Hints to Accelerate Symbolic Traversal*

Kavita Ravi[1] and Fabio Somenzi[2]

[1] Cadence Design Systems
kravi@cadence.com
[2] Department of Electrical and Computer Engineering
University of Colorado at Boulder
Fabio@Colorado.EDU

Abstract. Symbolic model checking is an increasingly popular debugging tool based on Binary Decision Diagrams (BDDs). The size of the diagrams, however, often prevents its application to large designs. The lack of flexibility of the conventional breadth-first approach to state search is often responsible for the excessive growth of the BDDs. In this paper we show that the use of *hints* to guide the exploration of the state space may result in orders-of-magnitude reductions in time and space requirements. We apply hints to *invariant checking*. The hints address the problems posed by difficult image computations, and are effective in both proving and refuting invariants. We show that good hints 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. We present an algorithm for guided traversal and discuss its efficient implementation.

1 Introduction

Great strides have been made in the application of formal methods to the verification of hardware The most successful technique so far has been model checking [19]. Model checking exhaustively explores the state space of a system to ascertain whether it satisfies a property expressed in some temporal logic.

Given the exponential growth of the number of states with the number of state variables, several techniques have been devised to turn model checking into a practical approach to verification. The most fundamental technique is abstraction: Verification is attempted on a simplified model of the system, which is meant to preserve the features related to the property of interest [11,16]. Compositional verification [1,21,15] in particular applies abstraction to hierarchically defined systems so that the environment of each subsystem is summarized by the properties that it guarantees.

Abstraction and compositional verification can be used to reduce the cost of model checking experiments. The modeling effort required of the user, however, grows with the degree of abstraction. Other techniques are therefore needed to increase the intrinsic efficiency of the state exploration. Explicit model checkers

* This work was supported in part by SRC contract 98-DJ-620.

L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, pp. 250–266, 1999.

use clever hashing schemes and external storage. Implicit model checkers, on the other hand, rely on Binary Decision Diagrams (BDDs) [5] to represent very large sets of states and transitions. BDD-based model checkers can sometimes analyze models with over 5000 state variables without resorting to abstraction. In other cases, however, even models with 40 state variables prove intractable. This large variability is ultimately due to the fact that only functions with high degrees of regularity possess compact BDDs. In many instances, however, the occurrence of large BDDs in model checking experiments is an artifact of the specific state search strategy and can be avoided. The analysis of the conditions that cause the occurrence of large BDDs during the exploration of the state space, and the description of a strategy to prevent those conditions are the topics of this paper.

Our discussion will focus on the form of model checking known as *invariant checking*, which consists of proving that a given predicate holds in all reachable states of a system. This form of verification is the most commonly applied in practice. We aid invariant checking by guiding reachability analysis with *hints*. Hints specifically address the computational bottlenecks in reachability analysis, attempting to avoid the memory explosion problem (due to large BDDs) and accelerate reachability analysis. Hints may depend on the property to be verified, but they are successful at speeding up the proof as well as the refutation of invariants.

Hints are applied by constraining the transition relation of the system to be verified. They specify possible values for (subsets of) the primary inputs and state variables. The constrained traversal of the state space proceeds much faster than the standard breadth-first search (BFS), because the traversal with hints is designed to produce smaller BDDs. Once all states reachable following the hints have been visited, the system is unconstrained and reachability analysis proceeds on the original system. Given enough resources, the algorithm will therefore search all reachable states, unless a state that violates the invariant is found. In model checking, constraints sometimes arise from assumptions on the environment. Consequently, these constraints need to be validated on the environment. In our algorithm, since hints are eventually lifted, they leave no proof obligations.

Hints are reminiscent of simulation stimuli, but the distinguishing feature of our application is that the state space is exhaustively explored. Simulation is a partial exploration of the state space and can only disprove invariants—both concrete and symbolic simulation suffer this limitation. Using hints is similar to applying stimuli to the system, but temporarily. Moreover, hints are designed to make symbolic computations with BDDs easier, whereas simulation stimuli are only chosen to exercise the circuit with respect to a property.

It has been observed in the reachability analysis of many systems that the BDDs at completion are smaller than the intermediate ones. The BFS curve of Fig. 2 illustrates this phenomenon for the traversal of the circuit Vsa (described in Section 6). The constrained traversal sidesteps the intermediate size explosion. When the hints are removed, the unconstrained traversal is expected to be

past the intermediate size explosion. Our algorithm also takes advantage of the information gathered in the constrained traversal.

Some invariants can be checked directly by induction (if the predicate holds in all initial states and in all successors of the states where it holds). In general, however, proving invariants entails performing reachability analysis. Our algorithm is designed to benefit the most general case. It is compatible with abstraction techniques like *localization reduction* [16], which is particularly useful when the invariants describe local properties of the system.

