Fast Falsification Based on Symbolic Bounded Property Checking *

Prakash M. Peranandam, Pradeep K. Nalla, Jürgen Ruf, Roland J. Weiss, Thomas Kropf and Wolfgang Rosenstiel
University of Tübingen, Germany
{peranand,nalla,ruf,weissr,kropf,rosenstiel}@informatik.uni-tuebingen.de

ABSTRACT

Symbolic property verification is an increasingly popular debugging method based on Binary Decision Diagrams (BDDs). The lack of optimization of the state space search is often responsible for the excessive growth of the BDDs. In this paper we present an accelerated symbolic property verification by means of a new *guiding* technique that automatically finds the set of interesting variables by exploiting the property and the transition relation of a design. Our property based state space guiding can substantially speed up the verification process. The heuristic picks up the interesting state or the input variables automatically and utilizes them in guiding the state space traversal. This guiding approach is a novel one as it is automatic, efficient and stable for fast falsification. Furthermore it does not degrade as much for full validation.

Categories and Subject Descriptors: B.5.2 Hardware: Design Aids

General Terms: Verification

Keywords: Property Checking, Fast Falsification, Guiding.

1. INTRODUCTION

According to statistics verification takes nearly 75% of the digital design time. Simulation has problems in catching the corner cases, and thus formal verification, e.g. model checking [10] complements it in order to cope with growing design size. Binary Decision Diagrams (BDDs) [5] are used as data structures to represent the state space in order to improve the scope of model checking.

The symbolic representations of state spaces [7] and bounded model checking (BMC) [2] have dramatically increased the design sizes that can be handled by formal verification tools. However, large designs cause memory overflow during exploration of the state space. In [19] a combination of symbolic simulation and bounded model checking is introduced, which is implemented in the tool SymC. Only the frontier set is kept in memory, i.e. no fix-point iterations are performed, there by controlling the memory overflow problem to some extent as compared to the traditional model checking algorithm. Although this approach performs well for certain classes of models and properties, the tool also faces memory exhaustion for large designs beyond a certain limit.

Another proposal to control this memory problem is to partition the BDDs (POBDDs) [20] into two or more pieces that are handled separately during further traversal. The typical problem of redundant computation due to the overlap of state space in the partitions are avoided by partitioning and traversing by the *windowing technique* [13]. Although this method has a couple of tradeoffs, the recent proposals of windowing technique aim at fast falsification, by handling the cross over states at the end. However, in this windowing algorithm there is no means of intelligent partitioning of windows and steering the traversal towards the target states.

This paper contributes to this area of partitioning the BDD by presenting a scalable algorithm that results in efficient and fast partitioning eventually boosting the verification process of large designs. We propose a *Guiding* algorithm based on splitting that relies on Shannon expansion. The partitions can be traversed either sequentially [9] or in parallel [12]. In this paper we deal only with the sequential handling of partitions. The *Guiding* algorithm aims at fast falsification by steering the traversal. The fast falsification approach tries to identify some traces as high priority traces and traverses those traces first, with the aim of finding either validation or violation of the property depending on whether it is an existential or universal quantification, respectively. Our heuristic is implemented and tested with the symbolic bounded property verification tool SymC. Our experimental results shows time gain due to the acceleration obtained by our guiding algorithm. The paper is organized as follows: Section 2 briefs the preliminaries of this work, Section 3 discusses related work, Section 4 summarizes the symbolic bounded property checking tool SymC. Section 5 details the guiding algorithm, and Section 6 discusses our experimental results. Finally, Section 7 concludes and outlines our future work.

2. PRELIMINARIES

The Shannon expansion $f = (a \wedge f_a) \vee (!a \wedge f_{!a})$ forms the basis for many BDD based manipulations. Our divide-and-conquer approach that partitions the state space is based on this expansion. The state-of-the-art partitioning technique *windowing* [13] identifies its windows by a unique combination of variables. Each window restricts its state space to capture its uniqueness at every traversal step. In comparison, the splitting technique restricts its uniqueness at only one step whenever the threshold is reached.

