SynAbs: Model Reduction Tool for Verilog Verification

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— **Current model checking tools suffer from the state space explosion problem due to the large number of generated states. In general, approaches like compositional verification and model reduction are used to tackle this problem. In this paper, we present a model reduction tool, called SynAbs, for Verilog verification. The reduction algorithms implemented are based on syntactic analysis and are applied prior to model checking. We have tested the efficiency of the algorithms using small examples. We have achieved reduction in both space and time requirements.**

I. Introduction

Model checking [6] is a fully automatic approach to verify a finite state machine against its temporal specifications. However, its application is limited by the size of the system under verification. Current model checking tools [16], [12], [10], [3] suffer from the state space explosion due to the large number of generated states. In general, *compositional verification* [6] and *model reduction* are used to tackle this problem. In compositional verification, one can verify separately each module in the system then composes the local results into a global property. However, in today's multi-million-gate designs, the size of one single module is usually beyond the capability of a model checking tool. Model reduction approaches are then used in order to reduce the module size prior to verification.

Model reduction approaches are based on abstract interpretation [7] which allows to reduce a concrete system (M) under verification to a more abstract and smaller one (M'). Both systems M and M' are connected by an abstraction relation which is *safe* with respect to a given property φ, namely it preserves the property. This means if the property holds for the abstract system, it holds for the concrete one as well. Syntactic abstraction techniques are usually based on the extensive analysis of the program syntax. They have several advantages as mentioned in [13]:

- They are more efficient than symbolic minimization algorithms such as the one described in [4], where explicit transition graph of the minimized system has to be constructed. The output of a syntactic abstraction tool is a text file which can be directly parsed and analyzed by a model checker.
- Other reduction methods based on BDD or partial reduction can be applied on the output of the abstraction tool.

Verilog HDL (Hardware Description Language) is a popular language for hardware specification, design and testing. Verilog programs exhibit a rich variety of behaviors including event-driven computation and shared variables concurrency. In this paper, we describe a tool for reducing and abstracting hardware designs written in Verilog. We will deal with a synchronous subset of Verilog accepted by the SMV model checker but without considering concurrency.

The rest of the paper is structured as follows. In Section II, we present the model reduction tool **SynAbs**, describing its implementation and its different modules. Some experimental results are discussed next in Section III. Section IV describes some relevant related works. Finally, section V concludes the paper.

II. The Reduction Tool

SynAbs (Syntactic Abstraction) is a tool implementing model reduction algorithms based on syntactic analysis. **SynAbs** accepts as inputs two text files: a Verilog file that includes the design to be verified and a property file that includes the specification described in ACTL [5]. The reduction algorithms analyze the input Verilog program, based on the property provided by the user, and generate a reduced Verilog code. The process is fully automated, i.e, no interaction is required by the user. The reduced Verilog program file, can be input to a model checker such as SMV [12] and VIS [16] for verification.

The structure of **SynAbs**, as shown in Figure 1, is composed of several connected modules which will be briefly described next. The advantage of this modularity is the simplicity of upgrading the tool architecture by extending the Verilog subset or enhancing its performance for example.

A. Parser

Upon reading the Verilog and the ACTL temporal properties files, the parser checks the syntactic correctness of the codes and builds the parse tree representing the internal format of these files which is translated into linked structures describing the control flow and the data dependency graphs (CFG and DDG) of the program.

The Verilog program structure consists of set of variables with associated finite domain of values, continuous assignment statements, **initial** procedural blocks that specify the initial state and **always** procedural blocks that contain a set of statements. The statements supported by **SynAbs** are *blocking* and