The algorithmic core of BDD-based invariant checking is the computation of least fixpoints. Our guided search approach is applicable in general to least fixpoint computations and hence to a wider class of verification procedures, like those of [14,3]. The rest of this paper is organized as follows. In Section 2 we discuss background material and define notation. In Section 3 we discuss the main computational problems in image computation—the step responsible for most resource consumption in invariant checking. In Section 4 we introduce guided symbolic traversal by discussing case studies and presenting our algorithm. Section 5 is devoted to a review of the relevant prior art in relation with our new approach. Experimental results are presented in Section 6 and conclusions are drawn in Section 7.

2 Preliminaries

Binary Decision Diagrams (BDDs): (BDDs) represent boolean functions. A BDD is obtained from a binary decision tree by merging isomorphic subgraphs and eliminating redundant nodes. For a given variable order, this reduction process leads to a canonical representation. Therefore, equivalence tests are efficient, and, thanks to the extensive use of memoization, the algorithms that operate on BDDs are fast. Large sets can be manipulated via their characteristic functions, which in turn can be represented by BDDs. BDDs are used to represent sets of states and transitions in symbolic verification.

Though almost all functions have optimal BDDs of size exponential in the number of variables, the functions encountered in several applications tend to have well-behaved BDDs. This is not necessarily the case in model checking: Sometimes the sets of states or transitions that a model checker manipulates are irregular and have large BDDs.

The sizes of the BDDs for many functions depend critically on the variable orders. Good variable orders are hard to predict *a priori*. Heuristic algorithms like sifting [27] have been devised to dynamically change the order and reduce BDD sizes during the computation. However, the high cost of the reordering and short-sighted optimization (which may be bad for a later stage) impedes its effectiveness.

Finite State Machine: A sequential circuit is modeled as a finite state machine $(S, \Sigma, O, T, \lambda, I)$, where S is the set of states, Σ is the input alphabet, O is the output alphabet, $T = S \times \Sigma \times S$ is the state transition relation, $Z = S \times \Sigma \times O$ is the output relation, $I \subseteq S$ is the set of initial states.

S is encoded with a set of variables x. T is encoded with three sets of variable—the set of present state variables x (same as S), a set of next state variables y, and a set of primary input variables w. $T(x, w, y) = 1$ if and only if a state encoded by y is reached in one step from a state encoded by x under input w.

For deterministic circuits, the transition relation is customarily constructed as a product of the *bit relations* of each state variable, $\prod_{i=1}^{n}(y_i \equiv \delta_i(x, w))$, where y_i is the next state variable corresponding to the state variable x_i, δ_i is the next state function of the i-th state variable. When T is represented by a single BDD, it is said to be *monolithic*. The monolithic transition relation may have a large BDD, even for mid-size circuits. In practice, a more efficient *partitioned* representation [7,28] is used as an implicit conjunction of blocks of bit relations.

$$T(w, x, y) = \prod_{i=0}^{m} T_i(w, x, y),$$

Image Computation: With BDDs it is possible to compute the successors (predecessors) of a set of states symbolically (without enumerating them). This computation is referred to as an *image (preimage)* computation, and is defined as follows:

$$\text{Image}(T, C) = [\exists_{x,w} T(x, w, y) \wedge C(x)]_{y=x},$$

where $\text{Image}(T, C)$ is the image of a set of states $C(x)$ (expressed in terms of the x variables) under the one-step transition relation $T(x, w, y)$. Further details are discussed in Section 3.

BFS: The traversal of the state space of a circuit is accomplished by a series of image computations, starting from the initial states and continuing until no new states are acquired. This is a BFS of the state graph: All the states at a given minimum distance from the initial states are reached during the same image computation. Some states are considered multiple times: A commonly applied optimization is to compute only the image of the states that were first reached during the previous iteration. This optimization still guarantees a BFS of the graph. Such a set of states is called the *frontier*.

3 Image Computation

And-Exists: For a monolithic relation, image computation is carried out with a single BDD operation called *And-Exists*. *And-Exists*(f, g, v) produces $\exists_v(f \wedge g)$. The complexity of this operation has no known polynomial bound [19]. In the worst case, the known algorithms are exponential in the number of variables in the first two operands.

We adopt the approach of Ranjan *et al.*, [23] for image computation. A linear order is determined for the blocks of the transition relation, $T_i(x, y, w)$. A series of *And-Exists* operations are performed. At each step, the operands are the current partial product, next block of the transition relation $T_j(w, x, y)$ and the quantifiable variables. The initial partial product is set to $C(x)$.

Quantification Schedule: *And-Exists* has the property that $\exists_v(f \wedge g) = g \wedge \exists_v f$, if g is independent of v. This property can be used to quantify x and w variables earlier than the last *And-Exists* in image computation to reduce the worst-case sizes by reducing the peak number of variables. This has been observed to produce significant practical gains. The application of this property is called *early quantification*. Several researchers have proposed heuristics to optimally order the operands of the *And-Exists*, $T_i(w, x, y)$, $C(x)$ and scheduling the quantification of variables [13,23]. In Ranjan's approach, the life-span of variables during image computation is minimized—variables are quantified out as early as possible and introduced into the partial product as late as possible. However, the proposed techniques are far from being optimal.

