Vhdl Bounded Model Checker (VBMC): A Formal Verification Tool for VHDL Designs

Ajith John, A. K. Bhattacharjee
RCnD, BARC

Supratik Chakraborty, CFDVS, IIT Bombay
Introduction

- Design of modern computer based systems involves partitioning of system into *hardware subsystems* and *software subsystems*. 
Design of modern computer based systems involves partitioning of system into **hardware subsystems** and **software subsystems**.

**Hardware subsystems** are often implemented in Hardware Description Languages (HDL) such as **VHDL**, Verilog etc.
Introduction

- Design of modern computer based systems involves partitioning of system into *hardware subsystems* and *software subsystems*

- *Hardware subsystems* are often implemented in Hardware Description Languages (HDL) such as *VHDL*, Verilog etc.

The programs written in *VHDL* are converted to hardware realizations in *FPGA* through a sequence of transformations
Introduction

- Design of modern computer based systems involves partitioning of system into **hardware subsystems** and **software subsystems**

- **Hardware subsystems** are often implemented in Hardware Description Languages (HDL) such as **VHDL**, Verilog etc.

- The programs written in **VHDL** are converted to hardware realizations in **FPGA** through a sequence of transformations

- **Correctness** of the final **FPGA** implementation depends on the **correctness of** the **VHDL** program
Introduction

- Design of modern computer based systems involves partitioning of system into hardware subsystems and software subsystems.

- Hardware subsystems are often implemented in Hardware Description Languages (HDL) such as VHDL, Verilog etc.

- The programs written in VHDL are converted to hardware realizations in FPGA through a sequence of transformations.

- Correctness of the final FPGA implementation depends on the correctness of the VHDL program.

- Ensuring the correctness (verification) of VHDL programs extremely important when used in safety-critical applications.
Conventional Verification: Simulation

Specifications → VHDL Program → Synthesis → Place & Route → FPGA
Conventional Verification: Simulation

Specifications → VHDL Program

Simulation (ModelSim, NCSim)

Exhaustive simulation covering all possible input combinations impossible in practice

Synthesis → Place & Route → FPGA
Conventional Verification: Simulation

Simulation (ModelSim, NCSim)

Specifications

VHDL Program

Synt hesis

Place & Route

FPGA

Exhaustive simulation covering all possible input combinations impossible in practice

Specifications

VHDL Program

output

32-bit input

2^{32} \times 2^{32} = 2^{64} \text{ possible input combinations}!!
**Conventional Verification: Simulation**

- We cannot use exhaustive simulation to **prove** the functional correctness of the program.

Exhaustive simulation covering all possible input combinations is impossible in practice.

Specifications → VHDL Program → Synthesis → Place & Route → FPGA

- Exhaustive simulation covering all possible input combinations is impossible in practice.

32-bit input → VHDL Program → output

\[ 2^{32} \times 2^{32} = 2^{64} \text{ possible input combinations} \]
**Formal Verification**

- We cannot use exhaustive simulation to *prove* the functional correctness of the program.

- Exhaustive simulation covering all possible input combinations is impossible in practice.

- Formal verification *proves / refutes* the functional correctness of the program using *rigorous mathematical techniques*.
Vhdl Bounded Model Checker

- Indigenously developed tool for formal verification of VHDL designs

Simulation (ModelSim, NCSim)

Specifications

VHDL Program

Synt hesis

Place & Route

FPGA

Exhaustive simulation covering all possible input combinations impossible in practice

VBMC
**Vhdl Bounded Model Checker**

- Indigenously developed tool for formal verification of VHDL designs

1. Specifications
2. VHDL Program
3. Synthesis
4. Place & Route
5. FPGA

- Simulation (ModelSim, NCSim)

Exhaustive simulation covering all possible input combinations impossible in practice

Expressed as Formal Specification (Property)
**Vhdl Bounded Model Checker**

- Indigenously developed tool for formal verification of VHDL designs

---

**Specifications** → **VHDL Program** → **Synthesis** → **Place & Route** → **FPGA**

- Simulation (ModelSim, NCSim)

- Exhaustive simulation covering all possible input combinations impossible in practice

- Expressed as Formal Specification (Property)
  - Bound K (integer)
  - (number of clock cycles)
Vhdlg Bounded Model Checker

Indigenously developed tool for formal verification of VHDL designs

- **Simulation (ModelSim, NCSim)**
  - Exhaustive simulation covering all possible input combinations impossible in practice

