the relationship between the inputs and outputs of the design entity. Behaviorally, this description is given in terms of *processes* that communicate via *signals* and synchronize on *wait statements*. Alternatively, parts of the architecture can be described structurally as *components* which are instances of the design entities described previously.

VHDL models devices using a stimulus-response paradigm: When a stimulus occurs, the model responds and then waits for a new stimulus. Thus, the semantics of a VHDL program must be given in terms of the *simulation cycle*, which consists of two stages. In the first stage, time advances to the next moment at which a signal becomes active or a process resumes. Signals scheduled to obtain new values at that time are updated. During the second stage, all processes that are sensitive to signals that have been updated resume and execute in zero time until they suspend on a wait statement. This completes the simulation cycle, and a new cycle is started.

Because of the simulation semantics outlined above, one of the major issues in discussing a design style in VHDL, whether for the purpose of synthesis or verification, is the usage of *time*. The initial subset selected for model-checking is restricted to a purely causal description style. More precisely, the use of explicit time expressions, such as the *after clause* in signal assignments or the *timeout clause* in wait statements is disallowed. In spite of this apparent restriction, important classes of designs can still be modelled and their properties verified. This includes synchronous designs, where the occurrence of all actions is described with respect to a global clock, as well as asynchronous systems, where components react to external stimuli, and the reaction terminates before the next stimulus is applied.

The major features and restrictions of the language subset are discussed in this section. The discussion follows the outline given in the *IEEE Standard VHDL Language Reference Manual*. All major restrictions are stated explicitly. A BNF description of the syntax for the subset is given in an appendix.

VHDL is a very rich language. Certain language constructs can be expressed in terms of more basic constructs, while maintaining the same semantics. Therefore, for the first prototype, we have attempted to select the subset of language constructs which are most widely used, without restricting significantly the set of behaviors that can be described. In later stages of the project, we will augment the model checker to handle additional language features.

- Design entities

  The language subset provides *entity declarations* and *architecture bodies* as design units. The entity declaration is restricted to specifying a header with the interface ports. Any other declarations or passive statements within the entity can be described equally well in the architecture body and will not be included in the initial subset. *Generics* are not allowed in an entity declaration.

- Configurations

  This subset does not allow *configuration declarations*. Instead it requires that the binding of component instances to design entities be done at an earlier phase by *configuration specifications*.

- Subprograms

  Both kinds of subprograms, that is *procedures* and *functions* are accepted. The subprogram body declaration must precede any call to the subprogram, therefore no recursive subprograms are allowed. Moreover, the subset does not allow either *subprogram* *overloading* or *operator overloading* by the programmer. *Resolution functions* are not permitted, thus only a single process is allowed to assign to a signal.

- Packages

  Package declarations may contain *type*, *constant*, *subprogram* and *component declarations*, *attribute declarations* and *specifications*, and *use clauses*. *Package bodies* define the bodies of the subprograms declared in the package.

- Types

  Since the verification technique that we use is based on searching finite-state models, only discrete and finite data types and data structures are permitted. The subset supports *scalar types*: the predefined *integer type*, *enumeration types*, and *physical types*. No provision exists for *subtype declarations*, although *integer ranges* are permitted. Other predefined types include: the enumeration types *character*, *bit*, and *boolean*; the type *string*; and the physical type *time*. It is possible to declare unconstrained array types. However, it is not possible to declare unconstrained arrays. *Access* and *file types* as well as incomplete type declarations are not supported.

- Declarations

  Constant and *variable declarations* obey the rules of standard VHDL. As a consequence, the signal types *register* and *bus* are not supported. In interface declarations, only modes *in*, *out* and *inout* are allowed.

- Names

  Names supported by the subset are *simple names*, *selected names*, *indexed* and *slice names*, and *attribute names*.

- Expressions

  All operators predefined by standard VHDL are included in the subset, with the exception of *array concatenation*. *Aggregates*, i.e. composite values of record or array types, are not supported.

- Sequential statements

  All sequential statements of standard VHDL are supported: *wait statement*, *signal* and *variable assignments*, *procedure call*, *if*, *case*, *loop*, *next*, *exit*, *next*, *return* and *null statements*. For the *assertion statement*, an error will be reported if a state corresponding to an assertion violation is reached during verification.

- Timing

  The subset does not allow the specification of *time expressions*, either in the wait statement (*timeout clause*) or in the signal assignment statement (*after clause*).

- Loops

  Loop statements may not be prefixed by a label; as a consequence, the *next* and *exit statements* always refer to the innermost loop.

- Concurrent statements

  The subset supports the *block*, *process*, *concurrent procedure call*, *concurrent signal assignment* and *component instantiation* statements. The *concurrent assertion* and *generate* statements are not included in the subset. The *block statement* may not have a *guard expression* associated with it, and the *block header* must be empty (no generic or port map clauses are allowed). Only the *conditional* (not the selected) form of the concurrent signal assignment is available for use; the options *guarded* or *transport* are not allowed.

