SYNTAX CODE ANALYSIS AND GENERATION FOR VERILOG

Mohamed Zaki and Sofiène Tahar
Dept. of ECE, Concordia University
{mzaki, tahar}@ece.concordia.ca

Abstract

In this paper, we present a syntax analyser tool for Verilog programs which can be used as a front end to debugging and program verification tools.

1. INTRODUCTION

Current hardware design complexity has made the verification and testing process difficult and expensive. Increasing testability of a design becomes one of the important issues during a design cycle. Recently static analysis based techniques for hardware programs have emerged as one of the successful methods to help reducing the cost of testing and verification. Usually, static analysis is applied before the verification phase in order to detect code violations; syntax and semantic violations or remove redundant information. It can also extract properties about the behavior of the design in order to guide the verification.

Static analysis [2] 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 is 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 in order to reduce and abstract the model size in order to speed up the model checking process. Static analysis is divided into two parts: Data flow analysis and control flow analysis. The control flow graph shows the sequence in which the program will be executed, while data dependency is concerned with estimating properties related to program variables. Alongside both techniques a data dependency graph

Verilog [15] is one of the most popular programming languages used for building software models of hardware system. It can be used to specify concurrent and sequential designs at different levels of abstraction; low level structural elements as well as high level behavioural programming.

To analyze Verilog designs efficiently, a graphical representation of the program syntax must be created. A program can be modeled by two graphs, a control data flow graph (CFG) which is concerned with capturing the control dependency relation in the program and a data dependency graph (DDG) [2] representing the syntactic relationship between program variables.

In this paper we present an automated tool for analyzing the syntax of Verilog programs to generate the respective CFG and DDG. Using the generated graphs, static analysis techniques such as abstraction and reduction [11] can be applied for hardware verification purposes and data flow analysis or slicing techniques [4]. Once done with graphs analysis, the user can backward generate another Verilog code, which can be applied for testing and debugging. To facilitate the code analysis and generation, the control and the data flow graphs are displayed by our tool in the dot format which is accepted by the graph visualisation tool Graphviz [5], from ATT labs, to display the CFG and DDG diagrams.

The rest of the paper is structured as follow. In Section 2, we define our system model including the abstract syntax (the control flow diagram) and the semantic representation of a program. In Section 3, we present the architecture of the tool. Section 4 discusses some applications of the tool, and Section 5, describes some related work. Section 6 finally concludes the paper.

2. VERILOG BEHAVIORAL REPRESENTATION

2.1. Verilog Subset

Verilog [15] is a hardware description language (HDL) used to specify a circuit at low level structural elements as well as high level behavioural programming [6]. It can be used to specify concurrent and sequential designs at different level of abstraction.

Example 1:

```
module exp_1(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;
        out = 0;
    end
    always
    begin
  if(x<100)
    begin
    pc = 0;
    y = y+1;
    in = 0;
            end
            else
            begin
                pc = 1;
    in = 1;
        end
        x = x+1;
    end
    always
if (y ==100) out = in;

    endmodule
```

Fig 1. Example Verilog Program

In Verilog, a program can be described by modules executed concurrently. A *module* consists of a number of threads, all executed concurrently. Threads can run continuously like *always* or *continuous assignment* threads, or run only once like *initial* threads. A variable in Verilog *(input, output, internal variable)* can be defined as a *register* or a *net*. The only allowable assignment for variables of type wire is either through declaration or by using a continuous assignment statement. The right hand side expression on the continuous assignment is re-evaluated every time one of its variables is changed; the result is passed as a new value to the assigned variable inside the procedural block. The behavior of the continuous assignment can be approximated as an always procedural block with only one statement

Always procedural blocks execute concurrently and indefinitely as they have no exit statements. We have identified a subset of behavioural Verilog for high-level synthesis to which we will restrict the discussion in the following sections. This subset is similar to the one used in [11]. We will assume that explicit use of time is not allowed in the behavioural specifications. As a result, signal assignments cannot have explicit delay clauses. In the next subsections, we will present the general syntax and the semantic models of the Verilog program. We will use the Verilog Code example in Fig. 1 to illustrate our approach.

