Prioritized Traversal: Efficient Reachability Analysis for Verification and Falsification

Ranan Fraer, Gila Kamhi, Barukh Ziv, Moshe Y. Vardi*[1], Limor Fix

Logic and Validation Technology, Intel Corporation, Haifa, Israel
Dept. of Computer Science, Rice University*
{rananf, gkamhi, zbarukh, lfix}@iil.intel.com, vardi@cs.rice.edu

Abstract. Our experience with semi-exhaustive verification shows a **s**evere degradation in usability for the corner-case bugs, where the tuning effort becomes much higher and recovery from dead-ends is more and more difficult. Moreover, when there are no bugs at all, shifting semi-exhaustive traversal to exhaustive traversal is very expensive, if not impossible. This makes the output of semi-exhaustive verification on non-buggy designs very ambiguous. Furthermore, since after the design fixes each falsification task needs to converge to full verification, there is a strong need for an algorithm that can handle efficiently both verification and falsification. We address these shortcomings with an enhanced reachability algorithm that is more robust in detecting corner-case bugs and that can potentially converge to exhaustive reachability. Our approach is similar to that of Cabodi et al. in partitioning the frontiers during the traversal, but differs in two respects. First, our partitioning algorithm trades quality for time resulting in a significantly faster traversal. Second, the subfrontiers are processed according to some priority function resulting in a mixed BFS/DFS traversal. It is this last feature that makes our algorithm suitable for both falsification and verification.

1 Introduction

Functional RTL validation is addressed today by two complementary technologies. The more traditional one, *simulation*, has high capacity but covers only a tiny fraction out of all the possible behaviors of the design. On the contrary, *formal verification* guarantees full coverage of the entire state space but is severely limited in terms of capacity. A number of hybrid approaches that combine the strengths of the two validation technologies have emerged lately. One of them is *semi-exhaustive verification* [1,3,4,5,9] that aims at exploring a more significant fraction of the state space within the same memory and time limits. While maintaining high coverage, similar to exhaustive verification, this technique is reaching more buggy states, in less time and with smaller memory consumption. In this family we can include several heuristics, like the *high-density reachability* [1], and *saturated simulation* [4,5].

We have run extensive experiments using the high-density reachability on a rich set of industrial designs, and we have reported encouraging results [15]. As opposed

[1] Work partially supported by NSF grant CCR-9700061 and a grant from Intel Corporation.

to previous studies, where semi-exhaustive algorithms are evaluated on the basis of their state space coverage, our results show these algorithms to be highly effective in bug finding too. We have identified two classes of problems where semi-exhaustive verification is particularly beneficial:

- *Dense bugs*: Designs characterized by a high density of buggy states. The exhaustive verification usually finds these bugs too, but the semi-exhaustive one reaches the buggy states much faster.
- *Corner-case bugs*: Designs characterized by sparsely distributed bugs in the state space such that the exhaustive verification blows up long before reaching them. These bugs are now within the reach of semi-exhaustive algorithms that can go deeper in the state space, while keeping the memory consumption under control.

Nevertheless, the corner-case bugs require a much higher tuning effort. Experience shows that often only a specific subsetting heuristic and a particular threshold are leading the semi-exhaustive algorithm to a buggy state, and very small variations of this "golden" setting are bound to fail. Failure here means getting stuck in a *dead-end*, where no more new states are found, and yet we are not sure if all the reachable states have been covered. The dead-end recovery algorithm of [11] addresses this problem by using the *scrap* states (the non-dense subsets, ignored by previous subsettings) to regenerate the traversal. The scrap states are partitioned as well, similarly to [11]. However, we had little success with this algorithm, as the BDDs of the scrap partitions gets larger and larger and more difficult to use.

For the same reason, shifting from semi-exhaustive traversal to an exhaustive one is very expensive, and often impossible. This makes the semi-exhaustive approach unsuitable for verification tasks (where there are no bugs at all). The ability to address both verification and falsification problems is important in an industrial context, since after the design fixes each falsification task needs to converge to full verification. Moreover, a significant problem in industrial-size projects is to ensure that the process of fixing one design problem does not introduce another. In the context of conventional testing this is checked through regression testing [19]. If consecutive test suites check N properties, a failure in one property may require re-testing all the previous suites, once a fix has been made. Efficient regression testing, clearly, requires an algorithm powerful both for verification and falsification.

