BARC/2014/E/010

BARC/2014/E/010

VBMC: A FORMAL VERIFICATION TOOL FOR VHDL PROGRAMS

by
K.J. Ajith and A.K. Bhattacharjee
Reactor Control Division

BARC/2014/E/010

BARC/2014/E/010

GOVERNMENT OF INDIA
ATOMIC ENERGY COMMISSION

VBMC: A FORMAL VERIFICATION TOOL FOR VHDL PROGRAMS

by
K.J. Ajith and A.K. Bhattacharjee
Reactor Control Division

BHABHA ATOMIC RESEARCH CENTRE
MUMBAI, INDIA
2014

BIBLIOGRAPHIC DESCRIPTION SHEET FOR TECHNICAL REPORT
(as per IS : 9400 - 1980)

01	*Security classification :*	Unclassified
02	*Distribution :*	External
03	*Report status :*	New
04	*Series :*	BARC External
05	*Report type :*	Technical Report
06	*Report No. :*	BARC/2014/E/010
07	*Part No. or Volume No. :*	
08	*Contract No. :*	
10	*Title and subtitle :*	VBMC: a formal verification tool for VHDL program
11	*Collation :*	30p., 6 figs., 2 tabs.
13	*Project No. :*	
20	*Personal author(s) :*	Ajith K.J.; A.K. Bhattacharjee
21	*Affiliation of author(s) :*	Reactor Control Division, Bhabha Atomic Research Centre, Mumbai
22	Corporate author(s) :	Bhabha Atomic Research Centre, Mumbai - 400 085
23	*Originating unit :*	Reactor Control Division, Bhabha Atomic Research Centre, Mumbai
24	*Sponsor(s) Name :* *Type :*	Department of Atomic Energy Government

Contd...

30	*Date of submission :*	July 2014
31	*Publication/Issue date :*	August 2014
40	*Publisher/Distributor :*	Head, Scientific Information Resource Division, Bhabha Atomic Research Centre, Mumbai
42	*Form of distribution :*	Hard copy
50	*Language of text :*	English
51	*Language of summary :*	English, Hindi
52	*No. of references :*	24 refs.
53	*Gives data on :*	
60	*Abstract :* The design of Control and Instrumentation (C & I) systems used in safety critical applications such as nuclear power plants involves partitioning of the overall system functionality into sub-parts and implementing each sub-part in hardware and/or software as appropriate. With increasing use of programmable devices like FPGA, the hardware subsystems are often implemented in Hardware Description Languages (HDL) like VHDL. Since the functional bugs in such hardware subsystems used in safety critical C & I systems have serious consequences, it is important to use rigorous reasoning to verify the functionalities of the HDL models. This report describes the design of a software tool named VBMC (VHDL Bounded Model Checker). The capability of this tool is in proving/refuting functional properties of hardware designs described in VHDL. VBMC accepts design as a VHDL program file, functional property in PSL, and verification bound (number of cycles of operation) as inputs. It either reports that the design satisfies the functional property for the given verification bound or generates a counterexample providing the reason of violation. In case of satisfaction, the proof holds good for the verification bound. VBMC has been used for the functional verification of FPGA based intelligent I/O boards developed at Reactor Control Division, BARC.	
70	*Keywords/Descriptors :* NUCLEAR FACILITIES; SAFETY; REACTOR OPERATION; DESIGN; ON-LINE CONTROL SYSTEMS; PROGRAMMING LANGUAGES	
71	*INIS Subject Category :*	S22
99	*Supplementary elements :*	

सारांश

सुरक्षा महत्वपूर्ण अनुप्रयोगों में उपयोग होने वाली नियंत्रण और यांत्रिकरण योजनाओं की रचना अक्सर योजना की समग्र प्रणाली की कार्यक्षमता का विभाजन उप भागों में कर प्रत्येक उप भाग को उपयुक्त हार्डवेयर और / या सॉफ्टवेयर में लागू करने के द्वारा होती है । एफ. पी. जी .ए **(FPGA)** जैसे प्रोग्राम योग्य उपकरणों के बढ़ते प्रचलन के फलस्वरूप हार्डवेयर उपयंत्र अक्सर वी.एच.डी.एल **(VHDL)** जैसी हार्डवेयर वर्णन भाषाओं (Hardware Description **Language)** में लागू किये जाते हैं । सुरक्षा महत्वपूर्ण अनुप्रयोगों में इस्तेमाल होने वाले इन हार्डवेयर उपयंत्र में कार्यात्मक गलतियाँ के परिणाम गंभीर होते हैं, अतः इन एच.डी.एल (HDL) मॉडल की कार्यक्षमता सत्यापित करने के लिए सख्त तर्क का उपयोग करना महत्वपूर्ण है । यह विवरणी **VBMC** नामित सॉफ्टवेयर उपकरण की डिजाइन का वर्णन करती है । इस उपकरण की क्षमता **VHDL** में वर्णित हार्डवेयर डिजाइन की कार्यात्मक गुणों को नकारने अथवा सत्यापित करने में है । **VBMC** एक **VHDL** प्रोग्राम फाइल को रचना स्वरुप, **PSL** में लिखित कार्यात्मक गुण एवं सत्यापन सीमा (आपरेशन के चक्रों की संख्या) को इन्पुट के रूप में लेता है । **VBMC** बताता है की क्या यह वर्णन सत्यापन सीमा के दौरान कार्यात्मक गुण को संतुष्ट करता है, अथवा जवाबी उदाहरण उत्पन्न करता है जो उल्लंघन के कारण बताता है । कार्यात्मक गुण संतुष्ट होने की स्तिथि में सबूत बाध्य सत्यापन सीमा के लिए वैद्य रहता है ।

Abstract

The design of Control and Instrumentation (C & I) systems used in safety critical applications such as nuclear power plants involves partitioning of the overall system functionality into sub-parts and implementing each sub-part in hardware and/or software as appropriate. With increasing use of programmable devices like FPGA, the hardware subsystems are often implemented in Hardware Description Languages (HDL) like VHDL. Since the functional bugs in such hardware subsystems used in safety critical C&I systems have serious consequences, it is important to use rigorous reasoning to verify the functionalities of the HDL models. This report describes the design of software tool named VBMC (VHDL Bounded Model Checker). The capability of this tool is in proving/refuting functional properties of hardware designs described in VHDL. VBMC accepts design as a VHDL program file, functional property in PSL, and verification bound (number of cycles of operation) as inputs. It either reports that the design satisfies the functional property for the given verification bound or generates a counterexample providing the reason of violation. In case of satisfaction, the proof holds good for the verification bound. VBMC has been used for the functional verification of FPGA based intelligent I/O boards developed at Reactor Control Division, BARC.

Table of Contents

VBMC: A Formal Verification Tool for VHDL Programs

Ajith K.J., A. K. Bhattacharjee*,
Reactor Control Division,
Bhabha Atomic Research Centre,
Mumbai 400 085
***email: anup@barc.gov.in**

1. Introduction

Computer based Control and Instrumentation (C&I) systems used in nuclear power plants are mandated to undergo a rigorous verification process commensurate with the safety class of the system. The design of such systems involves partitioning of the overall system functionality into subsystems, and implementing each subsystem in hardware and/or software as appropriate. The recent generation of hardware components include designs involving field programmable devices commonly known as FPGA. Such devices are often implemented in Hardware Description Languages (HDL) like VHDL[1,2]. The designer starts with a description of the subsystem in HDL which is then systematically transformed into a hardware realization in FPGA or CPLD using a chain of electronic design automation tools. Hence, the correctness of the hardware subsystems realized in FPGA crucially depends on the correctness of the HDL design designer started with. Since the functional bugs in such hardware subsystems used in C&I of safety critical systems have serious consequences related to safety, it is important to use rigorous techniques to verify the functionalities of the HDL designs.

