Model Checking VHDL with CV *

David Déharbe[1], Subash Shankar[2], and Edmund M. Clarke[2]

[1] Universidade Federal do Rio Grande do Norte,
Natal, Brazil
david@dimap.ufrn.br
[2] Carnegie Mellon University,
Pittsburgh, USA
{sshankar,emc}@cs.cmu.edu

Abstract. This article describes a prototype implementation of a symbolic model checker for a subset of VHDL. The model checker applies a number of techniques to reduce the search space, 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. The most commonly used verification technique is simulation. However, in many cases simulation can miss important errors. A more powerful approach is the use of formal methods such as temporal logic model checking [3], which can guarantee correctness. However, most model checkers can verify only circuits that are written in their own language. This makes it difficult to apply the technique 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 an interface between the verification tools and some widely used hardware description language. VHDL is an obvious choice for such a language: it is used as input for many CAD systems, it provides a variety of descriptive styles, and it is an IEEE standard [6].

Much research has been conducted to give a formal semantics to VHDL and apply formal verification techniques (see e.g. [7,1]). While there exist formal VHDL semantics in a number of formalisms, it is difficult to use many of these in formal verification. Whereas 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. There have been

* 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.

some attempts to apply model checking to VHDL specifications, though it is still desirable to reduce the state space generated from the VHDL programs.

We have developed a VHDL verification system called `CV`. The verification system uses symbolic model checking. Our approach allows for a number of optimizations that result in dramatically smaller state spaces. We have incorporated several of these into CV, and our results show state space reductions of several orders of magnitude using these optimizations. The optimizations include cone of influence reduction to eliminate irrelevant portions of the circuit along with a new scheme for approximating the set of reachable states. This approximation is simple to compute, and results in additional large reductions in state space size. Finally, our models represent whole VHDL simulation cycles with single transitions, unlike several other VHDL model checkers, thus affording additional savings.

An initial prototype of CV has been implemented and used to verify several complex designs, including the control logic of a commercial RISC processor. In this demonstration, we describe this prototype and present results that demonstrate the applicability of the method to actual industrial designs.

2 Overview of the System

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 [2]. 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 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:

1. A general-purpose VHDL analyzer, called `cva`, which takes as input a text file that contains VHDL descriptions and generates library units in the intermediate format `.cv`. This compiler is constructed in much the same way as commercial VHDL front-ends. A library of C functions has been developed to access the intermediate format files. Thus, the front-end can be used independently of the model checker. `CV` does not require a commercial compiler. In Section 3 we describe the current VHDL subset.

2. A VHDL model checker, called `cvc`, which takes as input a specification file, builds the model of the corresponding VHDL description, and applies the verification algorithms to this model.
   (a) The model elaborator implements the semantics of VHDL in terms of state-transition graphs as described in [5]. 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.
   (b) The consistency checker takes as input a specification file composed of temporal logic properties. The specification language is defined in Section 4.
   (c) 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. This information can then be used to debug the circuit. It is also possible to output the counterexample as a sequence of states of the VHDL architecture. Details about the model checker implementation are provided in Section 5.

The structure of `CV` is depicted in Figure 1. Solid lines correspond to commands given by the user whereas dashed lines represent data automatically read or written by `cva` and `cvc`.

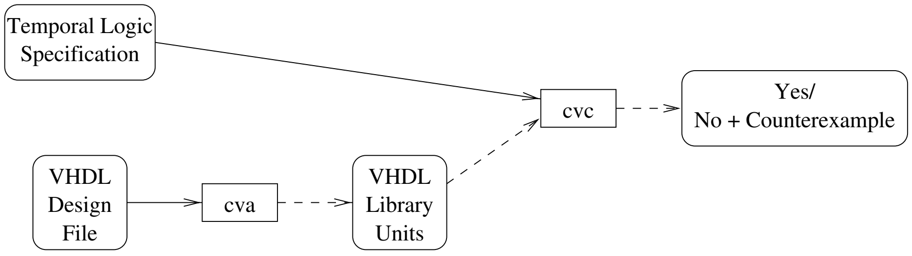

Fig. 1. CV Structure

3 Description of the VHDL Subset

VHDL is a very rich language. Some of its constructs generate infinite models and cannot be modeled by a finite-state tool such as `CV`, such as, for instance, combination of a 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 the core subset are

Design entities: Entity declaration, architecture body, package declaration.
Concurrent statements: Block statement and process statement.
Sequential statements: Wait statement, signal and variable assignment statements, if statement.
Libraries: Library, library clause and use clause.
Declarations: Signal (input, output and local), variable, constant and type declarations.
Types: Enumerated types.

