Formal Verification of VHDL
The Model Checker CV *

David Déharbe†
Universidade Federal do Rio Grande do Norte — Brazil
david@dimap.ufrn.br

Subash Shankar
Carnegie Mellon University — USA
sshankar@cs.cmu.edu

Edmund M. Clarke
Carnegie Mellon University — USA
emc@cs.cmu.edu

Abstract

This article describes a prototype formal verification system for a subset of VHDL. The behavior of a VHDL design can be specified with temporal logic formulas and be verified with an algorithm called symbolic model checking. The model checker applies a number of new techniques to handle larger designs, thus allowing for efficient verification of real circuits. We have completed an initial release of the VHDL model checker and have used it to verify complex circuits, including the control logic of a commercial RISC microprocessor.

1. Introduction

Ensuring the correctness of computer circuits is an extremely important and difficult task. As evidenced by several bugs in recent processors, there are serious ramifications resulting from bugs in released hardware. The most commonly used verification technique is simulation. However, simulation is by nature incomplete, and it can miss important errors in many cases. An alternative approach is to use formal methods to formally prove that a circuit implementation satisfies its specification. Temporal logic model checking [4] is a particularly popular and proven technique for proving hardware correctness.

Three major issues that must be addressed for model checking to be useful in a design environment are: ease of interfacing to existing hardware description languages (HDL) used to describe circuit implementations, an appropriate language for specifying the properties to be proven, and the efficiency of the model checker. VHDL is an obvious HDL choice: it is used as input for many CAD systems, it provides a variety of descriptive styles, and it is an IEEE standard [9]. Temporal logics such as Computation Tree Logic (CTL) have proven their usefulness for specifying hardware properties, and there has been much research on making CTL model checking more efficient. However, most existing model checkers can only verify circuits that are written in their own language. This makes it difficult to apply model checking in practice, since very few designers use these languages. For model checking to be accepted by industry, we believe that it is essential to provide a mechanism for automatically interfacing the model checker to a language such as VHDL.

We have developed a VHDL verification system called CV. The verification system automatically generates models from VHDL descriptions, and then uses symbolic model checking to prove that the specification is met. An initial prototype has been implemented and used to verify complex designs. Our approach allows for a number of optimizations that result in dramatically smaller state spaces, and we have implemented several of these. These optimizations allow our model (and thus, state space) to be smaller than in other approaches.

2. Overview

In the context of this work, a VHDL description can be considered to be an *implementation* that must be checked against a (partial) *specification* composed of a set of temporal logic formulas. Initially, the VHDL description is compiled into a state-transition graph represented internally by BDDs [3]. Model checking techniques are then used to determine if the specification holds in the circuit model.

We want the temporal logic specification and the VHDL implementation to be physically independent and reside in separate files. This independence property is quite important, since we might want to modify the specification (e.g. in incremental verification) without recompiling the VHDL implementation. Also, when using a structural description style, the same VHDL design can be instantiated several times as a component of a larger model. The system must be flexible enough to avoid recompiling the same description in such cases. Consequently, the compilation process is

*This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

† Contact author. Partly supported by CNPq and Projeto Nordeste.

split into a front-end that checks that the description is legal VHDL and a back-end that generates the symbolic model.

The system is composed of a general-purpose VHDL analyzer `cva` and a VHDL model checker `cvc`.

`cva` takes as input a text file that contains VHDL descriptions and generates library units in an intermediate format that can be accessed with a C library, in much the same way as commercial VHDL front-ends.

`cvc` takes as input a specification file and builds the model of the corresponding VHDL description, and applies the verification algorithms to this model. The *model elaborator* implements the semantics of VHDL in terms of state-transition graphs as described in [6]. It reads the intermediate format files produced by the compiler `cva` and builds a symbolic, BDD-based model of the VHDL design. This step is crucial for the entire verification process since the complexity of the model checking algorithms heavily depends on the size of the model (i.e. the number of state variables, and the size of the BDDs). To this end, considerable effort has been devoted to generating a compact model. The consistency checker takes as input a specification file composed of temporal logic properties. The *model checker* itself determines if the implementation satisfies the specification. When an error is detected, the model checker can produce a counterexample as a VHDL testbench. The testbench can be directly used with a conventional simulator to show an execution trace that violates the property.