Although the symbolic representation alleviated the memory explosion problem to some extent, the research concentrated on efficient representations, and thereby resulted in partitioned *Transition*

*This work has been funded in part by the German Research Council (DFG) within projects GRASP (KR 1869/2-2) and KOMFORT (KR 1869/4-1) and by the BMBF and edacentrum within project FEST.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

Relation(TR) [6]. Given a synchronous circuit design with m state variables and n input variables, there are 2^m possible states that belong to the set of states S. The set of states is represented as a BDD and is denoted as $S(E)$, where $E = \{e_1, \ldots, e_m\}$ and $I = \{i_1, \ldots, i_n\}$ are the set of Boolean state and input variables respectively. To represent the transition relation using BDD, we require a second set of states S' encoded by $E' = \{e'_1, \ldots e'_m\}$ called next state variables. The BDD representing the transition relation $\mathcal{T}$ is then denoted as $\mathcal{T}(E, E')$. For each state variable e_i, there is a piece of combinational logic which determines its next state value. Let f_i be the next state function computed by this logic for variable e_i. Then the transition relation $S \times S$ can be derived by the next state function vector, $f = \{f_1, \ldots, f_m\}$. Then the transition relation of a synchronous circuit is denoted as $\mathcal{T}(E, E')$ and it is defined as,

$$\mathcal{T}(E, E^{'}) = t_1(E, e_1^{'}) \wedge \ldots \wedge t_m(E, e_m^{'})$$
$$\text{where } t_i(E, e_i^{'}) = (e_i^{'} \equiv f_i(E))$$

In practice, each t_i can be represented by small and separate BDDs that are implicitly conjuncted and are referred as partitioned TR. It is further improved by the method called early quantification [8, 11] that is based on the circuit's locality. The guiding algorithm utilizes the influence factors defined in [14] and in this paper we extended the influence factor for input variables. The *lookback* influence factor definition and the input influence extension are given below,

$$\mathcal{D}_{\mathrm{S}}^{\downarrow}(e, 1) = \{v_1, \ldots, v_m\}$$
where $v_i \in supp(t_e) \cap E$, such that $t_e = (e^{'} \equiv f_e(E))$,
$1 \leq \mathrm{i} \leq \mathrm{m}$. and $supp(t_e)$ is the set of all variables occuring in t_e.

$$\mathcal{D}_i(e) = \{i_1, \ldots, i_n\}$$
where $i_j \in supp(t_e) \cap I$ and $1 \leq \mathrm{j} \leq \mathrm{n}$.

3. RELATED WORK

3.1 Partitioning

Many approaches for decomposing Boolean functions represented as BDDs exist in the literature. For sequential verification [9] partitioning algorithms aim at creating balanced partitions. The main distinguishing feature of these algorithms is the employed cost function for selecting the partitioning variable. The cost functions typically take into account the achieved memory reduction, the amount of sharing between the cofactors, and the memory balance of the cofactors. Also, the CUDD package [21] contains various decomposition algorithms, producing both balanced and unbalanced partitions. Furthermore, decomposition techniques allow representing the same function with multiple BDDs (POBDDs), thereby requiring less memory [13] by means of the windowing technique. Recent proposals of the windowing technique aim at fast falsification, where the cross over states (non-owned states) are handled only after the termination condition of windows are reached. The image computation algorithms have to be updated for these techniques. The more complex operations are set off by the reduced peak memory requirements of the BDDs [20]. As shown in [4], the reduction can even be exponential. Finally, dense under-approximations [16] try to reduce the memory requirements of the BDD while still capturing a large percentage of the state space. This dense approximation is of interest for the faster falsification approach.

3.2 Guiding

In [22] the topic of biasing the search towards reaching target states is addressed. Several heuristics are discussed, all of which aim at enhancing the verification tool with a bug finding capability called guiding. In general, the authors claim that the *Target enlargement* combined with *Tracks* and *Guideposts* heuristics can consistently find errors much faster than their individual performance or breadth first search. However, these guiding ideas are experimentally tried on explicit model checking. In [17], the authors addressed the usage of hints in symbolic traversal and also showed ways to identify the good hints. Apart from pointing the way towards the target states, the hints used in symbolic guided traversal try to address the difficulties of image computation. This idea was then extended in [3] to allow nested fix-points and to use hints to obtain over-approximation.

3.3 Contributions

POBDD based verification algorithms ideally reduce the memory overflow problems, and have the advantage of quitting the traversal if a target state is reached. The main contribution of our approach is a hybrid, POBDD based splitting and guiding, suited for fast falsification of properties. The first part of our algorithm collects all the interesting variables to validate (violate) an existential (universal) property. The interesting set of variables obtained by exploiting the influence factors is dependent on the property. Splitting is done based on those variables such that one of the splits contains a higher number of potential target states. The second part identifies the right split to be traversed first by steering given by our cost function, consequently accelerating the verification process.