Traditionally, the functional verification of HDL designs has been done by simulation or testing. However, exhaustive simulation covering all possible input combinations of the design is impractical for HDL designs of even medium size and complexity. Hence, simulation-based tools *can never* ensure that the design is tested for all corner cases of the input space. In contrast to this, formal verification tools exhaustively explore all possible input combinations of the design in an attempt to prove that the design satisfies the functional specifications. It is now recognized that formal verification tools provide a complementary approach to simulation (i) by finding corner case bugs that simulation may miss and (ii) in proving that design satisfies the functional specifications.

HDL designs fall into two different domains: *Control-dominated* and *Data-dominated*. A common definition of a "control-dominated" design is one which is realized by interacting finite state machines e.g. memory controller hardware. On the other side “data-dominated” designs like digital filters involve multi-bit data and word-level operations on the data.

The formal verification techniques for data-dominated designs differ significantly from those used for control-dominated designs. Control-dominated designs are normally verified using model checking. Model checking involves an exhaustive search of the entire state space of the design to ensure that the design can never be in an undesirable state. Model checking is typically fast and fully automatic. Unfortunately, the number of states in the state spaces of real-world designs is prohibitively large particularly due to the wide data paths present in the designs. This restricts the scope of model checking techniques in formal verification of data-dominated designs. Data-dominated designs are normally verified using proof-based approaches that focus on mathematically proving properties of the designs using proof engines called Theorem Provers. Theorem proving often requires human insight and creativity to complete proofs and is not fully automatic.

Real-life designs are a mix of both control and data dominated sub-parts and it is very difficult to separate them. Hence, it is usually left to the verification engineer to carve out suitable parts of the design that are then subjected to appropriate verification techniques. This makes the process tedious, error-prone, and highly dependent on the ingenuity and skills of the verification engineer. Hence, it is important to develop automated verification tools which can verify HDL designs with both control-dominated and data-dominated sub-parts.

As an example to HDL designs with both control-dominated and data-dominated sub-parts, consider a process controller containing several digital signal processing units, arithmetic logic units, and also a bus controller that controls how a shared bus is used by the different devices that communicate with the process controller. Note that the bus controller is control-dominated, whereas the digital signal processing units and the arithmetic logic units are data-dominated. There are several interesting properties one might want to check about this controller. Some properties may simply check whether the transactions on the bus happen as per a pre-determined protocol. Other properties could check whether the signal processing functions are being computed correctly. While the first property is control-oriented, the second one is highly data-oriented.

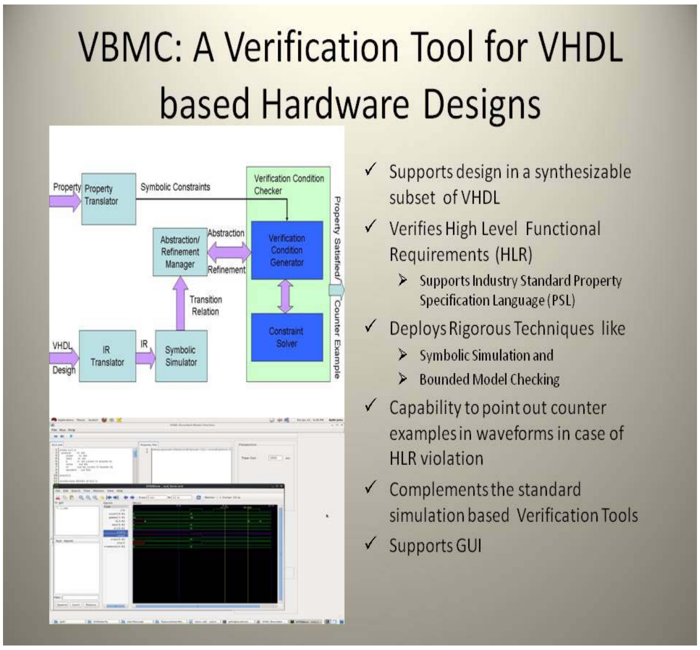

Fig. 1: Overview of VBMC

VBMC is an automated tool for mathematically proving/refuting functional properties of hardware designs with both control-dominated and data-dominated sub-parts described in VHDL. The basic features are shown in Fig. 1. It is based on the principle of bounded model checking. In bounded model checking, the model checker looks for a violation of the stated property while exploring a depth of k (integer) of the state space. If no bug is found in this depth of k, then one can increase k until (i) violation is found, (ii) the entire space is searched, or (iii) the computation time becomes prohibitive. In case of (ii), it has been proved that the design does not violate the property under all possible executions.

1.1 Motivation

The main motivation behind this development conforms to requirements of rigorous verification akin to software as per AERB SG-D25[3]. It has been recommended in D-25 to treat HDL based designs at the same rigour as software for the system safety class. Additionally, the Review Guidelines for Field-Programmable Gate Arrays in Nuclear Power Plant Safety Systems prepared by US NRC[4] have identified the following issues related to usage of FPGAs in design of safety systems:

1. FPGA devices are fundamentally complex *software* designs implemented by hardware engineers using HDL.
2. The main issue in achieving sufficient confidence in HDL based design arises from the complexity in hardware design life cycle and inadequate V&V process to account for the specific *software* characteristics of FPGAs. This issue is to be addressed taking difficulty in understanding the semantics of HDLs and analyzing the safety aspects of the design.
3. Even a relatively simple FPGA design in most cases prohibits 100% testing during code simulation and hardware verification. The guide recommends to treat FPGA-based safety systems as complex systems (ref: DO-254[5]), and argues to identify suitable design cycle including rigorous V&V process for FPGA-based safety-related designs.

The situation is complicated by the fact that at present, verifiers have no alternate means to verify a HDL design with respect to high level requirements. The motivation behind development of VBMC is to partially address the above concerns and is summarized below.

- ✓ An alternate verification strategy for high level requirements
- ✓ The tool is based on a technique which is not commonly used by traditional simulation tools and hence addresses common mode problem with design and verification.
- ✓ It addresses cycle accurate representation of HDL semantics.

1.2 State of art in HDL Verification

Most of the verification tools offered by FPGA manufactures and EDA tool vendors are simulation-based tools. Xilinx ISE Simulator[6], MentorGraphics ModelSim[7] and Cadence NCSim[8] are examples of simulation-based tools. Some vendors also offer formal equivalence checking tools. Examples of equivalence checkers include Synopsis Formality[9] , MentorGraphics FormalPro[7] and Cadence Conformal[8] . These tools focus on proving the equivalence between two different forms of a hardware design. For example, an equivalence checker can be used to prove that the netlist obtained after synthesis of an RTL design is functionally equivalent to the RTL design. However equivalence checkers cannot be used for formal property verification.