2.2. Program Syntax

Definition 1:
A Verilog program P can be specified as a *tuple* $< V, I, A, S >$, where:

- $V = \{v_1, ..., v_n\}$ 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.

- A is a set of always concurrent procedural blocks where each procedural block contains a set of statements S specified as follows:

$$
\begin{aligned}
S \quad ::= \quad & v{=}e \\
| \quad & If(B) \ \ S_1 \ else \ S_2 \\
| \quad & While(B)S \\
| \quad & CASE(v) \ \ e_1{:}S_1...e_n{:}S_n \ \ endcase \\
| \quad & wait(n) \\
| \quad & begin \ \ S_1;...;S_n \ \ end
\end{aligned}
$$

Where n is a natural number, B is a Boolean condition and e (with subscript) is an expression.

Definition 2:
The control flow graph (CFG) of a Verilog program P is a directed graph $< N, E, Initial, \omega, \varepsilon, L, SG >$ (see Fig. 2), where N is a finite set of nodes labelled by the program counter locations, E is a finite set of edges, $Initial$, ω and ε are specific nodes and

denoting respectively the beginning of initial block, the beginning and the end of the concurrent always procedural blocks. L is a labelling function that associates to each edge a statement, i.e., $L : E \rightarrow S$. SG is the set of subgraphs where each $sg_i \in SG$ representing an always procedural block. ω and ε are connecting to the start and end edges of the subgraphs in a join/fork fashion thru special parallel edges with no labelling.
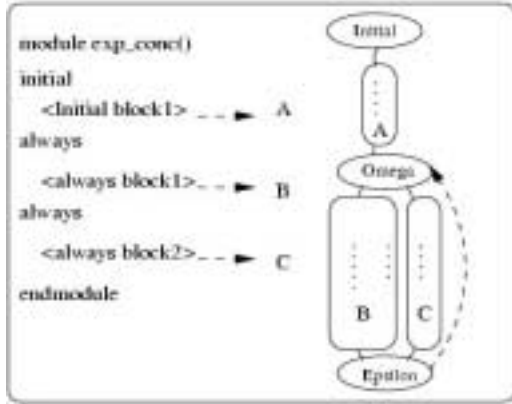

Fig 2.   Structure of the CFG

The control flow graph is built in a standard way. For example, the node that represents "if" statement has two out-going edges labelled 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 terms of "if" statements but we have preferred to keep it in the abstract syntax since its presentation is more readable than its coding (we assume all conditions are mutually exclusive). The CFG of the Verilog program of example 1 is shown in figure 3.

Data dependency graphs (DDG) represent the relationship between state variables. A DDG does not give any semantic information, only the type of dependency between variables whether it is an assignment dependency or a control dependency. The dependency between two variables is not always direct as it can be related through other variables. The direct dependency between variables can be divided into three categories:
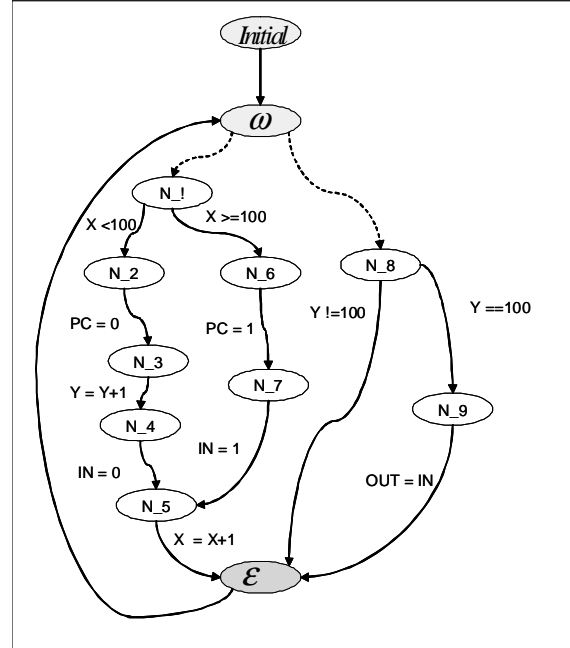

Fig 3.   CFG of the Verilog Example in Fig.1

- $v_1 \rightarrow v_2$: v_1 is used on an expression assigned to v_2.
- $v_1 \Re v_2$: v_1 and v_2 are related by a relational operator.
- $v_1 \mapsto v_2$: v_1 is used on a test statement affecting the evaluation of v_2.