To overcome the above drawbacks in the semi-exhaustive approach, we introduce in this paper an enhanced reachability analysis that is efficient both for falsification and verification and that is less sensitive to tuning. For falsification, we still want to use the mixed BFS/DFS strategy that is at the core of the semi-exhaustive approach. As for verification, we replace the dense/non-dense partitioning of [1] with a balanced partitioning as in Cabodi et al. [13]. As noted above, the density criterion is unsuitable for verification, due to the difficulty of exploring the non-dense part of the state space.

So our approach follows the lines of Cabodi et al. [13], producing at each step a set of balanced partitions instead of one dense partition, but differs in two respects. First, our partitioning algorithm trades quality for time resulting in a significantly faster traversal. The partitioning is based as in [13] on selecting a splitting variable. We have witnessed that the selection of the splitting variable can be very expensive computationally and significantly increase the overall traversal time. Therefore, we make use of a new heuristics to select only a subset of the variables as candidates for

the splitting as opposed to [13] where all the variables are involved in the selection process. Although the new partitioning algorithm may generate less balanced partitions, it is faster than the original and reduces the overall traversal time.

Second, while [13] does a strict breadth-first search as in the classic reachability, we make use of a mixed breadth-first /depth-first traversal controlled by a prioritized queue of partitions. We check the correctness of the invariant properties *on-the-fly* during the reachability analysis. The mixed approach, in addition to enjoying the benefits reported by Cabodi et al. for balanced decomposition (i.e., low peak memory requirement and drastic CPU-time and capacity improvement), makes the search more robust in case of falsification, by getting faster to the bugs. On-the-fly verification [16] clearly reduces the time required to get to the bugs. Experiments on Intel designs show a marked improvement on the existing exact and partial reachability techniques.

The paper starts with a summary of related work in Section 2. Section 3 introduces the new prioritized traversal as well as the fast splitting algorithm. In Section 4 we report experimental results comparing prioritized traversal to existing traversal algorithms and the new splitting algorithm to the original one. We conclude in Section 5 by summarizing the contributions of this work.

2 Related Work

Semi-exhaustive verification addresses the concerns of practicing verifiers by shifting the focus from verification to *falsification*. Rather than ensuring the absence of bugs, it turns the verification tool into an effective bug hunter. This hybrid approach aims at improving over both simulation and formal verification in terms of state space coverage and capacity respectively.

Our usage of semi-exhaustive verification follows the lines of [1,3,5,9], being based on subsetting the frontiers during state space exploration, whenever these frontiers reach a given threshold by making use of various under-approximation techniques.

The effectiveness of the semi-exhaustive verification is clearly very sensitive to the nature of the algorithm employed for subsetting the frontiers. A large number of BDD subsetting algorithms have been proposed lately in the model checking literature. Each of them is necessarily a heuristic, attempting to optimize different criteria of the chosen subset. An important class of heuristics takes the *density* of the BDDs as the criterion to be optimized, where density is defined as the ratio of states represented by the BDD to the size of the BDD. This relates to the observation that large BDDs are needed for representing sparse sets of states (as it is the case for the frontiers). Removing the isolated states can lead to significant reductions in the size of the BDDs.

Ravi and Somenzi [1] have introduced the first algorithms for extracting dense BDD subsets, *Heavy-Branch* (HB) and *Short-Path* (SP). Independently, Shiple proposed in his thesis [8] the algorithm *Under-Approx* (UA) that also optimizes the subset according to the density criterion. Recently, Ravi et al. [9] proposed *Remap-Under-Approx* (RUA) as a combination of UA with more traditional BDD

minimization algorithms like *Constrain* and *Restrict* [10]. A combined algorithm is *Compress* (COM) [9], which applies first SP with the given threshold and then RUA with a threshold of 0 to increase the density of the result. Although more expensive, the combination of the two algorithms is supposed to produce better results.

The *Saturation* (SAT) algorithm [3,5] is based on a different idea. Rather than keeping as *many* states as possible, it attempts to preserve the *interesting* states. In the context of [3,5] the control states are defined as the interesting ones. The heuristic makes sure to *saturate* the subset with respect to the control states, i.e. that each possible assignment to the control variables is represented exactly once in the subset. In terms of BDDs, this is implemented by Lin and Newton's *Cproject* operator [12].

Previous studies [1,3,5,9] advocate the merits of dense frontier subsetting techniques on the basis of their coverage of the reachable state space and the density of the approximated frontiers. We have evaluated in [15] the effectiveness of these techniques for bug hunting and confirmed their usefulness in the fast detection of design or specification errors. However, a major shortcoming of these techniques is the difficulty, and in many cases the inability, to provide formal guarantees of correctness, in case no bug is found. In other words, these techniques, although useful for *falsification,* are not practical for *verification*. Furthermore, these techniques suffer from high tuning effort in case they are used to find the corner-case bugs.