Formal property verification tools in the commercial domain include FormalCheck, Incisive Formal Verifier from Cadence, and Questa Formal Verifier from MentorGraphics. SMV [10], NuSMV [11], and VIS[12] are examples of academic formal property verification tools. However most of these tools operate at bit-level which poses serious scaling issues when reasoning about designs with wide data paths. Moreover all of these tools are based on unbounded model checking which is found to be practically less scalable compared to bounded model checking which VBMC uses. VCEGAR[13] is an academic tool for formal verification of Verilog HDL programs which works at word-level. VCEGAR is based on an idea called predicate abstraction. The identification of right set of predicates is crucial in the successful application of this tool, which is an undecidable problem in general.

The remaining part of this report is organized as follows. Chapter 2 gives the overview of VBMC, and describes the subset of VHDL and PSL it accepts. Chapter 3 focuses on the major technical contributions behind the design of VBMC and internals of VBMC. Chapter 4 concludes the report with some applications of VBMC and future work.

2. Overview of VBMC

VBMC accepts a VHDL design, a functional property in Property Specification Language (PSL), and verification bound (number of cycles of operation) as inputs (see Fig. 2). It either infers that the design satisfies the functional property for the given verification bound or generates an execution of the design violating the property (called counterexample).

Note that in the former case, VBMC provides a guarantee that the design satisfies the property for a bounded number of cycles of operation. However, as discussed above, if no counterexample is found with this bound, then one can increase the bound until (i) a counterexample is found, (ii) the entire space is searched, or (iii) the computation time becomes prohibitive. In case of (ii), it has been proved that the design does not violate the property under all possible executions.

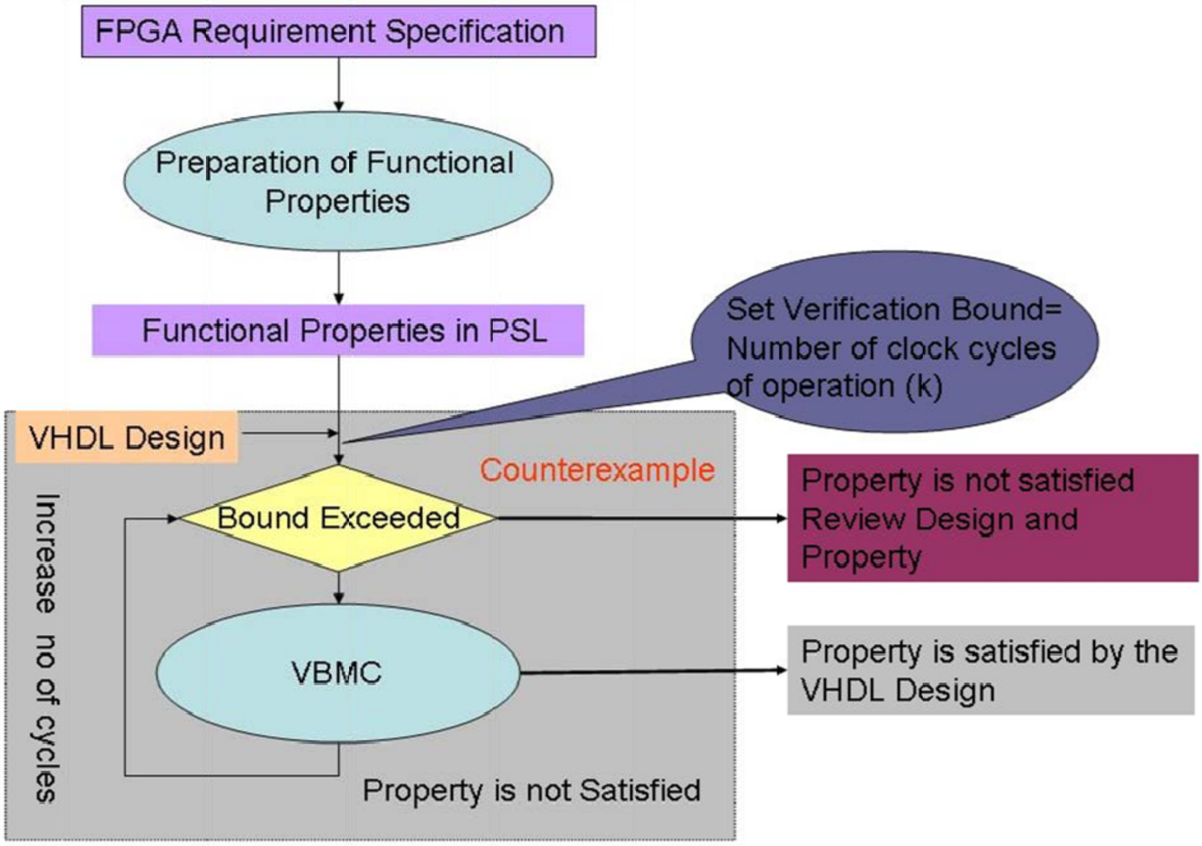

Fig. 2: Verification Steps and Flow

The counterexample generated by VBMC is displayed in the form of waveforms for input and output signals. An example screen-shot of such a waveform is shown in Fig. 3.