Issues in Image Computation: Image computation comprises the largest fraction of traversal time, typically 80-90%. Most of the BDD size explosion problems in symbolic traversal are observed during image computation. The main causes for size explosion are complex functions for the next state relations, insufficient early quantification, and conflicting variable ordering requirements.

Three sets of variables—x, y, and w are involved in image computation. Several ordering heuristics have been experimented with—the most common one being the interleaving of x and y variables (as done in VIS [4]). This controls the intermediate BDD sizes in the substitution of the y variables with the x variables at the end of image computation but may be sub-optimal for the partial products.

In the monolithic transition relation, the interleaving order may cause a size blowup since all the x and y variables interact in the same BDD. In a partitioned transition relation, normally the size stays under control since every block $T_i(x, w, y)$ has few variables and the blocks are kept separate. However, during image computation, depending on the set whose image is computed and the quantification schedule, all the x and y variables may interact. The situation may arise due to two reasons—the heuristics used may not find the optimal quantification schedule or, a good quantification schedule may not exist when many partitions depend on most of the variables.

Example: An ALU is an example of function that causes large intermediate sizes. ALUs usually have operations such as ADD, SHIFT that make the output bits depend on many of the input bits in complex functions, resulting in a bad quantification schedule and large BDDs. In Section 4 we show how the use of hints can address this problem.

4 Guided Symbolic Traversal Using Hints

We address the main problem of traversal—large intermediate BDD sizes during image computation. We propose to simplify the transition relation in order to reduce these sizes. The simplification addresses the issues discussed in Section 3—reducing the peak number of variables involved in image computation and (or) reducing the complexity of the transition relation by removing some transitions.

The simplification is achieved by the use of *hints*. As explained in Section 1, applying hints to traversal is equivalent to constraining the environment or the

operating mode of the FSM. Breath-first search (BFS) explores all possible states at each iteration of traversal i.e., all possible transitions from the current set of states are considered. With *hints*, only a subset of these transitions considered.

Hints are expressed as constraints on the primary inputs and the states of the FSM. The transition relation simplified with hints (conjoined with the constraints) may have fewer variables in its support or fewer nodes or both. This leaves room for a better clustering of the bit relations based on support and size considerations. A improved quantification schedule may result from this modified clustering. Every iteration of traversal performed with the simplified transition relation benefits in terms of smaller intermediate BDD sizes and faster image computations.

In invariant checking, all reachable states are checked for violation of the property. The order in which the states are checked for the violation is irrelevant for the correctness of invariant checking. Traditional methods employ BFS, starting at the initial states, as their search strategy. Applying hints results in a different search order. Removing the hints at the end ensures that the invariant is checked on all states. Counterexamples can be produced for invariant checking with hints in the same manner as invariant checking without hints. An error trace, though correct, may not be the shortest possible error trace since hints do not perform a BFS.

We use the following grammar to express hints.

$$hint ::= atomic\_hint \;\mid\; hint : hint \;\mid\; repeat(hint, n)$$

where an *atomic_hint* is a predicate over the primary inputs and states of the FSM. The ":" operator stands for concatenation. The *repeat* operator allows the application of hints for n (finite) iterations or infinitely many times ($n = \infty$). repeat(*hint*, ∞) results in exploration of all states reachable from the given set in the hint-constrained FSM.

We illustrate the effectiveness of hints by studying their application to some circuits.

Am2901: This model is a bit-sliced ALU and contains sixteen 4-bit registers organized into a register file, along with a 4-bit shift register. The ALU operations include LOAD, AND, OR, XOR, and ADD. The registers are fed by the output of the ALU. The operands of the ALU are either primary inputs or registers.

The hint given to this circuit restricts the instruction set to LOAD. Traversal without hints runs out of memory whereas with hints completes in 3.93 seconds. The LOAD instruction is sufficient to generate the entire reachable set. The hint results in every register being fed by primary inputs and removes the dependence on other bit registers. The size of each bit relation decreases, the number of blocks in the transition relation reduces due to better clustering and the quantification schedule improves, allowing more variables to be quantified earlier.

Another hint that enhances traversal is the sequencing of destination registers. Since there is only one write port, only one register gets written into in each cycle. In BFS, at the k-th iteration, sets of k registers are filled with data values of the ALU output and $17 - k$ registers retain their initial value. The

circuit takes 17 iterations to complete reachability analysis. In sequencing the destination address, a specific register is chosen as the destination for the k-th iteration. At the end of 17 iterations, the effect is the same as that of not sequencing addresses. The time for this traversal is 53 seconds.

The two hints mentioned above indicate that the transition relation can be simplified in different ways. In reducing the instruction set to LOADs only, the peak number of variables in the computation decreases. Sequencing the addresses resulted in only one set of registers being fed by the ALU output while the rest retain their original values.