Our approach is closely related to the work of Cabodi et al. [13], which considers as a key goal the good decomposition of state sets. They adopt a technique aimed at producing balanced partitions with a possibly minimum overall BDD size. This size is often slightly larger than the original one, but this drawback is largely overcome by the benefits derived from partitioned storage and computations. We have observed that the balanced partitioning approach is very effective in reducing peak memory requirements and it can drastically decrease the overall BDD size. From our experience, the benefits of balanced partitioning over dense subsetting is that it requires less tuning effort and it can much more easily and efficiently converge to exact reachability. On the other hand, the BDD splitting algorithm used in [13] may be computationally very expensive. For average designs the time spent in BDD splitting may outweigh the overall benefits of balanced partitioning. Cabodi et al. make use of a partitioned breadth-first search. For falsification we have observed the mixed breadth-first/depth-first approach to be much more effective. The enhanced reachability analysis that we propose enjoys the benefits of balanced partitioning and addresses its shortcomings.

Finally, the work of Narayan et al. [14] on partitioned ROBDDs takes a more radical approach where all the BDDs in the systems (not just the frontier and the reachable states) are subject to partitioning. Moreover, different partitions can be reordered with different variable orders. This approach can cope better with the BDD explosion problem, but it involves significant effort in maintaining the coherency of the infrastructure where several BDD variable orders coexist simultaneously.

3 Improved Reachability Search

A finite state machine is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 5-tuple $M = (I,S,\delta,\lambda, S_o)$, where I is the input alphabet, S is the state space, δ is the transition relation contained in S x I x S, and $S_o \subseteq S$ is the initial state set. BDDs are used to represent and manipulate functions and state sets, by means of their *characteristic functions*. In the rest of paper, we make no distinction between BDDs and set of states.

3.1 Invariant Verification

A common verification problem for hardware designs is to determine if every state reachable from a designated set of initial states lies within a specified set of "good states" (referred to as the *invariant*). This problem is variously known as *invariant verification*, or *assertion checking*. Invariant verification can be performed by computing all states reachable from the initial states and checking that they all lie in the invariant. This reduces the invariant verification problem to the one of traversing the state transition graph of the design, where the successors of a state are computed according to the transition relation of the model. Moreover, traversing the state graph in a breadth-first order makes possible to work on sets of states that are symbolically represented as BDDs [6]. This is an instance of the general technique of *symbolic model checking* [7].

Given an invariant *inv* and an initial set of states S_o, *reachability analysis* starts with the BDD for S_o, and uses BDD functions to iterate up to a fixed point, which is the set of all the states reachable from S_o using the *Img* operator. If the set of reachable states R is contained in *inv*, then the invariant is verified to be true, otherwise the invariant is not satisfied by the model and a counter-example needs to be generated.

The primary limitation of this classic traversal algorithm is that the BDDs encountered at each iteration F, commonly referred as *frontiers*, and R, referred as *reachable states*, can grow very large leading to a blow-up in memory or to a verification time-out. Moreover, it may be impossible to perform image computation because of the BDDs involved in the intermediate computations.

Subsetting traversal introduced by Ravi and Somenzi [1] and *Partitioned traversal* proposed by Cabodi et al. [13] (Figure 1) tackle these shortcomings of the classic traversal by decomposing state sets when they become too large to be represented as a monolithic BDD or when image computation is too expensive.

Subsetting traversal keeps the size of the frontiers under control by computing the image computation only on a dense subset of the frontier, each time the size of the current frontier reaches a given threshold. When no new states are produced, the sub-traversal may have reached the actual fixed-point (the one that would be obtained during pure BFS traversal) or a *dead-end*, which arises from having discarded some states during the process of subsetting. Theoretically, termination could be checked by computing the image of current reached set of states (as in Figure 1). In practice, this is rarely feasible, given the size of the BDD representing the reachable states. A dead-end resolution algorithm was proposed in [11] that keeps the *scrap* states (ignored by

previous subsettings) in a partitioned form to regenerate the traversal. However, we had little success with this algorithm, as the BDDs of the scrap partitions (the non-dense part of the state space) get larger and larger and more difficult to use.

