Automatic Generation of Hints For Symbolic Traversal[⋆]

David Ward[1] and Fabio Somenzi[2]

[1] IBM Printing Systems Division
daveward@us.ibm.com
[2] University of Colorado at Boulder
Fabio@Colorado.EDU

Abstract. Recent work in avoiding the state explosion problem in hardware verification during breath-first symbolic traversal (BFST) based on Binary Decision Diagrams (BDDs) applies *hints* to constrain the transition relation of the circuit being verified [14]. Hints are expressed as constraints on the primary inputs and states of a circuit modeled as a finite transition system and can often be found with the help of simple heuristics by someone who understands the circuit well enough to devise simulation stimuli or verification properties for it. However, finding *good* hints requires one to constrain the transition system so that small intermediate BDDs arise during image computations that produce large numbers of reachable states. Thus, the ease of finding *good* hints is limited by the user's ability to predict their usefulness. In this paper we present a method to statically and automatically determine *good* hints. Working on the control flow graph(s) of a behavioral model of the circuit being analyzed, our algorithm extracts sets of related execution paths. Each set has a corresponding enabling predicate which is a candidate hint. Program slicing is employed to identify execution paths. Abstract interpretation and model checking are used to ascertain properties along these paths. Hints generated automatically using our technique result in orders-of-magnitude reductions in time and space requirements during state space exploration compared to BFST and are usually as good as those produced by someone who understands the circuit.

1 Introduction

Reachability analysis plays a central role in formal verification of sequential circuits. One of the state-of-the-art approaches for reachability analysis and formal verification of circuits modeled as finite transition systems exploits symbolic computations based on Binary Decision Diagrams (BDDs). However, the known state explosion problem may cause large intermediate BDDs during the exploration of the state space of a system. The conventional breadth-first search (BFS) strategy, used in most implicit model checking algorithms, is the main culprit. Researchers have approached this problem by devising techniques [5, 11, 1, 12, 14] that simplify the system model employed during BFS.

In [14] a method is proposed to use hints to guide the exploration of the state space. Hints are expressed as constraints on the primary inputs and states of a circuit modeled as a finite transition system. In [14], hints are classified into those that depend on

⋆ This work was supported in part by SRC contract 2003-TJ-920.

the invariants being checked and those that capture knowledge of the design. Hints are applied by constraining the transition relation of the system; the constrained traversal of the state space proceeds much faster than that of the unconstrained system (original transition relation). This method obtained orders-of-magnitude reductions in time and space requirements. Hints can often be found by someone who understands the design well enough to devise simulation stimuli or verification properties for it. However, identifying good hints can be a labor-intensive process requiring many attempts, and in many cases does not avoid the state space explosion problem. One reason it is hard to identify good hints immediately is due to the user's inability to predict, in most cases, the impact the hint will have on the intermediate BDDs during the image computations. Acceptance of this method by designers and verification engineers will certainly benefit from an efficient technique to devise good hints from a system being verified. Our purpose in this paper is to demonstrate how such hints can be automatically determined statically using program analysis techniques.

One effective way to attack the state explosion problem is to construct small transition systems to make automatic checking tractable, yet large enough to capture the information relevant to the property being checked—reachability in our case. Our method exploits these observations and can be summarized as follows: First, we translate the behavioral model into its control flow graph(s) and augment control and data dependency edges. We then partition the control flow graph(s) into subgraphs consisting of sets of valid execution paths. (Execution paths are paths that begins with the start node and end with the exit node whose enabling predicates are satisfiable.) The enabling predicates of these subgraphs are the candidates hints; pruning criteria are applied to discard inferior candidates. Finally, the surviving candidates are sorted to produce the final list of hints.

One feature of our approach is to borrow techniques from program analysis and apply them to hardware verification. The same ideas could be used to generate hints for software verification, but that is outside the scope of this paper. We regard the behavioral model in this paper as a program. The program analysis techniques employed to accomplish our objective are program slicing [15] to extract the subgraph(s) from the original control flow graph(s); abstract interpretation [5] to obtain relevant properties for each subgraph—e.g., checking whether program variables can be influenced by the primary inputs; and model checking of abstract models to identify false data dependencies, and estimate the impact of candidate hints on reachability analysis. The program dependence graph (PDG) [6, 13] is chosen for its efficient representation of control and data dependencies of program operations and its rich set of supporting algorithms.

Analysis of models using our technique is mostly achieved at a high level of abstraction (program dependency graph). Therefore, it remains feasible when the BDD-based analysis (BFS) of the original model is not. We validated our technique using a subset of the Verilog hardware description language (behavioral Verilog); however, we argue that it can be easily extended to any simple imperative language.

This paper is organized as follows: Section 2 reviews the background material. Section 3 discusses the procedure to extract valid execution paths (subgraphs) from the original control flow graph. Section 4 presents our experimental results, and Sect. 5 summarizes, outlines future work, and concludes.

