

Shobha Vasudevan · E. Allen Emerson · Jacob A. Abraham

Improved Verification of Hardware Designs through Antecedent Conditioned Slicing

Abstract Static slicing has shown itself to be a valuable tool, facilitating the verification of hardware designs. In this paper, we present a sharpened notion, *antecedent conditioned slicing* that provides a more effective abstraction for reducing the size of the state space. In antecedent conditioned slicing, extra information from the antecedent is used to permit greater pruning of the state space. In a previous version of this paper, we applied antecedent conditioned slicing to safety properties written in propositional logic, of the form $G(\textit{antecedent} \implies \textit{consequent})$. In this paper, we use antecedent conditioned slicing to handle safety and bounded liveness property specifications written in temporal logic. We present a theoretical justification of our technique. We provide experimental results on a Verilog RTL implementation of the USB 2.0 functional core, which is a large design with about 1100 state elements (10^{331} states). The results demonstrate that the technique provides significant performance benefits over static program slicing using state-of-the-art model checkers.

Keywords Hardware Verification, Model Checking, Program Slicing, LTL property, Antecedent Conditioned Slicing, Hardware Description Languages, Verilog RTL

1 Introduction

Abstraction techniques have been widely used to reduce the state space explosion problem in model checking [6]. Although model checking has been successfully employed in hardware verification, formal verification at the full chip level is still a challenging problem. Simulation based verification techniques are applied for large parts of the chip. Many state space reduction techniques have been applied to alleviate

the increasing space requirements of model checkers. These techniques have mostly been applied at the boolean netlist (gate) level. It is also possible to apply these techniques at the register transfer level (RT-Level), which is typically described using hardware description languages (HDLs). It has been shown that this leads to significant benefits over gate level manipulation of hardware. This is due to the modularity and the intuitiveness that accompany RT-Level operations. Additionally, these source-to-source transformations can borrow techniques from software because of their structural similarity to software programs.

Program Slicing, introduced by Weiser [34, 32] is a widely used abstraction technique used to statically analyze programs and retain parts of the source code relevant to the application [33, 15]. Program slicing (or static slicing) has been used for a variety of tasks, including debugging [33], maintainence [15] testing [12]. within a behavioral domain.

Program slicing has been applied to various software engineering disciplines where manipulation of large programs, and hence their decomposition is desirable. In the past, program slicing has been extended to HDLs [7, 17, 29]. Program slicing has also been successfully applied to hardware verification by Clarke et al [7, 30]. A variation of static program slicing, that improves over it is *conditioned slicing* [2, 4]. Conditioned slicing augments static program slicing by specifying some initial states of interest in the slicing criterion. Conditioned slicing has been shown to produce smaller and more meaningful abstractions than static slicing. Conditioned slicing has been used for program comprehension [21], reuse [4], migration [3] and re-engineering [5].

In this paper, we introduce an abstraction technique for property based hardware verification using conditioned slicing, called *antecedent conditioned slicing*. The principle of our verification technique is to use the information provided by the antecedent in a given LTL [22, 20] property to prune the verification state space. We use the information from the antecedent in a given property specified in temporal logic to apply conditioned slicing to programs written in Verilog RTL. We form source-code level abstractions, *antecedent conditioned slices* that are more precise and relevant than static program slices.

S. Vasudevan, J. A. Abraham
Computer Engineering Research Center, University of Texas at Austin,
Austin, TX 78712
E-mail: {shobha,jaa}@cerc.utexas.edu

E. A. Emerson
Department of Computer Sciences, University of Texas at Austin
E-mail: emerson@cs.utexas.edu

Conditioned slicing has been defined for programs in software. We extend conditioned slicing to HDLs. We argue that our technique is effective for hardware verification and provide a theoretical basis for it. We also provide experimental results to show the substantial performance gains of using this technique, as compared to state-of-the-art model checking techniques. Experiments are shown on an RTL implementation of the USB 2.0 protocol.

In a previous version of this work [26], we showed the application of our technique to safety properties of the form $G(\textit{antecedent} \implies \textit{consequent})$. Safety properties written in propositional logic form a simpler class of properties and have limited applications. In order to be useful at any reasonable scale, temporal properties have to be dealt with. In this paper, we extend this idea of conditioned slicing based verification to safety and bounded liveness properties written in temporal logic. We show the modifications and the clauses in our algorithm to incorporate these temporal properties. We handle properties written in LTL, of the form $G(\textit{antecedent} \implies X\textit{consequent})$, and $G(\textit{antecedent} \implies EF\textit{consequent})$, where the *antecedent* and *consequent* can also be temporal logic formulae.

The principal contributions of this paper are:

- We extend conditioned slicing to HDLs and apply it to verification.
- We introduce a novel abstraction technique for verification, antecedent conditioned slicing. To the best of our knowledge, this is the first time that information from the antecedent of an LTL property has been exploited for creating abstractions.
- We do not operate on a model of the hardware, but the actual RT-Level implementation of the design. This, according to [1] is a relevant issue in contemporary hardware verification environments.
- Our abstractions are relatively simple to apply, since they only involve statement deletion. They do not require any manual expertise or intervention.

The rest of this paper is organized as follows. Section 2 gives a background of slicing techniques and conditioned slicing that are necessary for understanding our technique. Section 3 show the dependence graph based techniques to obtain a conditioned slice. Section 4 extends conditioned slicing to HDLs. In Section 5, we introduce and describe antecedent conditioned slicing using examples. Subsection 5.1 gives an algorithm for computing antecedent conditioned slices, while Subsection 5.2 gives the theoretical basis for our abstractions. In Section 6, we provide the experimental results using our technique. We discuss the applications and limitations of our technique in Section 7. We conclude with Section 8.

2 Slicing Techniques

Slicing, in the most general sense, is a program transformation involving statement deletion, that preserves some projection of the semantics of the original program. The aspect

of the program that must be preserved is application specific, and is captured by the slicing criterion. We present here, some necessary background for program slicing.

Definition 1 Static slicing criterion

A slicing criterion of a program P with an input alphabet Σ , is a pair $\langle i, V \rangle$ such that i is a program point in P and $V \subseteq \Sigma$.