SUBSETTING_TRAVERSAL (δ, S_o, th)	PARTITIONED_TRAVERSAL (δ, S_o, th)

$R = F = S_o$;
while ($F \neq \emptyset$) {
F = IMG(δ, F) – R;
$R = R \cup F$;
if ($F = \emptyset$) // *check if dead-end*
F = Img(δ, R) - R;
if (size(F) > th)
F = subset (F);
}
	$R_p = F_p = S_o$;
	while ($N_p \neq \emptyset$) {
	T_p = {IMG(δ, f) | $f \in F_p$};
	F_p = SET_DIFF (T_p, R_p);
	R_p = SET_UNION (F_p, R_p);
	F_p = RE_PARTITION(F_p, th);
	R_p = RE_PARTITION(R_p, th);
	}

Figure 1. Subsetting traversal [1] and Partitioned Traversal [13]

Partitioned traversal is a BFS one, just like the classic algorithm, except that it keeps the frontier F as a set of partitions F_p, and the reachable states R as a set of partitions R_p. The T_p sets are the results of image computation for each of the subfrontiers F_p, which may be either re-combined or partitioned again. This is done by functions like SET_DIFF, SET_UNION, and RE_PARTITION that work on partitioned sets. The intuition behind this approach is to overcome the complexity issue of large frontiers by splitting the frontiers to balanced partitions with minimum overall size and decrease peak BDD size by performing the image computation on the partitions of the frontier. One advantage of partitioned traversal over subsetting traversal is that it can more easily converge to exhaustive reachability. During the whole computation, all the partitions are preserved and image computation is performed on all the partitions. The computation never gets into dead-ends and consequently there is no need to perform image computation on the reached states. On the other hand, a major drawback of this approach is the time spent in selecting the BDD variable to split the frontier to balanced partitions. Moreover, if this traversal is used for falsification purposes, since the reachable states are computed still in a BFS fashion, it is time consuming to get to deep bugs.

Our approach is related to both [1] and [13]. As [13] it aims at generating balanced partitions instead of dense subsets, since we have experienced that balanced partitioning requires less tuning effort than dense subsetting and makes it easy to converge to full verification. Similarly to [1], we make use of a mixed breadth-first/depth-first search strategy in order to reach the deep bugs faster. Moreover, we propose a fast splitting algorithm that reduces the overall traversal time.

Consequently, our approach enjoys the benefits of both subsetting and partitioned traversal, while addressing the shortcomings of the two approaches: unsuitability of subsetting traversal for full verification, inefficiency of partitioned traversal for falsification and performance penalty induced by the splitting algorithm in partitioned traversal.

3.2 Fast Frontier Splitting

Cabodi et al. [13] proposed a BDD splitting algorithm based on single variable selection, that aims at producing balanced partitions with minimum overall BDD size. The algorithm is based on a procedure estimating the size of the cofactors with respect to a given variable. The cost of splitting a BDD with variable v is computed making use of the estimated node counts. The cost function calculates the potential of a variable v to generate balanced BDDs with minimum overall BDD size. The main drawback of this approach is that the estimated node counts may be quite inaccurate, resulting in unbalanced partitions. The estimations are computed without considering the reductions and sub-tree sharing. Therefore, in [18] an enhanced procedure was proposed. The enhanced algorithm takes into consideration the sharing factor while estimating the size of the cofactors with respect to a given variable. This refinement yields very precise estimates but is much slower than the original one.

Moreover, both algorithms estimate the size of the BDD representing f constrained either by v or $\sim v$ for each variable v in the true support of f. The cofactor size estimation for each variable in the support is computationally very expensive, and therefore, is a major drawback of the partitioned reachability analysis.

Our frontier splitting algorithm aims at achieving a good time/accuracy trade-off. We still want to use the accurate cofactor estimation of [18], but only on a subset of the variables. Therefore, we propose a two-stage algorithm[2]. The first stage prioritizes the variables according to their splitting quality. The second stage calculates accurate cofactor size estimations as in [18] for the subset of variables chosen to be the best candidates at the first stage. The performance improvement is mainly due to the accurate cofactor size estimation of only a subset of the variables. Additionally, the first stage of the algorithm, which prioritizes the variables with respect to their splitting quality is quite fast. This first stage is comparable to the function counting the nodes of a BDD, so its time complexity is linear in the size of the BDD. Therefore, we get a significant performance gain that outweighs the degradation in splitting quality, as testified by the results in Section 4.