3. The VHDL Subset

VHDL is a very rich language. Some of its constructs generate infinite models and cannot be modeled by a finite-state tools such as `CV`. Examples of such constructs include combination of an increasingly delayed signal assignment nested in an infinite loop, or a non-halting recursive function with local variables. Also, many constructs can be expressed in terms of more basic constructs, while maintaining the same semantics. It is also desirable to concentrate on a synthesizable subset of VHDL, as most such subsets eschew some parts of VHDL (e.g. either delta delays or unit time delays). Our approach has been to identify and implement a core subset of the language and incrementally extend this subset based on these guidelines. The elements of current supported subset are:

Design entities: entity declaration, architecture, package.

Concurrent statements: block, process, (with optional sensitivity lists), selected and conditional signal assignment.

Sequential statements: wait (with arbitrary sensitivity and condition clauses), signal and variable assignment, if, case, while, null.

Libraries: library, library clause and use clause.

Declarations: signal (input, output and local), variable, constant and type declarations.

Types: bounded integers, enumerations, arrays, and records.

Quantitative timing aspects are not yet handled in the subset but work is under way to include them.

4. VHDL Semantics for Symbolic Model Checking

Much research has been conducted to give a formal semantics to VHDL and apply formal verification techniques (see e.g. [10, 2]). While there exist formal VHDL semantics in a number of formalisms, it is difficult to use many of these in formal verification. Many operational semantics approaches tend to lead to excessively large models. Many axiomatic semantics either restrict the VHDL subset too much or are in logics difficult to reason in.

In this section we give a general overview of the VHDL simulation model, define the finite-state machine model used for symbolic model checking and present the relationship between the simulation model and the formal verification model used in CV.

4.1. VHDL simulation model

In VHDL, digital systems are modeled as a set of real-time communicating, synchronizing, concurrent processes. The set of all possible simulations of these processes is a model of the behavior of these systems. The *communication* channels between processes are called signals. Whenever several processes simultaneously assign different values to the same signal, the simulator calls a user-defined resolution function and computes the effective value of the signal. Signal assignment may optionally be delayed by some time quantity. Processes *synchronize* by means of wait statements, which explicitly let time pass until some condition is realized or for some specific duration. VHDL *time* is discrete: the base unit is the femto-second. In a simulation cycle, time may pass or not, for instance when a zero-delayed signal assignment occurs. Therefore, several consecutive simulation cycles may occur at the same time.

[9] defines the event-driven simulation algorithm for VHDL. First, the model is initialized: signals are assigned their initial values and processes are executed until they suspend. The time counter is then advanced until a new signal attribution is scheduled or a process resumes. That may be the current simulation time, and in this case, the simulation cycle is called a delta cycle. If new signal assignments have taken effect, then the effective value of some signals may change. Processes scheduled to resume, or sensitive to such signals, are run, and new signal assignments may in turn be executed. For a theoretical discussion on properties of the VHDL simulation see e.g. [8].

4.2. Formal verification model

Definition 1 (Formal verification model) *A formal verification model $\mathcal{M}$ is defined as a tuple $\langle S, \sigma_0, \mathbf{f} \rangle$, where:*

- *S is a set of* state *boolean variables of $\mathcal{M}$: $S = \{v_1, \ldots, v_n\}$.*

– $\sigma_0 \in \mathbb{B}^n$ *is the* initial state *of* $\mathcal{M}$*:* σ_0 *is a valuation of the state variables* S *of* $\mathcal{M}$*;*

– $\mathbf{f}$ *is the* transition function *of* $\mathcal{M}$*:* $\mathbf{f} = [f_1, \ldots, f_n]$*, where* $f_k : \mathbb{B}^n \rightarrow \mathbb{B}$ *is the transition function of state variable* v_k*.*

A *state* s of $\mathcal{M}$ is a valuation $\{v_1, \ldots, v_n\}$ of the state variables. State $s' = \{v'_1, \ldots, v'_n\}$ is the *successor* of state s if $\bigwedge_{i=1}^{n} v'_i = f_i(v_1, \ldots, v_n)$. A *path* is a sequence of states $s_0, s_1, \ldots s_k, \ldots$ such that, for all i, s_{i+1} is the successor of s_i. In other words, a formal verification model is a state transition graph labeled with boolean propositions called state variables here. The computation tree of $\mathcal{M}$ is the branching structure obtained when unwinding this state transition graph: each branch of this tree is a path of the model.

