

OBDD Extraction from VHDL Gate Level Descriptions at Design Elaboration.

VISHWANATH RAMAN
Vishwa@Synopsys.com
Synopsys, Inc.

ALEX N. ZAMFIRESCU
A.Zamfirescu@IEEE.org
Alternative System Concepts, Inc.

Abstract

This paper deals with a declarative interface for VHDL in general and the use of such an interface for OBDD based verification of VHDL gate level designs in particular. It presents a solution that enables OBDD verification without external manipulation of the netlist which is well integrated into the standard VHDL environment. The information required for OBDD based VHDL verification, existing and possible future VHDL uses and some advantages of a general declarative interface are illustrated. First experiments with such an interface revealed the advantages of in-place OBDD extraction and helped identify good candidates for VHDL extensions.

Keywords – Verification, VHDL, OBDD, declarative interface, combinational logic, global netlist.

Terminology

OBDD	Ordered Binary Decision Diagram.
KIF	Knowledge Interchange Format.
KQML	Knowledge Query and Manipulation Language.
PLI	Programming Language Interface.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.

I. Introduction

With the growing complexity of designs and the cost of generating close to 100 percent test coverage new verification techniques besides simulation have emerged. OBDD based verification [1] continues to be the method of

choice for functional verification. The VHDL standard, supporting a good concurrent design methodology, was originally designed to support simulation, is widely used in synthesis today, but lacks direct mechanisms that will enable other kinds of verification. We report here our experience interfacing an OBDD generation module with a VHDL tool. The goal was to show that *tighter* integration between implementations incorporating formal verification methodologies and a VHDL environment, is feasible and identify its advantages.

This paper presents the results of a design and implementation of a declarative interface that can be used to perform OBDD extraction from VHDL designs, without the need for external manipulation of the netlist. It identifies the information required for such extraction, finds existing and some possible future VHDL enhancements that could ease handling of such information, and advocates a more general declarative interface for VHDL design manipulation.

The work done by Glunz, et al [5] addresses system level design from a top down perspective that involves translation of parts of the system, described using SDL, into VHDL. That approach requires translation from one representation to another. In contrast, we propose an approach well integrated into a VHDL representation of a design, with the capability for advanced assertions, specifically, equivalence determination and checking for consistency to advertised behavior, using a general declarative interface.

A set of prerequisites for OBDD based verification is identified in section II, a pragmatic solution is presented in section III, our implementation and a VHDL example are

described in section IV, and finally, some implications for future versions of VHDL are presented in section V.

II. Prerequisites for OBDD based verification

For a multiple variable boolean function and a given order of the variables, the OBDD associated with the function is unique. Therefore, the verification of a gate level description could be accomplished by constructing the OBDDs for each output variable and comparing them with the OBDDs representing the corresponding functionality.

The advantages of in-place OBDD extraction can be leveraged in model equivalence checking, optimized code generation targeting improved simulation performance, extraction of the logic network, fault coverage, and dependency analysis. Our example highlights only one of these domains, namely, equivalence checking.

A. Construction of the source OBDDs

In order to construct the OBDDs for the output variables of a specific gate level design, the following steps need to be performed:

- Identification of all variables.
- Ordering of the variables.
- Extraction of all the existing relationships between the variables based on the logic gates in the design.
- Construction of the OBDDs that represent the outputs.

B. “Global vs. Local” netlist information

A global view of the block in the design that implements a particular boolean function is needed in order to build the OBDD representation of that function. It is known that such a view is not available directly during VHDL simulation and the computation normally performed is local. In order to enable OBDD extraction from VHDL designs, the following are needed:

- A netlist extracted from the elaborated VHDL design.

- Association of signals with the OBDD variables used to represent them.
- Dependencies of the output variables on the input variables. These dependencies can be specified in terms of symbolic expressions and these expressions should be made available to the OBDD construction tool.
- A mechanism to maintain the global order of all variables.
- A mechanism to perform compositions and restrictions on OBDDs.

C. Equivalence checking

The equivalence of a function extracted from the model and represented by an OBDD, to the desired functionality, represented by a second OBDD, amounts to composing their XNOR and checking for a tautology, or by a topological comparison of the diagrams. When there are don't care conditions for the output variables, equivalence gets extended to a tautology check on the don't care expression.

This phase of verification has the following prerequisites:

- A mechanism to check for tautologies.
- A method to register the expressions representing don't care conditions for all outputs with the verification tool.

The ideas that are used to overcome some of the challenges mentioned above and their implementations are explained in the subsequent sections.