In order to prioritize the variables with respect to their splitting quality, we estimate the size of every sub-function f in a BDD (i.e., the size of the sub-DAG under every BDD node), which we refer to as $c[f]$. Clearly, $c[f] \leq c[f_0] + c[f_1] + 1$, and the equality holds only when there is no sharing between f_0 and f_1. Using a DFS traversal of the BDD representing the function f, the exact $c[f]$ may be calculated as $c[f] = c'[f_0] + c'[f_1] + 1$, where $c'[f_0]$ and $c'[f_1]$ are the number of unvisited nodes encountered during the traversal of f_0 and f_1, respectively. If f_0 is traversed first, then $c'[f_0] = c[f_0]$ and $c'[f_1] \leq c[f_1]$, since there may be node sharing between f_0 and f_1. Similarly, $c'[f_1] = c[f_1]$ and $c'[f_0] \leq c[f_0]$ if f_1 is traversed first. Therefore, the value of $c'[f_0]$ is different if the traversal starts from F_0 or F_1. The maximum of the two values gives an accurate estimate for $c[f_0]$). The same holds for $c[f_1]$.

---

[2] We would like to thank one of the reviewers for pointing us to a more recent paper of Cabodi et al. [17] proposing similar improvements to the splitting algorithm. However, their estimation of cofactor sizes is based on computing different metrics during the DFS traversal of the BDD.

Based on these facts, we make two traversals on the BDD of *F,* where at the first traversal we expand the 0-edge of every node *f* of *F*, while at the second pass the 1-edge is expanded first. At each pass, *c*[*f*] is updated such that its final value is the maximum value resulting from the two traversals. Note that the resulting *c*[*f*] is only an estimate of the real size of *f*. To measure the splitting quality of each variable *v* in the support of *F*, we make use of the estimate sizes of sub-functions in *F*. We experimented with several cost functions and we have found the following one to be satisfying:

$$COST(v) = \frac{w \cdot \sum_{f:\,\mathrm{var}(f)=v} \frac{\mathrm{abs}(c[f_v] - c[f_{\bar{v}}])}{\max(c[f_v],\, c[f_{\bar{v}}])} + (1-w) \cdot \sum_{f:\,\mathrm{var}(f)=v} \frac{c[f_v] + c[f_{\bar{v}}] + 1}{c[f]}}{\mathrm{card}(\{f \mid \mathrm{var}(f) = v\})}$$

where var(*f*) is the root variable of the sub-BDD *f*, so the summations in the nominator are done over all the BDD nodes *f* that are at the level of the variable *v*. The denominator is simply the number of all the nodes at the level of *v*. The first term in the nominator represents the balance between the cofactors, and the second term is the node sharing. The weight *w* of the balancing term was empirically determined to be 0.4.

We then proceed to the second stage, where variables are prioritized in increasing order of their COST(*v*), and an accurate estimation of cofactors' sizes is calculated on the best *N* variables using the size estimation algorithm in [18]. The variable with the minimum larger cofactor, that is min(max(|f_v|,|$f_{\sim v}$|)), is chosen as the splitting variable *v*. We experienced that calculating an accurate estimate to a small set of variables (for instance, *N* = 15) is sufficient to get to a nearly optimal splitting.

3.3 On-the-fly Verification

We have incorporated on-the-fly verification of the invariants [16] to our reachability algorithms. Instead of checking the invariant only after all reachable states are computed, this check is performed after each computation of a new frontier. This feature is critical for falsification problems, where the different algorithms are evaluated with respect to their success in bug finding. This should be contrasted with previous work ([1,11,13]) where the evaluation is strictly based on the number of reached states.

3.4 Prioritized Traversal

Figure 2 shows the pseudo-code for the prioritized traversal. The set of states that satisfy the invariant is represented by *inv*. Prioritized traversal can be performed with a transition relation δ in monolithic or partitioned form. The results in Section 4 are obtained when a partitioned transition relation is used.

R represents the set of states reached so far, initially equal to the set of initial states S_o. *Fqueue* is the prioritized queue of the frontiers that have yet to be processed. Initially *Fqueue* contains just the set S_o. As long as the queue is not empty, we pop its first element *F*, compute its image and reinsert the result into *Fqueue* according to the

priority function. Whenever a new frontier F is inserted into the queue, if its size exceeds the threshold *th*, F is completely partitioned (i.e., decomposed until the size of all its sub-partitions are below *th*) making use of the splitting algorithm explained in Section 3.2. The new partitions are inserted in the queue in the order imposed by the *priority function*. We have experimented with several priority functions (*minimum_size*, *density*, *frontier_age*). Our experience, as can be observed from the results reported in Section 4, shows that *minimum_size* is the most efficient and the least sensitive priority function. At each fixed-point iteration step, the correctness of the invariant is checked. The traversal ends if the invariant is falsified during the reachability analysis.

```
PRIORITIZED_TRAVERSAL
(δ,S_o, th, prio_func)
R = S_o;
INVARIANT_CHECK(So, inv);
INSERT(So, Fqueue, th, prio_func);
while (Fqueue ≠ []) {
F = POP (Fqueue);
F = IMG(δ, F) - R;
R = R ∪ F;
   INVARIANT_CHECK(F, inv);
   INSERT(F, Fqueue, th,
 prio_func);
}
```

```
INSERT(F, Fqueue, th, prio_func)
   if (size(F) > th) {
      F_p = COMPLETE_PARTITION(F th);
      Foreach f ∈ F_p
        insert f in Fqueue using prio_func;
}
else if (F ≠ ∅)
   insert f in Fqueue using prio_func;

INVARIANT_CHECK(F, inv)
  if (F ⊄ inv) report the bug and exit;
```