Pipelined ALU (PALU): This model is a simple ALU with a three-stage pipeline. The ALU allows boolean, arithmetic and shift operations. It also reads from and writes to a register bank. The register bank is 4-bits wide and has 4 registers. There are three pipeline stages—fetch, execute and write-back stage. A register bypass is allowed if the destination address in the write-back stage matches the source address in the fetch stage. The pipeline is stalled if one of the inputs is asserted. On stalling, the data and addresses are not allowed to move through the pipeline. This creates dependencies between the latches in different stages of the pipeline.

The traversal of this circuit takes 1560 seconds. With the hint that disallows pipeline stalls, traversal completes in 796 seconds. In this case, the hint chooses a common mode of operation of the circuit. Disabling stalls changes the quantification schedule since many registers in the original FSM depend on the stall input. Additionally, the constrained FSM explores a denser [25] set of states due to the absence of simultaneous dependencies introduced by stall and addresses. There is no instruction such as LOAD to generate enough data values to explore a large portion of the reachable states.

Summary: The above examples demonstrate the usefulness of hints in traversal. Hints may make traversal possible where it was not or may speed up traversal considerably. In the following section, we describe how these hints are applied and a modified algorithm for hint-enhanced traversal.

4.1 Algorithm

Our algorithm for traversal is illustrated in Fig. 1. *hints* is a parse tree of hints expressed in the grammar described at the beginning of Section 4. Every node of this parse tree has a type, that may be "*atomic_hint*", ":" or "*repeat*". An "*atomic_hint*" node has no children, a ":" node has two children and a "*repeat*" node has one child and an associated field, n, for the number of repetitions.

The algorithm is called with P set to I and T set to the transition relation of the FSM. The hints are automatically suffixed with *:repeat* $(1, \infty)$. This serves to restore the original transition after applying all the hints. (The first argument here implies no constrain on the transition relation.) The algorithm in Fig. 1 recurs on the parse tree of *hints*. The leaf of the recursion is the case of the atomic hint $(H(w, x))$. In this case, the algorithm computes a constrained transition relation $T_H(x, w, y)$ with respect to the hint and computes the image of the current set P. In the case of a node of type ":" (concatenation), the procedure

```
Guided_Traversal (hints,T, P)
    switch type(hints)
    case "atomic_hint"
        T_H(x,w,y) = T(x,w,y) ∧ H(w,x)
        P = P ∨ Image(T_H, P)
    case ":"
        P = Guided_Traversal (left_child(hints), T, P)
        P = Guided_Traversal (right_child(hints), T, P)
    case "repeat"
        for (counter =0; counter < read_n(hints); counter++)
            P_prev = P
            P = Guided_Traversal (left_child(hints), T, P)
            if (P_prev == P) break
    return P
```

Fig. 1. Guided traversal algorithm.

recurs on each subgraph (concatenated hint) and updates P. With a "*repeat*" node, the procedure recurs as many times as required by n or until P converges, whichever comes first. The *atomic_hint* produces new successors while ":" and *repeat* order the application of hints.

The case of *repeat* with $n > 1$ involves repeatedly computing the image of P using the same transition relation $T_H(x, w, y)$ (generated using this *atomic_hint*). The implementation of *repeat*(*atomic_hint*, n) can be made more efficient by using the frontier states (as mentioned in Section 2). Our implementation makes use of this optimization.

The correctness of the algorithm in Fig. 1 is established by the following theorem.

Theorem 1. *1. Algorithm* `Guided_Traversal` *terminates.*

2. The set of states computed by Guided_Traversal *is contained in the reachable states and no less than the reachable states are computed.*

Proof Sketch: (1) is proved by induction on the depth of *hints*. The proof of (2) relies on the fact that application of hints to the transition is monotonic and the operation $P \vee Image(T_H, P)$ is monotonic. Finally, since the original transition relation is restored, all reachable states are computed.

Optimization. Touati [28] proved that the BDD *Constrain* operator has the property

$$\exists_{x,w} T(x, y, w) \wedge C(x) = \exists_{x,w} T(x, y, w) \downarrow C(x).$$

In applying the hint, the same result can be used as

$$P(x) \vee (\exists_{x,w}(T(x, w, y) \wedge P(x)) \downarrow H(w, x))|_{y=x}.$$

If the hint depends on primary inputs only, the computation can be further simplified to

$$P(x) \vee (\exists_{x,w}(T(x, w, y) \downarrow H(w)) \wedge P(x))|_{y=x}.$$

Images of Large Sets. Every iteration in the algorithm computes the image of P. If the size of the BDD of P increases, computing the image of P is likely to

cause a memory blowup. To address this problem, we decompose P disjunctively and compute the image of each individual part [9,22,26]. Specifically, we adopt the *Dead-End* computation approach of [26]—a two-way recursive disjunctive decomposition of P is done and an image is computed only when the size of a disjunct are below a certain threshold. A tradeoff may occur in terms of CPU time when the decomposition does not produce enough reduction in size and many disjuncts are generated before the allowed size for image computation is attained. However, the image computation of a large BDD is avoided and in many cases this allows traversal to complete.