The following extensions have already been implemented:

Concurrent statements: Selected signal assignment statement, concurrent signal assignment statement, processes with sensitivity lists.
Sequential statements: Case statement, while statement, null statement.
Types: Integer, array, and record types.

4 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. For instance:

```
abbreviation PUSHED is (PUSH and PUSH_RDY);
```

Two categories of assumptions are possible: invariant and fairness. An invariant is defined with the keyword `always` followed by a condition on the signals of `in` mode of the specified VHDL design. The effect of an invariant definition is to restrict the behavior to the set of simulations where the associated condition holds for every cycle. For instance, the following invariant assumption states that inputs `PUSH` and `POP` are never simultaneously true.

```
assume always not ((PUSH = '1' ) and (POP = '1'));
```

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. For instance, the following fairness assumptions can be used to state that `CLOCK` infinitely often changes value:

```
assume recurring (CLOCK = '0');
assume recurring (CLOCK = '1');
```

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 instance, the CTL formula $\mathbf{AG}f$ states that f must hold at all states reachable from the initial states in all possible simulations. f must always hold.

For example, suppose it is necessary to check that a bad situation never happens. An abbreviation can be used to denote this bad situation:

```
abbreviation BAD_SITUATION is <some condition> ;
```

The following commitment states that `BAD_SITUATION` never happens:

```
commit safe: ag not BAD_SITUATION;
```

Suppose it is necessary to show that whatever the current state of the system is, it can reach the restart state. Let `RESTART` be the condition that identifies the restart state:

```
commit RESTARTABLE: ag ef RESTART;
```

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

5 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 that use multiple transitions for each simulation cycle.

The models built by CV use a boolean functional vector representation for the transitions [4]. 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 to consider only the transition functions of the variables that are in the cone of influence of the specification. This cone of influence is simply constructed from the true support set of the transition functions.

Computing the reachable states of the model proves useful in practice for enhancing the performance of symbolic model checking. We have devised a heuristic based on the observation that a conservative approximation, or *overestimation*, of the reachable states, can also be used to simplify the transition representation, and consequently simplify the computation of the reachable states. We have devised and implemented a heuristic to compute overestimations of the valid states much faster than the valid states themselves. Indeed, an overestimation can be simply and efficiently computed considering the valid states of the submachine corresponding to a subset of the vector of transition functions.

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.

Therefore, given a specification, the model checker performs the following steps:

1. Parse the specification file.
2. Load the corresponding VHDL library unit, and build its BDD-based model using a boolean functional vector representation for the transitions.
3. Invoke dynamic variable reordering to reduce the size of the BDDs.
4. Compute overestimations of the reachable states and simplify transition representations.
5. Compute the reachable states of the model, and use the result to further reduce the BDDs representing the transitions.
6. Check the specification, using symbolic model checking techniques.
7. Report the results and generate counterexamples if necessary.

6 Conclusion

In its current version, our system is already able to handle some realistic designs. One of the most interesting examples is a model of the control logic of a commercial RISC processor. The VHDL description counts 25 different processes, and spans several hundred lines of code.

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. We then compute an approximation of the set of reachable states which further reduces this figure to 4,000 BDD nodes. The 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 that affect the specification, a simple property can be checked in just 30 seconds. Without this optimization, verification would take two orders of magnitude longer.

We are currently continuing the development of CV and expect further developments along two lines:

1. performance: automated abstraction, compositional reasoning, low-level implementation improvements.
2. usability: extension of the VHDL subset (timing aspects, structural design), design of a graphical user interface.

References

1. R.K. Brayton, E.M. Clarke, and P.A. Subrahmanyam, editors. *Formal Methods in System Design*, volume 7, (1/2). Kluwer, August 1995. Special Issue on VHDL Semantics – Guest Editor: D. Borrione.
2. R.E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions on Computers*, C(35):1035–1044, 1986.
3. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
4. 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, January 1991. ACM/IFIP WG10.2.
5. D. Déharbe and D. Borrione. Semantics of a verification-oriented subset of VHDL. In P.E. Camurati and H. Eveking, editors, *CHARME'95: Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, Frankfurt, Germany, October 1995. Springer Verlag.
6. IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987. Std 1076-1987.
7. C. Delgado Kloos and P. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1995.