Improved Reachability Analysis of Large Finite State Machines

Gianpiero Cabodi [†] Paolo Camurati [‡] Stefano Quer [†]

[†] Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

[‡] Università di Udine
Dip. di Matematica e Informatica
Udine, ITALY

Abstract

BDD-based symbolic traversals are the state-of-the-art technique for reachability analysis of Finite State Machines. They are currently limited to medium-small circuits for two reasons: peak BDD size during image computation and BDD explosion for representing state sets. Starting from these limits, this paper presents an optimized traversal technique particularly oriented to the exact exploration of the state space of large machines. This is possible thanks to: 1) temporary simplification of a Finite State Machine by removing some of its state elements, 2) a "divide–and–conquer" approach based on state set decomposition. An effective use of secondary memory allows us to store relevant portions of BDDs and to regularize access to memory, resulting in less page faults. Experimental results show that this approach is particularly effective on the larger ISCAS'89 and ISCAS'89–addendum'93 circuits.

1 Introduction

Finite State Machines (FSMs) are a popular model for *control-dominated* ASICs. FSMs are identified by their input/output alphabets, initial state sets, and next state and output functions.

A *forward traversal* of a FSM identifies its *reachable state set*. Intuitively, a state is reachable if a sequence of inputs forces the FSM to evolve from an initial state to that state. The next state function determines the evolution along time. The next states are the image, for all inputs, of the current states according to the next state function. The process terminates as soon as a fixed-point is reached, i.e., no newly reached states are found.

Symbolic approaches compute images by implicitly enumerating the inputs [1], [2], [3]. Although quite successful, symbolic methods cannot complete the reachability analysis of large FSMs, because they require too much memory and are computationally expensive. Focussing on memory requirements, the two major problems are peak BDD size during image computation and size of the BDDs representing reached states. Virtual memory could be a solution to such a problem but, if the working set size for a program is large and memory accesses are random, an extremely large number of page faults significantly modifies the performance of the software.

We improve standard techniques by:

- temporarily simplifying FSMs by latch removal and reinsertion at the end of the traversal process
- decomposing sets of states and carrying out costly traversal steps in a decomposed form.

Latches not feeding the next state functions may be extracted from the original FSM. A simplified FSM is obtained and a traversal can be carried out on that machine reducing the size of state sets during intermediate traversal iterations. The extracted latches are eventually reinserted to compute the full reachable state set. When applicable, this temporary simplification is particularly effective if intermediate steps are more expensive that the final ones, and this is often the case as shown by Ravi *et. al.* [4].

Moreover, when the monolithic BDDs of state sets become too large or when image computations are too expensive, we decompose the sets in subsets. We perform fixed point computations and other operations on the decomposed form, and this allows us to deal with just one subset at a time. The resulting technique is no more a pure breadth–first traversal, but it is partially depth–first, reducing both the peak number of BDD nodes required and the complexity of the operations.

Good decomposition of state sets is a key issue. We 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 deriving from partitioned storage and computations. Partitioning is based on selecting a splitting variable, it is possibly recursive, and it produces non overlapping subsets. Conversely, BDD operators (e.g. image computation) applied to partitioned sets may produce overlapping results, that are either combined (through logical union) or partitioned again directly in the decomposed form.

Despite the possibility of repeated computations, decomposition lowers peak memory requirements and often also decreases overall complexity, with remarkable benefits in terms of CPU time.

The decomposed approach has also some other advantages over the standard one:

- it exploits secondary memory and compression techniques to download large BDDs for non active partitions

ICCAD '96
1063-6757/96 $5.00

- it allows different dynamic ordering strategies to be applied to each decomposition.