Definition 3:

The data dependency graph DDG of a program P is a directed graph $< D, F >$, where:

- D is a set of nodes, each labelled by a variable $v_i \in V$
- $F \subseteq D \times D$ is a set of edges connecting the nodes based on one or more of the above three dependency categories. Each edge can be labelled by the types of dependency between the nodes' variables.

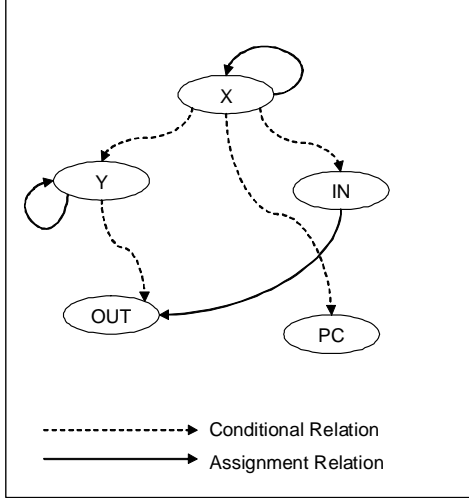The data dependency graph for the Verilog example in Fig. 1is shown in Fig 4.

Fig 4.   DDG of the Verilog Example of Fig. 1

2.3. Program Semantic

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

Definition 4:
   Given an edge η from a node n labelled with location λ and another node n' labelled with λ', we define the transition rule according to the statement associated to the edge η, written $L(\eta)$ as follows:

- $v = e$ $\quad < t,\lambda,\sigma > \rightarrow < t,\lambda',\sigma[e/v] >$

   where $\sigma[e/v]$ is the same as σ except the value of the variable v is now associated with the value e.

- WAIT(n)

$$\frac{t<n}{<t,\lambda,\sigma>\rightarrow<t+1,\lambda,\sigma>} \qquad \frac{t\geq n}{<t,\lambda,\sigma>\rightarrow<t,\lambda',\sigma>}$$

- A Boolean condition $\dfrac{\sigma(B)\ is\ true}{<t,\lambda,\sigma>\rightarrow<t,\lambda',\sigma>}$

- $\lambda = \varepsilon$ or $\lambda = Initial$ and $\lambda' = \omega$
$< t,\lambda,\sigma > \rightarrow < t+1,\lambda',\sigma >$

- Parallel Composition $\quad S = [S_1 \parallel ... \parallel S_n]$, $\lambda = \omega$ and $\lambda' = \varepsilon$

$$\frac{<t,\lambda_i,\sigma>\rightarrow<t,\lambda_i',\sigma'>}{<t,[\lambda_1\parallel..\parallel\lambda_i\parallel..\parallel\lambda_n],\sigma>\rightarrow<t,[\lambda_1\parallel..\parallel\lambda_i'\parallel..\parallel\lambda_n],\sigma'>}$$

where $i \in \{1,...,n\}$.

Let an Action *A* be statement of Boolean expression within a procedural block of parallel composition $S = [S_1 \parallel ... \parallel S_n]$ with shared variables [17]. This action can be identified as atomic action if no other blocks will be able to update this variable. We will extend this notion assuming that parallel blocks can share more than one variable and define a so-called high level synchronization where a block is dealt with as a critical section and is executed without interruption. Then the process has an exclusive access to the shared variables with no other blocks allowed to modify them during this period. Syntactically we will identify atomic regions as statements enclosed in angle bracket, i.e., $\left\langle S_i \right\rangle$ [17]

$$\frac{<t,\lambda_i,\sigma>\rightarrow^*<t,\xi,\sigma'>}{<t,\langle\lambda\rangle,\sigma>\rightarrow<t,\xi,\sigma'>}, \text{ where } \xi \text{ denotes termination.}$$

As each process is identified as an atomic region, its computation is reduced to one step computation. This reduction prevents interference from other parallel blocks. We will put the constraint that variables can only be updated within one procedural block, while others procedural blocks can only use this variable. This allows us to treat the execution of the parallel blocks as a sequential execution, while the final state of the program will not be updated until all process has been executed successfully.