4.3. Relationship between the simulation and formal verification models

The formal verification model of a given simulation model is such that state variables represent the value of signals, process variables, and process program counters. Therefore, the initial state represents the initial valuation of variables, signals and program counter. The transition function represents the state transformation operated in the body of the processes as well as the kernel activity. The goal is that the behavior of the simulation model be represented by the computation tree of the verification model. Therefore, a state of the verification model represents a valuation of the signals and variables, and a transition models a simulation cycle. Also, a path of the model corresponds to one possible simulation run.

The reader interested in technical details is referred to [6], where it is shown precisely how formal verification models can be constructed from VHDL descriptions and pointers to other approaches to finite-state modelling of VHDL designs can be found.

5. The Model Checker

5.1. Specification Language

The language used in CV for specifying the expected behavior of a VHDL design in a given environment is essentially the logic CTL with fairness constraints. The environment is described in terms of assumptions on the values taken by the input signals of the design. A valid simulation of the design is a simulation where all assumptions on the input signals are verified. The behavior of a VHDL design in an environment is then defined to be the set of all valid simulations with respect to this environment. Therefore, a specification is composed of a set of *assumptions* and *commitments* about a VHDL description. An assumption is a condition on the inputs of the design under verification. Commitments describe the expected behavior of the system provided all assumptions hold. It is the role of the model checker to verify that the VHDL description satisfies the commitments provided the assumptions hold.

For convenience purposes, the specification language also makes it possible to define abbreviations. An abbreviation is an identifier that denotes an expression and may be used to simplify assumptions and commitments.

Two categories of assumptions are possible: invariant and fairness. The effect of an invariant definition is to restrict the behavior to the set of simulations where the associated condition holds for every cycle. The effect of a fairness definition is to restrict the behavior to the set of simulations where the associated condition holds infinitely often. Fairness is often needed to prove properties about the progress of a system. Generally, fairness constraints increase verification time and memory consumption.

Commitments are expressed in the temporal logic CTL (Computation Tree Logic). A CTL operator is composed of a path quantifier (**A**, **E**) followed by a linear temporal operator (**X**, **G**, **F**, **U**, **W**). Since the (standard) semantics of a VHDL design is defined in terms of simulation, we use temporal logic to express properties of the possible simulations of a design. The path quantifier **A** (**E**) selects all (some) simulations, and the linear temporal operator **X** (**G**, **F**, **U**, **W**) selects the next simulation cycle (all cycles, some cycle, until some cycle, unless some cycle) in a given simulation. A formula is true for a description if it holds in all initial states. For example, the CTL formula **AG**f states that f must hold at all states reachable from the initial states in all possible simulations.

For example, suppose it is necessary to check that a bad situation never happens. An abbreviation can be used to denote this bad situation and later reused in a commitment to state that it shall not happen:

```
abbreviation BAD is <some condition> ;
commit safe: ag not BAD;
```

More complex properties involving, for instance, events on signals can also be expressed in the logic.

5.2. The Model Checker

Transitions in the BDD-based models built by CV correspond to whole simulation cycles in the VHDL program. This allows our models to have fewer variables and fewer bits to represent the program counter, compared to approaches which use multiple transitions for each simulation cycle.

The models built by CV use a boolean functional vector representation for the transitions [5]. This limits considerably the explosion of the size of the transition representation for large systems. The representation also makes it easy to eliminate parts of the model that are not relevant with respect to the specification. The model checker needs only to consider the transition functions of the variables that can potentially affect the specification, which we denote as the *cone of influence* of the specification, constructed from the true support set of the transition functions.

Computing the reachable states of the model proves useful to improve the performance of symbolic model checking. A heuristic based on the observation that a conservative approximation, or *overestimation*, of the reachable states, is used to simplify the transition representation, and

to simplify the computation of the reachable states. Overestimations can be computed much faster than the valid states themselves.

To the best of our knowledge, these optimizations are unique for a VHDL-based model checker. Our experience was that, for some examples, each of the optimizations reduces the computation time by an order of magnitude and decreases the amount of space used. It is possible to handle significantly larger examples when both optimizations are combined.

6. Analysis

A good measure to evaluate the complexity of the model is the number of boolean variables in the formal verification model of a given VHDL design, since the size of the formal verification model is related to this quantity, as is the complexity of the symbolic model checking algorithms used in the verification process.