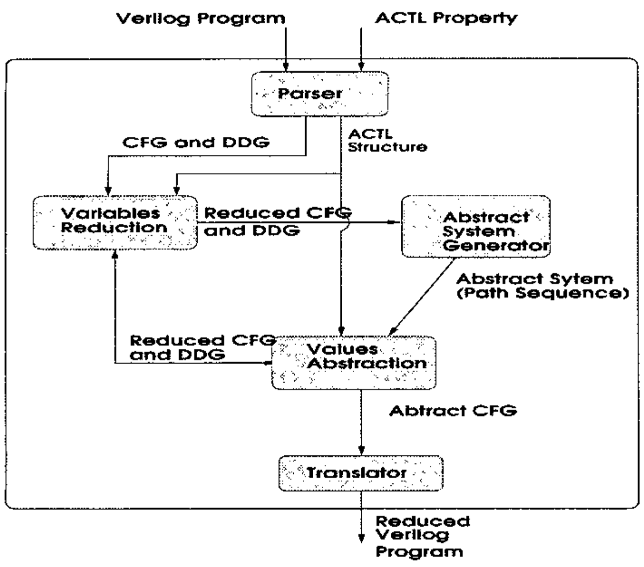
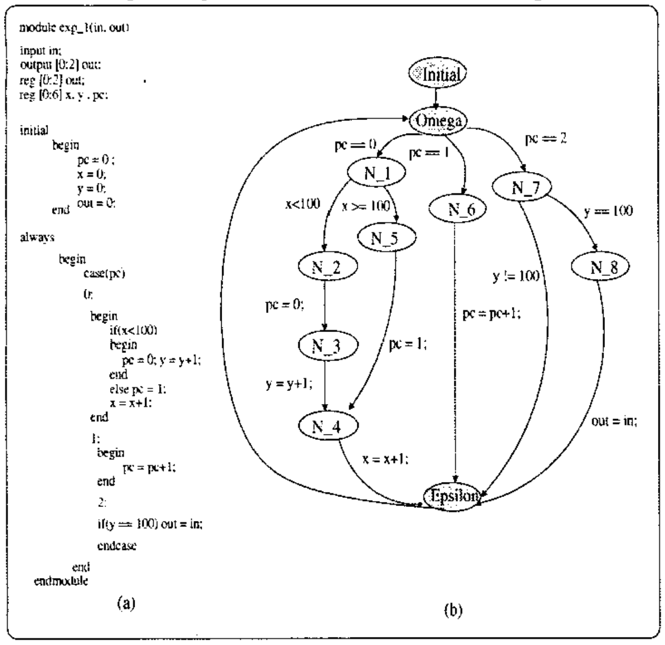


Fig. 1. SynAbs Architecture

non-blocking assignments, *if/else* statements, *case* statements, *wait* statements and *@(condition)* statements.

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 denoting respectively the beginning of initial blocks, the beginning and the end of the always procedural blocks. L is a labeling function that associates to each edge a statement i.e. $L : E \rightarrow S$. A Verilog example and its CFG are shown in figure 2



```
module exp_1(in, out)

input in;
output [0:2] out;
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;
                 end
                 else pc = 1;
                 x = x+1;
             end
             1:
              begin
                    pc = pc+1;
              end
              2:
              if(y == 100) out = in;
              endcase
          end
   endmodule
```

Fig. 2. An example Verilog program and its CFG

Data dependency graph (DDG) is a graph which represents the relationship between state variables, whether it is an assignment or control dependency. DDG of a Verilog program is a directed graph $\langle D, F\rangle$, where D is a set of nodes, each labeled by a variable and $F \subseteq D \times D$ is a set of edges connecting the nodes based on one or more of the dependency types. Each edge can be labeled by the types of dependency between the nodes' variables.

We consider the temporal logic specification language **ACTL** [5]. **ACTL** is a subset of **CTL** [18], where only *universal* path quantifier, namely only **A** is allowed. The set of well-formed universal computation tree logic (**ACTL**) are constructed from a set of atomic propositions AP which represent properties of individual states, the standard boolean operators, the temporal operators **X**, **U** and **V**, and the universal path quantifier **A**.

B. Variable Reduction

In order to enhance the reduction of programs prior to verification, the Cone of Influence algorithm (COI) [17] is implemented inside **SynAbs**. Using the set of variables in the property (of interest), the *variables reduction* module removes irrelevant variables to the property and generates reduced CFG and DDG which are input to the *Path Sequence Generator.*

The COI set of a certain variable is built by fixpoint iteration on the CFG; each iteration adds new variables to the list. The iteration stops when there are no more variables to be added. After the COI set is created, only edges with variables included in the set are kept, other edges are removed and a reduced CFG is created.

C. Values Abstraction

A values abstraction algorithm [15] is implemented in **SynAbs**, where we will partition the value domains of the program variables into active and deactive values which respectively affect and do not affect the property. The active values are kept in the program while the deactive values are replaced by one typical deactive value.

In order to achieve our goal, we will use two semantic functions adapted from the Floyd proof system [9]:

- The reachability condition RC_τ, associated with every path τ of the CFG, is a boolean condition under which this path is traversed.
- The state transformation function ST_τ computes the values of the program variables at the end of the path, provided that this path is traversed.

These functions are obtained by backward induction over the paths of the CFG. We can represent any path π as $\pi := RC_\pi(V) \wedge V' = ST_\pi(V)$.

The domains partition is done first by selecting the nodes (key nodes) influencing the property. From these key nodes, backward to the node ω constitute the key paths. In addition, the variables (nodes) that appear along those paths are also considered as key nodes and hence their paths are also added to the set of the selected paths. Next, we use the reachability condition and the state transformation of each path to partition the domain of each variable v into disjoint active domain, written ACTIVE (v), and deactive domain, written DEACTIVE (v).