III. A pragmatic solution

Our work lead to the development of a solution that provides for the verification of gate level designs. It is simple, can be easily adopted into a verification methodology and is relatively fast. The trade-off made affects the preparation of the libraries and is transparent to the designer. The significant trade-off is that the library developer has to document the intended behavior of each gate. The alternative would be to either extract the behavior from the intermediate format or to validate the documented behavior against the intermediate format. While this alternative is not precluded by our design, the current implementation relies on the consistency of the

library. This simplifies the tool and based on our experience does not reduce the usefulness of the verification. The arrival of the VHDL PLI [6] makes the extraction of gate behavior feasible directly from the library cells. Both intent as well as implementation will be verifiable this way using a portable PLI application.

The verification is performed during VHDL elaboration and initialization (Figure 1(i)) of a gate level description. The netlist is extracted through a declarative interface, the logic behavior of each gate is transmitted to the OBDD tool and the list of don't care expressions (if any) is also described to the tool. Based on this information and using composition and restriction algorithms, the tool constructs OBDDs for the logic behavior functions as well as the don't care expressions for each output. The interface can be used to trigger verification that compares the extracted OBDDs with the corresponding golden OBDDs. Vector based validated behaviors or expected behaviors are conveyed in a file format. An OBDD utilities module is needed to ensure global ordering of all the variables required for OBDD verification. In

the current implementation this module supplies initial values for the constants that are declared in VHDL.

The known approach in VHDL based OBDD verification is to extract the netlist and pass that to a separate verification tool. This requires a separate netlist extractor, a reader of that format and the whole library information or other conventions about the behavior and a special mechanism to scope in the don't care conditions. Therefore for larger designs the design database is visited several times. With our approach the OBDDs are generated on the fly with no extra passes over the elaborated design. Therefore, selective verification becomes feasible, don't care conditions are conveyed in the same scope as the signals for which they apply, the OBDD extraction is simpler, without necessitating an external netlist format. An extra benefit is that the interface we used to pass information is a possible starting point for a future standard VHDL declarative interface for more involved verification, design optimization, component selection during elaboration or the new verification phase.

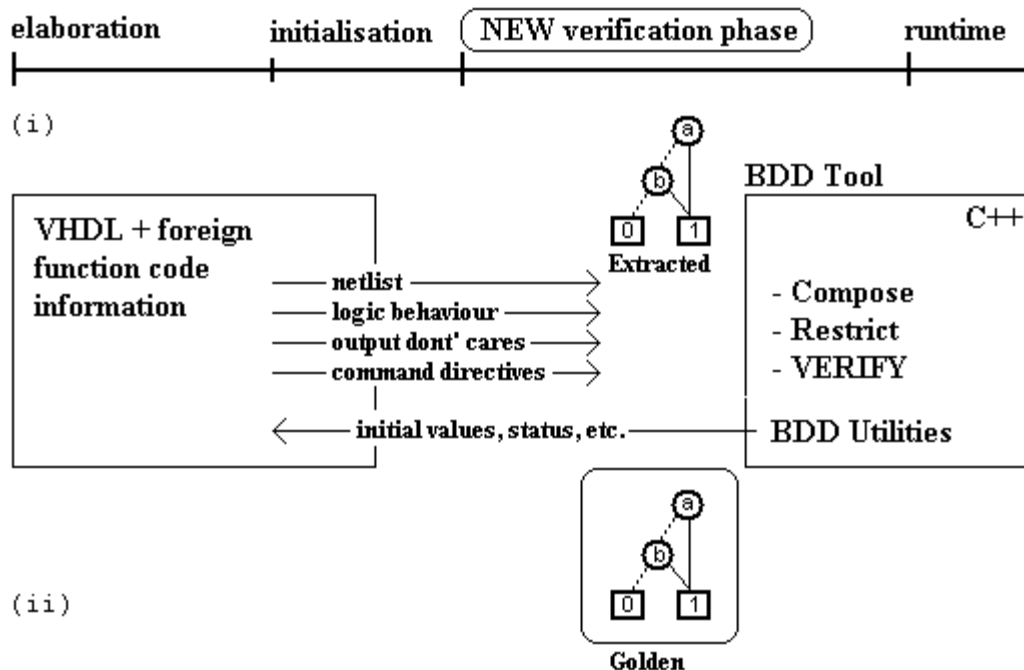


Figure 1. (i) New verification phase in the standard VHDL flow; (ii) Information flow for OBDD verification;

IV. Example

A. A four bit parallel adder with a fast carry look ahead [4]

The design we used to demonstrate the interface is purely structural with all leaf level components being gates. A package containing the foreign interface and a set of components implementing AND and OR logic has been defined. A full

adder making use of a couple of half adders has been implemented (Figure 2). Four full adders have been used to define a four bit adder. The fast carry look ahead stage is shown in Figure 3.