The interesting set of variables is an approximation of target enlargement and the variables can be either state or input variables resulting in state or input variable guiding. Our input variable guiding can be compared to the hint guiding [17] approach except for the fact that our approach is fully automatic.

4. BOUNDED PROPERTY CHECKING AND PARTITIONING

The formal verification algorithms in [19] combine bounded property checking and symbolic traversal. The algorithms have been implemented in the tool SymC, whose general operation is shown in Fig. 1. This tool takes the design as a Verilog gate list or in a SMV-like format. The Gate list format is then translated into a symbolic simulatable format and represented using BDDs. The property for this tool is specified in Finite Linear Time Logic (FLTL) which is an enrichment of pure Linear Time Logic (LTL) [19] with time limits. This FLTL property is also translated into a special automata called Accept-Reject automata (AR-automata) [18], which can be symbolically represented by BDDs and simulated. AR-automata allow finding violations or validations of properties on finite sequences, thus they are well suited for bounded property checking. The checking algorithm manipulates both the system description and the AR-automata represented as BDDs. In order to

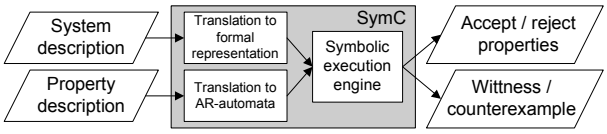

Figure 1: Overview of SymC operation.

avoid the construction of the complete transition relation, a set of conjunctively partitioned transition relations is built. The verification is done by traversing the product machine of the design and the AR-automata. Verification stops when finding either a validation or a violation of the property. The termination condition differs if one checks the property on all paths, i.e., universal quantification, or on one path, i.e., existential quantification (CTL* interpretation). Informally, the sequential termination condition is defined as follows,

- **Universal** If one reject state is detected in the current state set, a violation of the property is found. If all states in the current state set are accepting states, a validation of the property is found. Otherwise, the property is still pending.

- **Existential** If one accept state is detected in the current state set, a validation of the property is found. If all states in the current state set are rejecting states, a violation of the property is found. Otherwise, the property is still pending.

A central optimization technique for the algorithm is state set splitting. Whenever a threshold for the size of the BDD representing the current state set is reached, the set is split into two disjoint parts and the algorithm continues working on these subsets in a divide-and-conquer manner. One of the splits is considered for further traversal, while the other split is stacked for future exploration. The terminating condition of the SymC also has to be adopted for this splitting. In the case of universal correctness, if one of the splits is proven for failure then the whole verification process can be stopped. Similarly, in the case of existential correctness, if one of the splits is proven for acceptance then the whole verification process can be stopped. Otherwise, the verification process is repeated for all the stacked subsets. The following sections describe the guiding algorithm for the accelerated traversal of the splits for fast falsification of the property.

5. GUIDING

Guiding is based on the property that is supposed to be verified. There are two different ways of guiding in *SymC*:

- State variable guiding : The actual state set is partitioned into two parts using one of the state variables that are collected.

- Input variable guiding : The actual state set is restricted with the set of input variables (hints) such that the transitions satisfying those set of variables are allowed and the others are discarded.

Both types of guiding basically aim at fast finding of target states defined by the property of the kind "if A then C", i.e., $A \rightarrow C$, where A and C are LTL/FLTL expressions. For guiding, only the C part of the property is of interest. Because the tool *SymC* is highly optimized for the traversal, the traces that are not satisfying the A part of the property will be removed from the traversal and in some cases the assumption part can also be empty, i.e., *True*. Hence, the guiding algorithm collects all the signals of the C part.

The property is taken and manipulated as a string in order to identify the set of output signals and other state variables that are involved in the property, which is the set of interesting signals. All the interesting output signals that are collected are then reduced to sets of state variables as per the definition. This interesting set of variables are the ones that jointly define the property to be valid or invalid. Hence, we decompose the validation of one property to the validation of a set of variables. This set of variables are the target variables that have to be restricted in order to guide the traversal. This set of interesting variables are usually small and thus restriction can not benefit much. However this set of variables can be improved by the approximation of target enlargement method [22], in order to obtain a broader scope of guiding by collecting the set of variables that influence this small set of target variables. This set of new influencing variables can be either the state or input variables.

