A Path Dependency Graph for Verilog Program Analysis

Mohamed Zaki, Yassine Mokhtari and Sofiène Tahar
Electrical and Computer Engineering Dept.
Concordia University
Montreal, Quebec, Canada
Email: {mzaki, mokhtari, tahar}@ece.concordia.ca

***Abstract*— In this paper, we present a path dependency graph for Verilog-HDL programs called *path sequence*. This path sequence can be used in static analysis based techniques, such as program slicing, and model abstraction as well as in functional test generation. We have implemented this graph inside a model reduction tool. It enhanced the capability of the tool without adding significant timing overhead.**

I. Introduction

Verilog [12] HDL (Hardware Description Language) is a popular way for specifying, designing and verifying hardware systems. Classically, verification and error detection depends heavily on simulation of the designs. However, with the increasing complexity of the designs, detecting errors becomes extremely difficult and simulation becomes less efficient in terms of coverage and long simulation periods. Several approaches based on program analysis have been proposed to help the verification problem among them we cite static analysis [4] and model checking [2], where a semantic model of the design is verified against some temporal specifications.

Static analysis is a program analysis technique initially used for software debugging and testing. Static analysis means the analysis of the program semantics to extract information about the system at compile time such that properties can be proven about the system without the need to actually execute the program. The information collected from the analysis are very important for discovering and correcting possible errors, inconsistencies, deriving test patterns that could be used to validate the circuit under verification, identify the critical code part in the design as well as unreachable code. It can be also used to reduce and abstract the model size in order to speed up the model checking process.

There are two main properties desirable in static analysis algorithms [13]; *soundness* and *completeness*. In order to be *sound*, an algorithm must never delete (ignore) a statement from the original program which could have an effect upon the analysis. In order to be *complete*, the algorithm must delete (ignore) all statements which do not affect the analysis. In general, completeness is unachievable (for theoretical reasons dealing with undecidability [13]). Therefore, the goal of analysis algorithms is to delete (ignore) as many statements as possible, without giving up soundness. The closer an algorithm approximates completeness, the more accurate the analysis results will be.

The aim of this paper is to present a syntactic model for enhancing the static analysis of Verilog HDL programs. The model is mainly used in order to enhance the accuracy of the analysis based on control and data flow analysis. For instance, the idea of the model is based on mapping each state in the transition system of the Verilog program by the control flow graph path producing a path dependency graph. This graph is an abstract representation of the semantic model.

The rest of the paper is structured as follows. In Section II, we present some related work about HDL static analysis. In Section III, we define our system model including the abstract syntax (the control flow diagram) and semantic representation of a program. In Section IV, we present the proposed path dependency graph and Section V finally concludes the paper.

II. Related Work

One of the main static analysis approaches is *program slicing* [13]. Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics and deleting those parts of the program which can be determined to have no effect upon the semantics of interest [13]. There are few works dealing with slicing application for *hardware* description languages. For instance, the work in [6] applies program slicing to VHDL [11] programs. In order to analyze the VHDL code, along the data and control dependencies, a new dependency called *signal dependency* is defined for inter-process dependence. A slice follows from these dependencies (though the details of the slice are not defined). Clarke *et al.* [1] extended program slicing to VHDL in the context of a verification tool that can reduce the VHDL code to be analyzed. This is a direct application for static slicing of sequential systems with the introduction of a new dependence graph called *interference graph*, which represents the dependencies that may arise in concurrent procedure. The analysis of the program is done on the system dependency graph which is formed of data, control and interference dependencies.

Other related works include the work done by Baresi *et al.* [10], where data flow analysis techniques are used in order to identify deadlocks conditions within VHDL specifications. Hsieh *et al.* [5] applied data flow analysis techniques in their model abstraction algorithm, in order to abstract VHDL semantic. Liu *et al.* [7] created a high level finite state machine for coverage analysis of Verilog designs.