4.2 Identification of Good Hints

Hints to aid traversal fall in two main categories: Those that depend on the invariants being checked and those that capture knowledge of the design at hand, independent of the property. Invariant checking may profit from both property-dependent and machine-dependent hints. False invariants may be negatively impacted by hints that further the distance of the violating states from the initial states. Property-dependent hints will tend to avoid this situation. Counterexamples, using property-dependent hints, will tend to be shorter than those with general purpose hints.

In Section 4, we presented two circuits and appropriate hints that accelerated the traversal of these circuits. The hints fall in the category of machine-dependent hints. The acceleration was achieved due to reduced BDD sizes in image computation and exploration of a dense set of states for a majority of iterations. Figure 2 shows the comparison of BDD sizes for the reached set between BFS and traversal using hints of the circuit Vsa (described in Section 6). In this section, we try to summarize the kinds of hints that are easy to identify *and* achieve the desired effect. We believe that these hints are easily extracted from a reasonable knowledge of the circuit. (Sometimes it may even be possible to guess them.) We expect that traversal of circuits containing features (ALUs, register files) similar to the ones described in this paper will profit from the same kinds of hints.

The hints we have utilized can be classified into four categories.

1. Pick the most common operation mode of the circuit. One example is disabling stalls. If there is a counter in the circuit, the counter should be enabled first so that counting iterations have small BDDs.
2. Simplify complex functions such as ALUs: Pick an opcode that produces small intermediate sizes and many states. Loads, in this case, tend to be ideal as they reduce variables in the support of the ALU outputs and produce many next states.
3. Disabling latches: Disable output latches or latches with a small transitive fanin.
4. In the presence of addresses to access banks of registers, pick a sequence of addresses. This prevents the simultaneous interaction of variables belonging to independent registers. These registers tend to be quasi-symmetric in

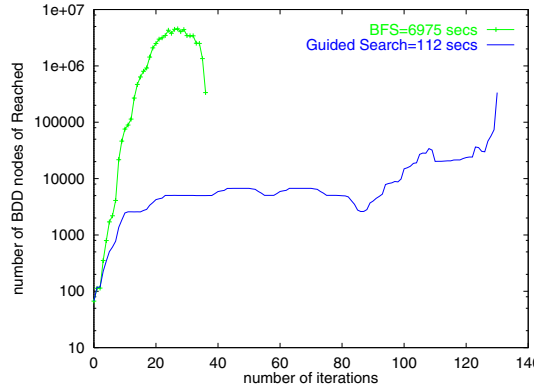

Fig. 2. Comparison of BDD sizes on a log scale of the reached states in the BFS traversal and hint-enhanced traversal of Vsa.

the circuit. Symmetry has traditionally been viewed problematic in model-checking because of its effects on variable order. Several researchers have studied the effects of symmetry [10,18]. The proposed hint breaks the symmetry in the circuit.

A combination of the above hints can also enhance traversal time. In the Vsa example (described in Section 6), a combination of opcode hints and sequencing of addresses reduced the BDD sizes and traversal time dramatically. For property dependent hints, it is useful to pick an input that is likely to cause the violation. While simulation requires the specification of all input stimulus, the property dependent hints require only a partial specification.

So far, we have discussed the extraction and application of hints by the user. It may also be possible to extract these hints automatically and apply them to traversal. The above mentioned guidelines may prove useful in the automatic detection of hints. In particular, the last two categories may be easy to extract by static analysis of the circuit. A property dependent hint for the same circuit decreased the

Effect on dead-end computations. Dead-end computations are expensive in comparison with image computations of frontier sets with small BDDs. Since repeated dead-end computations may slow down the overall traversal, a good choice of hints should try to minimize the number of dead-end computations. In this regard, retaining a frontier is desirable. The hint *repeat*(*atomic_hint*, n) is implemented to use frontier sets. Hints should also be chosen to produce dense sets of states, to ease dead-end computations. A particular hint may also be abandoned when the density of the reached state set begins to deteriorate.

5 Comparison to Prior Work

In this section we review the techniques that share significant algorithmic features with our approach.

High-density reachability analysis [25] is based on the observation that BDDs are most effective at representing sets of states when their *density* is high. The density of a BDD is defined as the ratio of the number of minterms of the function to the number of nodes of the BDD. By departing from pure BFS, one can try to direct the exploration of the state space so that the state sets have dense BDDs. Dense sets are obtained by applying subsetting algorithms [24] to the frontier states when the BDDs grow large. The increased density of the frontier sets often improves the density of the other sets as well. When this is not the case, corrective measures can be attempted [26].

Both high density reachability and guided traversal analysis depart from BFS to avoid its inefficiencies. The latter adopts the dead-end computation algorithm of the former. However, the search strategies differ in several respects. High density reachability is a fully automated approach, which can be applied regardless of the knowledge of the system, but occasionally exhibits erratic behavior. Also, it addresses directly the density of sets of *states*, but only indirectly the difficulties of image computation that may be connected to the representation of *transitions*. (When the intermediate products of image computation grow too large, they are subsetted. This leads to the computation of partial images [26].)