2 Preliminaries

2.1 Guided Search for Least Fixpoints

Hints are used to speed up symbolic reachability analysis of transition systems with set of states Q and inputs W defined by a *transition relation* $T \subseteq Q \times W \times Q$ and *initial state set* $I \subseteq Q$. A triple (q_1, w, q_2) is in T if and only if the transition system can proceed from q_1 to q_2 when the input is w; in this case q_2 is a *successor* of q_1. State q is *reachable* from state q' if there exists a sequence of states $q_1, \ldots, q_n$ such that $q = q_1$, $q' = q_n$, and for $1 < i \leq n$, q_{i+1} is a successor of q_i. The reachability analysis problem consists of finding all states that are reachable from some state in I. For $S \subseteq Q$, let $\mathsf{EY}\, S$ denote all states that are successors of some state in S. Then reachability can be computed as a fixpoint: $\mu Z \,.\, I \cup \mathsf{EY}\, Z$.

This fixpoint computation corresponds to *breadth-first search* (BFS) of the transition system starting from the initial states. In symbolic model checking, transition relations and sets of states are represented by their characteristic functions, which can be manipulated in various forms. In this paper we assume that (reduced, ordered) Binary Decision Diagrams (BDDs [3]) are used for this purpose. Success with symbolic computations depends on the algorithm's ability to keep the BDDs small. Several factors affect the size of BDDs, including the variable orders. Guided search, however, focuses on the facts that BFS may require the representation of sets of states that are intrinsically unsuitable for concise representation by BDDs, and that the BDDs that represent the full transition relation may be unwieldy while restrictions to subsets of transitions may dramatically shrink the BDDs.

Given a set of *hints*, $\tau_1, \tau_2, \ldots, \tau_k$ (where each τ_i is a transition relation obtained by constraining the inputs or state variables of the model) the computation of the reachable states can be decomposed into the computation of a sequence of fixpoints—one for each hint. If hints are chosen properly, the computation of least fixpoints can be substantially sped up [14]. If simple transition systems result for each τ_i, reachability analysis may proceed further compared to computing the model fixpoint directly, and in some cases go to completion by avoiding the memory explosion problem. There are several strategies to use hints. The one of [14] is based on the following result.

Theorem 1 ([14]). *Given a sequence of monotonic functionals $\tau_1, \tau_2, \ldots, \tau_k$ such that $\tau_i \leq \tau_k$ for $0 < i < k$, the sequence $\rho_0, \rho_1, \ldots, \rho_k$ of fixpoints defined by $\rho_0 = 0$ and*

$$\rho_i = \mu X \,.\, \rho_{i-1} \vee \tau_i(X), \quad 0 < i \leq k$$

monotonically converges to $\rho = \mu X . \tau_k(X)$; that is, $\rho_0 \leq \rho_1 \leq \cdots \leq \rho_k = \rho$.

The traditional BFS reachability analysis algorithm can be modified to take advantage of hints: first, each hint, in order, is used to constrain the original transition relation, the algorithm is allowed to run normally until all reachable states are reached. The starting point for each run is either the initial states, for the first hint, or the reached states from the previous run; finally, the original transition relation is restored and runs to completion or is terminated early due to time-space exhaustion. Its starting point is the set of reachable states produced by the last hint.

2.2 Control Flow Graph (CFG)

Many program analysis techniques work on graphs derived from the program text. Among these, the CFG is a directed graph that represents the flow of control of a program (hardware or behavioral model). Each node represents an assignment or branching statement S_i in a program P. Each directed arc represents flow of control from one node to another. A CFG can be extracted in a single pass traversal over P. (See the left part of Fig. 3.) In our implementation we create one CFG for each Verilog *always block*.

Definition 1 (Control Flow Graph (CFG)). *A control flow graph CFG is a directed graph $G = (V, E)$, where: (1) V is a finite set of nodes including two distinguished nodes of type* entry *and* exit*. All other nodes are of one of five types:* assignment, input, decision, no-op*, and* join*. (2) $E \subset V \times V$ is a control flow relation, whose elements are directed edges.*

We assume that the flow relation obeys restrictions. Specifi cally, we assume that the arcs can be partitioned into forward arcs and back arcs so that the forward arcs form a DAG in which all nodes are reachable from the entry node. We also assume that the exit node is reachable from all nodes in the CFG. Furthermore, each back edge goes from a node to another that dominates it. These assumptions imply *reducibility* of the CFG.

Intuitively, the different types of nodes map to the basic types of statements, and in fact we shall call the CFG nodes *statements*. The edges in E represent the transfer of control between statements. A path from the entry node to the exit node represent one clock cycle of a Verilog *always block*.

2.3 Control Dependence Analysis and Program Slicing

Control dependence represents the effect of conditional branches on the behavior of programs. Given two statements S_1 and S_2 in P, statement S_2 is control-dependent on S_1 if S_1 is a conditional branch statement and the control structure (enabling predicate) of P potentially allows S_1 to decide whether S_2 will be executed. Control dependence can be defi ned in terms of the CFG. Let S_1 and S_2 be two nodes of a CFG.