The leaf level components are programmed to communicate the dependency of the output variables on the input variables through symbolic boolean expressions. This information is gathered and processed by the verification tool.

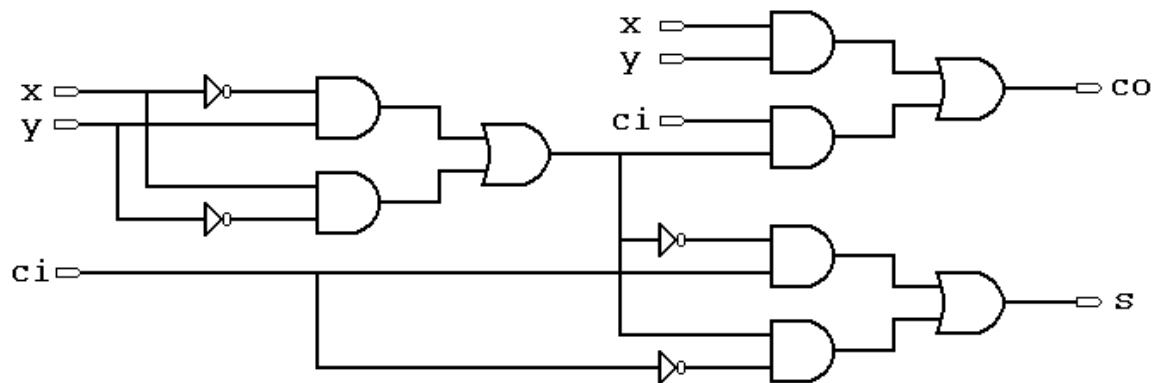


Figure 2. The implementation of the full adder. Two half adders have been used in conjunction with an OR stage. For brevity the sub components have been folded into this schematic.

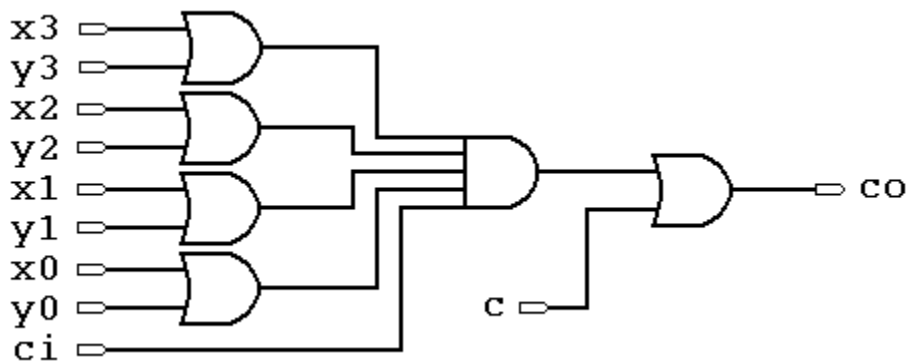


Figure 3. The design of the four bit parallel adder using four Full Adders. The fast carry is computed using a combination of five OR stages and an AND stage as shown.

A. OBDD extraction

Our mechanism to communicate the information that is required for the design OBDD extraction involves foreign function calls to register the netlist information as well as the input output relationship.

A foreign function call registers the path name of a design entity. This is used to establish context for subsequent registration calls used to communicate the connectivity as well as the symbolic expressions for each leaf level component of the design.

For the connectivity we provide a sequence of integer generics within every entity declaration, such that there is a generic that corresponds to each logical port of the entity. Further, every component instantiation contains a generic map such that, for every port that has an actual signal associated with it, the corresponding generic is associated with a constant that corresponds to the source signal (Figure 4). The values associated with the constants are automatically unique over the design and the uniqueness ensures that the netlist can be extracted at elaboration.

Foreign function calls are used to register generics and their corresponding ports. A simple language is used to express the relation of the output ports to the input ports. For example, a two input AND gate with input ports X and Y and output port Z, will register “Z = X and Y” as the symbolic expression. This string, provided by the library developer, could be further validated with the intermediate format, but our approach relies on the consistency of the model. This is typically expected from the provider of a library and would therefore not be open to change by the designer.

The advanced library format standard from which VHDL library cells are generated could contain that information and the string could be automatically generated by the ALF reader. The string is parsed by the verification tool in order to determine the functionality of that gate and construct the corresponding OBDD.