3. TOOL ARCHITECTURE

In this section, we describe the developed tool which generates the CFG and DDG out of the Verilog subset discussed in section 2. The tool accepts at the input a Verilog file that includes the design to be analysed. After the analysis phase (which is performed by another tool), it will generate at the back end another Verilog program.

The tool was composed of different modules interacting together. The advantage of this modularity is the simplicity of updating its architecture by modifying the Verilog subset or enhancing its performance. The structure of the tool as shown in Fig. 5The modules of the tool architecture are:

Parser: Upon reading the Verilog file, the parser checks the syntactic correctness of the codes and

builds the parse tree representing the internal format of these files.

Verilog Pre-processor: Once the Verilog code is parsed and checked for any syntax violation, several modifications need to be conducted on the Verilog code before generating the CFG and DDG. A pre-processor flattens the Verilog code if is formed of several modules. It then transforms continuous assignments to always procedural blocks as mentioned in Section 2. It also transforms Non-blocking assignments to blocking assignments by introducing intermediate registers.

CFG and DDG generator The parse tree of the Verilog file is then translated into linked structures representing the control flow and the data dependency graphs (CFG and DDG) of the program.

Verilog Generator: At the back end of the tool, a translator generates the reduced synthesizable Verilog, which can be fed, e.g, to a model checking tool for verification.

In order to facilitate the analysis of the programs, the syntax representations (CFG and DFG) of the programs can be displayed by a graph visualization tool [5]. Graphical representation always helps understanding the behaviour of the designs.

4. APPLICATIONS

This tool can be used on the top of slicing tools, model checker tools. Program analysis is a process of estimating properties of the program at specific points. The information provided by a program is useful in compiler optimization, code generation, program verification, testing and debugging.

The tool can also be used for Verilog program debugging where programmers trace backward from a statement which is emitting a wrong value to figure out which statement is emitting a wrong behaviour can be used to help analyze counter examples generated from model checkers. Suppose that an error in detected or generated by a counterexample, dataflow analysis techniques can analyze the control flow graph in order to identify the piece of code that generated that error. The tool can help in the test generation phase. Testing is aimed at testing all control flow structure of the program, such as testing branches and testing control branches.

We have already used the tool as a front end of a static analysis tool which implements the cone of influence [10]. The variable reduction removes irrelevant variables to the

property and generates a reduced CFG and DDG. Using a path dependency algorithm, we have been able to remove unreachable CFG paths as well as unreachable expression [18].
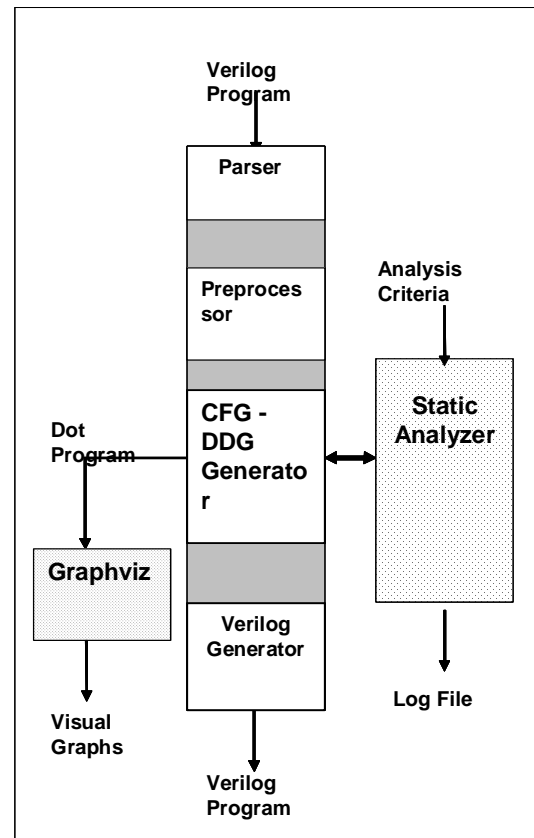