The active domain will contain the values affecting directly the property and will remain unabstracted, while the deactive domain can be abstracted by using a representative value and therefore will contain one single value.

We consider the example in Figure 2 and we assume that our property includes the statement $out == in$. The first path that influences the specification is $\pi_1 = \omega \rightarrow N_7 \rightarrow N_8 \rightarrow \varepsilon$ where $RC_{\pi_1} = pc = 2 \wedge y = 100$ and $ST_{\pi_1} = out = in$. Hence, we know that the value 2 and 100 are active values of the variable pc and y respectively. The variables that appear along this path are pc, y and out. By considering the other paths that influence these variables, we will partition the values as the following: ACTIVE $(y) = \{100\}$, DEACTIVE $(y) = \{0\}$, ACTIVE $(x) = \{100\}$ and DEACTIVE $(x) = \{0\}$ for the other variables their domains have not changed and are considered as active values.

After the abstraction, some redundant variables will be removed [1] and the Values Abstraction module creates an abstract CFG. This module also generates new un-initialized inputs which will be used to implement the non-deterministic choice.

D. Path Sequence Generator

Sometimes, it is not possible to know statically if a path of interest will be traversed during the execution. In our example, we have determined that $pc = 2$ will lead to a state that will verify the property which involves $out = in$. However, it is not clear if pc will have a value equal to 2 or not. Therefore, we will refine the analysis by introducing the dependency between the CFG paths called *path sequence*. This path sequence will be provided in case the Values Abstraction module needs to gather the domain of a certain variable [19].

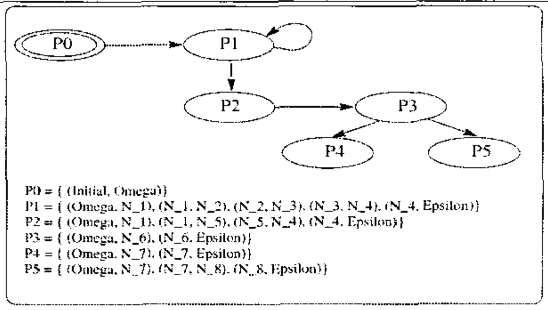
The path sequence is a tuple $\langle P_0, P, R\rangle$ where $P_0 \in P$ is an initial path, P is a set of paths in CFG and $R \subseteq P \times P$ such that $(\pi_1, \pi_2) \in R$ iff $RC_{\pi_1} \wedge RC_{\pi_2}[ST_{\pi_1}]$ is not false.

The path sequence represents another alternative to be used in the partition but it also provides a static check to ensure that some state will be reachable. We follow the routine used in classic static analysis techniques which assumes that a condition is true as long no information available that prove the inverse. Figure 3 shows the path sequence of our example in Figure 2. For example, 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.

E. Verilog Generator

At the back end of the tool, a translator [20] generates from the CFG a reduced Verilog program which can be input to model checker tools for verification as shown in Figure 4.

[1]Suppose these two assignments: $x = 10$ and $y = x$, and there is a path from the first assignment to the next one and the property to be checked involves y but not x. Then, the resulted control flow graph will include the following assignments $x = 10$ and $y = 10$. By sending this CFG to the variable reduction module, statements like $x = 10$ can be removed.



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. 3. Path Sequence of the Verilog example

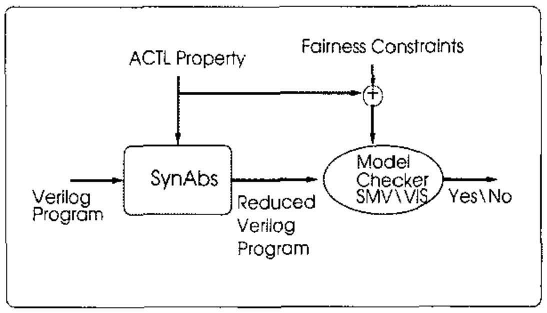


Fig. 4. **SynAbs** and model checking

III. Experimental Results

For performance evaluation, we used a modified version of the example in Figure 2 and with 32-bits registers. The program is formed of one module, which has two inputs *reset* and *in*, and two outputs *out1* and *out2*. The values of the outputs are changed depending on conditions on the inputs and the internal variables. Suppose we want to verify the following properties

Prop 1: $AG(\neg reset \rightarrow AF(out1 == 01))$.

Prop 2: $AG(x == 1001 \rightarrow AF(out1 == 01))$.

Prop 3: $AG(x == 1001 \rightarrow AF(out2 == 999))$.

A comparison between the verification with SMV reduction and with **SynAbs** reduction is shown in Table 1. A decrease is achieved in both the BDD size and the verification time.