Definition 2 (Postdominance and Control Dependence). *S_1 is postdominated by S_2 in a CFG if all paths from S_1 to the exit node include S_2. S_2 is control dependent on S_1 if and only if:*

1. *There exists a path p from S_1 to S_2 such that S_2 postdominates every node of p;*
2. *S_1 is not postdominated by S_2.*

If S_2 is control dependent on S_1 in a CFG, then S_1 must have two outgoing edges. Following one of the edges always leads to S_2, while there is a path that uses the other edge and bypasses S_2. A control dependence edge can be added to the CFG to show the dependence relation. We refer to the set of control dependence edges as E_{cd}.

Control dependence is used to extract a static program slice. During the determination of hints, our goal is to retrieve pertinent information (valid paths, over-approximate reachable states, $\ldots$) from the original model by analyzing the smallest subset of the original model that preserves the correct result. Program slicing allows us to effi ciently and effectively achieve this goal. Program slicing statically identifies all statements that *might* affect the value of a given variable occurrence [15]. The statements selected constitute a *subprogram* with respect to the variable occurrence. For a statement S_i in a CFG, the static program slice with respect to S_i is the set of statements $S_1, S_2, S_3, \ldots$ in the CFG augmented with control dependence information that can reach S_i via a path of flow or control dependence edges [8]. The program slice for a set of statements S is simply the union of the program slice of each statement S_i.

2.4 Data Dependence Analysis and the Program Dependence Graph (PDG)

Given any two statements S_1 and S_2 (containing variable x) of a CFG, a data dependence relation may hold between them if one statement is an assignment to variable x and the other is a read access (use) of the same variable x. Let OUT(S_i) be the left-hand side variable of S_i and IN(S_i) be the set of right-hand side variables of S_i. For any two statements S_1 and S_2 in a CFG, data dependence is defined as follows:

Definition 3 (Data Dependence). *S_2 is data dependent on S_1 if and only if: (1) $\exists x \in IN(S_2) \,.\, x = OUT(S_1)$; and (2) there exists a path in the CFG from S_1 to S_2 such that no intervening statement is an assignment to x.*

We refer to the set of data dependence edges as E_{dd}. In practice, it is easy to check for the first condition of the definition, but not for the second. Hence, we add one data dependency arc to G_{dd} whenever the first condition is met, and then try to identify as many false dependencies as computationally feasible. False data dependencies do not affect the correctness of our procedure only the quality of the result. Hence, a small number of false dependencies is tolerable. We use data dependence information to cluster paths of the CFG into candidate hints and to determine the relative order of hints.

The PDG represent the control and data dependencies of a program. It can be defined in terms of the CFG; PDG $= (V, E \cup E_{cd} \cup E_{dd})$. The PDG derived from the model text is a lossless transformation of the original program. We can go back and forth from one to the other. We employ the program dependence graph (PDG) [6, 13] as our intermediate representation.

2.5 Abstract Interpretation

Our goal is to statically (and cheaply) determine pertinent information of a program that would otherwise be ascertained during run-time. More specifically, we would like to calculate the run-time behavior of a program without having to run it on all input data, and while guaranteeing termination of the analysis. Abstract interpretation [5] provides the necessary framework to accomplish our goal. In our hint generation procedure we use abstract interpretation to determine if a decision node depends on primary inputs or not. (See Sect. 3.2.) This use effectively corresponds to a reaching definition analysis [7]. Our abstraction will therefore need to capture information about whether input information can reach the definition of a variable. We replace the set of possible values for each variable with the set of values NID and ID ("not input dependent" and "may be input dependent" respectively). Initially, each variable is assigned the value NID, unless it is an input variable. Arguments and values of functions (e.g., integer operators, boolean operators) are now from the set $\{NID, ID\}$. The encoding of the semantics of the functions is simple—any argument with a value of ID causes the function to return ID, otherwise the function returns NID. At the completion of the analysis we can safely conclude that a variable with a value of NID is not influenced by the inputs. Such a variable is labeled an *internal decision variable*. Using this technique on our running example in Fig. 2 results in $state$ being identifi ed as an internal decision variable.

3 Hint Generation Algorithm

The hints generated to help symbolic traversal of a model graph should select subsets of transitions that allow reachability analysis to visit sets of states with many elements and compact representations. Since these representations are usually Binary Decision Diagrams [3], we shall simply say that the objective is to have many states with small BDDs. When a model has several major modes of operation—as when it can execute a set of instructions—enabling one mode at the time is often effective in keeping the BDD sizes under control. Our approach to producing hints automatically is based on identifying the different modes of operation from the Control-Flow Graph (CFG) of the model, and merging and prioritizing them according to their dependencies and their promise to reduce time and memory requirements. The process can be divided in three major phases corresponding to the dashed boxes in Fig. 1.