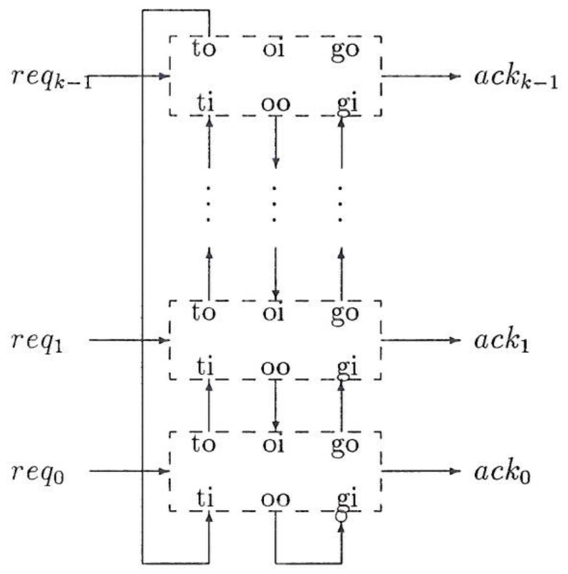
3 A VHDL example

This section presents a small VHDL design as an example for the types of descriptions that the proposed model-checking system will be able to handle. The VHDL code describes a synchronous bus arbiter circuit, adapted from the one given in McMillan's thesis. Its purpose is to grant access on each clock cycle to a single client among several that are competing for the use of the bus. The inputs to the circuit are the request signals $req_0, req_1, ...req_{k-1}$ and a clock clk, and the outputs are the acknowledge signals $ack_0, ack_1, ..ack_{k-1}$.

The arbiter uses two criteria to determine the request that will be acknowledged. The basic scheme is priority-based, the circuits having decreasing priorities starting with cell zero. To avoid starvation, the arbiter also implements a token-based round-robin scheme. If a client's request persists long enough to receive the token twice, that client is acknowledged, overriding other requests.

To coordinate the arbiter cells, override signals propagating along the chain to cell 0 and grant signals propagating in the opposite direction are used. The override input of a cell is asserted if any of the lower-priority cells receives the token while having a persistent request. The grant input of a cell is asserted if no cells assert override and none of the higher-priority cells has an outstanding request. Internal signals are used to sample the requests on the rising edge of the clock and to assert the acknowledge on the falling edge, avoiding effects due to propagation delay or hazards.

The desired properties of the arbiter circuit are expressed in the temporal logic CTL and included in the decription as special comments interpreted by the model-checker:
1. No two acknowledge outputs are asserted simultaneously
2. Every persistent request is eventually acknowledged
3. Acknowledge is not asserted without request.



Configuration of the synchronous arbiter circuit

```
entity arb_cell is
  port ( req: in boolean; clk: in bit; ack: out boolean;
                tok_in: in boolean; tok_out: out boolean;
                ovr_in: in boolean; ovr_out: out boolean;
                grant_in: in boolean; grant_out: out boolean );
end arb_cell;

architecture arb_cell_rr of arb_cell is
  signal persist: boolean; -- req persists after token in
  signal req_int: boolean; -- internal; sample req on clk = '1'
  signal ack_int: boolean; -- internal; output on clk = '0'
begin
  cycle: process -- executes once per clock
    begin
      wait until clk = '1';
      req_int <= req; -- store value of req
      persist <= req and (persist or tok_in);
      tok_out <= tok_in; -- moves down one cell per cycle
    end process;

  update: process ( req_int, persist, tok_in, ovr_in, grant_in )
    begin
      ack_int <= req_int and (persist and tok_in or grant_in);
      grant_out <= not req_int or grant_in; -- grant if no request
      ovr_out <= ovr_in or persist and tok_in; -- override if persist
    end process;
```

```
   ack: process -- executes once per clock
     begin
       wait until clk = '0'; -- output ack on falling edge
       ack <= ack_int;
     end process;
 end arb_cell_rr;

 entity main is
   port ( clk: bit; req0, req1, req2: in boolean;
                 ack0, ack1, ack2: out boolean );

 -- SPEC
 -- No two acknowledge outputs are asserted simultaneously
 --   AG ( !( ack0 & ack1) & !(ack1 & ack2) & !(ack2 & ack0) )

 -- SPEC
 -- Every persistent request is eventually acknowledged
 --   AG AF ( req0 -> ack0 ) & AG AF ( req1 -> ack1 ) & AG AF ( req2 -> ack2 )
  -- SPEC
 --   Acknowledge is not asserted without request.
 -- AG (!ack0 -> AX(ack0 -> req0)) & AG (!ack1 -> AX(ack1 -> req1))
 -- & AG (!ack2 -> AX(ack2 -> req2))

 end main;

 architecture main_arch of main is

   signal g0, g1, g2: boolean; -- grant
   signal no: boolean := true; -- negate override for cell 0
   signal o0, o1, o2: boolean; -- override
   signal zr: boolean := false; -- nth element override in
   signal t0, t1 : boolean := false; -- no token
   signal t2: boolean := true; -- initial token for cell 0

   component element
     port ( req: in boolean; clk: in bit; ack: out boolean;
                 tok_in: in boolean; tok_out: out boolean;
                 ovr_in: in boolean; ovr_out: out boolean;
                 grant_in: in boolean; grant_out: out boolean );
   end component;

   for all: element use entity arb_cell(arb_cell_rr);

 begin
   e0: element ( req0, clk, ack0, t2, t0, o1, o0, no, g0 );
   e1: element ( req1, clk, ack1, t0, t1, o2, o1, g0, g1 );
   e2: element ( req2, clk, ack2, t1, t2, zr, o2, g1, g2 );
   no <= not o0; -- grant in for cell zero is negated override out
 end main_arch;
```