- **Specifications**
- **VHDL Program**
- **VBMC**
  - Expressed as Formal Specification (Property)
  - Bound K (integer)
    - (number of clock cycles)
  - Two possible outcomes

- **Synthesis**
- **Place & Route**
- **FPGA**
VHDL Bounded Model Checker

- Indigenously developed tool for formal verification of VHDL designs

Simulation (ModelSim, NCSim)

Specifications

VHDL Program

Syntesis

Place & Route

FPGA

VBMC

Exhaustive simulation covering all possible input combinations impossible in practice

Property satisfied for K clock cycles/

Two possible outcomes

Expressed as Formal Specification (Property)

Bound K (integer) (number of clock cycles)
**Vhdl Bounded Model Checker**

- Indigenously developed tool for formal verification of VHDL designs

**Specifications** → **VHDL Program** → **Synthesis** → **Place & Route** → **FPGA**

**Simulation (ModelSim, NCSim)**

- Exhaustive simulation covering all possible input combinations impossible in practice

**VBMC**

- Property satisfied for $K$ clock cycles
- Property violated with Counterexample (execution of program violating property)

**Expressed as Formal Specification (Property)**

- Bound $K$ (integer) (number of clock cycles)
VBMC: Internals

VHDL program → VBMC

Property → VBMC

Bound, K → VBMC

Property satisfied/

Property violated with Counterexample (execution of program violating property)
Property satisfied/

Property violated with Counterexample (execution of program violating property)

Steps inside VBMC

- Generation of transition relation
- Bounded model checking
- Transition relation abstraction
Generation of Transition Relation

- Transition relation: relation between present and next states (R) of the program
Generation of Transition Relation

- Transition relation: relation between present and next states (R) of the program

- Using **symbolic simulation**: Involves simulation of the VHDL program with symbols as inputs, rather than concrete values
Generation of Transition Relation

VHDL program

```
input d; output t;
entity register is
    signal p;
    ....
    process(clk)
    if(clk'event)
        p<=d;
    process(clk)
    if(clk'event)
        t<=p;
    ....
end register;
```

- Transition relation: relation between present and next states (R) of the program
- Using symbolic simulation: Involves simulation of the VHDL program with symbols as inputs, rather than concrete values
**Generation of Transition Relation**

Transition relation: relation between present and next states (R) of the program

Using **symbolic simulation**: Involves simulation of the VHDL program with symbols as inputs, rather than concrete values

VHDL program

```vhdl
input d; output t;
entity register is
  signal p;
  ....
  process(clk)
    if(clk'event)
      p<=d;
  process(clk)
    if(clk'event)
      t<=p;
  ....
end register;
```

Transition relation $R(state_0, state_1)$

- If $clk_1 \neq clk_0$, $p_1 = d_1$, $p_1 = p_0$
- $\land$
- If $then_else(clk_1 \neq clk_0, t_1 = p_0$, $t_1 = t_0$)
Generation of Transition Relation

VHDL program

input d; output t;
entity register is
  signal p;
  ....
  process(clk)
  if(clk'event)
    p<=d;
  end process;
  process(clk)
  if(clk'event)
    t<=p;
  end process;
end register;

- Transition relation: relation between present and next states (R) of the program
- Using **symbolic simulation**: Involves simulation of the VHDL program with symbols as inputs, rather than concrete values

Transition relation R(state_0, state_1)

\[
\text{if}_\text{then}_\text{else}(clk_1 \neq clk_0, p_1 = d_1, p_1 = p_0) \land \\
\text{if}_\text{then}_\text{else}(clk_1 \neq clk_0, t_1 = p_0, t_1 = t_0)
\]
**Generation of Transition Relation**

**VHDL program**

```
input d; output t;
entity register is
    signal p;
    ...
process(clk)
    if(clk'event)
        p<=d;
    process(clk)
    if(clk'event)
        t<=p;
    end register;
```

- Transition relation: relation between present and next states (R) of the program
- Using **symbolic simulation**: Involves simulation of the VHDL program with symbols as inputs, rather than concrete values

**Transition relation** $R(\text{state}_0, \text{state}_1)$

```
if_then_else(clk_1 \neq clk_0, p_1=d_1, p_1=p_0)
\land
if_then_else(clk_1 \neq clk_0, t_1=p_0, t_1=t_0)
```
Bounded Model Checking