Figure 2. Prioritized traversal

Prioritized traversal is a mixed BFS/DFS traversal, which can be converged to full DFS or BFS by inserting the new frontiers always to the top or the bottom of the queue. Therefore, as a search traversal, it subsumes the partitioned traversal introduced by Cabodi et al. [13]. Note that our approach, unlike [13], does not re-partition the reachable states and the frontiers at each step. We estimate the re-partitioning to be very expensive, and we avoid using a partitioned SET_DIFF (Figure 1) to detect the fixpoint.

4 Experimental results

The results reported in this section come to support our main claims about the advantages of the new algorithms proposed in this paper. Section 4.1 compares our fast splitting algorithm against the one in [18] by measuring the impact of the time/quality tradeoff on the overall performance. Section 4.2 compares prioritized traversal with classic, subsetting and partitioned traversal on a set of verification and falsification problems. While these problems can be handled by most of the algorithms, our data shows that prioritized traversal behaves consistently well all over the spectrum. It is the only algorithm that is robust for both verification and falsification. Finally, section 4.3 reports successful results of prioritized traversal on a

number of challenging testcases (both verification and falsification) that no other algorithm can handle. This emphasizes the capacity improvement of our new algorithm.

4.1 Comparing splitting algorithms

For the purpose of this experiment we have run prioritized traversal using two different BDD splitting algorithms: the one implemented in [18] denoted by SPLIT below, and our fast splitting algorithm denoted by SPLIT+. Running the two algorithms on four full verification tasks (requiring exact reachability) allowed us to compare the impact of the splitting algorithms on the overall traversal time. Table 1 describes the number or variables of each circuit (latches and inputs), the threshold that triggers frontier partitioning, as well as the time (seconds) and memory (Mb) for the complete traversal.

ckt	ckt1		ckt2		ckt3		ckt4	
vars	81 lat/101 inp		79 lat/80 inp		136 lat /73 inp		129 lat /82 inp	
thresh	50000 nodes		100000 nodes		100000 nodes		100000 nodes	
Trials	time	mem	time	mem	time	mem	time	mem
SPLIT	327	169	368	91	3880	219	3522	190
SPLIT+	244	169	309	98	2840	219	2359	185

Table 1. Impact of splitting algorithms on overall traversal time

While the *speed* of the splitting algorithm obviously affects the traversal time, so does the *accuracy* of the splitting. Indeed, a poor split requires additional splitting steps and increases the number of iterations of the traversal. In this respect, the speed of SPLIT+ compensates its occasional loss in accuracy, causing it to perform systematically better than SPLIT.

Even more convincing is to compare the two algorithms on a few specific partitioning problems. In Table 2 we pick three representative cases and report the sizes of the initial BDD and of the two partitions resulting from the split, as well as the time required by the split. SPLIT+ is consistently 8-10X faster than SPLIT, without a significant loss in accuracy, although occasionally it can produce a poor decomposition (as seen in the first line of Table 2).

	SPLIT		SPLIT+	
Initial BDD	Partitions (BDD nodes)	Time	Partitions (BDD nodes)	Time
331520	224731 / 215579	98.83	63834 / 268414	8.86
100096	52527 / 52435	31.13	54293 / 47042	4.86
105003	53321 / 53312	27.96	53321 / 53312	2.81

Table 2. Comparing splitting algorithm on specific partitioning problems

4.2 Robustness for both verification and falsification

In this section, we selected for the purpose of evaluation several real-life verification and falsification problems that can be handled by all the algorithms: classic reachability (CLS), subsetting traversal using the Shortest Paths heuristic (SUB), partitioned traversal (PART) and prioritized traversal (PRIO). The point that we want to make here is that PRIO behaves consistently well all over the spectrum. It is the only algorithm that is efficient and robust for both verification and falsification. As opposed to [13], our implementation of PART does not re-partition the frontiers and reachable states.