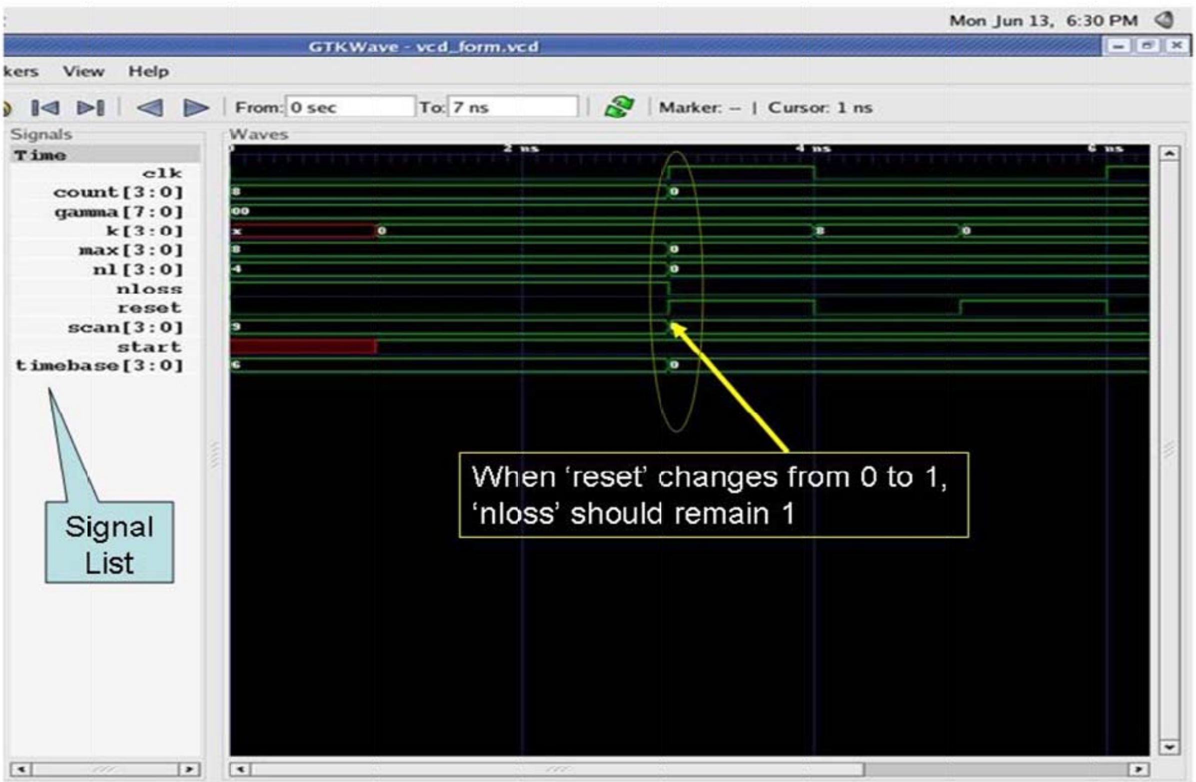

Fig. 3: Counterexample in waveforms generated by VBMC

2.1 Subset of VHDL accepted by VBMC

VHDL offers a rich collection of constructs. However, a large subset of these constructs offered by VHDL is not synthesizable[1]. Such constructs can be used only for modelling and simulation. VBMC does not support any of the non-synthesizable constructs. Here is a list of such constructs (the details of these constructs can be found in[2]).

1) access types
2) allocators
3) assertions
4) disconnection specifications
5) files
6) shared variables
7) groups
8) delays
9) floating-point types
10) physical types
11) incomplete types
12) report statements

Among the synthesizable constructs, VBMC currently accepts a large subset of the synthesizable constructs. Following is the list of the synthesizable constructs that are not supported by VBMC.

1) aliases
2) block statements
3) user-defined attributes: VBMC supports only the predefined synthesizable attribute 'EVENT.
4) loops, exit statements, and next statements
5) subprograms and return statements
6) packages
7) records
8) wait statements
9) overloading
10) arrays
11) variables
12) restrictions on ports: VHDL allows four types of ports – IN, OUT, INOUT, BUFFER. VBMC supports only IN ports and OUT ports.
13) restrictions on vector declarations: VBMC requires the vector declarations to be either 'N-1 DOWNTO 0' or '0 TO N-1', where N is the size of the vector.

2.2 Property Specification Language for VBMC

The properties accepted by VBMC are in a subset of PSL[14], extended with bounds. The syntax and semantics of the properties VBMC accepts is summarized in Table 1.

Table 1: PSL for VBMC

Property syntax	Property semantics
(*predicate***)**	Predicate is a Boolean expression. **(***predicate***)** asserts that the given Boolean expression is true in the first clock cycle. The permitted operators inside *predicate* are (1) Relational operators: **=, /=, <, <=, >, >=,** (2) Shift operators: sll, srl (3) Arithmetic, selection, and concatenation operators: +, -, *, [:], & (4) Bit-wise operators: bv-and (bit wise and), bv-or (bit-wise or), bv-not (bit-wise not). **Example**: The property (z=x+y) asserts that *In the first clock cycle, the value of the signal z is the sum of the values of the signals x and y*.
(next *formula***)**	Asserts that *formula* is true in the next clock cycle. **Example**: The property (next(z=x+y)) asserts that *In the second clock cycle, the value of the signal z is the sum of the values of the signals x and y*.
(next[*I***]** *formula***)**	Asserts that *formula* is true in the next I^{th} clock cycle. **Example**: The property (next[3](z=x+y)) asserts that *In the fourth clock cycle, the value of the signal z is the sum of the values of the signals x and y*.
(next_a[*I***:***J***]** *formula***)**	Asserts that *formula* is true in **<u>all</u>** cycles of a range of future cycles.

	Example: The property (next_a[1:3](z=x+y)) asserts that *From the second clock cycle through the fourth clock cycle, the value of the signal z is the sum of the values of the signals x and y*.
(next_e[*I*:*J*] *formula*)	Asserts that *formula* is true **<u>at least once</u>** within some range of future cycles. **Example**: The property (next_e[1:3](z=x+y)) asserts that *Starting from the second clock cycle until the fourth cycle, the value of the signal z is the sum of the values of the signals x and y, at least once.*
(always *formula* **[*I*])**	Asserts that *formula* is true in this cycle and is true in **<u>all</u>** of the next I-1 cycles. **Example**: The property (always (z=x+y) [3]) asserts that *The value of the signal z is the sum of the values of the signals x and y, from the first clock cycle through the third cycle.*
(eventually *formula* **[*I*])**	Asserts that *formula* is true either in this cycle or in **<u>at least one</u>** of the next I-1 cycles. **Example**: The property (eventually (z=x+y) [3]) asserts that *The value of the signal z is the sum of the values of the signals x and y, at least once, starting from the first clock cycle through the third cycle.*
(never *formula* **[*I*])**	Asserts that *formula* is **<u>false</u>** in this cycle and is **<u>false</u>** for **<u>all</u>** of the next I-1 cycles. **Example**: The property (never (z=x+y) [3]) asserts that *The value of the signal z is not equal to the sum of the values of the signals x and y, from the first clock cycle through the third cycle.*

(! *formula***)**	Asserts that *formula* is false. **Example**: The property (!(next[2](z=x+y))) asserts that *In the third clock cycle, the value of the signal z is* ***not equal to*** *the sum of the values of the signals x and y*.
(*formula*$_1$ **bin-op** *formula*$_2$**)**	This helps in connecting two formulas using a binary Boolean connective **bin-op. bin-op** here can be and, or, not, nand, nor, xor, xnor , with meaning indicated by the name. **Example**: The property ((z=x+y) and (next[2](x=1))) asserts that *In the first clock cycle, the value of the signal z is the sum of the values of the signals x and y, and in the third clock cycle, x is 1*.
(*formula*$_1$ **;** *formula*$_2$**)**	This is equivalent to **(***formula*$_1$ **and next(***formula*$_2$**))**
(*formula***) with** *signal*	This asserts that *formula* is true, and *signal* is the synchronizing signal (or clock signal). ***Example****: The property* *(always(next[1](err=0))[10]) with clk* asserts that the value of the signal *err* is 0 from the second clock cycle through the 11th clock cycle (i.e. for 10 clock cycles) with *clk* as the clock signal.

2.3 Example

This section explains the working of VBMC with the help of an example VHDL design and a property.

Consider the VHDL design shown in Fig. 4. The design comprises a controller with three states – 0, 1, and 2. In state 0, the controller checks if *start* is 1. If *start* is 1, the controller switches to state 1. Otherwise it remains in state 0. In state 1, the controller increments a variable *count* until *count* reaches 0x1000. Once *count* reaches 0x1000, *flag* is set to 1 and the controller switches to state 2. Once the controller reaches state 2, it stays there permanently.