Disjunctive partitioning of the transition relation has been proposed in [9,8,22,20]. In the original formulation, partitioning is applied to image computation only. This keeps the test for convergence to the fixpoint simple (no dead-end computation is required,), is effective in reducing the image computation cost, and is fully automated. However, it provides for no departure from BFS, and is therefore ineffective at controlling the sizes of the state sets. When the sub-images computed for the partition blocks have substantial overlap, the disjunctive approach may do unnecessary work.

[9] also introduces the notions of λ-latches and partial iterative squaring. These are latches on which no other state variables depend; they are constrained at their initial value until no new states are achieved. Guided traversal subsumes this technique. Freezing λ-latches is indeed a good hint. Partial iterative squaring may be effective when the sequential depth of the circuit is high.

Biasing the search towards reaching particular states is the subject of [30,2,29]. A technique called *saturated simulation* is proposed in [30,2]. It consists of dividing the state variables into control and data variables, and selectively discarding reached states to allow the exploration of a large number of values for the control variables. Since this technique concentrates on "control" states, its usefulness is mostly in disproving invariants in circuits with a clear separation between control and data.

In [29] the idea of *guidepost* is presented in the context of explicit state search. Given a set of target states, a guidepost is a set of states that the search should try to reach as an intermediate step towards reaching the targets. There is a clear relation between guideposts and the hints used in guided symbolic traversal

formulated in terms of state variables. However, there are important differences. In explicit state enumeration, guideposts only help if the target states are indeed reachable. By contrast, the hints used in the symbolic algorithm do not have this limitation. Apart from pointing the way towards the target states, the hints used in guided traversal try to address the difficulties of image computation—not addressed by explicit search.

Constraints on the inputs of the system being verified are also used in symbolic trajectory evaluation [6]. However, full-fledged reachability analysis is not the objective of that technique. Constraints are also used in [15] to implement an assume/guarantee approach to verification. The constraining of the transition relation proceeds in the same way as in guided traversal, but the constraints are not chosen to speed up verification; they are imposed by the environment. One consequence is that there is no need to lift the constraints, and therefore no dead-end computation.

A similar situation occurs in *forward* model checking [14,3]. The traversal of the state space is constrained by the formula being verified. This has been reported as one of the reasons why forward model checking often outperforms the classical algorithm based on backward analysis. However, as in the previous cases, the constraints are not deliberately chosen to improve performance of the algorithm.

6 Experimental Results

We implemented the algorithm described in Fig. 1 in VIS [4]. Experiments were conducted on a 400MHz Pentium II machine with 1GB of RAM running Linux. We conducted two sets of experiments: One for reachability analysis; the other for invariant checking. We report the results of these experiments in Tables 1 and 2. We used thirteen circuits in our experiments. Of these, Am2901, and PALU are described in Section 4. The Am2910 is a microprogram sequencer. It has a 5-deep stack to store 12-bit addresses, a stack counter and a microprogram counter. CRC computes a 32-bit cyclic redundancy code of a stream of bytes. Fabric is an abstracted version of an ATM network switching fabric [17]. BPB is a branch prediction buffer that predicts whether a branch should be taken depending on the correctness of the previous predictions. Rotator is a barrel shifter sandwiched between registers. Vsa is a very simple non-pipelined microprocessor that executes 12-bit instructions—ALU operations, loads, stores, conditional branch—in five stages: fetch, decode, execute, memory access, and write-back. It has four registers, with one always set to zero. DAIO is a digital audio input output receiver. Soap is a model of a distributed mutual exclusion protocol [12]. CPS is a flat description of a landing gear controller. s1269 and s1512 belong to the ISCAS89 Addendum benchmarks.

The set of atomic hints belong to one of the four categories described in Section 4.2. We guessed some hints for CPS, s1269, and s1512 as we had no information regarding their behavior. In all these circuits, constraints on the primary inputs alone were effective. The hints were expressed as a concatenation of *repeat* (*atomic_hint*, ∞) and computed with frontier sets. We conducted our

Table 1. Experimental results for reachability analysis.

Circuits	FFs	Reachable States	Peak Live Nodes		Times in seconds		
			without hints	with hints	without hints	with hints	constr. traversal
CRC	32	4.295e+09	>42,384,405	16,312	Mem. out	0.11	0.11
BPB	36	6.872e+10	1,884,853	46,904	236.9	1.04	1.04
PALU	37	2.206e+09	29,134,676	25,753,334	1560.78	796.48	40.9
s1269	37	1.131e+09	31,225,168	3,287,777	2686.08	47.07	25.31
s1512	57	1.657e+12	23,527,792	26,210,543	5036.9	2372.7	2372.05
Rotator	64	1.845e+19	>12,891,752	16,071	Mem. out	0.18	0.18
Vsa	66	1.625e+14	25,061,852	6,858,369	6974.7	111.8	23.2
Am2901	68	2.951e+20	>38,934,128	349,781	Mem. out	3.65	3.65
DAIO	83	3.451e+14	6,390,705	3,746,631	24584.3	1752.12	1591.36
Fabric	87	1.121e+12	14,220,404	16,197,453	340.12	178.65	117.15
Am2910	99	1.161e+26	>36,696,241	26,238,783	Mem. out	1674.39	14.97
Soap	140	4.676e+08	1,011,972	628,959	36.93	13.87	9.67
CPS	231	1.108e+10	4,648,226	4,032,671	108.86	105.9	68.9

experiments without dynamic reordering in order to compare without variability introduced by it.