As for PRIO, the priority queue is sorted by BDD size (the smallest frontiers are traversed first). This setting was uniformly a good choice, but different priority functions perform better on different examples, which justifies the need for a general priority queue mechanism. Also, the results obtained with the different priority functions, and for each of them several thresholds have been consistently good, which supports our claim about the robustness of this approach. The results for other priority functions are not included here, due to space constraints.

ckt	ckt1		ckt2		ckt3		ckt4	
vars	81 lat/101 inp		79 lat/80 inp		136 lat /73 inp		129 lat /82 inp	
thresh	50000 nodes		100000 nodes		100000 nodes		100000 nodes	
	time	mem	time	mem	time	mem	time	mem
CLS	174	169	253	92	2923	265	2271	224
SUB	2763	245	568	108	Out	Out	Out	Out
PART	299	169	457	95	3844	219	2550	190
PRIO	244	169	309	98	2840	219	2359	185

Table 3. Performance comparison on verification problems

Let us look first at the verification results in Table 3 (ckt 1–4). We notice that CLS is still the fastest algorithm. However, PRIO comes close behind – for larger examples, it is only 10-20% slower and occasionally it beats the classic algorithm (as seen for ckt3). Also, PRIO has lower memory consumption and this trend gets more emphasized as we lower the threshold or run more challenging examples. This is similar to the time/memory trade-off observed in the usage of partitioned transition relations [20] compared to monolithic ones.

PART takes more time to complete these examples, we suspect that PRIO wins over PART due to its mixed BFS/DFS nature that allows to reach deeper states faster and converge in less iterations. As for SUB, its dead-end resolution algorithm rarely succeeds to converge to full reachability and when it does is much slower than both PRIO and PART. This is due to the unbalanced partitioning used in SUB – the dense partition is explored first, but when it comes to traversing the non-dense one we are dealing with increasingly larger BDDs and a rapid degradation in performance.

Table 4 reports falsification results on three buggy test cases (ckts 5–7). The time and memory data are measured only until the bug is encountered, due to the use of "on-the-fly" verification. As opposed to previous evaluations [1,13] of subsetting traversal that aimed at high state coverage, our evaluation criteria measures the efficiency and robustness of the different algorithms with respect to bug finding.

ckt	ckt5		ckt6		ckt7	
vars	195 lat/67 inp		129 lat/54 inp		136 lat /73 inp	
thresh	50000 nodes		50000 nodes		100000 nodes	
	time	mem	time	mem	time	mem
CLS	944	234	1307	344	4655	494
SUB	Out	Out	810	220	Out	Out
PART	1262	155	1012	227	5650	210
PRIO	470	128	540	157	1600	210

Table 4. Performance comparison on falsification problems

PRIO is clearly faster than both CLS and PART, again due to its mixed BFS/DFS nature. It is no surprise that PART is even slower than CLS, since both do a BFS traversal only that PART does it slower and with lower memory consumption (just as noticed in Table 3). When comparing performance, there is no a priori winner between PRIO and SUB, but PRIO is definitely more robust. When SUB's approximations miss the closest bugs, it is harder and harder to recover from dead-ends and to encounter the buggy states ignored previously. This is why SUB requires a high tuning effort – often only the combination of a specific subsetting heuristic and a specific threshold succeeds in finding the bug. By contrast, the usage of balanced partitioning in PRIO allows it to explore more states and eventually hit the bug. Different priority queues or different thresholds have a smaller impact on the chances of finding the bug, although they may affect the time required for it.

4.3 Capacity improvement

One of the main benefits of PRIO is the capacity improvement over CLS, SUB and PART. This can be noted in Table 5 for both verification (ckt 8-10) and falsification problems (ckt 11-12). Both SUB and PART were run with the best tuning (i.e. different thresholds, approximation heuristics) and SUB actually succeeded to handle the two falsification problems, but only with a very specific configuration: the RUA approximation [9] and a threshold of 400000 nodes.

	Verification						Falsification			
ckt	ckt8		ckt9		ckt10		ckt11		ckt12	
# vars	90 lat /104 inp		145 lat/55 inp		139 lat /41 inp		129 lat /82 inp		71 lat /58 inp	
thresh	50000 nodes		50000 nodes		20000 nodes		100000 nodes		50000 nodes	
	time	mem	time	time	time	mem	time	mem	time	mem
CLS	Out	Out	Out	Out	Out	Out	Out	Out	Out	Out
SUB	Out	Out	Out	Out	Out	Out	2236*	41	2763*	59
PART	Out	Out	Out	Out	Out	Out	Out	Out	Out	Out
PRIO	14768	851	54324	237	86522	228	512	48	739	63

Table 5. Capacity comparison on verification and falsification problems

These results only confirm the trend noticed in section 4.2. While CLS had a slight edge for average verification problems, for the difficult examples PRIO has better chances for success than both SUB and PART. All the arguments mentioned above still apply here: PRIO wins against SUB due to the balanced partitioning, and is better than PART due to its mixed BFS/DFS strategy.