Suppose one is interested in proving the property "*flag* is 0 as long as *start* is 0" for the first 10 clock cycles of operation. This property written in PSL using the syntax and semantics illustrated in Table 1 is shown in Equation 1.

```
entity detect is
 port
  (        reset : in std_logic;
      clock: in std_logic;
      start           : in std_logic;
      flag : out std_logic
      );
end detect;
architecture detect of detect is
 signal state : std_logic_vector(3 downto 0);
 signal count : std_logic_vector(15 downto 0);

begin
 process( clock, reset)
 begin
  if reset = '0' then
    flag <= '0';
    state<=  x"0";
    count <= x"0000";

  elsif clock'event and clock = '1' then
    case state is
     when x"0"=>
       count <=x"0000";
       flag <= '0';
       if (start = '1') then
         state       <= x"1";
       else
         state <= x"0";
       end if;

     when x"1" =>
             if count = x"1000" then
         count <= x"0000";
         flag <='1';
         state <= x"2";
       else
         count <= count + '1';
         state       <=  x"1";
       end if;

     when others =>
       flag <= '1';
    end case;
  end if;
```

Fig. 4: Example VHDL design

(always(start=0)[10] => always(flag=0)[10]) with clock

(Equation 1)

The counterexample in Fig. 5 is generated by VBMC when this property is tried on the design.

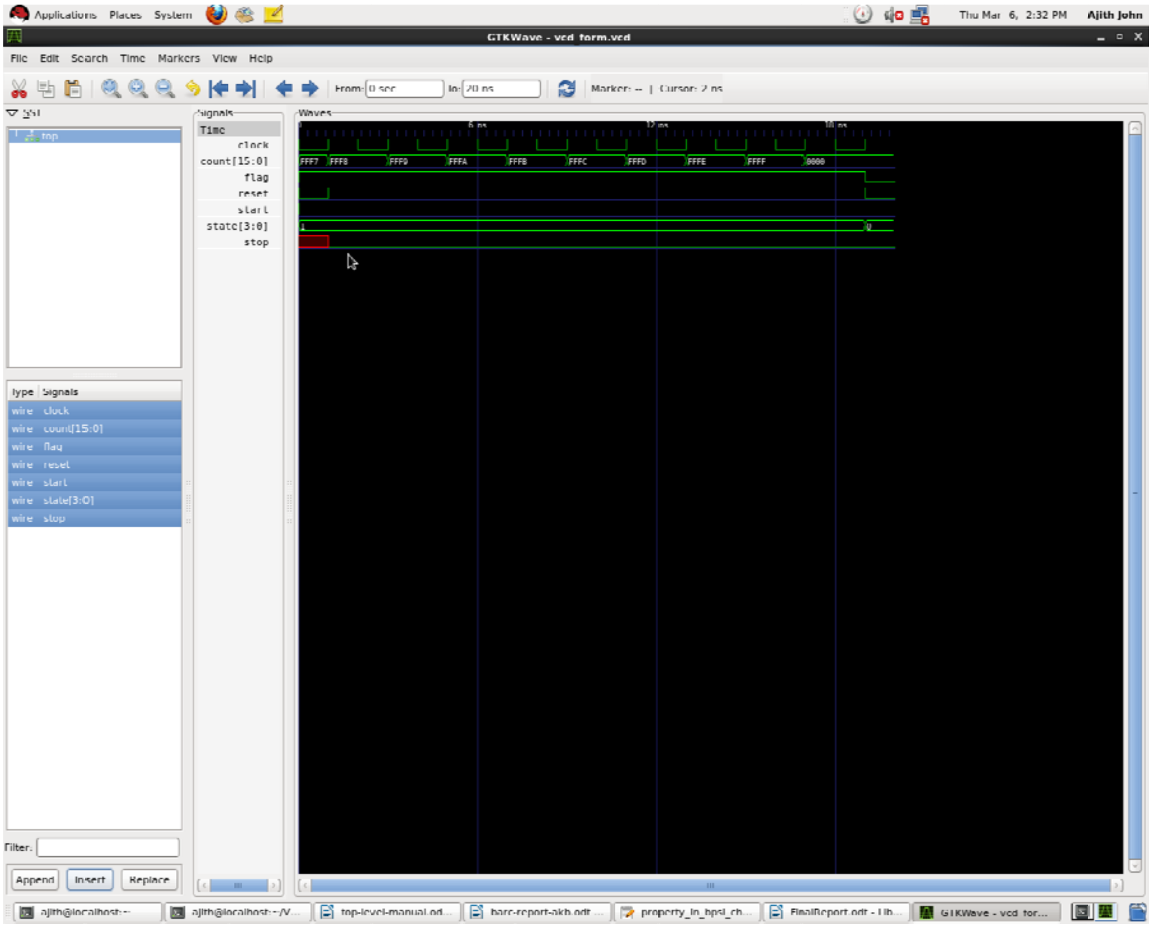

Fig. 5: Counterexample generated by VBMC

The analysis of the counterexample reveals that the VHDL design violates the property if the execution of the controller starts in state 1. Note that (see Fig. 4) if the execution of the controller starts in state 1, it can remain in state 1 for 10 clock cycles with *start* = 0 and *flag* = 1, thus violating the property. Hence the property needs to be changed to “once started in state 0, *flag* is 0 as long as *start* is 0”. The changed property for the first 10 clock cycles is shown in Equation 2.

(((reset=1) && (next[1](reset=0))) =>
(always(next[1](start=0))[10] => always(next[1](flag=0))[10])) with clock

(Equation 2)

Note that the PSL property is changed by adding an antecedent “**(((reset=1) && (next[1](reset=0)))**”, which asserts that the execution of the design starts at state 0. The changed property in Equation 2 is proved by VBMC for the given design.

3. Conceptual Framework and Internals of VBMC

VBMC performs the verification of functional properties of VHDL designs with both control-dominated and data-dominated sub-parts by combining some powerful techniques used by the formal verification community, namely symbolic simulation[15], bounded model checking[16] , word-level constraint solving[17], and counterexample guided abstraction refinement[18].

Symbolic simulation involves simulating a design's behaviour using symbols for the design’s inputs. This helps in deriving relations between inputs, outputs, and internal variables of the design as symbolic expressions. By substituting concrete values for the symbols in these symbolic expressions, the result of traditional (concrete valued) simulation can be obtained. This technique has been shown to be effective in reasoning about designs with wide data paths as well as significant control complexity.

The symbolic expressions generated by symbolic simulation can be used to derive the transition relation (R) of the design. Transition relation of a design is a relation between present and next states of the design. Suppose it is required to ensure that the design has a functional property P for k cycles of operation. This can be done by unrolling the transition relation k times, conjoining the unrolled relation with the negation of the property P, and then checking for satisfiability of the resulting constraint using a word-level constraint solver. The process of checking for the satisfiability of such a constraint (called verification condition C) gives one of the two outcomes: (i) a set of assignments of values to the signals in the design that demonstrates a concrete execution of the design violating the property, i.e., a counterexample, or (ii) inference that the verification condition cannot be satisfied, implying that the design satisfies the property under verification up to k cycles of operation.

The process of checking the satisfiability of the verification condition C can pose serious scaling issues if one tries to reason about all variables and expressions. VBMC alleviates this problem using an abstraction (over-approximation) C' of the verification condition C which is simpler to reason about compared to C. C' is obtained from the transition relation R in the following way: Initially an abstraction of the transition relation (R') is obtained from the transition relation R by hiding the details of selected internal signals of the design. C' is obtained by unrolling the abstract transition relation k times and conjoining the unrolled relation with the negation of the property P. The set of solutions of R' is a superset of the set of solutions of R; hence the set of solutions of C' is a superset of the set of solutions of C. If C' does not have any solution then, C also does not have any solution, and the design satisfies the property up to k cycles of operation. However, if C' has a solution π, then π may or may not be an execution which the design can follow violating the property. If π is an execution of the design which violates the property, then π is called a real counterexample. Otherwise π is called a spurious counterexample. Hence, if a counterexample is obtained, it needs to be analysed, to check if it is a real or spurious counterexample. If it is a real counterexample, it is proved that the design violates the property under verification. However, if it is a spurious counterexample, it needs to be eliminated by refining the abstraction R' (and consequently C') using hints from the spurious counterexample. This process of checking for the satisfiability of the abstract verification condition and refining it continues iteratively, until (i) a real counterexample is obtained, or (ii) it is proved that the design satisfies the property. This technique of starting from an initial abstraction and refining it iteratively in a counterexample guided manner is called counterexample guided abstraction refinement (CEGAR).