Coudert and Madre [3] resort to a full domain or co-domain partitioning for image computation. They apply these techniques to the next state function and they recursively decompose the problem until a terminal case is found.
Ravi *et. al.* [4] use a mixed breadth–first/depth–first traversal focused on *dense* BDDs. The objective of the method is to take large *subsets* of the state sets representable with a small number of BDD nodes. This technique is quite effective in finding large portions of the reachable state set, but it overestimates the distance of a state from the reset states and this often causes too many traversal iterations, aborted to obtain an under-estimation of reachable states.
Our approach partitions BDDs until their size is lower than a threshold, not until a terminal case is reached. The method is *exact*, as it computes the set of reachable states, not an over- or under-estimation. Experimental results show that it is particularly effective on the larger ISCAS'89 and ISCAS'89–addendum'93 circuits.
The remainder of the paper is organized as follows. Section 2 summarizes some useful concepts and describes enhanced symbolic traversal based on latch removal and set decomposition. Section 3 describes the decomposition procedures. Section 4 describes the overall traversal strategy. Section 5 reports and discusses experimental results. Section 6 closes the paper with a brief summary.

2 Improving Symbolic Traversal

A *finite state machine* is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 6-tuple $M = (I, O, S, \delta, \lambda, S_0)$, where I is the input alphabet, O is the output alphabet, S is the state space, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function, and $S_0 \subseteq S$ is the initial state set.
BDDs are used to represent and manipulate functions and state sets, by means of their *characteristic functions*. With abuse of notation, in the rest of this paper we make no distinction between the BDD representing a set of states, the characteristic function of the set, and the set itself.

2.1 Standard Symbolic Traversal

A standard *symbolic traversal* (figure 1) is a breadth-first search that returns at each iteration the set of states T reached from the current one F. Initially F equals S_0. T is computed by means of a symbolic image computation IMG(δ, F)[1]. Reached states are accumulated in R. Set N contains the T states that have not yet been visited. The set F for the next iteration is selected choosing a suitable BDD that represents all newly reached states and possibly some of the already visited ones, as in [2] (procedure BEST_BDD). The termination condition is to find a least fixed-point or equivalently to test for the emptiness of N at each step. The number of iterations of this algorithm gives the *sequential depth* of the machine.

```
TRAVERSAL (δ, S0)
{
R = F = N = S0;
while (N ≠ ∅)
    {
    T = IMG (δ, F);
    N = T · R̄;
    R = R + N;
    F = BEST_BDD (N, R);
    }
return (R);
}
```

Figure 1: Exact Forward Traversal.

Moving from one step to the next one in symbolic traversal, the BDDs of all the sets, in particular F and R, become larger and very complex. Symbolic traversals experience two bottlenecks: 1) a monolithic BDD representing a set may be too large, 2) it may be impossible to perform an image computation because of the size of the BDDs involved in intermediate computations.
Our approach faces the above problems by:

- temporarily simplifying FSMs by latch removal and reinsertion at the end of the traversal process
- decomposing state sets when, during traversal, they become too large to be represented as a monolithic BDD or when image computation is too expensive.

2.2 Simplification Based on Temporary λ–Latch Removal

Detecting latches not feeding the next state functions may reduce the size of state sets at intermediate traversal levels. We call these latches *λ–latches.*
We partition latches in two subsets A and B, such that the s_B variables are not in the true support[2] of the δ functions:

$$s_A = \{s_i \mid s_i \in supp(\delta)\}$$
$$s_B = s - s_A$$

Conversely, also the δ functions are partitioned in the corresponding δ_A and δ_B subsets. We restrict traversal to the sub-space of S described by s_A variables, using only δ_A, and we compute $R_A(s_A)$.
The full reached state set is eventually computed as:

$$R = \text{IMG}(\delta, R_A)$$

The advantage of this delayed use of the δ_B functions is that the BDD of R at the fixed point is normally smaller than the state sets at intermediate traversal levels [4]. Thus we globally simplify the traversal procedure by temporary removing some latches, and we re-introduce them

[1] Let $f : \mathcal{B}^i \rightarrow \mathcal{B}^j$ be a Boolean function and $C \subseteq \mathcal{B}^i$ a subset of its domain. The *image* of C according to f is:

$$\text{Img}(f, C) = \{y \in \mathcal{B}^j \mid \exists x \in C \ \wedge \ y = f(x)\}$$