IV. Related Work

Many abstraction techniques have been implemented to help the verification of hardware and software designs. The degree of automation, the degree of abstraction and property preservation are major factors that affect the implementation and the usage of the abstraction. Many surveys discussing those issues can be found in the literature [7], [8].

In [13], Namjoshi and Kurshan extended and automated a syntactic abstraction approach, called *predicate abstraction*, which translates a variable with large value domain into a set of predicates. The produced output model is a reduced program text. The algorithm proposed was implemented in a tool called **AutoAbs** [14]. A similar implementation of predicate abstraction was applied on VHDL programs in [2].

Yorav and Grumberg [11] proposed two forms of syntactic abstraction; "path reduction", which is based on suppressing

		Model Checking					
		With SMV reduction			With **SynAbs** reduction		
Example	Property	Status	BDDs	Time	Status	BDD	Time
E32	(1)	Verified	164712	11.87 sec	Verified	2870	0.17 sec
	(2)	Verified	638328	281.98 sec	Verified	2870	0.17 sec
	(3)	Verified	542553	175.59 sec	Verified	1191	0.17 sec

TABLE I
VERIFICATION RESULTS OF SAMPLE PROPERTIES USING SMV

the control flow graph paths that do not affect the property and "dead-variable reduction", where some of the successors of states for which the variables of interest are not used are excluded. However, the abstraction we proposed in this tool is more aggressive in the sense that not only the paths not affecting the variables of interest are removed, but also the variables of interest domains are abstracted. Another advantage of our technique is the absence of termination restriction, which makes it more suitable for hardware design.

In [1], Bharadwaj and Heitmeyer proposed a syntactic approach based on variable hiding, where variables only used on assignment expressions of the variables of interest, are replaced by their domains. However, such approach is restricted to a limited subset of the program variables.

Our proposed reduction is also related to other work like localization reduction [17]. However, our approach is an extension of it, because we analyze the dependency between the values of variables in addition to the dependency between variables, thus the dependency relation is more accurate.

V. CONCLUSION

In this paper, we have presented **SynAbs**; a tool that applies model reduction algorithms on Verilog designs. The approach uses syntactic analysis and generates a smaller program compared to the original one. In order to evaluate the performance of **SynAbs**, we verified some examples, using the SMV model checker, before and after being abstracted by **SynAbs**. Primary results were satisfactory. We are now using **SynAbs** to verify larger examples.

In our approach, the solution is based on the syntactic analysis of the program source code. The code reduction can be combined with other model checking or reduction tools to improve its reduction efficiency even more. Moreover, since the size of the source code is usually smaller than that of its state space or other intermediate forms, this approach uses little resources (CPU time and memory space). The reduction approach implemented can be extended for other HDLs like VHDL.

REFERENCES

[1] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1):37–68, January 1999.

[2] M. Bourahla and M. Benmohamed. Predicate abstraction and refinement for model checking vhdl state machines. In *Proc. Seventh International Workshop on Formal Methods for Industrial Critical Systems*, University of Malaga (Spain), July 2002.

[3] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169, Chicago, IL, USA, July 2000.

[5] E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, Vol.16(No. 5):1512–1542, Sept 1994.

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

[7] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics, 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[8] D.Dams. abstraction in software model checking: Principles and practice. *Model Checking of Software, 9th International SPIN Workshop*, April 2002.

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

[10] R. P. Kurshan. Formal verification in a commercial setting. In *Design Automation Conference*, pages 258–262, 1997.

[11] K.Yorav and O. Grumberg. Syntax-directed model checking of sequential programs. *Journal of Logic and Algebraic Programming*, 52-52:129–162, 2002.

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

[13] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer-aided Verification*, volume 1855 of *LNCS*, pages 433–449, Chicago, IL, USA, July "2000". Springer Verlag.

[14] K.Namjoshi N.Amla, R.Kurshan. Autoabs: Syntax-directed program abstraction. 2002.

[15] H. Peng, Y. Mokhtari, and S. Tahar. Model reduction based on value dependency. In *Proc. of IEEE International ASIC/SOC Conference*, Washigton, DC, USA, September 2001.

[16] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 428–432, New Brunswick, NJ, USA, 1996. Springer Verlag.

[17] R.P.Kurshan. *Computer Aided Verification of Coordinating Processes: The Automata Theoritic Approach*. Princeton University press, 1994.

[18] T. Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999.

[19] M. Zaki, Y. Mokhtari, and S. Tahar. A path dependency graph for Verilog program analysis. In *Proc. First Northeast Workshop on Circuits and Systems*, pages 109–112, Montreal, Quebec, Canada, June 2003.

[20] M. Zaki and S. Tahar. Syntax code analysis and generation for verilog. In *IEEE Canadian Conference on Electrical and Computer Engineering*, pages 235– 240, Montreal, Quebec, Canada, May 2003.