A set of statements I_s is said to *affect* the values of V at i in a given slicing criterion $\langle i, V \rangle$, if I_s defines a subset of V that is used in i .

Definition 2 Static slice for programs

A slice S of a program P on a slicing criterion $\langle i, V \rangle$ is a subset of the statements of P that might affect the values of V at i .

A static slice preserves the projection of the semantics of the original program for every possible execution of the program. This can result in very large slices. [21, 18]. Many slicing algorithms have been proposed as variations of static slicing, to create smaller slices [18, 27, 31]. A detailed survey of program slicing techniques can be found in [28].

2.1 Conditioned Slicing

Canfora et al [2] introduced the notion of *conditioned slicing*, that forms a theoretical bridge between static and dynamic slicing. Conditioned Slicing augments static slicing by introducing a condition that specifies the initial set of states in the criterion. It does not give specific inputs, unlike dynamic slicing. This slicing technique, therefore allows slicing with respect to the initial constraints in the program. We present some basic definitions of conditioned slicing that appear in the literature.

Definition 3 Conditioned Slicing criterion

Let Σ be the input alphabet of the program P . Let C be a first order predicate logical formula on the variables in Σ . A conditioned slicing criterion is a triple $\langle C, i, V \rangle$, where i is a statement in the program, and $V \subseteq \Sigma$.

Definition 4 Conditioned Slicing for programs

A conditioned slice of a program P on a conditioned slicing criterion $\langle C, i, V \rangle$ consists of all the statements and predicates of P that might affect the values of the variables in V at i , when the condition C holds true.

Tip [27] introduced a more restricted form of conditioned slicing called constraint based slicing. In all these cases, the *condition* that specifies the set of initial states, and is used for slicing is a first order predicate logic formula.

In situations where the initial set of constraints for the program analysis are known, this technique can be employed to get much smaller slices than those produced by static slicing.

Conditioned slicing has been automated with significant success on C and WSL code [10,9]. More recently, analogous to backward and forward slicing [25], backward and forward conditioning were introduced. [14]. In forward conditioning, a statement is deleted if, when the condition is satisfied, it cannot be executed. In backward conditioning, a statement is deleted if, when executed, it cannot lead to the condition being satisfied. In this paper, wherever it appears, conditioning refers to forward conditioning.

3 Dependence graph based slicing

Ottenstein and Ottenstein define slicing as a reachability problem in a dependence graph representation of the program [24]. They use Program Dependence Graphs (PDGs) [13] for static slicing of single procedure programs. The statements and predicates of a program correspond to the vertices of the PDG and the edges correspond to data and control dependences between statements. In dependence graph based slicing approaches, the slicing criterion is identified with a vertex v in the PDG. The i in the slicing criterion $\langle i, V \rangle$ corresponds to v in the PDG, while V stands for the set of all variables defined or used at v .

For slicing of multi-procedure programs, System Dependence Graphs (SDGS) were introduced [16]. An SDG combines the *procedure dependence graphs* of all the called procedures of a program, along with the *program dependence graph* of the main program by allowing edges that can model procedure calls.

Let the program dependence graph for a program P , denoted by G_P be a directed graph.

Definition 5 Control Dependence Edge

A control dependence edge from v_1 to v_2 , where v_1 is a predicate vertex, denoted by $v_1 \rightarrow_c v_2$, implies that the truth of the predicate expression represented by v_1 determines whether or not v_2 is executed.

Definition 6 Flow Dependence Edge

A flow dependence edge from v_1 to v_2 , denoted by $v_1 \rightarrow_f v_2$, implies that there is some variable x , that is defined at v_1 and used at v_2 and there is an execution path from v_1 to v_2 along which, there is no assignment to x .

3.1 Conditioned Slicing using PDGs

We present here, conditioned slicing, using a dependence graph approach. To the best of our knowledge, such a treatment of conditioned slicing has not been shown before.

Figure 1 shows an example program written in pseudocode. The PDG for the program is shown in Figure 2. In order to find a static slice for the program with respect to slicing criterion $\langle 11, B \rangle$, we find all the reaching definitions of B at node 11. Then, we find the set of all reachable nodes from these nodes in the PDG. This set $\{1, 2, 3, 4, 6, 7, 9\}$,

```

begin
1:      read(N);
2:      A = 1;

3:      if (N < 0)
4:      {
5:          B = f(A);
6:          C = g(A);
7:      }
8:      else
9:          if (N > 0)
10:         {
11:            B = f'(A);
12:            C = g'(A);
13:         }
14:         else
15:            {
16:               B = f''(A);
17:               C = g''(A);
18:            }
19:         print(B);
20:         print(C);
21:     end

```

Fig. 1 Example Program written in pseudocode

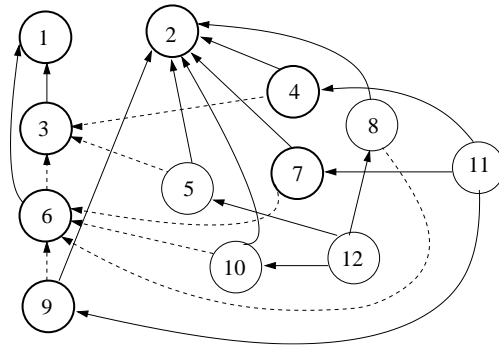


Fig. 2 Program Dependence Graph of the program. The solid edges denote data dependency and the dashed edges denote control dependencies. The vertices in bold denote the Static slice of the program with respect to the variable B at statement 11.

gives us the desired static slice. The nodes are shown in bold in the figure. set of ALL reachable may be wrong write vertices instead of nodes

Now, consider the conditioned slicing of the program with respect to the slicing criterion $\langle C, 11, B \rangle$, where C corresponds to the predicate $(N < 0)$. To obtain the conditioned slice for a given predicate C , we *project* the PDG with respect to the predicate, and then use the static slicing algorithm on the projected PDG. Figure 3 shows the application of this technique to obtain the conditioned slice for the mentioned criterion. Initially, all the vertices in the graph are drawn dotted. All the statements that would get executed when the predicate C is satisfied, are marked, and the corresponding vertices in the graph are made solid. The graph is then traversed only for solid (marked) vertices. The set of all vertices reached during the traversal are made bold. This set $\{1, 2, 3, 4\}$ gives the desired conditioned slice. The con-