[2] Given a vector of Boolean variables $x = (x_1, x_2, \ldots, x_n)$ and a Boolean function $f(x)$, the *"true support"* of f is the set of variables on which the function depends, i.e. the positive and negative cofactors of f with respect to x_i differ: $supp(f) = \{x_i | f_{x_i} \neq f_{\bar{x}_i}\}$.

only at the last image computation step, when the reached state set has possibly a simpler BDD representation.
Latch removal can be accomplished by recursively removing the s_A variables that are not in the true support of δ_A (but are in the true support of δ_B) and so on. This procedure requires two or more final steps of image computation.
A limit of the latch removal/reinsertion technique appears in symbolic traversals used for verification, when two machines are checked for equivalence and a counterexample is possibly needed. Equivalence is normally checked during traversal, the latter is stopped if the check fails, and a counterexample is found by means of pre–image steps across previously computed reached states sets. With our technique, a complete knowledge of the reached state set is only available at the fixed point, equivalence check must thus be delayed to the end of traversal, and we cannot guarantee minimum length counterexamples.

2.3 BDD Operations on Partitioned State Sets

Our purpose is to simplify a costly operation between two BDDs a and b, a op b, resorting to a "*divide–and-conquer*" strategy. The operation is performed as $(a_1 + a_2)$ op $b = (a_1$ op $b) + (a_2$ op $b)$ where a_1 and a_2 can be recursively partitioned. Decomposition is done by selecting a splitting variable v, that induces a partitioning also on the b operand: a op $b = (a_v$ op $b_v) + (a_{\overline{v}}$ op $b_{\overline{v}})$.
Recursive splitting is a very common practice with BDDs, as it characterizes almost all BDD operators, but it normally follows a fixed variable selection scheme: variable ordering. Our method is independent from variable ordering, and it possibly chooses different splitting variables when recurring in different set partitions.
The advantage of applying op on decomposed sets stems from lowering overall complexity in terms of memory and execution time. This is not achieved with elementary BDD operations like Apply, ITE, Cofactor, because they work in single depth–first traversal of BDDs, and complexity is close to the size of the BDD of the result. With such operations, partitioning the operands by means of splitting variables is equivalent to pushing them onto the top of variable ordering. Other evaluations are implemented as chains of elementary operators, and the cost of intermediate steps may be significantly larger than the size of the result. This is the kind of operators we optimize. In particular, we addressed to image computations and/or inner conjunction–abstraction steps of image computations themselves (see section 4). Partitioning is a good solution in both cases, because applying all the steps in sequence on each input partition can drastically reduce the cost of intermediate steps.
A further improvement is attained by downloading partitions to mass storage when not directly involved in computations.

3 Set Decomposition

Good decomposition of state sets is a key issue. As BDD operations have a complexity depending on the size of the operands, our target is to obtain balanced partitions with minimum overall BDD size.
We experimented with Ravi's *et al.* strategies [4] but as their purpose is to obtain *dense*, not balanced BDDs, we didn't obtain good results. Instead, we use a decomposition based on single splitting variable selection. The pseudo code of the partitioning procedure is shown in figure 2. Given a BDD f, representing a set of states, partitioning is attempted if its size exceeds a threshold: we heuristically choose a splitting variable v, with procedure SELECT_VAR, then we partition f in two subsets f_v and $f_{\overline{v}}$, and we possibly recur.
Single variable selection is simpler than multiple variable or cube selection and it allows partitioning a set in non overlapping subsets, by taking the right and left cofactors of the selected variable. In general the splitting variable

```
SPLIT (f, th)
{
if (|f| < th)
    return (f);
v = SELECT_VAR (f);
return (v · SPLIT (f_v, th), v̄ · SPLIT (f_v̄, th));
}
```

Figure 2: Splitting a BDD.

returned by SELECT_VAR (figure 3) is not the top variable. It is chosen depending on heuristic estimations done on the BDD of the input function f.

```
SELECT_VAR (f)
{
min = +∞;
foreach v ∈ supp(f)
    {
    n_v = COUNT (f, v);
    n_v̄ = COUNT (f, v̄);
    if (COST (|f|, n_v, n_v̄, v) < min)
        {
        v_min = v;
        min = COST (|f|, n_v, n_v̄, v);
        }
    }
return (v_min);
}
```

Figure 3: Selecting the best splitting variable.