Check if the property is violated in k (bound) clock cycles

Step-1: Find the states reachable in k clock cycles
Bounded Model Checking

Check if the property is violated in k (bound) clock cycles

Step-1: Find the states reachable in k clock cycles

Formula for initial states; i.e, state\(_0\) represents all initial states

- Formula I derived from the property
**Bounded Model Checking**

Check if the property is violated in $k$ (**bound**) clock cycles

Step-1: Find the states reachable in $k$ clock cycles

\[ l(state_0) \land R(state_0, state_1) \]

**Transition Relation**: Relation between present state $state_0$ and next state $state_1$ reachable in 1 clock cycle from initial states

- Formula $I$ derived from the property
**Bounded Model Checking**

Check if the property is violated in k (bound) clock cycles

Step-1: Find the states reachable in k clock cycles

\[ I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \land \ldots \land R(\text{state}_{k-1}, \text{state}_k) \]

- **Transition Relation:** Relation between present state \( \text{state}_0 \) and next state \( \text{state}_1 \)

- **Formula I derived from the property**

- **reachable in k clock cycles from initial states**
**Bounded Model Checking**

Check if the property is violated in $k$ (bound) clock cycles

Step-1: Find the states reachable in $k$ clock cycles

\[ l(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \]

Step-2: Check if a bad state $\neg$Property is reachable in $k$ clock cycles
**Bounded Model Checking**

Check if the property is violated in k (bound) clock cycles

Step-1: Find the states reachable in k clock cycles

\[ I(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \land \neg\text{Property} \]

Step-2: Check if a bad state \( \neg\text{Property} \) is reachable in k clock cycles
**Bounded Model Checking**

Check if the property is violated in k (bound) clock cycles

**Step-1:** Find the states reachable in k clock cycles

**Verification Condition**

\[ I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \land \ldots \land R(\text{state}_{k-1}, \text{state}_k) \land \neg \text{Property} \]

**Step-2:** Check if a bad state \( \neg \text{Property} \) is reachable in k clock cycles
**Bounded Model Checking**

Check if the property is violated in k (bound) clock cycles

**Step-1: Find the states reachable in k clock cycles**

Verification Condition

\[ I(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \land \neg \text{Property} \]

- **Solution of Verification Condition** => Execution violating Property

- **Verification Condition has no solution** => No execution violating Property => Program satisfies property for K clock cycles

**Step-2: Check if a bad state \(\neg \text{Property}\) is reachable in k clock cycles**
**Bounded Model Checking**

Check if the property is violated in $k$ (bound) clock cycles

**Step-1:** Find the states reachable in $k$ clock cycles

Verification Condition

$$I(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \land \neg Property$$

**Step-2:** Check if a bad state $\neg Property$ is reachable in $k$ clock cycles

No solutions: design safe up to $k$ cycles of operation

Solution: counterexample

No solutions: design safe up to $k$ cycles of operation

**SMT Solver**

Step-2: Check if a bad state $\neg Property$ is reachable in $k$ clock cycles
Simulation Vs Bounded Model Checking

Consider a VHDL program and Property

Initial states: I

Buggy states: ~Property
Simulation Vs Bounded Model Checking

Consider a VHDL program and Property

Initial states: I

Buggy states: ~Property

Counterexample trace
Simulation

Consider a VHDL program and Property

Counterexample is missed in simulation

Initial states: \( I \)

Buggy states: \( \sim \text{Property} \)

Counterexample trace

Trace followed in simulation
Bounded Model Checking

\( I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \)

Initial states: \( I \)

Buggy states: \( \neg \text{Property} \)

Counterexample trace

\( \text{state}_0 \): initial states
\( \text{state}_1 \): all states reachable in 1 clock cycle from initial states
Bounded Model Checking

\[ I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \land R(\text{state}_1, \text{state}_2) \]

Initial states: \( I \)

Buggy states: \( \neg \text{Property} \)

Counterexample trace: \( \text{state}_0 \): initial states

\( \text{state}_2 \): all states reachable in 2 clock cycles from initial states
Bounded Model Checking

\[ I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \land R(\text{state}_1, \text{state}_2) \land R(\text{state}_2, \text{state}_3) \]

- **Initial states:** \( I \)
- **Buggy states:** \( \neg \text{Property} \)
- **Countereexample trace**
- **\( \text{state}_0 \):** initial states
- **\( \text{state}_3 \):** all states reachable in 3 clock cycles from initial states
Bounded Model Checking

\[ I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \land R(\text{state}_1, \text{state}_2) \land R(\text{state}_2, \text{state}_3) \land R(\text{state}_3, \text{state}_4) \]

Initial states: \( I \)

Buggy states: \( \sim \text{Property} \)