This set of influencing variables is obtained using the influence factors. Fig. 2 portrays the pseudo algorithm of collection of the interesting variables. The identification of the property and its decomposition is shown in lines 7 and 8. The target variables set collection and its corresponding influence variables grouping is shown in lines 10 to 19. The influencing variables are categorized into state and input variables and are collected in the set $\mathcal{G}$ and $\mathcal{I}$, respectively.

```
// P is the property                                                          1
// G is the interesting state variables                                       2
// I is the interesting input variables                                       3
// sst1, sst2, sst3, sst4 are set of string                                   4
// st, s are present state variables                                          5
Best−Var−Set(in: P; out: G, I)                                                6
commit = getCommitment(P) // Collect C part                                   7
sst1 = decompose(commit) // Decompose to set of vars.                         8
for all st ∈ sst1                                                             9
  if st == output_var                                                         10
    sst2 = getDefinition(st)// collect. state var. for the output var.        11
    for all s in sst2                                                         12
      sst3.insert(D_S^↓(s, 1)) // state var. inf. collection.                 13
      sst4.insert(D_I(s)) // input var. inf. collection.                      14
  else                                                                        15
    sst3.insert(D_S^↓(st, 1)) // state var. inf. collection.                  16
    sst4.insert(D_I(st)) // input var. inf. collection.                       17
G = sst3                                                                      18
I = sst4                                                                      19
```

Figure 2: Guiding variables collection algorithm.

One can see that the guiding technique is more efficient if the property contains only the subset of the actual set of state or output variables. So that the set $\mathcal{G}$ and $\mathcal{I}$ will contain only a subset of the actual set of state or input variables, respectively. Otherwise, every variable will be of interest, leading to no special restriction that could be applied for fast falsification. However, the image computation is a major and time consuming factor and it is generally proportional to the BDD node count. Therefore, as an optimization we utilize the *SubsetShortPaths* [15] to under approximate the actual state set and then apply the guiding technique.

5.1 State Variable Guiding

Guiding based on state variables entails partitioning the state space into two by splitting using a selected state variable in such a way that one of the splits has more concentration of potential error states and is comparatively smaller. Let us give an example in order to explain the state variable guiding. Assume the partitioned TR in equation 1. Let us define a property $P_1 = Xe_3$ that is to be verified, where the assumption part is empty.

$$\begin{aligned} T_1 &= (e_1' \equiv f_1) \text{where } f_1 = (\neg e_2 \vee e_3) \vee i_1 \\ T_2 &= (e_2' \equiv f_2) \text{where } f_2 = e_2 \vee (\neg i_2 \vee i_1) \\ T_3 &= (e_3' \equiv f_3) \text{where } f_3 = e_2 \wedge (\neg e_3 \vee i_2) \end{aligned} \quad (1)$$

The property P_1 expresses that variable e_3 should be eventually true within 1 clock cycle. Hence the interesting variable set is only $\{e_3\}$, therefore, the guiding set of variables is $\mathcal{G} = \{ e_2, e_3 \}$. First, let us assume the property is to be proved universally, hence the guiding should steer the traversal to the traces that do not accept the property. As we know that the set $\mathcal{G}$ is the set of variables that decides the e_3's next state values, we partition the state space with only those variables in the set $\mathcal{G}$.

If the state space is assumed to be S = {010, 100, 110}, then splitting the set S with the variable e_3 ends up with one empty split as shown in Fig. 3 (a). Trivially, due to the balancing condition defined as $max(|S_v|, |S_{\overline{v}}|) \leq \frac{2}{3}|S|$, where $|S|$ is the BDD node

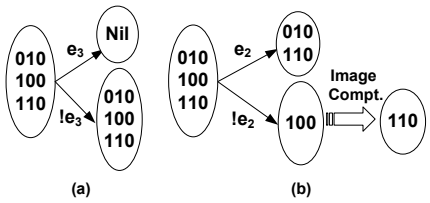

Figure 3: State variable guiding for universal property

count of S, it is not the right choice. Hence we try the next possible variable e_2, and now we have a better partition with S_1 = 100 and S_2 = 010, 110 as shown in Fig. 3 (b). Therefore, the first part of guiding is to split with the interesting variable that satisfies the balancing condition. The second part is our cost function (see section 5.3) that identifies the split that contains more of the potential target states. Fig. 3 (b) shows that traversing the negative cofactor of the variable e_2, identified by the cost function, leads to the failure of the property. The above two parts, splitting with interesting variables and the cost function, together defines the state variable guiding algorithm. Suppose that the property P_1 should be proved existentially, then only the interesting set of variables will be updated as $\overline{\mathcal{G}} = \text{E} - \mathcal{G}$, which in our example is $\overline{\mathcal{G}}$ = { e_1}. In this case both the splits leads to the validation independent of the cost function.