For each variable v in the true support of f, function COUNT is called to estimate the size of the f function constrained either by v or by $\overline{v}$. This is done directly on the BDD of f without computing the f_v and $f_{\overline{v}}$ cofactors (figure 4): the BDD of f is analyzed by restricting the visit to the proper state subspace (represented by the c constraining parameter). Estimating the node count has a relevant impact on time performance, because explicit computation of f_v and $f_{\overline{v}}$ for all v implies generating many unused BDD

nodes, with several activations of the garbage collection procedure. The returned value is an overestimation of the correct value, because it ignores possible BDD reductions in f_v and $f_{\overline{v}}$. Reductions occur at BDD nodes above v in variable ordering. Due to this, the overestimation tends to grow as the position of v increases in variable ordering.
Given the estimated node counts n_v and $n_{\overline{v}}$, we compute the cost of splitting with v (function COST): a split should produce balanced BDDs with the smallest global number of BDD nodes.
We experimented with several cost functions, and a good compromise was obtained with:

$$\begin{aligned} \text{COST}(|f|, n_v, n_{\overline{v}}, v) &= w_1 \frac{n_v + n_{\overline{v}} - |f|}{|f|} + \\ & \quad w_2 \frac{abs(n_v - n_{\overline{v}})}{|f|} + w_3 \alpha(v) \end{aligned}$$

where the three terms represent, respectively, the overall

```
COUNT (f, c)
{
if (ISCONSTANT (f) or VISITED (f))
    return (0);
v = TOPVAR (f);
if (v = TOPVAR (c))
    return (1 + COUNT (f_c, c));
else
    return (1 + COUNT (f_v, c) + COUNT (f_v̄, c));
}
```

Figure 4: Counting the Number of Nodes of the Cofactors.

expected node count, the size imbalance, and a correction term taking into account the error introduced in COUNT. The overall node count is computed without considering subtree sharing between f_v and $f_{\overline{v}}$, because operations will be applied to single partitions, and because each partition is possibly downloaded to a distinct file, without sharing nodes with other partitions. The correction term $\alpha(v)$ is presently limited to a linear function of the variable ordering, returning 1 for the top variable and 0 for the bottom one: this heuristically compensates the overestimation done by COUNT. We plan to extend α to include a measure of the importance of the variable during image computation (e.g. if and when it is existentially quantified), and also a sort of *history*, granting lower α values to variables used with profit in previous partitioning processes. w_1, w_2, and w_3 are generic weights: good values for them have been determined experimentally as $w_1 = 1$, $w_2 = 0.8$ and $w_3 = 0.3$.

4 Partitioned Symbolic Traversal

The basic idea of the partitioned traversal is to perform each step using a "*divide–and-conquer*" approach. Every time complexity exceeds a threshold, we split the problem in sub-problems whose complexity is smaller. This is done at two levels:

- image computations
- conjunction–abstraction steps within image computations.