As the transition relation R is in general a bit-vector (word-level) formula, the problem of generating an abstraction of the transition relation boils down to variable elimination (more commonly known in literature as quantifier elimination) problem for bit-vector formulas. VBMC uses a novel quantifier elimination algorithm[19, 20], which tries to keep the quantifier-eliminated formula at word-level as much as possible, resorting to blasting only when it is absolutely necessary**.** This helps in keeping the abstract transition relation and the abstract verification condition at word-level. Keeping the abstract verification condition at word-level is appealing, as it allows efficient word-level reasoning to be applied by the back-end word-level constraint solvers. This improves the overall scalability of the verification scheme particularly when verifying designs with wide data paths.

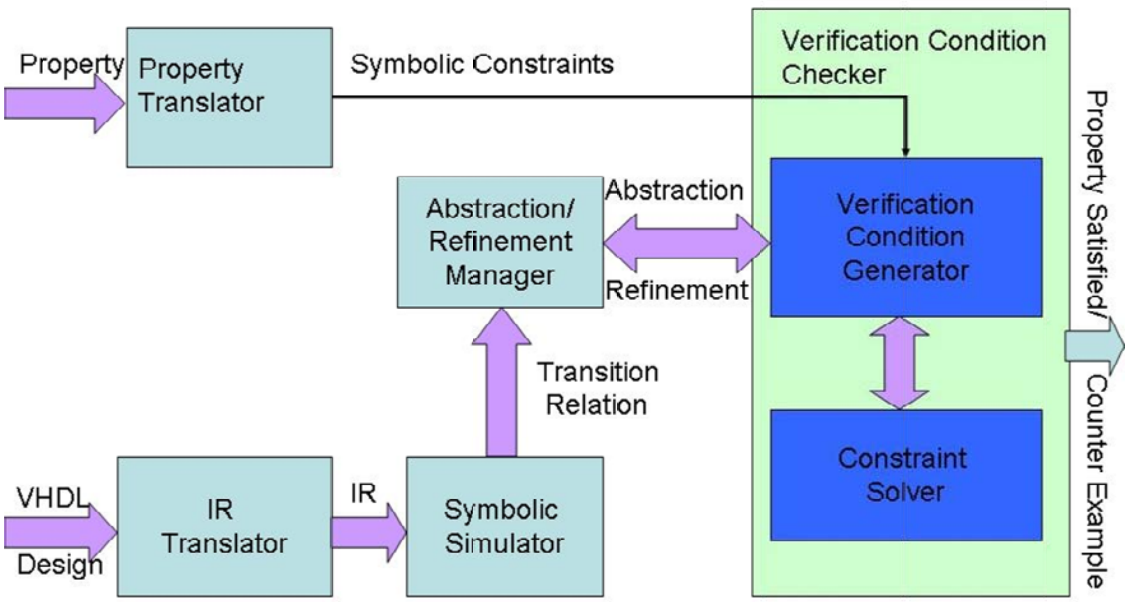

Fig. 6: Architecture of VBMC

The block diagram in Fig. 6 shows the internal details of VBMC. A brief description of the internal components of VBMC is given in the following subsections.

3.1 IR Translator

The IR translator translates the VHDL design into an Intermediate Representation (IR). The symbolic simulator reads the IR and generates symbolic expressions by symbolically simulating the IR. The IR is basically a control and data flow graph (CDFG) that represents the control flow structure of the VHDL program along with the data flow among variables, signals and ports in the program. The IR has two types of nodes, namely, the Control Nodes that represent the statements in the program (including process statements, assignment statements etc.), and the Data Nodes that represent the data in the program (including variables, constants, signals, ports, etc.).

The IR Translator works in three phases. In the first phase, the input VHDL files are pre-processed and a single file (which is equivalent of all source files) is outputted. The pre-processing phase ensures that the input VHDL files do not contain any constructs disallowed by the subset. The output of the pre-processing phase is used as input by the parser, the second phase, to create the initial IR. However, the initial IR may be incomplete as filling up some fields like ranges of signals and ports, addresses of instantiated entities in port maps etc. may require a look up of the symbol table created by the parser. Hence a third phase (post-processing phase) is required which fills up these blank fields and makes the IR complete and final.

3.2 Symbolic Simulator

The symbolic simulator i) performs the symbolic simulation of the design generating symbolic expressions, and ii) generates transition relation of the design from the symbolic expressions.

Symbolic simulation involves mimicking the conventional simulation (concrete valued simulation) with symbols as inputs rather than concrete values. This generates symbolic expressions that are symbolic relations between inputs, outputs, and internal variables of the design.

A VHDL design is composed of a collection of basic statements called process statements. The conventional simulation of a VHDL design involves the repetitive execution of the process statements in the VHDL design in an event-triggered fashion (execution of a process statement generates events, which trigger the execution of other process statements), until a fix-point is reached, where there are no more events or execution of process statements. The execution of the each process statement is called a simulation cycle. VHDL associates an internal delay with the simulation cycle called δ. In short, the conventional simulation of a VHDL design involves a sequence of simulation cycles (δs).

```
entity register is
  port(clk, din : in bit; qout : out bit);
end register;

architecture register of register is
signal q : bit;
begin
  DFF1 : dff port map(clk, din, q);
  DFF2 : dff port map(clk, q, qout);
end register;

entity dff is
  port(clk, d : in bit; q : out bit);
end dff;

architecture dff of dff is
begin
  process(clk)
  begin
    if (clk'event and clk='1') then
      q<=d;
    end if;
  end process;
end dff;
```

Fig. 7: VHDL design of 2 bit

The symbolic simulator mimics the conventional simulation with symbols as the inputs in the different simulation cycles, thus generating symbolic expressions between inputs, outputs, and internal variables of the design in the different simulation cycles. The idea is explained with the help of a VHDL design which represents a 2-bit register (see Fig. 7). The symbolic expression for this VHDL design generated by the symbolic simulator appears as shown in Equation 3.

$$\begin{aligned} qout_0 &= ite(clk_{-\delta} \neq clk_{-2\delta} \mathbin{\&\&} clk_{-\delta} = 1, q_{-\delta}, qout_{-\delta}) \quad \&\& \\ q_{-\delta} &= ite(clk_{-2\delta} \neq clk_{-3\delta} \mathbin{\&\&} clk_{-2\delta} = 1, din_{-2\delta}, q_{-2\delta}) \end{aligned} \qquad \textit{(Equation 3)}$$

In Equation 3, $x_{t\delta}$ denotes the value of the signal x in the t^{th} simulation cycle and a=ite(b, c, d) represents (b && (a=c)) || (¬b && (a=d)). Intuitively, the symbolic expression above relates the value of the output qout in the 0^{th} simulation cycle ($qout_0$) with the values of the other signals in previous simulation cycles. We call this δ-transition relation (R_δ).