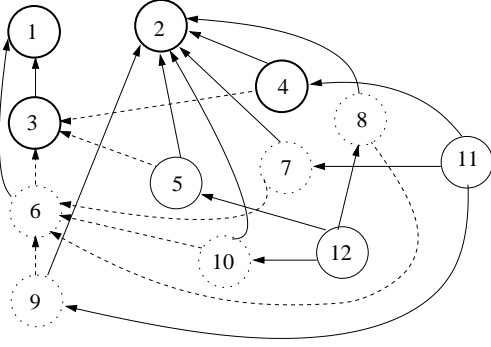


Fig. 3 Conditioned Slice with respect to the predicate $N \neq 0$. A vertex is made solid if it is ever executed and made bold if it gets traversed while computing the slice. The dotted vertices are not executed when the predicate is true

ditioned slice, therefore, contains only those statements that get executed when C evaluates to *true*. It is evident from this example that the conditioned slice is typically much smaller than the static slice.

4 Conditioned Slicing for HDLs

We present here, an extension of conditioned slicing to HDLs. The HDL computational paradigm differs fundamentally from traditional languages, since HDLs model non-halting, reactive systems with concurrently running processes. The processes are not explicitly called, as they are in a program's procedures. Instead, they communicate through signal dependency[7]. In order to extend conditioned slicing for HDLs, we use a few definitions from earlier work in program slicing [34] and its extension to HDLs [7, 29] with minor modifications. We also use extend the definitions from earlier work in conditioned slicing [4, 2].

Let M be a Verilog program² with k concurrent processes P_i , such that $\parallel_{i=1}^k P_i$, where \parallel is the parallel composition operator [29]. The processes communicate with each other through shared signals. The sensitivity list of a process p is the list of shared signals that can cause the "calling" or re-evaluation of p . When a signal in the sensitivity list gets assigned, p is re-evaluated or called with the new value of the signal.

Definition 7 Signal dependence

A process region P is said to be signal dependent on a statement i , if i assigns a value to a signal which is present in the sensitivity list of P .

Definition 8 Inter process CFG

An inter-process control flow graph G for a Verilog program is the structure $\langle G_1, G_2 \dots G_k, E_{sd} \rangle$ where $G_1, G_2 \dots G_k$

² The term "program" is used with the same meaning as in [7] for VHDL designs. A Verilog program is a set of concurrent processes that executes as a series of simulation cycles.

are the control flow graphs representing the processes in the program, and E_{sd} is the set of edges representing the signal dependencies between the processes.

Definition 9 Inter process SDG

An inter-process def/use graph for a Verilog program is a structure $\langle G, \Sigma, D, U \rangle$, where Σ is the set of signals in the program, D is a function mapping the nodes of G to $\Delta(\Sigma)$ and U is a function mapping the nodes of G to $\gamma(\Sigma)$, where $\Delta(\Sigma)$ is the set of signals defined and $\gamma(\Sigma)$ is the set of signals used in the statements corresponding to the nodes.

This graph is also called a system dependence graph. Program Slicing of HDLs has been modeled as a dependence graph node reachability problem in [7]. Each concurrent process has a corresponding PDG. The PDG for each process is modified for HDLs by making provision for inter-process communication through signals. Implicit procedure calls are generated in the inter-process control flow graph, whenever a signal is potentially assigned. The PDGs of the processes are connected appropriately to form a System Dependence Graph (SDG). The slices are computed by following the chains of dependences represented in the edges of the SDG. We extend the node reachability problem to include conditioning of HDLs.

Definition 10 Static Slicing for HDLs

An inter-process slice S_p within a M , on a given criterion $\langle i, V \rangle$ is an executable subset of M obtained recursively containing all the statements that may affect the values of V at i within the process P in which i is defined, and all the slices on the slice criterion $\langle i_s, V_s \rangle$ where i_s is the set of statements defining the set of signal V_s on which process P is dependent.

Definition 11 Conditioned process

A conditioned process $P(C)$, for a process P and a condition C , is an entity containing the set of statements of P that can be executed when C holds true.

Definition 12 Conditioned Program

A conditioned program with respect to the condition C is represented as $M(C) \equiv \parallel_{i=1}^k P(C)_i$ where $P(C)_i$ is a conditioned process.

Definition 13 Inter-process Conditioned Slicing

An inter-process conditioned slice S_c for a Verilog program M , on a given criterion $\langle C, i, V \rangle$ is a subset of M obtained recursively containing

- all statements in the scope of condition C , that affect the values of V at i within the process P where i is defined
- all the conditioned slices with respect to the slice criterion $\langle C, i_s, V_s \rangle$, where i_s is the set of statements that define the set of signals V_s on which P is signal dependent.

The conditioned slice of a Verilog program M can be computed using its SDG representation. The condition C in

the slicing criterion is applied to all the PDGs of all the processes P_i in M . The resulting conditioned process $P(C)_i$ is marked in its PDG. The conditioned program $M(C)$, is thus obtained from all the conditioned processes. $M(C)$ is now marked on the SDG. The conditioned slices are computed by finding the transitive closure of the control, flow and signal dependencies of the conditioned program, $M(C)$ in the SDG.

Figure 4 illustrates an example program of a hardware system, in Verilog HDL. The three processes $P1$, $P2$ and $P3$ correspond to the concurrently executing `always@...` blocks in Verilog. The parentheses of the always blocks list the signals on which each process is dependent. Now, consider the slicing criterion $\langle C, end_{P1}, result \rangle$, where C corresponds to the predicate $valid = true$ and end_{P1} corresponds to the `end` statement of process $P1$. We apply this predicate on each of the three processes, to obtain the corresponding conditioned processes, $P(C)_1$, $P(C)_2$ and $P(C)_3$. The resulting conditioned program, $M(C)$, is shown in Figure 5. The portions of the code which can be executed, when the predicate $valid$ is true, are shown. The conditioned program is now analyzed for determining the slice with respect to the given criterion. The transitive closure of the data, control and signal dependencies of the relevant variables yields the program of Figure 6. It can be seen that $P2$ is included in the slice due to the signal dependence of $P1$ on $P2$ with respect to the variable `reset`. The statement in $P3$ is eliminated, since the variables in the slicing criterion do not have any dependencies on it. However, the process stub is maintained, in order that the behavior of the slice with respect to fairness constraints remains the same as the original program.