4 Milestones for the project

Below we describe how we expect to implement the model checker. We have broken the expected two year time-frame for the project into six-month stages.

1. During the first six month phase, we expect to build a compiler that translates the VHDL code into an internal representation from which the corresponding finite-state model will be built. To speed up the development of the first prototype, subprograms and loops with a dynamic iteration counts will not be included.

2. In the second phase, we will develop the first working model-checker. This involves developing the program to construct a finite-state model of the system and implementing the model-checking algorithms. There are two possible ways of accomplishing this phase. The first is to compile VHDL into the SMV language for which a good model checking system already exists. While this solution would possibly be faster to implement, it would be less flexible in the long run. Therefore, we currently favor a second approach which involves constructing the finite state model directly from the VHDL program. This will allow an easier extension of the language subset as well as permit counterexamples to be given at a level that the user can easily understand.

3. The third phase of the project will be primarily devoted to the extending the accepted language subset. We intend to implement procedures as well as loop coonstructs in their full generality. Other features like generics that do not require major changes in the way models are constructed from programs will be added if time permits. Furthermore, at this stage we hope to have some feedback from potential users concerning the additional language features that should be supported.

4. In the fourth phase, we plan to improve the interface of the model-checker. One of the main tasks in this phase is improving the readability of the counterexamples that are produced by the model-checker (the current SMV verifier simply prints the list of states in the counterexample trace). We also plan to investigate how to make the specification language more expressive and easier to use. One way of achieving this goal is to allow *timing diagram notation* be used in specifications in addition to temporal formulas.

115. block_declarative_
116. block_statement_p
117. process_statement
[*process*_label
process [(se
process_d
begin
process_s
end process
118. process_declarativ
119. process_declarativ
subprogram_b
| type_declarat
| constant_dec
| variable_decl
| attribute_dec
| attribute_spe
| use_clause
120. process_statement
121. concurrent_proced
122. concurrent_signal_
[label :] con
123. conditional_signal_
124. conditional_wavefo
125. component_instant
*instantiation*_l
126. use_clause ::= **use**
127. abstract_literal ::=
128. decimal_literal ::=
129. integer ::= digit {
130. exponent ::= E [+
131. based_literal ::= ba
132. base ::= integer
133. based_integer ::= e
134. extended_digit ::=
135. character_literal ::=
136. string_literal ::= "
137. bit_string_literal ::=
138. bit_value ::= exten
139. base_specification ::
140. physical_literal ::=

```
115. block_declarative_part ::= { block_declarative_item }
116. block_statement_part ::= { concurrent_statement }
117. process_statement ::=
         [ process_label : ]
         process [ ( sensitivity_list) ]
             process_declarative_part
         begin
             process_statement_part
         end process [ process_label ] ;
118. process_declarative_part ::= { process_declarative_item }
119. process_declarative_item ::=
         subprogram_body
         | type_declaration
         | constant_declaration
         | variable_declaration
         | attribute_declaration
         | attribute_specification
         | use_clause
120. process_statement_part ::= { sequential_statement }
121. concurrent_procedure_call ::= [ label : ] procedure_call_statement
122. concurrent_signal_assignment_statement ::=
         [ label : ] conditional_signal_assignment
123. conditional_signal_asignment ::= name ¡= conditional_waveforms
124. conditional_waveforms ::= { waveform when condition else } waveform
125. component_instantiation_statement ::=
         instantiation_label : component_name [ port_map_aspect ] ;
126. use_clause ::= use selected_name { , selected_name } ;
127. abstract_literal ::= decimal_literal | based_literal
128. decimal_literal ::= integer [ exponent ]
129. integer ::= digit { [ underline ] digit }
130. exponent ::= E [ + ] integer | E [ − ] integer
131. based_literal ::= base # based_integer # [ exponent ]
132. base ::= integer
133. based_integer ::= extended_digit { [ underline ] extended_digit }
134. extended_digit ::= digit | letter
135. character_literal ::= ' graphic_character '
136. string_literal ::= " { graphic_character } "
137. bit_string_literal ::= base_specification " bit_value "
138. bit_value ::= extended_digit { [ underline ] extended_digit }
139. base_specification ::= B | O | X
140. physical_literal ::= [ abstract_literal ] unit_name
```