III. Verilog Modeling

In this section, we will present the syntax and the semantic model of the Verilog [12] programs. We are considering a synchronous subset of Verilog accepted by the SMV tool [8] but without considering concurrency.

A. Program Syntax

Definition 3.1: A Verilog program *P* can be specified as a *tuple* $\langle V, I, S\rangle$, where:

- *V* = $\{v_1, \cdots, v_n\}$ is a set of variables, where v_i has an associated finite domain of values, i.e., dom(v_i).
- I is an **initial** procedural block that specifies the initial state.
- S is an **always** procedural block that contains a set of statements specified by the following, where n is a natural number, B is a boolean condition and e (with subscript) is an expression:

$$\begin{array}{lll} S & ::= & v = e \\ & | & \textbf{wait}\ (n) \\ & | & \textbf{if}\ \ (B)\ S_1\ \textbf{else}\ \ S_2 \\ & | & \textbf{while}\ (B)\ S\ \textbf{end} \\ & | & \textbf{case}\ (v)\ e_1 : S_1 \ldots e_n : S_n\ \textbf{endcase} \\ & | & \textbf{begin}\ \ S_1; \ldots ; S_n\ \textbf{end} \end{array}$$

Definition 3.2: The *control flow graph* (CFG) of a Verilog program P is a graph $\langle N, Initial, \omega, \varepsilon, L\rangle$, where *N* is a finite set of nodes labeled by the program counter locations, *E* is a finite set of edges, $Initial$, ω and ε are specific nodes and denote respectively the beginning of initial block, the beginning and the end of the always procedural block and L is a labeling function that associates to each edge a statement, i.e., $L : E \rightarrow S$.

The control flow graph is built in a standard way. For example, the node that represents an “if” statement has two out-going edges labeled with the testing condition and its negation, pointing to the program locations of the “then” and “else” statements, respectively. Note that the case statement can always be rewritten in term of “if” statement but we have preferred to keep it in the abstract syntax since its presentation is more readable than its coding (we assume all the conditions are mutually exclusive).

B. Program Semantics

The behavior of the Verilog programs can be described using operational semantics. We define a set of transition rules over configurations. A configuration has the form $\langle t, \lambda, \sigma\rangle$ where t is a simulation time (a natural number), λ is a program counter location and σ is mapping from program variables to values.

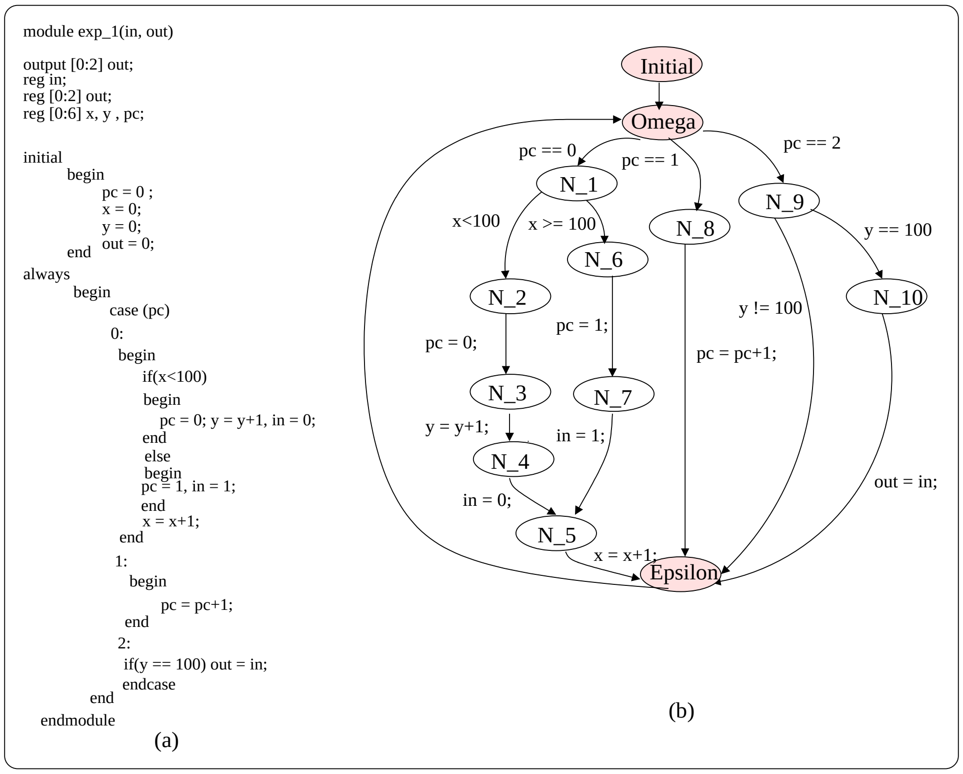
Definition 3.3: Given an edge η from a node n labeled with location λ and another node n labeled with λ', we define the transition rule according to the statement associated to the edge η, written $L(\eta)$ as follows:

- $v = e \qquad \langle t, \lambda, \sigma\rangle \longrightarrow \langle t, \lambda', \sigma[e/v]\rangle$
  where $\sigma[e/x]$ is the same as σ except the value of the variable v is now associated with the value e.
- **wait** (n)
  $$\frac{t < n}{\langle t, \lambda, \sigma\rangle \longrightarrow \langle t+1, \lambda, \sigma\rangle} \qquad \frac{t \geq n}{\langle t, \lambda, \sigma\rangle \longrightarrow \langle t, \lambda', \sigma\rangle}$$
- a Boolean condition $\dfrac{\sigma(B) \text{ is true}}{\langle t, \lambda, \sigma\rangle \longrightarrow \langle t, \lambda', \sigma\rangle}$
- $\lambda = \varepsilon$ or $\lambda = Initial$ and $\lambda' = \omega \qquad \langle t, \lambda, \sigma\rangle \longrightarrow \langle t+1, \lambda', \sigma\rangle$

IV. Path dependency Graph

Unfortunately, it is not always possible to know statically if a path of interest will be traversed during the execution. Consider the example in Figure 1 and assume that our property to check includes the statement $out == in$. We need to check if this statement can be evaluated to 0. As in is assigned more than one value throughout the program , it is not clear, by just analyzing the CFG, which value(s) will reach this statement. Therefore, we will refine the analysis by introducing the dependency between the CFG paths, which we call *path sequence*.

We will construct the path sequence by using two semantic functions [3]: The reachability condition and the state transformation functions [9]. Given a finite path π in the control flow graph, the reachability condition RC_π specifies a Boolean condition under which the path will be traversed while the state transformation ST_π



```
module exp_1(in, out)

output [0:2] out;
reg in;
reg [0:2] out;
reg [0:6] x, y , pc;


initial
        begin
              pc = 0 ;
              x = 0;
              y = 0;
        end   out = 0;
always
         begin
              case (pc)
              0:
               begin
                   if(x<100)
                   begin
                      pc = 0; y = y+1, in = 0;
                   end
                   else
                   begin
                   pc = 1, in = 1;
                   end
                   x = x+1;
               end
              1:
                 begin
                        pc = pc+1;
                 end
              2:
                if(y == 100) out = in;
                endcase
           end
   endmodule
                (a)
```

Fig. 1. An example Verilog program and its control flow graph

specifies the valuation of program variables obtained at the end of traversing this path. Both of these functions can be computed as follows[1]:

Definition 4.1: Let $\pi = n_1 \to \ldots \to n_m$ be a finite path in the CFG, we backwards define RC_π and ST_π by induction

- *Induction basis*: RC_π^m = **true** and $ST_\pi^m = V$. Being at the end of a path π, at time n, to traverse this path implies a true reachability condition and an identity state transformation.
- *Induction steps*: We define RC_π^k and ST_π^k according to the statements at $L(k, k+1)$
  - *v=e*: $RC_\pi^k = RC_\pi^{k+1}[e/v]$ $ST_\pi^k = ST_\pi^{k+1}[e/v]$
  - *Boolean condition*: $RC_\pi^k = RC_\pi^{k+1} \wedge B$ $ST_\pi^k = ST_\pi^{k+1}$