5 Antecedent conditioned slicing

We use conditioned slicing for verification of hardware designs described in Verilog HDL. Our technique aims at reducing state space of the design, by slicing away the parts of the design irrelevant to the property being verified. We assume that the properties are specified as temporal logic formulae specified in LTL.

For these properties, we can use the antecedent to specify the set of initial states that we are interested in. *The antecedent therefore, forms the condition in the slicing criterion.* All the statements that would get executed when the antecedent is true (or the condition is satisfied) are included in the slice. The statements on the paths that cannot get executed when the antecedent is false, are removed. The reduced program still preserves its behavior *with respect to the property being checked.* We therefore create property preserving abstractions using conditioned slicing.

All prior art in verification using program slicing uses static program slicing techniques. While slicing property specifications written in temporal logic, these techniques retain the set of all statements of the program where the antecedent is true, *as well as those where it is not.* This is because static

slicing retains all possible executions of the relevant variables.

However, in property based verification, *we do not need to check the states where the antecedent is false.* In these cases, static slices might be too large and include statements that are not of interest. We introduce a precise abstraction on the basis of conditioned slicing, *antecedent conditioned slices.*

The key idea in these abstractions is that they utilize information from the antecedent. Antecedent conditioned slicing, therefore, forms more meaningful abstractions than static slicing.

These are also significantly smaller slices, that can reduce verification state space significantly. We describe them in detail in the next section.

5.1 Computing antecedent conditioned slices

In our property language, we permit LTL properties h of the form

$$\begin{aligned} G(a \implies c) \\ G(a \implies X^{=n}c) \\ G(a \implies aU_s c) \end{aligned}$$

where a and c are propositional formulas

$X^{=n}q$ means at distance $n \geq 0$, q holds, *i.e.* $XXX \dots n$ times.

U_s represents the strong until.

We also permit bounded LTL properties, that we represent by $[h]^k$ for a given bound k . We detail the theoretical basis for bounded LTL properties, since we want to justify our experiments that were performed using a bounded model checker. We permit bounded properties of the form

$$\begin{aligned} [G(a \implies c)]^k \\ [G(a \implies X^{=n}c)]^k \\ [G(a \implies Fc)]^k \end{aligned}$$

Definition 14 Antecedent conditioned slice

Let h be a bounded LTL formula with a bound k . Let $h = [G(a \implies C)]^k$, where C represents any of the permitted LTL formulas in our property language and a is a proposition. Let V_h be the set of all variables in h . The antecedent conditioned slice S_a with respect to the slicing criterion $\langle a, i, V_h \rangle$ is a subset of P such that,

If $C = X^{=n}$, $0 \leq n \leq k$, S_a consists of

- Conditioned slice of P with respect to the slicing criterion $\langle a, i, V_h \rangle$
- All the statements s that can be executed n time steps in the future
- All static slices with respect to $\langle i_s, V_s \rangle$ where V_s is the set of variables in s and i_s constitutes the set of statements where every variable in V_s is defined.

If $C = F$, S_a consists of

```

      P1                P2                P3
always @(clk or reset)  always @(clk)      always @(clk or flag)
begin
  if (valid)
  {
    result = a+b;
  }
else
  {
    result = a-b;
  }
end

always @(clk)
begin
  reset = init;
  if (valid)
  {
    flag = 1;
  }
else
  {
    flag = 0;
  }
end

always @(clk or flag)
begin
  start = flag;
end

```

Fig. 4 Example Verilog program. The three “always” blocks represent concurrent processes.

```

      P1                P2                P3
always @(clk or reset)  always @(clk)      always @(clk or flag)
begin
  if (valid)
  {
    result = a+b;
  }
end

always @(clk)
begin
  reset = init;
  if (valid)
  {
    flag = 1;
  }
end

always @(clk or flag)
begin
  start = flag;
end

```

Fig. 5 The conditioned program, for the predicate (valid = true). Each process is a conditioned process.

```

      P1                P2                P3
always @(clk or reset)  always @(clk)      always@(clk or flag)
begin
  if (valid)
  {
    result = a+b;
  }
end

always @(clk)
begin
  reset = init;
end

always@(clk or flag)
begin
end

```

Fig. 6 The slice obtained by statically slicing the conditioned program.

- Conditioned slice of P with respect to the slicing criterion $\langle a, i, V_h \rangle$
- All the statements s that can be executed k time steps in the future
- All static slices with respect to $\langle i_s, V_s \rangle$ where V_s is the set of variables in s and i_s constitutes the set of statements where every variable in V_s is defined.

We outline the algorithm for computing the antecedent conditioned slice for different classes of LTL formulae in Figure 7. S_a is the required antecedent conditioned slice.

The procedure *antecedent_conditioned_slice()* computes the antecedent conditioned slice over all time steps n ¹. The antecedent conditioned slice at every time step $t + 1$, P_{t+1}

is recursively defined as the entity obtained by slicing the antecedent conditioned slice of the previous time step, P_t with respect to the antecedent in the current time step, C_t . (Other conditions in the slicing criteria are as given in the algorithm.)

This simply means that all the statements that would get executed when the antecedent is true in the current time step, are retained, along with all the statements in future time steps that would be executed when the antecedent is true.