In the first case, if we decompose the current state set $C(s)$ as $C_v(s) + C_{\overline{v}}(s)$, its image according to δ is equivalent to the union of the images of $C_v(s)$ and $C_{\overline{v}}(s)$:

$$\begin{aligned} \text{IMG}(\delta, C(s)) &= \text{IMG}(\delta, (C_v(s) + C_{\overline{v}}(s))) \\ &= \text{IMG}(\delta_v, C_v(s)) + \text{IMG}(\delta_{\overline{v}}, C_{\overline{v}}(s)) \end{aligned}$$

In the second case the approach is similar. We compute images resorting to partitioned transition relations. Let M be a FSM. The *transition relation* T_M associated to M is:

$$T_M(x, s, y) = \prod_{i=1}^{n} (y_i \equiv \delta_i(x, s)) = \prod_{i=1}^{n} t_i(x, s, y_i)$$

A transition relation is often called *partitioned* if conjunctions are not performed once and forall, but t_is are kept separate until image computations. The *image* of a set $C(s)$ is defined as:

$$\begin{aligned} \text{IMG}(\delta, C(s)) &= \exists_{x,s} \, (T_M(x, s, y) \cdot C(s)) \\ &= \exists_{x,s} \, (\textstyle\prod_{i=1}^{n} t_i(x, s, y_i) \cdot C(s)) \end{aligned}$$

We compute images with a partitioned transition relation by resorting to early quantification during conjunction steps. Suppose that some input and current state variables appear just in the first i partitions. Let E_i be sets of such variables. Early quantification eliminates variables belonging to the E_i sets before conjoining the t_{i+1} term:

$$\begin{aligned} \text{IMG}(\delta, C(s)) &= \exists_{x,s} (\textstyle\prod_{i=1}^{n} t_i(x, s, y) \cdot C(s)) \\ &= \exists_{(x,s) \in E_n} (t_n \cdot (\exists_{(x,s) \in E_{n-1}} \, (t_{n-1} \cdot \ldots \cdot \\ &\qquad \exists_{(x,s) \in E_1} (t_1 \cdot C(s))))) \end{aligned}$$

The atomic operation in image computation is conjunction-quantification. The decomposition technique can be applied to the i–th conjunction step $\exists_{(x,s) \in E_i} (t_i \cdot \ldots)$ as introduced before.
In general, the first method is better when the overall image computation is far too expensive whereas the second one optimizes the previous one when only a few steps of the image computation are particularly expensive. In both cases we can recompose the resulting set after a decomposed operation, or we can carry on operations in the partitioned form.

4.1 Traversal Procedure with Decomposed Image

Figure 5 shows the pseudo-code for the decomposed traversal. It is derived from the standard traversal of figure 1.
R_p, F_p, and N_p represent set R, F, and N in monolithic or partitioned form. They are initially set to S_0. At each step, for each subset of F_p the image computation procedure is called. Images are collected in T_p. This allows the image computation procedure to work on just a subset at a time, decreasing peak BDD size. Internally the image function can decompose sets as previously introduced but this is not shown in the pseudo-code.
After image computation, functions SET_DIFF, SET_UNION, and RE_PARTITION are called. These functions are relatively simple and perform the computation of sets N_p, F_p, and R_p for the next iteration. In particular, function RE_PARTITION carries out set union and set splitting, according to the size of their BDD representation and to

```
PARTITIONED_TRAVERSAL (δ, S_0, th)
{
R_p = F_p = N_p = S_0;
while (N_p ≠ ∅)
    {
    T_p = ∅;
    foreach f ∈ F_p
        T_p = (T_p, IMG (δ, f));
    N_p = F_p = SET_DIFF (T_p, R_p);
    R_p = SET_UNION (N_p, R_p);
    F_p = RE_PARTITION (F_p, th);
    R_p = RE_PARTITION (R_p, th);
    }
return (R_p);
}
```

Figure 5: Exact Partitioned Forward Traversal.

parameter *th*. It calls directly procedure Split (see section 3) when necessary. The *th* parameter controls the complexity of the image computation procedure and the size and number of the state set partitions.

5 Experimental Results

Symbolic techniques were until now restricted to medium–small circuits. Our main novelty is to propose a technique that can deal with some of the larger ISCAS'89 and ISCAS'89–addendum'93 circuits.