5 Conclusions

The ability of the same algorithm to address both verification and falsification problems is critical in an industrial context. The prioritized traversal proposed here achieves this goal by combining the best features of subsetting traversal [1] and partitioned traversal [13]. As in [1], the mixed BFS/DFS strategy makes the algorithm efficient for falsification problems. As in [13], using balancing instead of density as the partitioning criterion makes partitioned traversal suitable for verification problems. The results reported on Intel designs show a marked improvement on existing exact and partial traversal techniques.

Another important contribution of this paper is the fast splitting algorithm. As it addresses the general BDD decomposition problem, we suspect that such an algorithm might be useful for many other applications relying on BDD technology. In the specific context of partitioned traversal the speed of the algorithm outweighs the loss in the quality of the partitions, resulting in a significant reduction of the overall traversal time.

Also it is worthwhile to note that the usage of prioritized traversal is not limited to reachability analysis and invariant checking. It can be easily adapted to other least-fixpoint computations, like the evaluation of CTL formulas of type EU or EF. Since the evaluation of such formulas involves a backward traversal of the state space, one only has to replace the image operator with the pre-image one in order to get the corresponding dual algorithm.

6 References

[1] K.Ravi, F. Somenzi, "High Density Reachability Analysis", in Proceedings of ICCAD'95

[2] M.Ganai, A Aziz, "Efficient Coverage Directed State Space Search", in Proceedings of IWLS'98

[3] J. Yuan, J.Shen, J.Abraham, and A.Aziz, "On Combining Formal and Informal Verification", in Proceedings of CAV'97

[4] C.Yang, D.Dill, "Validation with Guided Search of the State Space", In Proceedings of DAC'98

[5] A.Aziz, J.Kukula, T. Shiple, "Hybrid Verification Using Saturated Simulation", In Proceedings of DAC'98

[6] R.Bryant, "Graph-based Algorithms for Boolean Function Manipulations", IEEE Transactions on Computers,C-35:677-691, August 1986.

[7] K.L. McMillan. "Symbolic Model Checking", Kluwer 1993.

[8] T.R. Shiple "Formal Analysis of Synchronous Circuits". PhD thesis, University of California at Berkeley, 1996.

[9] K. Ravi, K.L. McMillan, T.R. Shiple, F. Somenzi "Approximation and Decomposition of Binary Decision Diagrams", in Proceedings of DAC'98.

[10] O. Coudert, J. Madre "A Unified Framework for the Formal Verification of Sequential Circuits", in Proceedings of ICCAD'90.

[11] K.Ravi, F. Somenzi, "Efficient Fixpoint Computation for Invariant Checking", In Proceedings of ICCD'99, pp. 467-474.

[12] B. Lin, R. Newton "Implicit Manipulation of Equivalence Classes Using Binary Decision Diagrams" in Proceedings of ICCD'91.

[13] G.Cabodi, P.Camurati, S.Quer. "Improved Reachability Analysis of Large Finite State Machines" in Proceedings of ICCAD'96.
[14] A.Narayan, J.Jain, M.Fujita, A.Sangiovanni-Vincentelli. "Partitioned ROBDDs – A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions" in Proceedings of ICCAD'96.
[15] R.Fraer, G.Kamhi, L.Fix, M.Vardi. "Evaluating Semi-Exhaustive Verification Techniques for Bug Hunting" in Proceedings of SMC'99.
[16] I.Beer, S. Ben-David, A.Landver. "On-the-Fly Model Checking" of RCTL Formulas", in Proceedings of CAV'98.
[17] G. Cabodi, P. Camurati, S. Quer, "Improving the Efficiency of BDD-Based Operators by Means of Partitioning," IEEE Transactions on CAD, pp. 545-556, May 1999.
[18] F.Somenzi, " CUDD : CU Decision Diagram Package – Release 2.3.0", Technical Report, Dept. Electrical and Computer Engineering, University of Colorado, Boulder
[19] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds and N. J. A. Sloane, "Efficient Regression Verification", Int'l Workshop on Discrete Event Systems (WODES '96), 19-21 August , Edinburgh, IEE, London, 1996, pp. 147-150.
[20] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification", IEEE Transactions on Computer-Aided Designs of Integrated Circuits and Systems, 401-424 Vol.13, No. 4, April 1994.