The procedure *get_conditioned_slice()* computes the conditioned slice for a given conditioned slicing criterion $\langle C, e, V \rangle$. A statement is retained in the slice if it is executed when the condition C holds true, and deleted if it is executed when C is false. If it does not depend on the truth value of C , it is retained in the slice. The resulting *conditioned program* is then statically sliced with respect to the slicing criterion $\langle e, V \rangle$. This ensures that all the statements that affect the

¹ In Verilog programs describing sequential hardware circuits, a clock is explicitly modeled in the design. Successive time steps are, therefore, according to the progression of this clock. In terms of the Kripke structure of program P , computing n time steps corresponds to n transitions.

Algorithm antecedent_conditioned_slice (P : Verilog program,
 h : LTL property)

```

begin
   $h = [G(a \implies C)^k]$ 
   $P_0 = P$ ;
  if  $C = c$ ,
     $S_a = \text{get\_conditioned\_slice}(P, \langle C, e, V_h \rangle)$ ;
  else if  $C = X^n c$ 
    for every time step  $t > 0$ , while  $t < n$ 
      begin
         $P_{t+1} = \text{get\_conditioned\_slice}(P_t, \langle C_{t+1}, e, V_h \rangle)$ ;
         $S_a = P_n$ ;
      end
  else if  $C = Fc$ 
    for every time step  $t > 0$ , while  $t < k$ 
      begin
         $P_{t+1} = \text{get\_conditioned\_slice}(P_t, \langle C_{t+1}, e, V_h \rangle)$ ;
         $S_a = P_k$ ;
      end
  end

get_conditioned_slice ( $P$ ,  $\langle C : \text{condition}, e, V \rangle$ )
begin
   $S = P$ ;
  for every process  $p$  in  $P$  /*process where e is
    defined */
    for every statement  $x$  in  $p$ 
      if  $x$  is executed when  $C$  is true
        retain( $x$ ); /*retain x in S*/
      else if  $x$  is executed when  $C$  is false
         $S = S - x$ ; /* slice x */
      else if  $x$  does not depend on a truth value of  $C$ 
        retain( $x$ );
      for every variable  $v_{sens}$  in process  $p$ 's sensitivity list
        /* conditioned slice of signal
        dependencies */
         $S = \text{get\_conditioned\_slice}(S, \langle C, e_{sens}, v_{sens} \rangle)$ 
         $S = \text{get\_static\_slice}(S, \langle e, V \rangle)$ ;
    return( $S$ );
end

```

```

get_static_slice ( $P$  : Verilog program,  $\langle e : \text{program point},$ 
   $V : \text{set of variables} \rangle$ )
begin
   $Q = P$ 
  for every variable  $v \in V$ 
    if  $v$  is not an input
      for every statement  $x$  is the statement at point  $p$  in  $P$ 
        if  $x$  defines  $v$  as  $d(v)$ 
          retain( $x_v$ )
           $Q = \text{get\_static\_slice}(Q, \langle p, d(v) \rangle)$ 
        else
           $Q = P - x$ 
  return  $Q$ 
end

```

Fig. 7 Algorithm for antecedent conditioned slicing.

variables V in the conditioned program are retained and the others are deleted.

The procedure `get_static_slice()` describes the procedure to obtain a static program slice. We have given a high level overview of this algorithm, as applied to programs, to ease understanding. The details of applying this to HDLs can also be found in [29, 7].

In the verification step, the antecedent conditioned slice S_a is passed through a model checker along with the prop-

erty h . In case the property is not verified, a counterexample is returned.

Figure 8, shows a typical Verilog state machine that we use to illustrate antecedent conditioned slicing.

```

always @ (clk)
begin
  state = next;
  case(state)
  S1:
    begin
      count = 1;
      next = S2;
    end
  S2:
    begin
      count = 2;
      next = S3;
    end
  S3:
    begin
      count = 3;
      next = S4;
    end
  S4:
    begin
      if (flag)
        next = S1;
      else
        next = S4;
    end
  endcase
end

```

Fig. 8 Example Verilog code showing a state machine

Let $h1$ be an LTL formula where $h1 = [G((state = S4) \wedge (flag) \implies (next = S1))]^k$. This is of the form $[G(a \implies c)]^k$. Now, the static slice will retain all the statements which define the variable `state` along with `next`, `flag`, `S4` and `S1`. However, in the antecedent conditioned slice, we can get rid of the statements that define `state`, but do not appear in the antecedent, as explained in the algorithm. So, the antecedent conditioned slice is an intersection of the antecedent conditioned slices with respect to the criteria $\langle (state = S4), end, \{state, S4, flag, next, S1\} \rangle$ and $\langle (flag, end, \{state, S4, flag, next, S1\}) \rangle$. This is shown in Figure 9(a). It may be noted that according to Definition 13, although `next` is defined in the other states, it is in the scope of the condition (antecedent) only in the portion of code included in the slice.

Let $h2$ be an LTL formula such that $h2 = [G((state = S1) \implies X(state = S2))]^k$. This is of the form $[G(a \implies Xc)]^k$. In the antecedent conditioned slice, we will include all the statements that correspond to the conditioned slice with respect to $\langle (state = S1), end, \{state, S1, S2\} \rangle$. To this, we include all the statements that will be executed in the next clock cycle, since there is an `X` operator in the formula. The resulting antecedent conditioned slice is shown as the code on the right in Figure 9(b).

```

always @ (clk)
begin
  state = next;
  case(state)
  S4:
    begin
      if (flag)
        next = S1;
      end
    endcase
end

```

(a) Antecedent conditioned slice for $h1$

```

always @ (clk)
begin
  state = next;
  case(state)
  S1:
    begin
      next = S2;
    end
  S2:
    begin
      next = S3;
    end
  endcase
end

```

(b) Antecedent conditioned slice for $h2$ **Fig. 9** Antecedent conditioned slices of Figure 8 for properties $h1$ and $h2$

5.2 Verification using antecedent conditioned slicing

Let $M = (S, R, I)$ be the Kripke structure representing a Verilog program P , where S and R represent the states and the transitions and I represents the set of initial states of the program.

Each LTL property h is interpreted over full paths of the underlying structure. Each bounded LTL property $[h]^k$ is interpreted over finite paths of length k in the underlying structure. Then, for M , we define $M \models h$ to mean \forall full paths $x \in M$ starting at any $s_0 \in I$, $M, x \models h$.