We included the improved techniques described in the previous sections in a traversal program written in C on top of the Colorado University Decision Diagram (CUDD) package [5]. The overall tool includes all recently published dynamic reordering techniques and a traversal method based on a partitioned and clustered transition relations with sorting heuristics and other optimizations derived from [1], [3], [7], [8], [9]. We use this tool both without and with the improved method based on simplification of λ–latches and partitioning: in the former case we consider our tool as "state-of-the-art", and we use it to measure the improvement attained with our novel techniques.

An important role in saving main memory is played by storing (and loading) BDDs on secondary memory. By doing this we decrease the number of BDD nodes we deal with to a minimum required by the active partitions. As a consequence we decrease the amount of working memory required and we avoid page faulting problems. BDDs are downloaded by resorting to compression techniques, and they require an average memory occupancy of 6 bytes per node.

Our experiments ran on a 200 MHz DEC Alpha with a 256 Mbyte main memory, by imposing a working memory limit of 228 Mbyte. In all the following tables Circuit indicates the name of the circuit, # FF the number of flip-flops, # Level indicates the number of traversal iterations (partial or total), # Nodes is the number of BDD nodes and # States is the number of states of the reachable state set. # λ–FF indicates the number of λ–latches removed and # Part. indicates the maximum number of partitions used. Disk indicates maximum mass memory (in Mbyte) used to download BDDs. Due to the compression technique this amount is about 1/4 of the space the same BDDs occupy in main memory. Mem. is the maximum quantity of main memory used (in Mbyte). Time indicates the total execution time (in seconds, unless otherwise stated).

Table 1 collects data for circuits that we traverse up to the fixed point. To the best of our knowledge, these are the first results on exact forward traversal for all these circuits. Table 2 shows data for s1423, s5378 and some others ISCAS'89–addendum'93 we couldn't traverse completely. Complete traversal of s3271 and s3330 cannot be achieved without our improved approach. In the case of circuits s1423, s5378 and s6669, although not able to complete traversal, we increase the number of traversal iterations. In almost all other circuits we improve memory and time performance.

Circuit s1269 is traversed with the standard approach. With the improved one we decrease memory usage from 74 to 28 Mbyte, with a minimum time overhead. The decomposition procedure works quite well in this case as the BDDs are easily partitioned in a balanced way. For example we are able to decompose the reached set at level 3, consisting of 78940 nodes, in 7 subsets, each having less that 10000 nodes. Levels 3 and 4 are the dominant ones for traversal, while after level 4 the size of the reachable state set decreases by two orders of magnitude and all other levels are very simple to compute. Only one λ–latch removal is possible: being the advantage attained negligible we do not present data for this technique.

Circuit s1512 is sequentially very deep but single image computations are not particularly expensive; partitioning is worthwhile from level 300 to 600: using up to 4 partitions we reduce memory usage from 96 to 41 Mbyte, and execution time from 64^h to 42^h.

Traversal of circuit s3271 is completed at level 17 by the improved approach, while the standard one stops at level 9. A comparison for the first 9 traversal iterations shows a memory reduction from 214 to 41 Mbyte, with a 10% gain in time. Maximum memory usage in the other traversal iterations (partitioned approach) is 149 Mbyte, plus 11.8 Mbyte of disk storage. The most critical levels are 11 and 12, requiring approximately 50 hours of CPU time.

In the case of circuit s3330, the improved approach obtains relevant gains over the standard one. Apart from completing traversal, this circuit is characterized by a dominant role of the λ–latch removal/re-insertion technique. We could remove 12 λ–latches and the improved approach needs only 24 Mbyte instead of 113 to reach level 4 and it is able to traverse completely the circuit with only 107 Mbyte. Table 3 shows the size (BDD nodes) of the reached state set and the execution time in the first 5 traversal iterations. As far as comparison is possible, BDD size and time are reduced by an order of magnitude or even more.

Circuit s4863 is not really critical: we obtain some advantages in terms of memory, counterbalanced by an overhead in execution time. This is essentially due to the fact that state sets are not suitable for balanced partitioning.