5.2 Input Variable Guiding

In contrast to state variable guiding the input variable guiding basically restricts the possible next state set into a smaller subset, rather than splitting. This restriction is in such a way that this subset has a higher concentration of the potential target states. Further traversing this smaller subset is faster and it is easier to reach the error states.

Let us use the same TR in equation 1 and the property P_1 in order to explain the guiding by input variables. Let us assume the set of input variables I = { i_1, i_2 }. From the equation T_3 in 1 we observe that the next state value of the interesting variable e_3 is partially dependent on the input variable i_2. Hence, the set of input guiding variables is $\mathcal{I}$ = {i_2}. If we assume the property is to be proven universally, the guiding should steer the traversal to the traces that do not accept the property. As we know that the set $\mathcal{I}$ is the set of variables that influences the e_3's next state value, we partition by restricting the state space with only transitions with those variables in the set $\mathcal{I}$. If the present state space is assumed to be S = {001, 011, 111, 100} then Fig. 4 (a) shows the actual set of next states as a result of image computation. In order to guide using the input variables, we restrict the present state space with combination of input variables so that it only takes a subset of the actual transitions. The cofactor of the input variables is identified by the cost function.

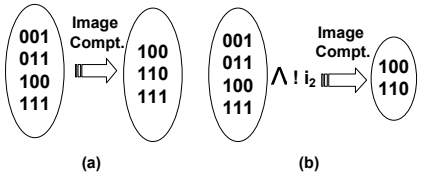

Figure 4: (a) Without guiding (b) Guiding with input var. $!i_2$

In our example, the restriction is done by conjuncting the input variable $!i_2$ to the set S. During the image computation the transition with the i_2 gets removed automatically, hence restricting the next state space as shown in Fig. 4 (b). Restricting the set S with $!i_2$ therefore leads to the next state space S_1 = { 110, 100} and this state space rejects the property P_1. Suppose that the property P_1 should be proved existentially, then the interesting variable set is $\overline{\mathcal{I}} = \text{I} - \mathcal{I}$, which in our example is $\overline{\mathcal{I}}$ = {i_1}. One can easily compute to see that both restrictions have a trace that leads to the acceptance of the property independent of the cost function identification.

5.3 Cost function

Guiding in SymC is basically composed of two steps: one is to partition the state space into two and the second is to identify the right partition to traverse first. The second step of the guiding procedure is called *Steering*. The steering algorithm is shown in Fig. 5. The steering procedure depends on the commitment signals that are collected from the property. The transition relation of every signal of the commitment is identified as shown in the lines 3 and 4. Then the set of interesting variables is restricted to be positive in that transition and the number of minterms are counted (line 8). The original transition is again restricted with the negative cofactor and minterms are counted (line 12).

The minterm counting is the key factor of guessing the cofactor of the variable in the transitions. For example, if the variables are connected as positive cofactor with some other variables, then restricting the variable to be true will leave the remaining variables in the function. That remaining variables forms the minterm of the function. If the same is present as a negative cofactor, then restricting it as true will result the whole function to be false, i.e., no minterm. Hence, higher the minterm count of a transition with a specific cofactor restriction then the variables are present in that form. Therefore to falsify the property one has to follow the negation of the cofactor. Of course, the argument here clearly assumes the variables in the transitions are connected conjunctively. This fact is not always true, hence resulting in the best and worst cases of the algorithm. However, the experiments show good results for this assumption in most cases.

```
// T is the Partitioned transition relation                          1
Steer(in: T, Φ; out: Res)                                            2
 for all c ∈ commitment signals from the property                    3
   t_c = T.find(c')                                                  4
   temp = t_c                                                        5
   for all v ∈ Φ // the set of variables to be checked for its cofactor 6
     temp = temp.restrict(v)                                         7
   Pcofactor = temp.CountMinterm()                                   8
   temp = t_c                                                        9
   for all v ∈ Φ                                                     10
     temp = temp.restrict(!v)                                        11
   Ncofactor = temp.CountMinterm()                                   12
   if Pcofactor ≥ Ncofactor                                          13
     return follow negative cofactor                                 14
   else                                                              15
     return follow positive cofactor                                 16
```

Figure 5: Cost function - Steering algorithm for guiding.