For bounded formulas $[h]^k$, we define $M \models [h]^k$ to mean \forall finite paths $x = s_0, s_1 \dots s_k \in M$ starting at $s_0 \in I$, $M, x \models h$ in the standard temporal semantics for finite timelines.

Let h be a bounded LTL formula of the form $[G(a \implies c)]^k$. The antecedent conditioned slice of P with respect to the criterion $\langle a, i, V \rangle$ is shown by $P|_a$. From the algorithm in Definition 14, $P|_a$ comprises only those set of states of P , where the antecedent a is true. Let $N = (S', R', I')$ be the Kripke structure representing the Antecedent conditioned slice $P|_a$, such that S' and R' represent the states and the transitions and $I' = I$. In general, the slice structure N for M and bound k is the substructure of M comprising $M|_a$, or the set of states in M that satisfy a , and those states of M at a distance at most k from $M|_a$.

For $h = G(a \implies c)$, a bound of 0 suffices, and the slice N for M is just $M|_a$.

For $h = G(a \implies X^k c)$, $h = [G(a \implies c)]^k$, $h = G(a \implies F^{\leq k} c)$, $h = [G(a \implies Fc)]^k$, $h = [G(a \implies aU_s c)]^k$, a bound of k suffices and slice N corresponds to $M|_a$ and all the states of M at a distance of at most k from $M|_a$.

For the special case when $h = [G(a \implies X^n c)]^k$ and $n \leq k$, we can define slice N to consist of $M|_a$ and all the states of M at a distance of at most n from $M|_a$. We will use this streamlining in our experimental results where n is typically of the order of 2 or 3 and $k = 50$.

In order to prove the correctness of antecedent conditioned slicing, we need to prove that the property h holds on the original program if and only if holds on the antecedent conditioned slice.

We state the correctness theorem for all the LTL formulas discussed above and outline an illustrative proof.

Theorem 1 $M \models h \iff N \models h$ where $h = [G(a \implies c)]^k$.

Proof:

We say a state $t \in M$ is “close” if there a path $x \in M$ from some initial state $s_0 \in I$ to t of length at most k . Let $M \models h$. Then, $N \models h$ due to the following reasoning. All “close” states in M satisfy $a \implies c$. Since N is a substructure of M , this is also true of all the close states in N . Now, let $N \models h$. Then, $M \models h$ due to the following reasoning. All close states in N satisfy $a \implies c$. These states include all the close states of $M|_a$. Thus all close states of M that satisfy a must also satisfy $a \implies c$. All states of M that satisfy $\neg a$, including the close states, satisfy $a \implies c$ vacuously.

Therefore, we infer that all close states of M satisfy $a \implies c$. QED.

Theorem 2 $M \models h \iff N \models h$ where h is one of the following

$$\begin{aligned}
&G(a \implies X^k c) \\
&G(a \implies F^{\leq k} c) \\
&[G(a \implies Fc)]^k \\
&[G(a \implies X^n c)]^k, 0 \leq n \leq k \\
&[G(a \implies aU_s c)]^k
\end{aligned}$$

We outline the proof for the correctness of the antecedent conditioned slice for an illustrative formula, where $h = [G(a \implies Fc)]^k$.

Proof:

We need to prove that $M \models h \iff N \models h$. Let k -shell be the states reachable from forward or backwards paths of length k from every state in $M|_a$. $N = M|_a + k$ -shell.

Let $M \models h$. Then, $N \models h$ due to the following reasoning. All “close” states in M satisfy $a \implies c$. Since N is a substructure of M , this is also true of all the close states in N .

Let $N \models h$. Then, $M \models h$ due to the following reasoning. Pick an arbitrary path x of length k in M .

Case 1: $x \in M|_a$

All close states in N satisfy $a \implies Fc$. These states include all the close states of $M|_a$. Thus all close states of M that satisfy a must also satisfy $a \implies Fc$. Therefore all states of x satisfy $a \implies Fc$.

Case 2: $x \notin M|_a$ This means x starts in a state that satisfies $\neg a$. Let x remain in $\neg a$ within the k -shell. In this case, the distance is too much to cover in k steps, and x will never reach $M|_a$. Therefore, all states of x satisfy $\neg a$, and thereby satisfy $a \implies Fc$ vacuously.

Case 3: $x \notin M|_a$ This means x starts in a state that satisfies $\neg a$. Let x not remain in $\neg a$ within the k -shell. Let x exit the k -shell and enter $M|_a$. In this case, by the reasoning in *Case 1*, all states in x satisfy $a \implies Fc$. QED.

The proofs for the other formulas are similar.

6 Experimental Results

We provide experimental results on the Verilog RTL implementation of the USB 2.0 Function Core. The USB is a standard interconnect between computers and peripherals. This core operates at full and high speed rates (12 and 480 Mb/s). The source code can be found at [8]. The properties chosen for verification were from the USB 2.0 core specification document [11]. These properties were involved with the many state machines in the implementation, and were essentially control based properties. The safety and bounded liveness properties expressed in temporal logic have been listed below as LTL formulae, where k is the bound (in this case $k = 50$) and also explained in English, for the sake of readability. The variables used in the LTL formulae are the signal names in the Verilog code. The Verilog state machines that correspond to the given property are given in parentheses.