Fig 5.   Tool Architecture

5. RELATED WORK

One of the main static analysis approaches is program slicing [16]. Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics and deletes those parts of the program which can be determined to have no effect upon the semantics of interest [16]. There are a few works dealing with slicing application for hardware programming languages. For instance, the work in [8] applies program slicing to VHDL programs [14]. In order to analyze the VHDL code, along the data and control dependencies, 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.* [4]

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 system 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 one done by Baresi *et al.* [13] where the authors used data flow analysis techniques in order to identify deadlocks conditions within VHDL specifications. Hsieh *et al.* [7], applied data flow analysis techniques in their model abstraction algorithm, in order to abstract VHDL semantic. Commercial Verilog analysis tools include V2C which is a Verilog to C translator, and the LEDA static analysis tool.

6. CONCLUSION

In this paper, we have presented a syntax analyzer tool for Verilog programs. The approach is used as part of a static analysis tool which, based on a certain criteria, removes redundant information from the source code before the verification phase. Primary experiment results are optimistic. We are in process of extending our work for the analysis and reduction of a commercial ATM switch.

7. REFERENCES

[1] J. Bhasker, *A Verilog HDL Primer*, Star Galaxy Press, 1999.

[2] C.Hankin. Program Analysis Tools, Software Tools for Technology Transfer, 2(1), 1998.

[3] S. T. Cheng. Compiling Verilog into Automata. Technical Report UCB/ERL M94/37, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1994.

[4] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, *Program slicing for VHDL* In Correct Hardware Design and Verification Methods , LNCS 1703, Springer Verlag, 1999.

[5] E. Koutsofios and S. C. North. *Drawing Graphs with dot*. AT&T Bell Laboratories, Murray Hill NJ, U.S.A., October 1993.

[6] J. Fiskio-Lasseter and A. Sabry, *Putting operational techniques to the test: A syntactic theory for behavioral Verilog*. In Proc. of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99), Paris, 1999.

[7] Y.-W. Hsieh and S. P. Levitan, *Control/data-flow analysis for vhdl semantic extraction*, in Proc. of The 4th Asia-Pacific Conference on Hardware Description Languages, pp. 68--75, August 1997.

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

[9] B. W. Kernighan and D. M. Ritchie. The C *Programming Language*. Prentice Hall, 1988.

[10] R. P. Kurshan, *Formal verification in a commercial setting*. In Proc. of the Design Automation Conference, pages 258--262, Anaheim, CA, USA, June 1997.

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

[12] R. K. Brayton et al. *VIS: A system for verification and synthesis*. In Computer Aided Verification, LNCS 1102, Springer Verlag, 1996, pp.428-432.

[13] A. Balboni, M. Mastretti, and M. Stefanoni. *Static Analysis of VHDL Model Evaluation*. In Proc. of Euro-VHDL, pages 586--591, Grenoble, France, 1994.

[14] IEEE Standard VHDL Language Reference Manual: IEEE Standard 1076-1993, revised ed., 1994.

[15] IEEE Standard Verilog Language Reference Manual: IEEE Standard 1364-2001

[16] F. Tip, *A Survey of Program Slicing Techniques*, Journal of Programming Languages, Vol.3, No.3, pp.121-189, September 1995.

[17] K.-R. Apt and E.-R. Olderog, "*Verification of Sequential and Concurrent Programs*," Springer-Verlag, 1997, 2nd edition.

[18] Zaki, M.H, "*Syntactic Model Reduction for Hardware Verification*", M.A.Sc thesis, Dept. of ECE, Concordia University, Canada, April 2003.