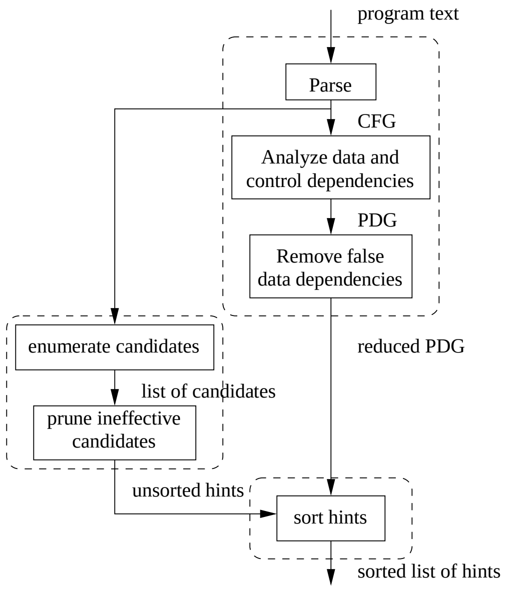

Fig. 1. Automatic hints generator methodology

- From the program text a CFG is extracted and from it a reduced program dependency graph (PDG) is created.
- From the CFG a list of candidate hints is compiled; ineffective hints are pruned.
- Using the reduced PDG, hints are sorted according to their dependencies and usefulness.

It is possible in principle for the list of hints produced by this process to be empty. This may result from exhaustion of computational resources, or because the procedure deems all candidates unworthy. However, we have not yet observed this outcome in practice, except for trivial models that contain no control flow statements. We now describe the three phases in more detail. Figures 2 and 3 show an example that is used throughout the rest of the paper to illustrate the algorithm.

On the left of Fig. 2 we show a Verilog model. The statements of the model are annotated with labels so that they can be traced through the transformations. The model contains a false data dependency (between S3 and S5) as well as cyclic data dependencies and therefore allows us to illustrate various aspects of the hint generation process. The PDG for this model with the control and data dependencies extracted is given in Fig. 3. To save space, the join nodes, where the two branches emanating from a decision node meet, have been merged into their successors. Moreover, to avoid clutter, the data dependency arcs are shown separately on the right, while control flow and control dependencies are jointly presented in one graph, by using thick lines for those arcs that represent both control flow and control dependency. Finally, the right part of Fig. 2 shows the serialized polar graph, which is introduced in Sect. 3.2.

3.1 Removing False Data Dependencies

The program that defines the model to be analyzed is translated into a CFG, which is augmented with data dependency information to produce a PDG. (See Sect. 2.4.) Since the analysis is conservative, some data dependency arcs in the PDG are false. Since more data dependency arcs result in fewer degrees of freedom in the merging and prioritization of modes of operation, it is desirable to remove as many false arcs as possible, without incurring excessive costs. This is accomplished as follows. Each data dependency arc is tested in turn to determine whether the variable definition at the tail of the arc can actually reach the usage at its head by augmenting the program with *token variables*. The program slice corresponding to the token variable of the usage variable is extracted from the PDG. The check whether the definition can reach the usage is thus translated into the check for an invariant on the token variable of the usage variable.

Specifically, suppose that the dependency on the definition $S_1 : x := v_1$ of x by its use $S_2 : y := x$ is investigated. Two token variables, t_1 and t_2, are added to the program by making the following changes.

- Token variables t_1 and t_2 are added to the program defines.
- Assignments $t_1 := 0$ and $t_2 := 0$ are added to the beginning of the program.
- S_1 is changed to $x := v_1; t_1 := 1$.
- Every other assignment $S_i : x := v$ is changed to $S_i : x := v; t_1 := 0$.
- S_2 is changed to $y := x; t_2 := t_1$.

```
module example(clock,reset,chg,load_data,indat);
    input         clock,reset,chg,indat;
    input[4:0]   load_data;

    reg           state,data;
    reg[4:0]     a,b,c;

    initial begin
S1:    state=0;a=0; b=0; c=0; data=indat;
    end

    always @ (posedge clock) begin
P1:    if (state == 0) begin
P2:       if (reset) begin
S2:         a=0; b=0; data=indat;
          end
          else begin
P3:         if (data) begin
S3:           a = load_data;
            end
            else begin
S4:           a = b + 1;
            end
          end
       end
       else begin
P4:      if (!data) begin
P5:        if (chg && !reset) begin
S5:          b = a + 1;
           end
         end
       end
S6:    state = ~state;
    end

    always @ (posedge clock) begin
P6:    if (reset) begin
S7:          c = 0;
       end
       else begin
S8:          c = a;
       end
    end
endmodule
```

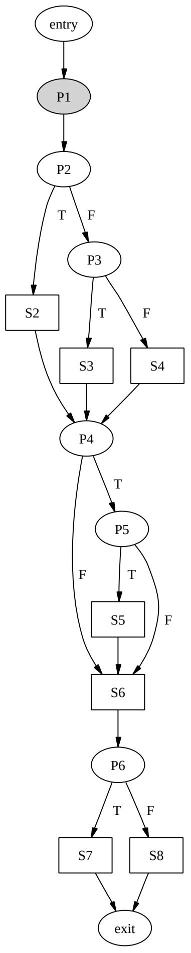

Fig. 2. Example Verilog model (left) and serialized CFG (right)