In our example for state variable guiding, constraining the variable e_2 ($!e_2$) in the transition T_3 results in 1 (0) minterm. Hence, the cost function returns *follow negative cofactor*, identifying to follow the "$!e_2$". Similarly, the cost function is also applied to the input guiding.

6. EXPERIMENTAL RESULTS

We implemented the guiding algorithms in SymC. Experiments were conducted on a Sun Blade 1500, SunOS 5.8, with 1GB RAM. We used 14 circuits in our experiments, of which 9 circuits are from a IBM benchmarks suite and 5 are from the ISCAS'89 benchmarks. We conducted three sets of experiments with static variable ordering, one for comparing the guiding algorithms with other standard partitioning algorithms (as listed below) for both universal and existential properties. The other set of experiments is a fairness comparison that compares all the algorithms with and without the state space under-approximation (*u-approx*). The last set of experiments for completeness that shows guiding do not degrades for full validation. The *SubsetShortPath* is a *u-approx* technique used in our experiments. The properties for the IBM benchmarks are obtained from the suite itself and except the *29-batch* all others are altered in order to make the property universally rejected, in order to bring the guiding effect to the picture. For ISCAS'89 circuits we had no information regarding their behaviour. Therefore we had to guess some properties, except for the circuit *s1423* where we utilized the properties from the paper [1] with a time bound of 14. The preprocessing time, i.e., BDD construction and the influence factor collection time, is hardly 2% of the total verification, except *30-batch* which is more than 15%. The properties that are used express the reachability of set of states. Fig. 6 and 7 lists the verification runtime (least runtime is highlighted) in seconds for different partitioning algorithms for the universal and the existential properties. We used different set of properties for universal and existential results. The algorithms listed from third to the seventh columns are as follows: *VD* is the variable disjunctive decomposition algorithm provided by CUDD. *Win-8* is the windowing technique with 8 windows. *SP* is the *u-approx* algorithm that is also provided by CUDD, and *IG* is the guiding with input variables and *SG* is the guiding algorithm with state variables presented in this paper. The term C_{over} refers that the target state can only be reached by traversing the cross-over states. The first column lists the design name and the second column shows the number of flipflops. The last column S_{Up} is the speed up obtained by our guiding algorithms compared to the highest in the table.

Design	FFs	VD	Win-8	SP	IG	SG	S_{Up}
s1269	37	>2 hrs	>2 hrs	3546.0	**2.5**	>2 hrs	1418.4
s1423-I	74	>1 hr	708.8	300.5	**122.5**	232.7	5.7
s1423-II	74	469.6	364.9	175	**85.78**	146.8	5.4
s3271	116	245.8	C_{over}	322	**26.3**	282.3	12.2
s4863	104	>1 hr	>1 hr	1198.2	**579.6**	1093.7	2.0
18-batch	143	921.3	1691.6	909.5	637.6	**522.2**	3.2
19-batch	181	1194.9	3362.4	1433.9	1206.9	**846.4**	3.9
20-batch	148	913.3	1675.6	869.1	662.5	**582.0**	2.8
23-batch	192	205.9	C_{over}	177.6	**73.9**	109.3	2.7
28-batch	109	239.6	562.3	324.1	204.8	**169.1**	3.3
29-batch	95	322.4	C_{over}	128.1	139.9	**88.4**	3.6
30-batch	191	493	587	294	284	**205**	2.8

Figure 6: Verification time comparison for *Universal* prop.

Design	FFs	VD	Win-8	SP	IG	SG	S_{Up}
s1423-I	74	> 1 hr	806.9	336.1	**199.9**	331.8	4.0
s1423-II	74	> 1 hr	689	320.1	**181.9**	249.3	3.7
10-batch	297	447.8	215.7	247	202.2	**201.4**	2.2
23-batch	192	243.6	401.1	190.1	**139.2**	155.8	2.8
28-batch	109	266.9	704.8	329.1	**251.7**	307.2	2.8
29-batch	95	569.1	600.7	257.7	**175.6**	190.1	3.4
30-batch	191	2623.6	1810	872.9	**360.8**	589.6	7.2

Figure 7: Verification time comparison for *Existential* prop.

The windowing technique in SymC is adopted for the fast falsification by avoiding the computation and distribution of cross-over states among windows. However, the results of the designs *s3271*, *23-batch* and *29-batch* shows that fast falsification by windowing technique is not by means of intelligent guiding of traversal but by chance of target state being in the way, as the target state is not found and can be reached only through cross-over (C_{over}) states. In general the results show that the *Input Var.* and *State Var.* guiding is efficient.