- **P1:** $G((state == SPEED_NEG_FS) \Rightarrow X((mode_hs) \wedge (T1_gt_3.0ms) \Rightarrow (next_state == RES_SUSP)))$
If the machine is in the speed negotiation state, then in the next clock cycle, if it is in high speed mode for more than 3ms, it will go to the reset/suspend state. (Main State Machine)
- **P2:** $G((state == IDLE) \wedge (ep_stall) \wedge (pid_PING) \wedge (mode_hs) \Rightarrow \neg(token == ACK))$. If the machine is in the IDLE state and high speed mode, if a stall is forced, then a PING token is ignored (or an acknowledgement is not sent out.) (Main Protocol State Machine)
- **P3:** $G((tokenout) \wedge (buf0_na) \wedge (buf1_na) \Rightarrow (signal == NACK))$. If the OUT token is received and both buffers are not available, the NACK handshake is issued. (Main Protocol State Machine)
- **P4:** $G(\neg(suspend_clr) \Rightarrow \neg(state == RESUME) \wedge \neg(state == RESUME_REQUEST))$. If the *suspend* bit is not cleared, then the machine is not resuming from the SUSPEND state. (Main State Machine)
- **P5:** $G((crc5err) \vee \neg(match) \Rightarrow (state == IDLE) \wedge \neg(send_token))$. If an packet with bad CRC5 is received,

or if there is an endpoint field mismatch in the IDLE state, then the token is ignored. (Main Protocol State Machine)

- **P6:** $G((state == CRC1) \wedge (tx_ready) \Rightarrow X(tx_ready \Rightarrow X(state == IDLE)))$
If the machine is ready to transmit in the CRC state, the IDLE state should be reached after two states. (Transmit/Encode State machine.)
- **P7:** $G((state == OUT2B) \wedge \neg(abort) \wedge \neg(pid_sequence_err) \wedge \neg(no_bufs) \wedge \neg(to_small) \wedge \neg(to_large) \Rightarrow (token_pid_sel_d == ACK))$
If the machine is in the OUT state, and it is not aborted, and there is no error in the process id, and buffers are available and the data is not too small or too large, then an acknowledgement token is sent out. (Main Protocol State Machine)
- **P8:** $G((state == SPEED_NEG_J) \wedge (chirp_cnt_inc) \wedge (chirp_cnt == 3'h1) \Rightarrow F(state == SPEED_NEG_HS))$
If the machine is in the "J" speed negotiation mode and a counter is initialized, then eventually, high speed mode is reached. (Main State Machine)
- **P9:** $G((state == RESUME_WAIT) \wedge \neg(idle_cnt_clr) \Rightarrow F(state == NORMAL))$
If the machine is waiting to resume operation, and a counter is set, eventually (after 100 mS) it will return to normal operation. (Main State Machine)
- **P10:** $G((state == OUT) \wedge (abort) \Rightarrow X(state == IDLE))$
In any OUT state, if the *abort* signal is asserted, the machine gets into IDLE mode. (Main Protocol State machine)
- **P11:** $G(((state == SPEED_NEG_K) \vee (state == SPEED_NEG_J)) \wedge (se0_long) \Rightarrow (XX((T1_gt_3.ms) \wedge (mode_hs)) \Rightarrow (state == RES_SUSP)))$
If the machine is in the speed negotiation state, and *se0* is asserted for a long time, then if after two cycles, the machine is in high speed mode and 3 mS have elapsed, the machine will either go into RESET or SUSPEND states. (Main State Machine)
- **P12:** $G(\neg(wb_req) \Rightarrow (state == IDLE))$
If a writeback request is not received, the machine remains in the IDLE state. (Interface State Machine.)
- **P13:** $G((state == IDLE) \wedge (match_r) \wedge \neg(ep_disabled) \wedge \neg(pid_SOF) \Rightarrow (state == IDLE))$
If there is an endpoint mismatch and an SOF token is not received, then the machine remains in the IDLE state. (Main Protocol State Machine)

We used the SMV [23] model checker for our experiments. All experiments were performed using a 3GHz Intel Pentium 4 processor with 1 GB ram. Since the USB is a large design with approximately 10^{331} state elements, model checking this design using SMV alone resulted in memory overflow. Hence, in order to provide a baseline for our technique, we use Bounded Model Checking with a uniform

Property Checked	BMC Original	BMC Static Sliced	BMC Conditioned Slicing	Proof Result
P1	11.47	11.02	0.96	Unsat
P2	43.59	41.31	30.37	Unsat
P3	136.3	87.95	44.14	Unsat
P4	27.36	27.02	0.75	Unsat
P5	227.27	201.07	39.27	Unsat
P6	304.51	100.14	0.76	Unsat
P7	68.95	42.69	9.86	Unsat
P8	15.57	8.26	1.17	Sat
P9	19.56	4.8	1.32	Sat
P10	477.53	361.37	13.87	Unsat
P11	2.16	2.01	0.96	Unsat
P12	85.24	81.12	0.7	Unsat
P13	85.49	44.55	4.69	Unsat

Table 1 Comparison of execution times (in seconds) taken for verification of properties by the original program, the static slice and the antecedent conditioned slice. A bound of 50 was given to BMC for all experiments.

bound of 50 for all properties. The results were generated using the SAT-based BMC utility of the Cadence-SMV tool.

The results of our experiments are presented in Table 1. The first and second columns provide the execution times of BMC when running on the original program, and when running on the static slice with respect to a given property. In the third column, we show the performance increase due to the application of conditioned slicing. We can observe that the performance increase in BMC due to static slicing is not very high as compared to the original program. In contrast, there is a tremendous gain in bounded model checking performance due to antecedent conditioned slicing of the design, when compared to the original as well as the statically sliced design. We will discuss the import of these results in the next section.

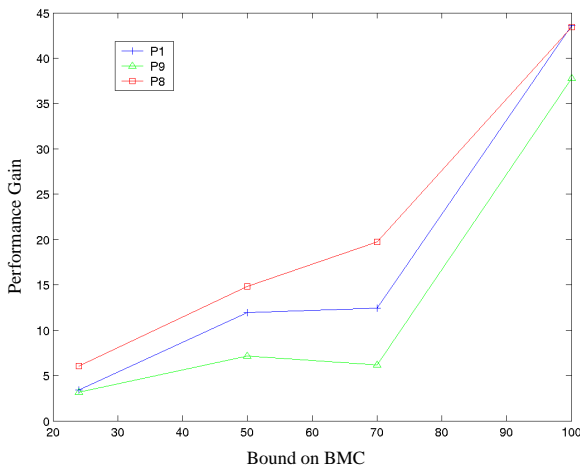


Fig. 10 Graph showing the performance gain of antecedent conditioned slicing for increasing bounds on BMC

Properties $P1$ to $P7$ and $P10$ to $P13$ are safety properties. $P8$ and $P9$ are liveness properties. Since no counterexamples to these properties are generated within the given

bound, the safety properties are partially verified. The liveness properties do not find a sequence that satisfies the property for the given bounds. We increased the bounds on these properties with significant performance gains.