If in the program slice for t_2 no state is reachable in which $t_2 = 1$, then the data dependency arc from S_1 to S_2 is removed. We employ model checking to check the invariant $t_2 = 0$ in the augmented model. We are only interested in direct dependencies: Consider $S_1 : x := 2$, $S_2 : y := x$, and $S_3 : x := x + 1$. If S_3 is always executed between S_1 and S_2, the dependency arc between S_2 and S_1 is removed. However, the dependencies of S_3 on S_1, and of S_2 on S_3 imply, by transitivity, the one of S_2 on S_1.

Though program slicing may greatly reduce the cost of checking the $t_2 = 0$ invariant, this is not always the case; hence, each model checking run is allotted a short time to complete. If it does not finish, a less accurate, but less expensive test is applied. The augmented program slice is analyzed with abstract interpretation. If abstract interpretation fails to prove the invariant, the arc is (conservatively) retained.

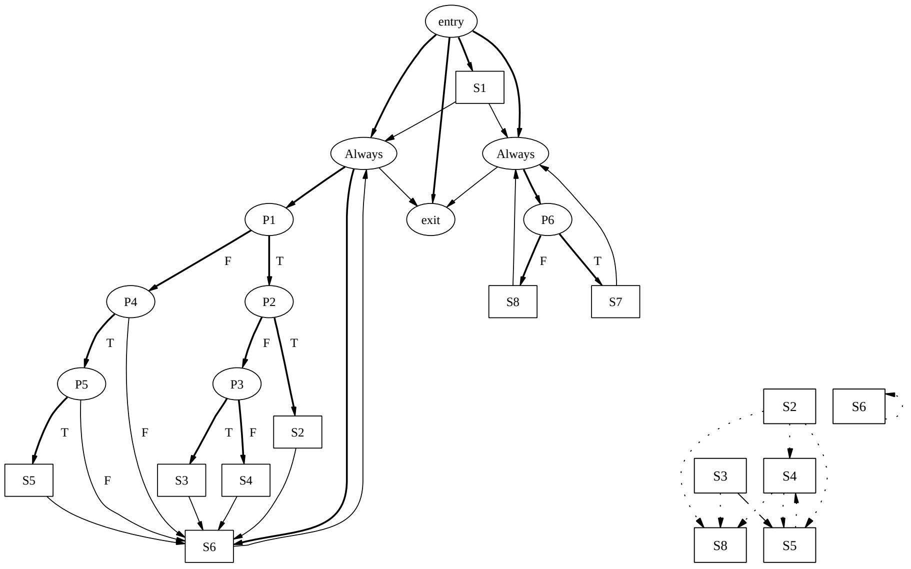

Fig. 3. Control flow and control dependency graph (left) and data dependency graph (right) for the model of Fig. 2

Referring to Fig. 3, it is not hard to see that the data dependency between S3 and S5 is false. In fact, S3 requires `data` to be true for its execution, while S5 is only executed if `data` is false. The edge in the data dependency graph is correspondingly shown as a dashed line. The algorithm based on the token variables identifies the false data dependency. As a result the dashed arc is removed from the PDG.

3.2 Generating Candidate Hints

The modified PDG with (some) false data dependencies removed is one of the two inputs to the final step of our procedure that outputs a list of hints. The other input is a list of subgraphs of the CFG, each corresponding to a mode of operation. The subgraphs are identified by a procedure based on the enumeration of the paths from the entry node to the exit node of the CFG.

Depth-first search (DFS) of a graph classifies the arcs of a directed graph into forward, backward, and cross arcs [4]. The classification depends in general on the order in which nodes are visited. In a *reducible* CFG, however, the result is unique: The back arcs are precisely the return arcs of the looping constructs. Therefore, in the following we refer to the back arcs without reference to a specific DFS.

The graph obtained from the CFG by removing the back arcs is *polar*, that is, all nodes are reachable from the entry node and have a path to the exit node. Our procedure produces a list of candidate hints by partitioning the set of paths connecting the two poles of the graph obtained by removing the back arcs from the CFG. Each subset in the partition is a candidate for producing a hint (the enabling predicate obtained by conjoining the predicates along the path from the entry node to the exit node).

The partitioning algorithm is based on the notion of internal decision. A variable is an *internal decision variable* if it does not depend directly on the external inputs. Abstract interpretation is used to identify internal decision variables. A decision node of a CFG is an *internal decision node* if any variables appearing in the predicate attached to the node are internal decision variables. A hint should not constrain internal decision variables, lest it may contribute very few states to reachability analysis. Consider, for instance, an internal decision variable that is incremented modulo n at each iteration through the CFG. A hint that specifi es the value for this variable will allow only one iteration of reachability analysis before reaching a fi xpoint: As soon as the variable gets incremented, all transitions are disabled (remember that one clock cycle corresponds to executing one path from the entry node to the exit node). Internal decision variables are therefore treated specially in two ways:

1. Paths that diverge at an internal decision node are kept together in the partitioning.
2. Internal decision variables are existentially quantifi ed from the predicates attached to decision nodes before they are used to form a hint.