Circuit $s5378_{opt}$ is an optimized version of the original s5378 ISCAS'89 benchmark, which has been obtained by iterative application of the redundancy removal procedure of [6], based on the approximate traversal techniques. The resulting circuit is sensibly smaller and contains 58 less latches than the original one. Ravi *et al.* [4] could traverse the circuit up to level 12. We could traverse it without optimizations but our λ–latch removal/re-insertion strategy decreases both peak BDD size and CPU time.

S1423 is the first circuit we describe with partial traversal data. Despite its relatively small size, it is very difficult to handle during reachability analysis, and this is proved by the extensive experiments described in [4] and [6]. Ravi *et al.* [4], using the state transition function, which they deemed to be more powerful than the transition relation, couldn't go beyond the 11^{th} level with a pure breadth-first traversal. To the best of our knowledge, nobody has already presented data beyond that limit. Our standard approach is able to deal with the circuit up to level 12. Afterwards, we have to resort to the decomposed approach. Advantages are obtained also before level 12, both in terms of CPU time (from 5410 to 2730 seconds) and memory required, decreased from 116 Mbyte (BDD peak size 2514569 nodes) to 32 Mbyte (BDD peak size 563251 nodes).

Table 4 shows the impact of different partitioning thresholds on this circuit from level 12 to level 13, using secondary memory. Column # Threshold indicates the number of nodes above which we partition sets. Peak Size is the size of the biggest BDD we produce during image computations. This table shows that a higher number of partitions may improve memory and time performance. For s1423 our decomposition technique works quite well, as, despite the relatively small number of state variables (74), the BDD sizes are enormous. We reach level 14; at that level the monolithic representation of the reachable state set is impossible to obtain.

Circuit s5378 (not optimized by means of redundancy removal) is even harder. The standard approach reaches level 3. The BDD of the reachable state set in this case is not large, but difficult to decompose in balanced subsets, because of the large number of state variables (179). In this case we can remove 17 λ–latches; this reduces the size of the reachable state set from 276102 nodes to 217509 at level 4. Due to the incomplete traversal, without reaching the fixed point, we present data without λ–latch removal for a correct evaluation of the reached state set. With our current implementation of decomposed traversal we reach level 4.

For circuit s6669 we are able to reach level 3 only with λ–latches removed.

6 Conclusions

Symbolic FSM state space exploration techniques represent one of the major recent results of formal verification. Their limit resides in the inability to deal with large circuits. In this paper we propose a simplified approach based on temporary latch removal and a decompose forward traversal. The overall approach is exact and decreases peak BDD size during image computation and the representation limits of the reachable state sets. Experimental results show that our approach works well with uniform and large sets. We therefore succeeded in handling circuits that were, up to now, beyond the scope of symbolic techniques.

Acknowledgments

We wish to thank Fabio Somenzi, CU Boulder, for providing us with the CUDD package. We also wish to thank Enrico Macii and Massimo Poncino, Politecnico di Torino, for providing us with the $s5378_{opt}$ benchmark.

References

[1] H. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli, "Implicit enumeration of finite state machines using BDDs," in *Proc. IEEE ICCAD'90*, November 1990, pp. 130–133

[2] H. Cho, G. Hachtel, S.W. Jeong, B. Plessier, E. Schwarz, F. Somenzi, "ATPG Aspects of FSM Verification," in *Proc. IEEE ICCAD'90*, November 1990, pp. 134–137

[3] O. Coudert, J.C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits" in *Proc. IEEE ICCAD'90*, November 1990, pp. 126–129

[4] K. Ravi, F. Somenzi, "High-Density Reachability Analysis," in *Proc. IEEE ICCAD'95*, November 1995, pp. 154–158

[5] F. Somenzi, "CUDD: CU Decision Diagram Package – Release 1.0.4," Technical Report, Dept. of Electrical and Computer Engineering, University of Colorado, Boulder, November 1995