To analyze the results we look at the number of input and state variables involved in the guiding process, that is listed in Fig. 8 and Fig. 9. The first column *Design* lists the design name, the second through the sixth are as follows: P_v gives the total number of variables involved in the property, I_n lists the total number of input variables, U_i lists the number of input variables utilized for *Input Var.* guiding, S_n lists the total number of state variables, U_s lists the number of state variables utilized for *State Var.* guiding.

Design	P_v	I_n	U_i	S_n	U_s
s1269	24	18	15	37	36
s1423-I & II	1	17	9	74	27
s3271	43	26	26	116	42
s4863	9	49	33	104	40
18-batch	40	111	4	143	93
19-batch	6	175	4	181	131
20-batch	40	123	5	148	85
23-batch	7	153	6	192	24
28-batch	8	35	7	109	30
29-batch	49	12	2	95	92
30-batch	5	36	3	191	45

Figure 8: Variables utilization for *Universal* prop.

For *Input Var.* guiding for universal properties the designs *s1269, s3271, s4863* utilizes from 67% to 100% (Fig. 8) of the total input variables, while for *s1423* it is 41% and for *28 and 29-batch* it is around 18% and for all other design it is less than 5%. For the existential property (Fig. 9) all the designs perform better with *Input Var.* guiding, although the designs vary in the percentage of utilization of input variables. Therefore one can generalize that if more number of input variables are involved then the *Input Var.* guiding is efficient for both universal and existential properties.

Design	P_v	I_n	U_i	S_n	U_s
s1423-I & II	1	17	8	74	27
10-batch	18	125	109	297	9
23-batch	2	153	153	192	24
28-batch	8	47	40	109	30
29-batch	49	10	2	95	13
30-batch	2	36	36	191	45

Figure 9: Variables utilization for *Existential* prop.

The number of state variables involved in *SG* (Fig. 8) are as follows: The designs *18, 19, 20, 29-batch*, utilizes more than 60% to 90% of state variables for guiding, and the results are better for these designs. The design *s1269* utilizes almost 100% but it degrades due to the fact that there is no special set of variables. In other words, all the state variables are in the interesting variable set thereby degrading the *State Var.* guiding approach to the usual ones. The other designs vary from 25% to 40% resulting in moderate time gain. However, as an exception, the design *23-batch* utilizes only 3% (8%) of input (state) variables but still gives a comparatively good result. For the existential property (Fig. 9) all the designs perform average with *State Var.* guiding, although the designs vary in the percentage of utilization of state variables. This reveals that although guiding in general increases the probability of finding a target state faster, it is never 100% accurate. Although

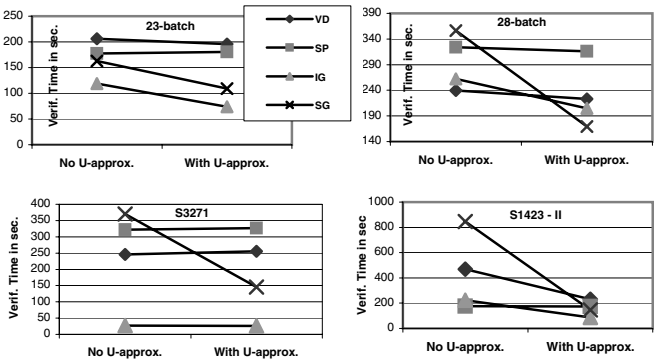

Figure 10: Impartial comparison.

the guiding algorithm picks up the right path in most cases BDD node count is large, resulting in expensive image computation step. Guiding without under-approximation works for some designs, in order to make it applicable for wide range of designs we combine the under-approximation along with guiding which leads to an efficient combination. Our experiments exhibited that guiding combined with under-approximation proved to be more efficient than the other combination as shown in Fig. 10. It is a impartial comparison that compares all the algorithms with and without (*u-approx*). For the *ShortPath* algorithm which itself is an *u-approx* technique it is under-approximated two times.

The guiding algorithm aims at fast falsification, i.e., to find the bugs as fast as possible. However, it should also perform better in case of no errors in the design. Therefore we conducted full validation experiments with two examples, *s1512* and *04-batch* (Fig. 11).