To account for internal decision nodes in the partitioning of the paths, the polar graph is *serialized*. Let v be an internal decision node with children v_0 and v_1, and u be the corresponding join node with parents u_0 and u_1. Assume that u_0 is a descendant of v_0 and u_1 is a descendant of v_1. Then serialization of v replaces its predicate node with a no-op node; makes v_0 the only child of v; makes v_1 the successor of u_0, and u_1 the only parent of u. Which of the two children of v is regarded as v_0 is immaterial. The effect of serialization is to merge paths that should be kept in the same block of the partition into a single path. We call the result of serialization the *serialized polar graph.* DFS from the start node of this graph enumerates the paths connecting the two poles. The search procedure maintains the conjunction of the predicates of all the decision nodes currently on the stack. This conjunction is kept as a BDD; if it ever becomes false, then the search backtracks to skip the infeasible execution path. Serialization is applied to each always block individually. Finally, the resulting polar graphs are concatenated to form one polar graph for the whole model.

On the right in Fig. 2 one can see the serialized polar graph for our running example. There is one internal decision variable in this case: `state`, which causes P1 to be an internal decision node. The two children of P1, P2 and P4, are serialized. P1 becomes a no-op node (shown by the shaded background). In addition, the two serialized polar graphs from the two always blocks get concatenated.

Two additional techniques are used to reduce the number of blocks in which the paths of the serialized polar graphs are partitioned. When blocks are merged by these techniques, their predicates are disjoint. The two techniques are:

1. Distinct paths of the serialized polar graph may be merged if there are data dependencies among them. The merging takes place after the paths have been extracted from the graph. For each path, merging with another path is considered if there is some data dependency between the two. The merged path is evaluated by reachability analysis. If it does not require more time than the individual paths, and if it does not time out, it replaces the two candidates that were merged.

2. Array variables are treated specially. Addressing into an array can be regarded as conditional access to a set of scalar variables. For instance, $a[i] := 0$ corresponds to $a[0] := 0$ if $i = 0$, $a[1] := 0$ if $i = 1$, and so on. This interpretation of array accesses is required in cases in which hints prescribe the order in which array elements should be enabled. (Such hints are sometimes effective in the presence of symmetry, since they help in reducing the sizes of BDDs that otherwise would encode all permutations of certain set of states.) However, in case of large arrays, expanding their accesses may greatly increase the number of paths. Therefore, the expansion is not initially performed. This leads to ignoring the address variable in the hints. If the resulting subgraphs time out during their evaluation by reachability analysis, the array assignments are gradually refined until a set threshold is reached.

The partitioning of the paths in the serialized polar graph is maintained as a list of predicates, each annotated with a list of CFG nodes, which carry the data dependency information.

The CFG of a complex model may have many candidates that would not contribute enough states to pay for themselves. Therefore, the set of candidates produced by the enumeration procedure is pared down. The selection of the best candidates is heuristic. We currently employ two criteria. The first favors subgraphs with more states reached during abstract interpretation or model checking. Very often, subgraphs in which variables can be controlled via primary inputs produce the best results. To see why, consider an assignment $x := y + z$, where y depends on other variables, whereas z is read from an external input. During reachability analysis, for any value of x and any value of y there is a value of z such that $x = y + z$. (Assuming signed integers.) Hence, the dependency on y, while present, is voided by the dependency on z.

The second selection criterion favors those subgraphs that result in smaller BDDs for the transition relations of the corresponding models. Ideally, we would have a criterion that accounts for both the number of states and the BDD sizes. However, it is difficult to accurately estimate both without going all the way to guided search. The pruning eliminates all candidates that prove inferior according to at least one heuristic.

The serialized polar graph of Fig. 2 has a total of 18 paths from entry to exit. Of these, only six are viable (their predicates are not false). For each of these paths we list the predicate and the list of assignment nodes appearing in them.

1. $\texttt{reset} \wedge \neg\texttt{data} \wedge \texttt{chg}$ (S2, S6–7)
2. $\texttt{reset} \wedge \neg\texttt{data} \wedge \neg\texttt{chg}$ (S2, S6–7)
3. $\texttt{reset} \wedge \texttt{data}$ (S2, S6–7)
4. $\neg\texttt{reset} \wedge \texttt{data}$ (S3, S6, S8)
5. $\neg\texttt{reset} \wedge \neg\texttt{data} \wedge \texttt{chg}$ (S4–6, S8)
6. $\neg\texttt{reset} \wedge \neg\texttt{data} \wedge \neg\texttt{chg}$ (S4, S6, S8).