Table 1 compares reachability analysis with hints against BFS runs. 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. A memory limit of 1GB was set on each traversal run. Columns 4 and 5 show the peak number of live nodes during traversal. These numbers are reflective of the peak intermediate sizes during image computation. Columns 6 and 7 compare run times for reachability analysis without and with hints. Column 8 shows the portions of the times in Column 7 spent in traversing the constrained machines.

The circuits in this table are mid-sized, but four of these circuits—CRC, Rotator, Am2901, and Am2910 run out of memory for BFS. The hints described in Section 4 provide dramatic improvements to the traversal of these circuits, enabling completion times of a few seconds. With the remaining circuits in Table 1, BPB, s1269, DAIO, and Vsa demonstrate 1-2 orders of magnitude improvement. For Am2901 and s1269, different hints give comparable reduction in traversal times. The best times are reported in the table. With PALU, Fabric and s1512, the improvement in traversal time is roughly a factor of 2. In cases of dramatic improvement in traversal time, the difference peak number of live nodes support the notion that small computation times result from manipulation of small BDDs (see Fig. 2). CRC, BPB, s1269, Am2901, and Rotator also display this behavior.

The difference in traversal times for CPS is small. The traversal of the constrained machine takes only 69 seconds. Most of the remaining traversal time is taken up in the dead-end computation (only one additional image computation after the dead-end computation is required to prove convergence). This is an

Table 2. Experimental results for invariant checking.

Circuits	Invariant TRUE/ FALSE	Reached States		Times in seconds	
		without hints	with hints	without hints	with hints
CRC-1	FALSE	Mem. out	4.295e+09	Mem. out	0.2
BPB-1	FALSE	2.76347e+08	2.41592e+09	20.0	0.9
PALU-1	FALSE	6.05235e+06	5.27424e+06	2.3	1.3
PALU-2	FALSE	5.50225e+08	5.05979e+08	71.0	8.6
PALU-3	FALSE	1.50252e+09	1.33688e+09	164.8	19.2
PALU-4	TRUE	2.20562e+09	2.20562e+09	1529.9	824.5
Rotator-1	FALSE	Mem. out	1.84467e+19	Mem. out	0.3
Vsa-1	FALSE	5.81677e+12	1.23589e+11	806.8	5.6
Vsa-2	FALSE	2.92565e+11	9.47068e+13	173.2	40.5
Vsa-3	TRUE	1.62485e+14	1.62485e+14	6813.5	111.1
Am2901-1	FALSE	Mem. out	2.95148e+20	Mem. out	19.6
Fabric-1	FALSE	4.37925e+09	2.73703e+08	10.4	8.0
Fabric-2	TRUE	6.88065e+10	6.88065e+10	7.0	14.2
Fabric-3	TRUE	1.11004e+12	1.11004e+12	77.9	65.6
Am2910-1	FALSE	Mem. out	3.45961e+18	Mem. out	872.6
Am2910-2	TRUE	24576	24576	3.5	2.8
Am2910-3	TRUE	Mem. out	1.161e+26	Mem. out	1734.8
Soap-1	TRUE	4.676e+08	4.676e+08	31.5	16.1

example of a case where hints speed up traversal but the dead-end computation offsets these gains. It illustrates the importance of a hint producing a dense set of reachable states in addition to simplifying each image computation.

Table 2 shows comparisons for invariant checking runs without and with hints. Both passing and failing invariants were tried on the various circuits. Entries in Column 1 indicate the circuit name and the number of distinct invariants tried on them. Column 2 indicates whether an invariant failed or passed. Some invariants led to a reduced FSM based on the transitive fanin of the variables involved in the invariant property. Such invariants required exploration of fewer states to pass or fail. For a fair comparison the same reduced FSM was provided to the algorithm without and with hints. Columns 3 and 4 of the table indicate the number of reachable states that needed to be explored to prove these invariants. The last two columns give time comparisons. Failing invariants are reported as soon as they are detected.

The results show that hints may help both passing and failing invariants. For failing invariants where using hints was faster, hints steered the search toward the violating states. For example, in PALU, invariants involving the register values are checked faster when stalls are disabled. Invariants PALU-2 and PALU-3 benefit from this. On the other hand, a user should be careful that the hints provided do not conflict with the invariants being checked, thereby resulting is slower computation times. Invariant checking with hints on CRC, Rotator, Am2901, Am2910, and Soap enjoys the same benefits as reachability analysis.

Sometimes hints have less of an impact since the violating states are close to the initial states, or the reduced FSM is small, like in the case of Fabric.