B. Implementation in VHDL

Sections of the VHDL code that was used to realize the adder are shown in Figure 5.

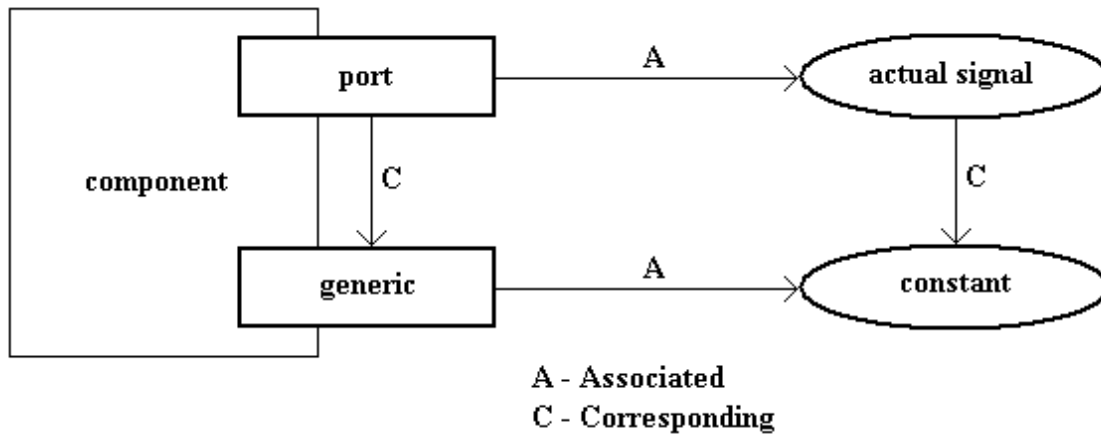


Figure 4. The relationship between ports, generics, actual signals and constants.

```
-- entity declaration and architecture body of the full adder.
use WORK.FORMAL.all;

entity FA is
  generic (
    g1 : integer; -- generic g1 corresponds to port x
    g2 : integer; -- g2 corresponds to port y
    g3 : integer; -- g3 corresponds to ci and so on
```

```

        g4 : integer;
        g5 : integer);

    port (
        x : in bit;
        y : in bit;
        ci : in bit;
        s : out bit;
        co : out bit);
end FA;

architecture STRUCTURAL of FA is
    -- internal signals used for connectivity

    signal sint : bit;
    signal cint1 : bit;
    signal cint2 : bit;

    -- constant c1 corresponds to signal sint
    -- constant c2 corresponds to cint1 and so on...
    -- this correspondence is in fact registered only at the
    -- time of call of the tell foreign function

    constant c1 : integer := ask;      -- ask returns an
    constant c2 : integer := ask;      -- unique integer
    constant c3 : integer := ask;      -- with each call

begin

    -- registration process executes once
    -- registers the component as well as the verification
    -- generics with the verification package

    process
        -- variable to handle context
        variable instanceId : POSITIVE;

        -- variable to handle return status
        variable result : POSITIVE;
    begin
        -- register component with path name
        instanceId := tell(STRUCTURAL'path_name);

        -- register all generics with the corresponding ports
        result := tell(instanceId, g1, x'simple_name);
        result := tell(instanceId, g2, y'simple_name);
        ...

        -- register the constants with the corresponding signals
        result := tell(instanceId, c1, sint'simple_name);
        ...

        wait;
    end process;

    -- component instantiations

    HA1 : entity work.HA(GATELEVEL)
        generic map (g1, g2, c1, c2)
        port map (x, y, s1, s2);
    ...

end STRUCTURAL;

-- entity declaration of a two input OR gate.

entity OR2_FORMAL is
    generic (
        g1 : integer;
        g2 : integer;
        g3 : integer);

    port (
        x : in bit;
        y : in bit;
        o : out bit);

    -- definition of a string constant containing the symbolic
    -- expression that relates the output to the inputs
    constant expression : string := "o = x or y";

```

```

end OR2_FORMAL;

architecture EXPR of OR2_FORMAL is
begin
  process
    variable instanceId : POSITIVE;
    variable result : POSITIVE;
  begin
    instanceId := tell(EXPR'path_name);

    -- registration of the symbolic expression
    result := tell(instanceId, expression);

    -- calls to register generics come here
    ...

    wait;
  end process;

  -- gate behavior
  o <= x or y;
end EXPR;

```

Figure 5. Sections of the VHDL code that was used to realize the adder

For the OR gate a single call is made to register the symbolic Boolean expression. The unique constant values are obtained through invocations of the ask function. These values are generated through a method that requires access to a global state during simulation initialization. Shared variables, file I/O or a foreign function can be used in VHDL. The current implementation uses a foreign function.