The first three paths do not generate new states; therefore, they are pruned. The last two candidates have mutual data dependencies through S4 and S5; hence, they are merged into $\neg\texttt{reset} \wedge \neg\texttt{data}$, (S4–6, S8). The result of the merger and the fourth path are forwarded to the final phase of the algorithm.

As another example consider the model CRC used in the experiments of Sect. 4. It is challenging for reachability analysis because of its complex transition relation. This model has four main modes of operation. One in which it resets its register; another in which it holds the current value; a third mode in which it loads data from the outside;

and a fourth mode in which it computes the cyclic redundancy code. Four candidates are produced by the analysis of the CFG, one for each of these modes. The first two candidates are discarded because they yield no new states. The fourth candidate is discarded both because the corresponding transition relation is too large and because its reachability analysis times out. In fact, it is this mode of operation that makes BFS reachability analysis hard. The surviving candidate enables the load operation, which is exactly what a knowledgeable designer writing a hint would do. The load mode allows every state to be reached. Hence, guided search terminates using only the constrained transition system without having to restore the full transition relation.

3.3 Sorting the Hints

The final step of the procedure sorts the list of candidates using the information on data dependencies provided by the PDG. The order in which hints are applied may greatly influence their effectiveness. This is particularly the case when there are data dependencies between the variables occurring in different subgraphs. Suppose subgraphs P_1 and P_2 are such that variable x is assigned in P_1 by an input statement, while in P_2 it is assigned a constant value v. Suppose also that x is used in an assignment $y := x$ in P_2, and that that is the only assignment to y. Then, if the hint extracted from P_2 is applied before the hint derived from P_1, all the states reached after the two hints have been applied have $y = v$, whereas, if the order of application is reversed, there will be reachable states for each possible value of y.

In general, there will be cyclic dependencies among subgraphs. Hence, we proceed as follows. We form a Subgraph Dependency Graph (SDG) with one node for each subgraph. Each node of the SDG is the set of nodes in the PDG that make up the corresponding subgraph. An arc connects nodes u and v of the SDG iff there exists a data dependency arc (a, b) in the PDG such that $a \in u$, $b \in v$, and $a \neq b$. The ordering of the candidate subgraphs is obtained from the SDG. In particular the strongly connected components (SCCs) [4] of the SDG define a preorder on the subgraphs: We say that $u \preceq v$ if there is a path from u to v in the SDG. The final order $\leq$ is always a refinement of this preorder in the following sense: if $u \preceq v$ and $v \not\preceq u$, then $u \leq v$. However, an arbitrary total order that refines the preorder may not work well, if there are just a few large SCCs.

We decompose the problem of deriving a total order from the preorder defined by the SDG into two subproblems. The first is the one of linearizing the partial order defined by the SCC quotient graph of the SDG. The second is to find total orders for the nodes of each SCC. The total order of the subgraphs results from combining the solutions of these two subproblems in the obvious way.

Any topological sort of the nodes of the SCC quotient graph would satisfy the definition of order refinement. However, different orders result in BDDs of different sizes. It is normally advantageous to keep subgraphs adjacent in the order if they operate on common variables. Therefore, to sort the SCCs of the SDG we perform a depth-first search from the source nodes of the SCC quotient graph.

Sorting the nodes of an individual SCC is based on identifying a starting node, and then enumerating the elementary circuits of the SCC [10]. As we enumerate elementary circuits from the designated start node, we add nodes to the total order as they appear in some elementary circuit. We rely on the fact that the enumeration algorithm outputs short circuits first. We equate short circuits to tight interaction, and therefore put those nodes that have tighter interaction with the start node earlier in the order. The start node is the entry point to the SCC in the DFS that computed the quotient graph.

The SDG for the example of Fig .2 consists of two disconnected nodes. In this case the order of the two hints is chosen arbitrarily.

4 Experimental Results

We extended VIS 2.0 [2] as outlined in Sect. 3 to automatically produce hints. Experiments were conducted on a 1.8GHz Pentium IV machine with 512MB of RAM running Linux. We report the results of our experiments in Tables 1 and 2. We used ten circuits in our experiments. CRC computes a 32-bit cyclic redundancy code. BPB is a branch prediction buffer. S1269 is an 8-bit ALU. Rotator and Spinner are barrel shifters sandwiched between registers. B04 is a Verilog translation of the original b04 circuit from the ITC99 benchmark set [9]. It computes the minimum and maximum of a set of numbers. Vsa is a simple non-pipelined microprocessor that executes 12-bit instructions—ALU operations, loads, stores, conditional branches—in five stages: fetch, decode, execute, memory access, and write-back. Am2901 is a bit-sliced ALU and contains sixteen 4-bit registers organized into a register file, along with a 4-bit shift register. Am2910 is a microprogram sequencer.