However, the simulation cycles (δs) are important only for proper ordering of events, and are not observable externally. In other words, the values of the signals in the different simulation cycles are not necessarily the stable values of the signals. The transition relation R relates the present and the next stable values of the signals. However, considering the transient signal values is important for correctly obtaining the stable values of the signals.

The transition relation R is obtained from R_δ by imposing the following constraints on R_δ: (i) inputs of the design change exactly once i.e. at δ, and (ii) for every internal signal and output, the values at all δs are transient except the one at the maximum δ. The transition relation R of this design derived by the symbolic simulator from R_δ appears as shown in Equation 4.

$$\begin{aligned} &qout' = ite(clk' \neq clk \;\&\&\; clk' = 1, q, qout) \;\&\& \\ &q' = ite(clk' \neq clk \;\&\&\; clk' = 1, din', q) \end{aligned}$$

(Equation 4)

The unprimed variables in Equation 4 denote the initial values of the signals and the primed variables denote the new values of the signals. The transition relation indicates how the output and the internal signals in the design change when there is a change in any of the input signals.

3.3 Abstraction/Refinement Manager

Abstraction/refinement manager generates an abstraction (over-approximation) R' of the transition relation by hiding/eliminating the details of a selected set of internal variables of the design. The set of solutions of R' is a superset of the set of solutions of R. The degree of abstraction depends on the set of internal variables chosen for elimination. The larger the set of internal variables chosen for elimination, the larger is the degree of abstraction.

As the transition relation R is in general a bit-vector (word-level) formula, the problem of generating an abstraction of the transition relation boils down to variable elimination (more commonly known in literature as quantifier elimination) problem for bit-vector formulas. Currently, the most popular technique for eliminating quantifiers from bit-vector formulas involves converting (blasting) bit-vectors in the formula into individual bits (Boolean variables), followed by quantifier elimination from the resulting Boolean formula. However this approach has some undesirable features. For example, blasting involves a bit-width-dependent blow-up in the size of the problem. This can present scaling problems in the usage of tools for quantifier elimination from the resulting Boolean formula, especially when reasoning about wide words. Similarly, given an instance of the quantifier elimination problem for a bit-vector formula, blasting the variables to be eliminated may transitively require blasting other variables (that need not eliminated) as well. This can cause the quantifier-eliminated formula to appear like a Boolean formula, instead of being a bit-vector formula. Since reasoning at the level of bit-vectors is often more efficient in practice than reasoning at the level of blasted bits, quantifier elimination using blasting may not be the best option as the quantifier-eliminated formula (R' in our case) is intended to be used in reasoning by a word-level constraint solver. VBMC uses a novel quantifier elimination algorithm **QE_LMDD**[16,17], which tries to keep the quantifier-eliminated formula at word-level as much as possible, resorting to blasting only when it is absolutely necessary**.**

QE_LMDD represents the transition relation R as a DAG based data-structure called LMDD[19]. The LMDD is traversed in a top-down manner converting the problem of quantifier elimination from the transition relation into a set of simpler sub-problems each of which involves quantifier elimination from a conjunction of bit-vector equations, disequations and inequations. **QE_LMDD** uses an algorithm **Project** [20] to perform quantifier elimination from a conjunction of bit-vector equations, disequations and inequations. **Project** uses a layered approach for quantifier elimination - sound, but relatively less complete, cheaper layers are invoked first, and expensive but more complete layers are called only when required. The cheaper layers involve simplification of the given conjunction of bit-vector equations, disequations and inequations using the equations. Subsequently, an efficient combinatorial heuristic to identify unconstraining inequations and disequations that can be dropped from the conjunction without changing the set of satisfying solutions is applied. Finally, those cases that are not solved by application of the above computationally cheap techniques are handled by a variant of the classical Fourier-Motzkin quantifier elimination algorithm for reals adapted for bit-vector equations, disequations and inequations.

It is experimentally found that **QE_LMDD** performs quantifier elimination without any blasting for the fragment of bit-vector formulas involving bit-vector equations, disequations and inequations. As the transition relations of real life VHDL designs encountered largely belongs to this fragment, the use of **QE_LMDD** for abstraction generation (i) is considerably efficient compared to blasting, and (ii) keeps the quantifier eliminated formula at word-level helping in efficient reasoning by the word-level constraint solvers.

As an example, eliminating the internal variable q from the transition relation R in Equation 4, using **QE_LMDD** generates the abstract transition relation R' shown in Equation 5.

$$((clk \neq clk)$$
$$\&\&$$
$$(clk'=1))$$
$$||$$
$$(\neg ((clk \neq clk)$$
$$\&\&$$
$$(clk'=1))$$
$$\&\&$$
$$(qout' = qout))$$

(Equation 5)

It is easy to see that the set of solutions of R' in Equation 5 is a superset of the set of solutions of the given R.

Initially the abstraction/refinement manager chooses to eliminate all the internal variables in the design from the transition relation in order to generate the (initial) abstract transition relation. Subsequently, the set of internal variables to be eliminated is chosen by the verification condition generator/checker. Hence, the abstraction/refinement manager initially generates a weak abstraction (abstraction with a large degree of abstraction) by eliminating all the internal signals of the design from the transition relation, and subsequently generates stronger abstractions (abstractions with lesser degree of abstraction) by eliminating the internal signals of the design chosen by the verification condition generator/checker.

3.4 Property Translator

Property translator converts the property in PSL into the input language of the constraint solvers called the SMTLIB format[18.] Table 2 summarizes the translation from PSL to the SMTLIB format. Here f represents a PSL formula and f' represents the SMTLIB translation of the formula.

Table 2: Translation scheme in property translator

PSL construct	SMTLIB format
variable	variable_k, where k represents the point where the property starts
constants, logical operators, shift operators, addition/multiplication operators, bit-wise operators, implication operators	corresponding constant/operator in SMTLIB format
(t1 R t2) where R is a relational operator and t1 and t2 are terms	(t1' R' t2') and (t1' R' t2' >> 1), where t1' and t2' represents t1 and t2 converted to SMTLIB format, R' represents the operator corresponding to R in SMTLIB format, and >> k denotes shifting the formula (t1' R' t2) by k steps. For example, (x_0 = y_1) >> 1 gives (x_1 = y_2).
(**next** f)	(f' >> 2)
(**next[I]** f)	(f' >> 2.I)
(**next_a[I:J]** f)	(f' >> 2.I) and (f' >> 2.(I+1)) and … and (f' >> 2.J)
(**next_e[I:J]** f)	(f' >> 2.I) or (f' >> 2.(I+1)) or … or (f' >> 2.J)
(**always** f[**I**])	(f') and (f' >> 2) and … and (f' >> 2.(I-1))
(**eventually** f[**I**])	(f') or (f' >> 2) or … or (f' >> 2.(I-1))
(**never** f[**I**])	(not f') and (not f' >> 2) and … and (not f' >> 2.(I-1))
(f) with signal	(T => f'), where T denotes the SMTLIB representation of the statement *signal toggles for a number of clock cycles equal to the bound of the property*

3.5 Verification Condition Generator/Checker

Verification condition generator/checker (i) generates the abstract verification condition, (ii) checks the satisfiability of the abstract verification condition using a word-level constraint solver, and (iii) either reports that the property is proved/refuted or finds the set of internal variables in the design to be exposed (not to be eliminated) before generating the next abstract transition relation by analysing the result from the constraint solver.