Finally, $RC_\pi = RC_\pi^1$ and $ST_\pi = ST_\pi^1$. We can represent any path by single transition as follows:

$$\pi := RC_\pi(V) \wedge V' = ST_\pi(V)$$

Definition 4.2: Let π be a finite path in the CFG and $b \in \text{dom}(v)$ a value, we say that a value b is active iff $RC_\pi(v)$ is true or $ST_\pi(v) = b$. Similarly, we say that a value b is deactive iff $RC_\pi(v)$ is false or $ST_\pi(v) \neq b$.

[1]Without loss of generality, we will not consider the simulation cycle time

Now, consider for example the paths ψ_0 and ψ_1 from Figure 1:

$\psi_0 : (Initial, \omega)$

$\psi_1 : (\omega,\ N_1),\ (N_1,\ N_2),\ (N_2,\ N_3),\ (N_3,\ N_4),\ (N_4,\ N_5),\ (N_5, \epsilon)$

We want to check whether ψ_1 will be executed after ψ_0 or not. To achieve this, we run a backward computation on the paths to get the ST_ψ and RC_ψ at each edge.

Let V be the ordered set of the program variables in, x, y, pc, out, for the example in Figure 1;

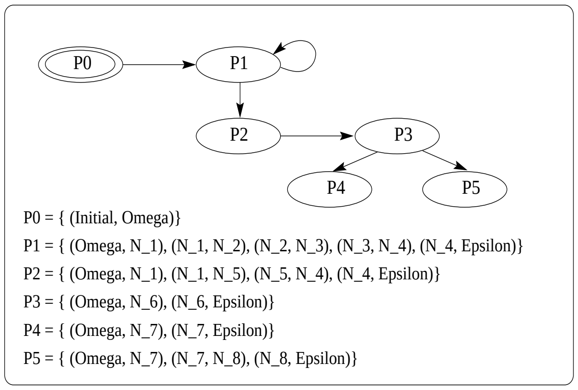
- $RC_{\psi_0}^{Initial}\{V\}$ = *true* = $RC_{\psi_0}, ST_{\psi_0}^{Initial}\{V\} = in, 0, 0, 0, 0, 0\} = ST_{\psi_0}$
- $RC_{\psi_1}^{\epsilon}\{V\}$ = *true*, $ST_{\psi_1}^{\epsilon}\{V\} = \{in, x, y, pc, out\}$
- $RC_{\psi_1}^{N_5}\{V\}$ = *true*, $ST_{\psi_1}^{N_5}\{V\} = \{in, x+1, y, pc, out\}$
- $RC_{\psi_1}^{N_4}\{V\}$ = *true*, $ST_{\psi_1}^{N_4}\{V\} = \{0, x+1, y, pc, out\}$
- $RC_{\psi_1}^{N_3}\{V\}$ = *true*, $ST_{\psi_1}^{N_3}\{V\} = \{0, x+1, y+1, pc, out\}$
- $RC_{\psi_1}^{N_2}\{V\}$ = *true*, $ST_{\psi_1}^{N_2}\{V\} = \{0, x+1, y+1, 0, out\}$
- $RC_{\psi_1}^{N_1}\{V\}$ = *true*$\wedge(x < 100) = (x < 99), ST_{\psi_1}^{N_1}\{V\} = \{0, x+1, y+1, 0, out\}$
- $RC_{\psi_1}^{\omega}\{V\} = (x < 99) \wedge (pc == 0) = (x < 99) = RC_{\psi_1}, ST_{\psi_1}^{\omega}\{V\} = \{0, x+1, y+1, 0, out\} = ST_{\psi_1}$

We can see from the above that ψ_1 will be executed after ψ_0. Using the methodology in the above example, we will build the path sequence which is the path between the control flow graph paths.