Table 1. Experimental results for reachability analysis

Circuits	FFs	Reachable States	Times in seconds		
			BFS	Manual Hints	Auto Hints
CRC	32	4.295e+09	Mem. out	0.48	0.48
BPB	36	6.872e+10	124.22	0.55	0.22
s1269	37	1.31e+09	22.91	0.65	0.65
DAIO	56	2.95e+11	21.02	7.94	22.05
Rotator	64	1.845e+19	Mem. out	0.15	0.15
Spinner	65	3.689e+19	Mem. out	0.15	0.15
B04	66	5.650e+15	5883.28	1892.76	1892.76
Vsa	66	1.625e+14	4859.33	153.38	224.22
am2901	68	2.951e+20	Mem. out	1.79	1.87
am2910	99	1.610e+26	Mem. out	60.08	Mem. out

Table 1 compares reachability analysis with automatically generated hints against BFS runs and manual hints supplied by an expert user. Columns 1, 2, and 3 give the name of the circuit, number of flip-flops (state variables) and number of reachable states of the circuit. Columns 4, 5, and 6 compare run times for reachability analysis for BFS, manual hints, and automatic hints, respectively. The circuits in this table are mid-sized, but three of these circuits—CRC, Rotator, and Spinner—run out of memory for

Table 2. Auto hint generation

Circuits	Total number of hint candidates	Final number of hints	time to produce hint(s)	Statistics	
				timeouts	Reachability completed
CRC	5	1	25	1	1
BPB	165	4	806	2	0
s1269	32	1	381	0	0
DAIO	576	1	422	0	0
Rotator	32	1	4	0	32
Spinner	32	1	4	0	32
B04	6	1	96	0	0
Vsa	23	7	61	3	0
am2901	82	3	123	0	0
am2910	48	1	76	0	0

BFS. The automatic and manual hints were able to provide dramatic improvements to the traversal of these circuits, enabling completion times of a few seconds. Circuit B04 completes in about one third of the time taken by BFS when using hints. The automatically generated hint is in this case exactly the same as the manual hint. Three remaining circuits in Table 1, BPB, s1269, Vsa, and am2901, demonstrate 1–2 orders of magnitude improvements over BFS. Finally, circuit DAIO and am2910 does not show any improvement over BFS. It is remarkable that the quality of the automatically generated hints is, with one exception, quite comparable to that of the manual hints. (In a few cases, the hints are indeed the same.) The times to generate the hints are non-negligible for some examples, but quite acceptable especially considering that it is incurred only once. The hints, on the other hand, may be used many times.

Table 2 shows information collected during the experiments. Column 2 shows the total number of hints generated during the analysis of the example's CFG (the total number of acyclic paths in the serialized polar graph). The sizable number of hint candidates produced for BPB is attributed to the need to expand the array elements to compensate for the lack of robustness of our parser. However, this was not a limiting factor in that we were able to generate competitive hints after pruning and sorting. By contrast, the number of candidates for Vsa is small because expansion was not necessary.

The total number of hints after pruning and sorting is shown in Column 3. We were able to forego the pruning and sorting steps for CRC and Rotate after a completed reachability analysis of the candidate hint was realized (within a 15 CPU seconds). This reachability analysis showed that all states were reached with a hint, thereby eliminating the need to continue the generation process.

5 Conclusion

In this paper we have shown that state traversal guided by automatically generated hints can substantially speed up reachability analysis relative to BFS and produces hints comparable to manual hints. We have presented a procedure that analyzes the control flow graph derived from the program text (e.g., behavioral Verilog description) and partitions the execution paths into subgraphs corresponding to hint candidates. The candidates are ranked and ordered to produce a fi nal list. Though our implementation is still a prototype, it has produced very encouraging results because the quality of the hints it generates rivals that of hints written by expert users. The times required to automatically generate hints sometimes exceed the guided search time, but remain acceptable, and should be reduced as our implementation matures.

Considerable work remains to be done in the area of automatic generation of hints. We need to strengthen our parser so that we can confi rm the initial encouraging results on a larger selection of examples. We also need to address model checking of more general properties than just invariants and study the generation of property-specifi c hints.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] R. K. Brayton et al. VIS. In *Formal Methods in Computer Aided Design*, pages 248–256. Springer-Verlag, Berlin, Nov. 1996. LNCS 1166.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 238–250, 1977.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[7] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Input-dependency analysis for hard real-time software. In *International Workshop on Object-Oriented Real-Time Dependable System*, Oct. 2004.

[8] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Fourteenth International Conference on Software Engineering*, 1992.

[9] ITC'99 benchmark home page. http://www.cerc.utexas.edu/itc99-benchmarks/bench.html.

[10] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4:77–84, 1975.

[11] R. P. Kurshan. *Computer-Aided Verifi cation of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[12] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verifi cation (CAV'98)*, pages 110–121. Springer-Verlag, Berlin, 1998. LNCS 1427.

[13] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Symposium on Practical Software Development Environments*, pages 177–184, New York, NY, 1984.

[14] K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *Correct Hardware Design and Verifi cation Methods (CHARME'99)*, pages 250–264, Berlin, Sept. 1999. Springer-Verlag. LNCS 1703.

[15] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.