For example, it is required to verify that the VHDL design has a functional property P for k cycles of operation. The verification condition generator/checker generates the abstract verification condition by unrolling the abstract transition relation k times, and conjoining the unrolled relation with the negation of the property P. The abstract verification condition (converted to SMTLIB format) is given to a word-level constraint solver to check its satisfiability. VBMC can use any constraint solver which supports bit-vector arithmetic such as simplifyingSTP [23], Yices[24] etc., to check the satisfiability of the abstract verification condition. If the abstract verification condition does not have any solution then, the verification condition generator/checker reports that the design satisfies the property up to k cycles of operation. If the abstract transition relation has a solution π then, π is analyzed to check if it is a real or spurious counterexample. This is achieved by checking if π is a solution to the verification condition. If π is a solution of the verification condition, then π is a real counterexample. Otherwise, π is a spurious counterexample. If π is a spurious counterexample, Verification Condition Generator/Checker finds the set of internal variables in the design to be exposed (not to be eliminated) in order to avoid this counterexample. This set of internal variables to be exposed becomes the refinement hint for abstraction/refinement manager to generate the next abstract transition relation.

4. Conclusions

VBMC was used for the functional verification of VHDL programs developed for the FPGAs used in EHS hardware boards developed in Reactor Control Division. The functional properties specified in the FRS document were taken up for verification. Following the verification cycle in Fig 2., properties were initially written in English and then translated into PSL, which were then submitted for verification. In cases, where the tool reported the falsification of the property in the design, the counterexamples produced by the tool were analyzed to understand the reason for failure. One important outcome of the tool supported analysis was in increasing accuracy of specification.

The performance of VBMC was also evaluated on a set of benchmark VHDL designs. This included a set of public domain VHDL designs[22]. Using VBMC, functional properties of these designs up to 2000 cycles of operation were checked within 1800 seconds on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB RAM running Linux, with simplifying STP[23] as the back-end constraint solver. This outperformed the existing techniques, which were unable to scale beyond a few tens of cycles with a time limit of 1800 seconds.

Given a VHDL design and a property in PSL, VBMC either generates a counterexample, or infers that the design satisfies the given functional property for k cycles of operation. It can be observed that, in the latter case, VBMC only provides an incomplete (bounded) guarantee that the design satisfies the property for a bounded number of cycles of operation (Note that in the experiments, VBMC was successfully applied for the verification of properties of real life VHDL designs up to 2000 cycles of operation within 1800 seconds, which gives a reasonable although bounded guarantee of correctness). This is in contrast with the complete (unbounded) guarantee that a given design can *never* be made to violate the property (for any number of clock cycles), as provided by traditional model checkers such as SMV or NuSMV. It is interesting to see if the approach behind VBMC can be extended to give such unbounded guarantees. Empirically it is observed that unbounded model checking is less scalable compared to bounded model checking. One of the approaches for scalable unbounded model checking that has shown initial promising results is based on decomposition of the transition relation into independent components that can be model checked in parallel. The authors are presently exploring this approach in collaboration with CFDVS, IIT Bombay.

It will be our endeavour to extend VBMC for the verification of other HDLs such as Verilog.

Acknowledgement

Some part of the work was carried out collaboratively with CFDVS under a BRNS Project Grant 2008/36/14-BRNS/426. The authors wish to thank Prof. Supratik Chakraborty, CFDVS/CSE, IIT Bombay, Mumbai for his guidance and collaboration in developing the core verification engine of the tool. Authors are also grateful to BRNS for this support. Thanks are due to Shri Amol Wakankar, SO/E and Smt Pratibha Sawhney, SO/D, for their contribution in implementing the GUI and Counterexample extraction, without which this integrated framework would not have been possible.

Authors would like to thank Shri S.D. Dhodapkar, RRF, who initiated the development activity. Authors also wish to thank and acknowledge the support and encouragement of Shri Y.S. Mayya, Head, Reactor Control Division and Shri C.K. Pithawa, Director, E&IG, BARC during this development work. Thanks are also due to Shri G. Ganesh, OS, Head, TTCD and Shri Mukesh Sharma, SO/F, RCnD for readily sharing the VHDL designs used in the EHS boards as a case study.

References

1. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis IEEE Std 1076.6-2004
2. IEEE Standard VHDL Reference Manual IEEE Std 1076-2002
3. AERB Safety Guide COMPUTER BASED SYSTEMS OF PRESSURISED HEAVY WATER REACTOR, AERB/NPP-PHWR/SG/D-25, 2010.
4. Review Guidelines for Field-Programmable Gate Arrays in Nuclear Power Plant Safety Systems, Oak Ridge national Laboratory, USNRC, NUREG/CR-7006, 2010.
5. Design Assurance Guidance for Airborne Electronic Hardware (DO-254), RTCA, 2005.
6. Xilinx website. http://www.xilinx.com (accessed July 1, 2014)
7. MentorGraphics website. http://www.mentor.com (accessed July 1, 2014)
8. Cadence website. http://www.cadence.com (accessed July 1, 2014)
9. Synopsys website. http://www.synopsys.com (accessed July 1, 2014)
10. SMV website. http://www.cs.cmu.edu/~modelcheck/smv.html (accessed July 1, 2014)
11. NuSMV website. http://www.nusmv,fbk.eu/ (accessed July 1, 2014)
12. VIS website. http://www.vlsi.colorado.edu/~vis/ (accessed July 1, 2014)
13. VCEGAR tool website. http://www.cs.cmu.edu/modelcheck/vcegar/(accessed July 1, 2014)
14. B Cohen, S. Venkataramanan, A. Kumari. *Using PSL/Sugar for formal and dynamic verification*, *2nd edition, VhdlCohen Publishing, USA*, 2004
15. R. Bryant. "Symbolic simulation—techniques and applications". *Proceedings of 27th*

ACM/IEEE Design Automation Conference (DAC), 1990

16. Bierre, A. Cimatti, E. Clarke, Y. Zhu. “Symbolic model checking without BDDs”, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1999.
17. Kroening, O. Strichman. *Decision procedures: an algorithmic point of view, Texts In Theoretical Computer Science, Springer*, 2008
18. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Counterexample-guided abstraction refinement”, *Proceedings of 12th International Conference on Computer Aided Verification (CAV)*, 2000
19. Ajith John and Supratik Chakraborty. “A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations”, *Proceedings of 23rd International Conference on Computer Aided Verification (CAV)*, 2011
20. Ajith John and Supratik Chakraborty. “Extending Quantifier Elimination to Linear Inequalities on Bit-Vectors”, *Proceedings of 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013
21. SMTLIB website. http://www.smtlib.org (accessed July 1, 2014)
22. ITC’99 benchmarks. http://www.cad.polito.it/downloads/tools/itc99.html (accessed July 1, 2014)
23. STP website. http://sites.google.com/site/stpfastprover/ (accessed July 1, 2014)
24. Yices website. http://yices.csl.sri.com/ (accessed July 1, 2014)