Definition 4.3: The *path sequence* is a tuple $\langle P_0, P, R\rangle$ where

- $P_0 \in P$ is an initial path
- P is a set of paths in CFG.
- and $R \subseteq P \times P$ such that $(\pi_1, \pi_2) \in R$ iff $RC_{\pi_1} \wedge RC_{\pi_2}[ST_{\pi_1}]$ is not false.

The path sequence provides a static check to ensure that some program state will be reachable. On the other hand, we follow the routine used in classic static analysis techniques, which assumes that a condition is true as long as no information that proves the inverse is available. Figure 2 shows the path sequence of our example in Figure 1. For instance, there is a transition between P_3 and P_4 because $RC_{\pi_4}[ST_{\pi_3}] \wedge RC_{\pi_3}$ is $pc = 1 \wedge pc + 1 = 2 \wedge y \neq 100$ which could be true.



P0 = { (Initial, Omega)}

P1 = { (Omega, N_1), (N_1, N_2), (N_2, N_3), (N_3, N_4), (N_4, Epsilon)}

P2 = { (Omega, N_1), (N_1, N_5), (N_5, N_4), (N_4, Epsilon)}

P3 = { (Omega, N_6), (N_6, Epsilon)}

P4 = { (Omega, N_7), (N_7, Epsilon)}

P5 = { (Omega, N_7), (N_7, N_8), (N_8, Epsilon)}

Fig. 2. Path Sequence of the example in Figure 1

Let's consider again the property about the expression $out == in$ in the example of Figure 1. We needed to check if this statement can be evaluated to 0. By examining the path sequence in Figure 2, we find out that the only value of in that can be reached is 1, hence we conclude that out will never be evaluated to 0.

V. Conclusion and Future Work

We have demonstrated in this paper how the idea of path sequence can achieve dependency analysis between program variables better than approaches based on syntactic relations only. It can also be used in order to remove redundant paths; paths which will never be traversed, or detect design violation, as well as other analysis goals. The path sequence was implemented as a part of a model reduction tool for hardware designs. Primary results showed that building the path sequence do not add a considerable time overhead for the verification process. Currently, we are extending the tool to support Verilog code with multiple procedural blocks.

References

[1] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, Germany, 1999.

[2] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[3] N. Francez. *Program Verification*. Addison-Wesley, 1992.

[4] C. Hankin. Program analysis tools. *Iternational Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[5] Y. Hsieh and S. Levitan. Control/data-flow analysis for vhdl semantic extraction. In *Proceedings of the 4th Asia-Pacific Conference on Hardware Description Languages*, pages 68–75, Hsin-Chu, Taiwan, August 1997.

[6] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on VHDL descriptions and its applications. In *Proceedings of Asian Pacific Conference on Hardware Description Languages*, pages 132–139, Bangalore, India, January 1996.

[7] Jing-Yang Jou and Chien-Nan Jimmy Liu. Coverage analysis techniques for hdl design validation. In *6th Asia-Pacific Conference on Hardware Description Languages*, Software Research Park, Fukuoka, Japan, Octobe 1999.

[8] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[9] H. Peng, Y. Mokhtari, and S. Tahar. Model reduction based on value dependency. In *Proceeding of IEEE International ASIC/SOC Conference*, pages 220–224, Washigton, DC, USA, September 2001.

[10] D. Sciuto, L. Baresi, and C. Bolchini. Software methodologies for VHDL code static analysis based on flow graphs. In *Proceedings of the conference with EURO-VHDL'96*, pages 406 – 411, Geneva, Switzerland, September 1996.

[11] IEEE standard 1076-1993. IEEE standard description language based on the VHDL hardware description language, 1993.

[12] IEEE standard 1364-2001. IEEE standard description language based on the Verilog hardware description language, 1995.

[13] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.