In the case of the false invariant for Vsa-2, the 40.5 second run-time was generated with hint based on the knowledge of the circuit. A property-dependent hint is more effective, completes traversal in 6.5 seconds and generates the shortest error trace. This hint also works for the other invariants, but is about three times slower than the general-purpose one. So general purpose and property-dependent hints may be also useful in the reverse situations.

7 Conclusions

In this paper we have shown that state traversal guided by hints can substantially speed up invariant checking. Orders-of-magnitude improvement have been obtained is several cases, and visible gains have been achieved in most experiments we have run. The hints prescribe constraints for the input and state variables of the system. Deriving the hints requires some knowledge of the circuit organization and behavior, at the level a verification engineer needs to devise simulation stimuli or properties to be checked. Simple heuristics often allow one to find hints that are effective for both properties that fail and properties that pass. There is no guarantee, however, that a hint will be beneficial, as we have discussed in Section 6. We are therefore investigating methods that would allow guided traversal to recover from an ineffective hint with minimum overhead.

The success obtained in deriving hints from a rather small catalog of hint types seems to suggest the automatic derivation of hints as a fruitful area of investigation. We are interested in both methods that work on the structural description of the system (e.g., the RTL code), and methods that analyze the transition relation for information concerning function support and symmetries. We are also investigating the application of hints to model checking more general properties than invariants.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. 250
2. A. Aziz, J. Kukula, and T. Shiple. Hybrid verification using saturated simulation. In *Proc. of the Design Automation Conference*, pages 615–618, San Francisco, CA, June 1998. 260
3. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verification (CAV'98)*, pages 184–194. Springer-Verlag, Berlin, 1998. LNCS 1427. 252, 261
4. R. K. Brayton et al. VIS. In *Formal Methods in Computer Aided Design*, pages 248–256. Springer-Verlag, Berlin, November 1996. LNCS 1166. 254, 261
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. 251
6. R. E. Bryant, D. L. Beatty, and C. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proc. of the Design Automation Conference*, pages 397–402, 1991. 261

7. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. of the Design Automation Conference*, pages 46–51, June 1990. 253
8. G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitionining and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In *Proc. of the Design Automation Conference*, pages 728–733, Anaheim, CA, June 1997. 260
9. G. Cabodi, P. Camurati, and S. Quer. Improved reachability analysis of large finite state machines. In *Proc. of the International Conference on Computer-Aided Design*, pages 354–360, Santa Clara, CA, November 1996. 258, 260
10. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verification (CAV'98)*, pages 147–158. Springer-Verlag, Berlin, 1998. LNCS 1427. 259
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints. In *Proc. of the ACM Symposium on the Principles of Programming Languages*, pages 238–250, 1977. 250
12. J. Desel and E. Kindler. Proving correctness of distributed algorithms using high-level Petri nets: A case study. In *International Conference on Application of Concurrency to System Design*, Aizu, Japan, March 1998. 261
13. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 299–310, Berlin, 1994. Springer-Verlag. LNCS 818. 254
14. H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *Proc. of the International Conference on Computer-Aided Design*, pages 82–87, San Jose, CA, November 1996. 252, 261
15. M. Kaufmann, A. Martin, and C. Pixley. Design constraints in symbolic model checking. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verification (CAV'98)*, pages 477–487. Springer-Verlag, Berlin, 1998. LNCS 1427. 250, 261
16. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994. 250, 252
17. J. Lu and S. Tahar. On the verification and reimplementation of an ATM switch fabric using VIS. Technical Report 401, Concordia University, Department of Electrical and Computer Engineering, September 1997. 261
18. G. S. Manku, R. Hojati, and R. K. Brayton. Structural symmetry and model checking. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verification (CAV'98)*, pages 159–171. Springer-Verlag, Berlin, 1998. LNCS 1427. 259
19. K. L. McMillan. *Symbolic Model Checking*. Kluwer, Boston, MA, 1994. 250, 253
20. K. L. McMillan. Personal Communication, February 1998. 260
21. 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 Verification (CAV'98)*, pages 110–121. Springer-Verlag, Berlin, 1998. LNCS 1427. 250
22. A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned ROBDDs. In *Proc. of the International Conference on Computer-Aided Design*, pages 388–393, November 1997. 258, 260

23. R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. IWLS95, Lake Tahoe, CA., May 1995. 253, 254
24. K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of decision diagrams. In *Proc. of the Design Automation Conference*, pages 445–450, San Francisco, CA, June 1998. 260
25. K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, November 1995. 256, 260
26. K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *Proc. of the International Conference on Computer Design*, Austin, TX, October 1999. To appear. 258, 260
27. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of the International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, November 1993. 252
28. H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proc. of the IEEE International Conference on Computer Aided Design*, pages 130–133, November 1990. 253, 257
29. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of the Design Automation Conference*, pages 599–604, San Francisco, CA, June 1998. 260
30. J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 376–387. Springer-Verlag, Berlin, 1997. LNCS 1254. 260