Let *trig* denote the set of signals to which at least one process statement is sensitive. If o is a signal or a process variable, $type(o)$ denotes the set of values that this object can take. If S is a set, let $|S|$ denote its cardinality. The number of state variables in the formal verification model of a VHDL design unit with n_s signals $s_1, \ldots, s_{n_s}$, n_p processes $p_1, \ldots, p_{n_p}$, each with v_i variable declarations $p_{i,v_1}, \ldots, p_{i,v_i}$, and w_i `wait` statements ($1 \leq i \leq r$) is:

$$\left(\sum_{k=1}^{n_s} \lceil \log |type(s_k)| \rceil\right) + \left(\sum_{s_k \in trig} \lceil \log |type(s_k)| \rceil\right) + \sum_{k=1}^{n_p} \left(\begin{array}{c} \lceil \log |type(w_k)| \rceil \\ + \quad \sum_{j=1}^{v_k} \lceil \log \left| type(p_{k,v_j}) \right| \rceil \end{array} \right)$$

The first term of this sum is the number of variables that represent the effective value of signals. The second term of this sum is the number of variables that model the delayed effective value of signals to which some process is sensitive. The third term corresponds to the number of variables needed to represent the local data of processes. It is itself the sum of the variables representing the value of the process program counter and the value of the process variables.

These are worst case figures. For example, a process statement with a single final `wait` statement need not have a wait counter variable. This situation occurs quite frequently, for example, in processes with sensitivity lists or in concurrent signal assignments.

7. Results and Further Work

We have applied CV to several designs. One particularly interesting circuit is a model of the control logic of a commercial RISC processor. The VHDL description consists of 25 different processes and spans several hundred lines of code. CVC was used to check two properties related to the handling of the reset signal and a safety property about the data and instruction register not being loaded simultaneously.

Building the BDD-based model of this description takes only about 5 seconds. Then, dynamic variable reordering is invoked to reduce the size of the model. This step takes approximatively 75 seconds, and reduces the size of the transition function to only about 5,000 BDD nodes, despite the fact that the model has 157 variables. An approximation of the set of reachable states is then computed, resulting in a further reduction to 4,000 BDD nodes. The resulting model has about 10^{28} reachable states out of 10^{47} possible states. By using the optimization that considers only the variables in the model which affect the specification (cone of influence reduction), a simple property can be checked in just 30 seconds. Without this optimization, verification would take two orders of magnitude longer. These results were measured on a Sun Sparc 10 with 128 MB of RAM.

We are currently continuing the development of CV and expect further developments to enhance performance (automated abstraction, compositional reasoning, low-level implementation improvements) and usability (extensions of the VHDL subset to timing aspects and structural design, graphical user interface). Also, we wish to compare the performances of CV other VHDL model checkers[11, 1, 7]. However because of commercial reasons, it will not be an easy task.

References

[1] J. Bormann, J. Lohse, M. Payer, and G. Venzl. Model checking in industrial hardware design. In *DAC'95*. IEEE Press.

[2] R. Brayton, E. Clarke, and P. Subrahmanyam, editors. *Formal Methods in System Design*, volume 7, (1/2). Kluwer, Aug. 1995. Special Issue on VHDL Semantics – Guest Editor: D. Borrione.

[3] R. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Trans. Comput.*, C(35):1035–1044, 1986.

[4] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[5] O. Coudert and J.-C. Madre. Symbolic computation of the valid states of a sequential machine : algorithms and discussion. In *International workshop on formal methods for correct VLSI design*, Miami, Jan. 1991. ACM/IFIP WG10.2.

[6] D. Déharbe and D. Borrione. Semantics of a verification-oriented subset of VHDL. In *CHARME'95*, volume 987 of *Lecture Notes in Computer Science*. Springer Verlag.

[7] E. Encrenaz. A symbolic relation for a subset of vhdl'87 descriptions and its application to symbolic model checking. In *CHARME'95*, volume 987 of *Lecture Notes in Computer Science*. Springer Verlag.

[8] K. Goossens. Reasoning about VHDL using operational and observational semantics. In *CHARME'95*, volume 987 of *Lecture Notes in Computer Science*. Springer Verlag.

[9] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987. Std 1076-1987.

[10] C. D. Kloos and P. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1995.

[11] R. Kurshan. Formal verification in a commercial setting. *Verification Times*, I, 1997.