C. Verification algorithm

The schedule of activities, related to the OBDD extraction, that occur after analysis are listed below.

- The invocation of foreign functions takes place during elaboration and initialization.
- The expression parsing and OBDD generation for the leaf level components is performed at the end of the initialization phase.

Generating OBDDs for the design is a two pass process.

- The first pass generates OBDDs from the symbolic expressions.
- The second composes the design OBDDs by eliminating all internal variables through a process of restriction and

composition. This latter process is based on the Shannon factoring of boolean expressions [1, 2].

D. OBDD implementation

A two weight system has been used for constructing the OBDDs. The weight system is similar to the one proposed by Bryant [1]. All edges lead to subgraphs, which are restrictions of the OBDD graph with respect to the variable corresponding to the initial vertex. In Figure 4, dotted lines lead to restrictions when the variable assumes the value of logic '0' and solid lines lead to restrictions when the variable corresponding to the initial vertex assumes a value of logic '1' (terminal vertices are listed multiple times to avoid cluttering). Single terminal vertices for logic '0' and logic '1' are implemented.

The OBDDs are stored as vertices lists. Each vertex contains an edge list. Within each edge the address of the terminal vertex and a weight are stored. OBDD graphs also contain a value attribute, which is set to logic '0' or logic '1', if during restriction or composition operations, the OBDD is reduced to a tautology.

Figure 6, represents the OBDD obtained through our implementation for the output carry of the

compose OBDDs. The emerging PLI standard [6] has a provision for the declaration and registration of foreign functions as part of the standard, which makes the implementation completely portable.

V. Implications for future versions of VHDL

A. Netlist extraction

A global view of the netlist of a VHDL design is not easily available at runtime. Today's tools can extract the netlist from a VHDL configuration or from the elaborated design using for example a PLI, but that information is not obtainable from within the execution of the elaboration, initialization and/or simulation. A second obstacle is difficulty in acquiring local netlist information (fanin, fanout) from within a component or a block. To counter these obstacles we had to utilize the pairing of signals and constants (generics and ports), but this could have been avoided if there was VHDL support for one of the following:

- A mechanism to pair signals and constants.
- Functions to recognize fanout and fanin ports and centralized communication through a declarative interface.
- A VHDL PLI accessible from within VHDL code during initialization.

B. Object ordering

We encountered the need to order the signals in a design. We used a global counter and a foreign function, but this could have been solved by any of the following:

- Existence of global counters that can be assigned to particular tasks.
- An interface function for VHDL objects to self register after the objects are created. For example, an initialization string could contain the name of a registration function.

C. Verify phase

A separate phase after initialization and before simulation could be used to perform initial

design checks, which could include logic verification, static time property verification, fanout rules verification or any other property that is important to the designer. Of those, logic verification can be tackled using the interface described in this paper. A standard VHDL verify phase could enable such verification in any VHDL compliant tool.

D. General declarative interface

Knowledge passing mechanisms based on standard agent languages KIF and KQML could provide for smart component selection, implementation of negotiation algorithms (for system level design), optimized elaboration and simulation, electronic commerce and verification. The simple interface (tell, ask, etc.) described here is in support of such future enhancements.

VI. Final remarks

It can be seen clearly that using the approach outlined in this paper, we can enable the extraction and use of OBDDs within the confines of an existing VHDL based verification environment with minimal changes. The two principle contributions with our approach are the achievement of in-place OBDD extraction and the seeding of further development of a declarative interface to capture user intent, that can be used by system level design tools and new methodologies. The mechanism provides for verification consistent to the intermediate format, while opening up many opportunities to use the extracted OBDDs, such as optimizing verification or abstracting behavior for system level design.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments.

References

- [1] Randal E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, Vol. 24, No. 3 (September, 1992), pp. 293-318
- [2] Randal E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," *International Conference on Computer Aided Design* (November, 1995).
- [3] IEEE Standard VHDL Language Reference Manual STD 1076-1993. IEEE, New York, 1993.
- [4] Herbert Taub, Donald Schilling. *Digital Integrated Electronics*, McGraw-Hill Book Co. Singapore. 1987.
- [5] Wolfgang Glunz, Thomas Kruse, Torsten Rossel, Dieter Monjau, "Integrating SDL and VHDL for System-Level Hardware Design", *IFIP Conference on Hardware Description Languages and their Applications* (April, 1993).
- [6] The IEEE VHDL PLI draft standard at <http://www.vhdl.org/vhdlpli>.