Figure 10 shows the performance of some sample properties after applying our algorithm for increasing bounds of BMC. We find that there is a spectacular increase in performance due to antecedent conditioned slicing of the RT-Level design. We placed a reasonable bound of 50 on the number clock cycles we verified. Communication protocol designs, however, need to be verified for much higher bounds. As the bound increases, the performance gain scales too.

Our algorithm performs well even on safety properties that do not hold and produce a counterexample as well as liveness properties that hold within the given bound. We do not reproduce them here, since they require different bounds each time.

7 Discussions

An important issue we'd like to address is the improvement of conditioned slicing over static slicing. Static slicing has been applied to HDL verification before, with performance speedups. However, theoretically, static program slicing has not been shown to be different from the *cone of influence reduction* (COI) used by existing model checkers [19]. Clarke et al [7], while comparing their static slicing technique to COI reductions, mention that COI reduction is similar to building a dependence graph for the program, and then deciphering relevant variables using graph reachability. The dependence graph may be constructed on the HDL source code (slicing), or on the synthesized netlist. This shows that the only difference between static slicing and COI reductions, is their application domain (pre or post encoding). Semantically, the two are not different. Any performance gains due to static slicing, therefore, are due to the ease of *model generation*, as opposed to that of *model verification*.

In contrast, conditioned slicing creates a different program (or design) from static slicing or COI reductions. The antecedent conditioned slice forms a new entity that does not bear similarity in structure or meaning to the original program, but retains the behavior with respect to the property in question. The performance gains due to antecedent conditioned slicing, therefore, are due to the powerful abstractions created by this technique. Although when combined with static slicing, the overall performance gain may include the model generation component, the tremendous gains in performance are primarily due to the reduction in the complexity of model verification.

Our algorithm for antecedent conditioned slicing is entirely structural and based on syntactic transformations. Consequently, it suffers from some inherent weaknesses. For instance, the efficiency of the algorithm depends on the type of property being verified and the structure of the program being verified. In the case of semantic transformations, like abstract interpretation etc., the effectiveness of the abstraction

is not property or program dependent. Also, when a property requires reasoning over many time steps, the size of the antecedent conditioned slice increases. In the case of properties where the antecedent changes, all future behavior of the program would need to be retained for each time step. In these cases, the antecedent conditioned slice would not be too much smaller than the original program.

However, when applied to specific practical applications and properties, the abstraction can be very useful in reducing state space. In that respect, processor verification is an excellent application for our abstractions. The antecedent, which is the instruction word in the single instruction machine, does not change through the duration of the property.

8 Conclusions

In a preliminary version of this work [26], we have shown that our technique is effective for the simple case of safety properties written in propositional logic, of the form $G(\textit{antecedent} \implies \textit{consequent})$. In this paper, we have demonstrated that our abstraction based property verification technique using antecedent conditioned slicing is effective for safety and bounded liveness properties written in temporal logic. Our proposed methodology has been shown to maintain correctness. It lends itself easily to automation, especially to be built on existing model checkers. Our abstractions, antecedent conditioned slices, are exact, and therefore do not produce spurious counterexamples and false negatives. Of course, the accompanying disadvantage is the loss of generality due to the property specific nature of the abstraction. The experimental results show that the technique scales very well, and produces exponential performance speedups when compared to state-of-the-art techniques. The technique therefore seems very promising and can be applied to different domains of hardware and software verification, like verification of microprocessors and device drivers. Future work would focus on these varied verification application domains, where state-of-the-art model checkers are not very efficient.

References

1. Mark Aagaard, Vlad Ciobotariu, Jason Higgins, and Farzad Khalvati. Combining equivalence verification and completion functions. In *Formal Methods in Computer-Aided Design (FMCAD 2004)*, 2004.
2. G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607, 1998.
3. G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.
4. G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca. Software salvaging based on conditions. pages 424–433, 1994.
5. G. Canfora, A. De Lucia, and M. C. Munro. An integrated environment for reuse reengineering c code. *Journal of Systems and Software*, 42:153–164, 1998.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
7. E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.
8. USB Source Code. <http://www.opencores.org/pdownloads.cgi/list/usb>.
9. S. Danicic, C. Fox, M. Harman, and R. Hierons. Consit: A conditional program slicer. pages 216–226, 2000.
10. M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, Mark Marman, Chris Fox, and M. P. Ward. Consus: A scalable approach to conditional slicing. In *IEEE Proceedings of the Working Conference on Reverse Engineering*, pages 181–189, 2002.
11. USB Specification Document. <http://www.usb.org/developers/docs/>.
12. E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of Second Irvine Software Symposium*, pages 131–145, 1992.
13. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
14. C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In *9th IEEE International Workshop on Program Comprehension*, pages 89–97, 2001.
15. K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. In *IEEE Transactions on Software Engineering*, pages 751–761, 1991.
16. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, 1988.
17. M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on vhdl descriptions and its applications. pages 132–139, 1996.
18. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
19. R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
20. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, 1985.
21. A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. page 9, 1996.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
23. K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.
24. K. J. Ottenstein and L.M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.
25. T. Reps and T. Bricker. Illustrating interference in interfering versions of programs. In *Proceedings of the 2nd International Workshop on Software configuration management*, pages 46–55, 1989.
26. S.Vasudevan, E.A.Emerson, and J.A.Abraham. Efficient model checking of hardware using conditioned slicing. In *Preliminary Proceedings of 4th Int. Workshop on Automated Verification of Critical Systems*, 2004.
27. F. Tip. *Generation of Program Analysis Tools*. PhD thesis, 1995.
28. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
29. V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing: Theory and Applications*, 19(2):149–160, 2003.

-
30. V. M. Vedula, W. J. Townsend, and J. A. Abraham. Program slicing for atpg-based property checking. *International Conference on VLSI Design*, pages 591–596, 2004.
 31. G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 26(6):107–119, 1991.
 32. M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979.
 33. M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
 34. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.