[6] H. Cho, G.D. Hatchel, E. Macii, M. Poncino, K. Ravi, F. Somenzi, "Approximate Finite State Machine Traversal: Extensions and New Results," IWLS'95: IEEE International Workshop on Logic Synthesis, Lake Tahoe, CA, USA, May 1995

[7] D. Geist, I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," in *Proc. CAV'94*, June 1994, pp. 299-310

[8] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier, C. Pixley, "Efficient BDD Algorithms for FSM Synthesis and Verification," IWLS'95: IEEE International Workshop on Logic Synthesis, Lake Tahoe, CA, USA, May 1995, poster presentation

[9] G. Cabodi, P. Camurati, S. Quer, L. Lavagno, E. Macii, M. Poncino, E.M. Sentovich, "Enhancing FSM Traversal by Temporary Re–Encoding," in *Proc. IEEE ICCD'96*, October 1996

Circuit	# FF	# Level	Reached		Standard Approach		Improved Approach				
			# Nodes	# States	Mem.	Time	# λ–FF	# Part.	Disk	Mem.	Time
s1269	37	10	612	$1.1313{\cdot}10^{9}$	74	1322	0	7	0	28	1424
s1512	57	1024	1100	$1.6574{\cdot}10^{12}$	96	64^{h}	0	4	0	41	42^{h}
s3271	116	9	1319332	$4.9609{\cdot}10^{26}$	214	16.4^{h}	0	4	2.8	41	15.1^{h}
		17	383521	$1.3177{\cdot}10^{31}$	ovf	–	0	9	11.8	149	137^{h}
s3330	132	4	998886	$1.2915{\cdot}10^{16}$	113	1023	12	4	6.3	24	27
		8	28748	$7.2778{\cdot}10^{17}$	ovf	–	12	4	9.7	107	4155
s4863	104	5	17557	$2.1904{\cdot}10^{19}$	32.6	720	0	2	0	27.4	825
$s5378_{opt}$	121	43	24651	$2.5806{\cdot}10^{17}$	17	963	38	1	0	14.8	612

Table 1: Comparison on Traversal Results between Standard and Decomposed techniques on some ISCAS'89 circuits. ovf means overflow of BDDs nodes; – means data not available.

Circuit	# FF	# Level	Reached		Standard Approach		Improved Approach				
			# Nodes	# States	Mem.	Time	# λ–FF	# Part.	Disk	Mem.	Time
s1423	74	12	1361263	$2.3035{\cdot}10^{10}$	116	5410	0	16	22.6	32	2730
		14	13738871	$1.7945{\cdot}10^{11}$	ovf	–	0	24	125.9	106	8.4^{h}
s5378	179	3	20274	$1.7285{\cdot}10^{12}$	45	227	0	3	0	45	227
		4	276102	$2.6303{\cdot}10^{15}$	ovf	–	0	5	5.3	78	3.5^{h}
s6669	239	2	14593	$2.8657{\cdot}10^{45}$	13	61	35	2	0	12.3	48
		3	2494135*	*	ovf	–	35	9	22.7	97	530

Table 2: Comparison on Partial Traversal Results between Standard and Decomposed techniques on some ISCAS'89 circuits. ovf means overflow of BDDs nodes; – means data not available; * indicates that we use the partial reachable state set without re-inserting λ–latches.

# Level	Standard Approach		Improved Approach	
	Reached # Nodes	Time	Reached # Nodes	Time
1	7804	3.6	1285	2.5
2	9415	6.7	1411	3.7
3	358359	62.6	17432	7.6
4	998886	1023	102929	38.2
5	ovf	–	190958	2648.5

Table 3: Experiments on circuit s3330 from level 1 to level 5. Standard approach compared to the improved one with 12 latches removed and no partitioning. ovf means overflow of BDDs nodes; – means data not available.

# Threshold	# Part.	Peak Size	Mem.	Time
125000	17	500894	97.5	6388
150000	14	689007	110	6700
175000	11	784992	134	7589
200000	8	918679	142	7920

Table 4: Experiments with different partition thresholds on circuit s1423 from level 12 to level 13.