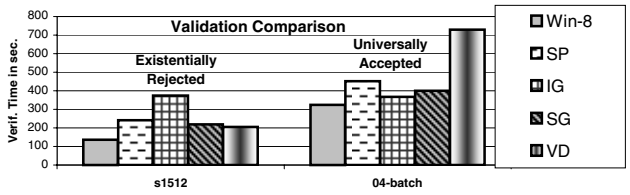

Figure 11: Full validation comparison.

For the example *s1512* the *Input Var.* guiding takes the maximum time, while *State Var.* guiding performs comparatively the same. For the example *04-batch*, both *Input Var.* and *State Var.* guiding performs better than other algorithm except the windowing. Fig. 11 shows that although the guiding algorithms is meant for fast falsification it does not degrade so much for full validation.

7. CONCLUSIONS AND FUTURE WORK

The fast falsification algorithm proposed in this paper utilizes the set of interesting variables that are identified automatically and guides the POBDD based state space traversal. The experimental results show that this approach allows gain in verification time. From our results we conclude that both of our guiding algorithms are effective as compared to other approaches. In general, input variable guiding gains if more input variables are involved, but state variable guiding is better if the number of state variables involved are around 30% to 60%. The state variable guiding is of advantage if the user expects the property to be universally valid or existentially rejected (full validation).

We are further investigating methods for steering and obtaining more information from properties that would allow guided traversal to be more stable and much faster. The success of fast falsification obtained by this guiding interests us to further explore the option of guiding by considering more interesting variables and using it in a defined pattern to improve and stabilize the guiding heuristic.

8. REFERENCES

[1] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference*, volume 1954 of *Lecture Notes in Computer Science*, pages 390–404. Springer, 2000.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In M. Zelkowitz, editor, *Highly Dependable Software*, volume 58 of *Advances in Computers*. Academic Press, 2003.

[3] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 29–34, New York, NY, USA, 2000. ACM Press.

[4] B. Bollig and I. Wegener. Partitioned BDDs vs. other BDD models. In *ACM/IEEE International Workshop on Logic Synthesis (IWLS)*, May 1997.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. Denyer, editors, *International Conference on Very Large Scale Integration (VLSI)*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[7] J. Burch, E. Clarke, K. L. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computing*, 98(2):142–170, June 1992.

[8] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th Conference on Design Automation*, pages 403–407. ACM Press, 1991.

[9] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In *34th Conference on Design Automation*, pages 728–733. ACM Press, 1997.

[10] E. M. Clarke, O. Grumberg, and D. E. Peled. *Model Checking*. The MIT Press, December 1999.

[11] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation. In D. L. Dill, editor, *Conference on Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310, Stanford, California, USA, June 1994. Springer-Verlag.

[12] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In W. A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference*, volume 2725 of *Lecture Notes in Computer Science*, pages 54–66. Springer Verlag, 2003.

[13] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned ROBDDs - a compact, canonical and efficiently manipulable representation for boolean functions. In *1996 IEEE/ACM International Conference on CAD*, pages 547–554. ACM and IEEE Computer Society Press, 1996.

[14] P. M. Peranandam, P. K. Nalla, R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel. Overlap reduction in symbolic system traversal. In *IEEE International High Level Design Validation and Test Workshop 2005 (HLDVT 05)*, November 2005.

[15] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *35th Conference on Design Automation*, pages 445–450. ACM Press, 1998.

[16] K. Ravi and F. Somenzi. High-density reachability analysis. In *1995 IEEE/ACM International Conference on CAD*, pages 154–158. ACM and IEEE Computer Society Press, 1995.

[17] K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 250–264, London, UK, 1999. Springer-Verlag.

[18] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on a multi-valued AR-automata. In W. Nebel and A. Jerraya, editors, *Design, Automation and Test in Europe 2001*, pages 742–748. IEEE Press, 2001.

[19] J. Ruf, P. M. Peranandam, T. Kropf, and W. Rosenstiel. Bounded property checking with symbolic simulation. In *Forum on Specification and Design Languages 2003*, 2003.

[20] D. Sahoo, S. K. Iyer, J. Jain, C. Stangier, A. Narayan, D. L. Dill, and E. A. Emerson. A partitioning methodology for BDD-based verification. In A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design, Fifth International Conference*, volume 3312 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2004.

[21] F. Somenzi. CUDD: CU decision diagram package, release 2.4.0. http://vlsi.colorado.edu/~fabio/CUDD, 2004.

[22] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Design Automation Conference (DAC)*, pages 599–604, San Francisco, CA, June 1998. ACM/IEEE.