Counterexample trace

\( \text{state}_0 \): initial states
\( \text{state}_4 \): all states reachable in 4 clock cycles from initial states
**Bounded Model Checking**

\[ l(state_0) \land R(state_0, state_1) \land R(state_1, state_2) \land R(state_2, state_3) \land R(state_3, state_4) \land \neg \text{Property} \]

*Initial states: I*

*Buggy states: \( \neg \text{Property} \)*

*Counterexample is found by BMC*
Transition Relation Abstraction

- Optional optimization step
Transition Relation Abstraction

- Optional optimization step
- Time to solve exponential in number of variables

\[ I(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \land \neg \text{Property} \]
Transition Relation Abstraction

- Optional optimization step
- Time to solve exponential in number of variables

\[ I(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \land \sim \text{Property} \]

- Can we simplify R intelligently?
**Transition Relation Abstraction**

- Optional optimization step
- Time to solve exponential in number of variables

\[ I(\text{state}_0) \land R(\text{state}_0, \text{state}_1) \land \ldots \land R(\text{state}_{k-1}, \text{state}_k) \land \neg \text{Property} \]

- Can we simplify R intelligently? Do transition relation abstraction
- Eliminate variables from R so that it has fewer variables
Transition Relation Abstraction

- Optional optimization step
- Time to solve exponential in number of variables

\[ I(state_0) \land R(state_0, state_1) \land \ldots \land R(state_{k-1}, state_k) \land \sim \text{Property} \]

- Can we simplify R intelligently? Do transition relation abstraction
- Eliminate variables from R so that it has fewer variables

- We use a novel algorithm for transition relation abstraction – to generate R', simplified (abstract) transition relation
Can we simplify $R$ intelligently? Do transition relation abstraction
- Eliminate variables from $R$ so that it has fewer variables
- We use a novel algorithm for transition relation abstraction – to generate $R'$, simplified (abstract) transition relation
Verification Steps using VBMC

1. FPGA Requirements Specification
2. Find specifications to be checked
3. Express as properties
4. Set Verification Bound (K) = Number of Clock Cycles
   - Property is not Satisfied
     - Analyse counterexample
     - Review program, property
   - Property is Satisfied for K cycles
5. Increase K, until resources are exhausted
Verification Steps using VBMC

1. FPGA Requirements Specification
2. Find specifications to be checked
3. Express as properties
4. Set Verification Bound (K) = Number of Clock Cycles
5. Property is not Satisfied
   - Analyse counterexample
   - Review program, property
6. Increase K, until resources are exhausted
7. Property is Satisfied for K cycles
   - Manually done
Other Formal Verification Tools

Existing Formal Verification Tools:

- Commercial: JasperGold, Incisive (Cadence), Questa (Mentor Graphics)
- Academic: SMV, NuSMV (CMU), VIS, ABC (Berkely), PdTrav (Politecnico di Torino)
Other Formal Verification Tools

Existing Formal Verification Tools:

- Commercial: JasperGold, Incisive (Cadence), Questa (Mentor Graphics)
- Academic: SMV, NuSMV (CMU), VIS, ABC (Berkeley), PdTrav (Politecnico di Torino)

Problem with existing academic tools:

- Take input in gate-level (bit-level) formats like AIGs
- Need to convert the RTL-level (word-level) constructs (eg: +, -) in the program to gate-level and perform verification at gate-level
- Scaling issues for verification of designs with wide data paths
Word-level constructs in the VHDL program are kept at word-level and verification at word-level (**word-level verification**)

*Transition relation abstraction* generates word-level abstractions
VBMC

- Word-level constructs in the VHDL program are kept at word-level and verification at word-level (word-level verification)

- Transition relation abstraction generates word-level abstractions

- Scalable for verification of VHDL programs with wide data-paths
Applications of VBMC

- Formal verification of VHDL programs used in boards developed at RCnD, ED in BARC and in NPCIL
Limitations of VBMC

- Supports only a subset of VHDL: Includes almost all synthesizable constructs

- Provides only bounded guarantee: Design is safe up to $K$ clock cycles, No guarantee after $K$ clock cycles
Publications


- Design and Application of a Formal Verification Tool for VHDL Designs, *In BARC Newsletter, ISSN: 0976-2108, 2012*

- A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations in *Computer Aided Verification (CAV) 2011*

- Extending Quantifier Elimination to Linear Inequalities on Bit-Vectors in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2013*

- Quantifier Elimination for Linear Modular Constraints in *International Congress on Mathematical Software (ICMS) 2014*

Thank you

Demo @ 3:30 pm