Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversals

Gianpiero Cabodi Paolo Camurati Stefano Quer
Politecnico di Torino
Turin, Italy

Abstract

Symbolic state space exploration techniques for Finite State Machines (FSMs) are a major recent result in CAD for VLSI. Most of them are exact and based on forward traversal, but limited to medium-size circuits. Approximate forward traversal deals with bigger circuits at the expense of exactness. Backward traversal takes into account many irrelevant, unreachable states. This paper combines the advantages of approximate forward traversal and of exact backward traversal. Cofactoring plays a key role in efficient function simplification. For the first time, we are able to symbolically manipulate some of the larger ISCAS'89 and MCNC circuits in an exact way and to generate test patterns for them. [1] [2]

1 Introduction

A popular model for "*control-dominated*" ASICs is the *Finite State Machine*.

Given two FSMs, proving or disproving the equivalence of their input/output behavior has applications to several fields, namely *automated synthesis*, *formal verification* of correctness, *test pattern generation*, *redundancy identification* and *diagnosis*.

Equivalence is a global property whose verification is better done on a local basis: the theoretical framework consists in *exploring the state space* of the *product machine*, checking at each step whether the outputs differ. State space exploration can either be a *forward* or a *backward traversal*.

The basic traversal operation is an image or a preimage computation. They are conceptually simple if the next state function is given as a "*transition relation*" [1].

Forward traversal is limited to medium-size circuits, because of state space explosion. Backward traversal considers too many unreachable states, thus the BDDs grow too large. Approximate traversals allow only to prove equivalence, not to disprove it, neither to generate a counterexample.

As neither forward nor backward traversals are definitely the best, a combination of both could be beneficial. Instead of a costly exact forward traversal, we have a fast and cheap *approximate* version combined with an *exact backward* algorithm. Efficient function simplification occurs thanks to the use of cofactoring, which is by far superior to simple set intersection-based techniques. The resulting algorithm is *exact*, *complete* and applicable to *large* circuits.

Experimental results show that this mix allows us to explore for the first time and exactly some of the larger ISCAS'89 and MCNC circuits, that have been until now outside the scope of exact symbolic techniques. We are also able to generate the test patterns for or to tag as undetectable all stuck-at faults with few exceptions.

Section 2 summarizes some useful concepts. Section 3 presents the symbolic state space exploration algorithms. Section 4 briefly recalls how the overestimation of the reachable state set is computed. The experimental results are in section 5. Section 6 closes the paper with a brief summary and future developments.

2 Preliminaries

Let two completely specified FSMs be $M = (I, O, S, \delta, \lambda, S_0)$ and $M' = (I, O, S', \delta', \lambda', S'_0)$, where I (O) is the common input (output) space, S and S' are the state spaces, δ and δ' (λ and λ') are the next state (output) functions, and S_0 and S'_0 are the initial state sets.

[1] This work has partially been supported by the ESPRIT Working Group 6018 CHARME-2.

[2] Contact Address: Paolo Camurati, Politecnico di Torino, Dipartimento di Automatica e Informatica, Corso Duca degli Abruzzi 24, I-10129 Turin (Italy), e-mail: camurati@polito.it.

When working on Boolean spaces, denoting with B the set $\{0,1\}$, I, O, S, and S' are (not necessarily the same) powers of B; δ, δ', λ, and λ' are functions from powers of B to powers of B.

The *product machine* $M^p = M \times M'$ is a 6-tuple $M^p = (I, B, S^p, \delta^p, \lambda^p, S_0^p)$ where $S^p = S \times S'$, $\delta^p((s,s'),x) = (\delta(s,x),\ \delta'(s',x))$, $\lambda^p((s,s'),x) = (\lambda(s,x) \neq \lambda'(s',x))$, and $S_0^p = \{(s_0, s_0')$ such that $s_0 \in S_0$ and $s_0' \in S_0'\}$.

Let us consider the representation of δ as a transition relation: its characteristic function $\delta_c : S \times S \times I \rightarrow B$ is 1 iff the next state $y \in S$ is the image of the current state $s \in S$ and of the input $x \in I$ according to δ. As we are often interested only in the existence of an input value, rather than in the value itself, the transition relation abstracts from the inputs. Writing hereinafter, with abuse of notation, the function's name for its characteristic function, the transition relation is:

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

The image (pre-image) of a set of states described by its characteristic function $C(s)$ ($C(y)$) according to δ is defined as: $\mathbf{Img}(\delta, C(s)) = \exists s \ (T_M(s,y) \cdot C(s))$ and $\mathbf{PreImg}(\delta, C(y)) = \exists y \ (T_M(s,y) \cdot C(y))$.

The next section describes how these concepts are applied in practice in forward and backward traversals.

```
FT(δ^p, λ^p, S_0^p); {
    set_of_states R, Curr, New, Next;
    R = Curr = New = S_0^p;
    do {
        if (λ^p · New ≠ ∅) return(Not_Equiv);
        Next = Img(δ^p, Curr);
        New = Next · R̄;
        R = R + New;
        Curr = New;
    } while (New ≠ ∅);
    return(Equiv);
}
```

Figure 1: Exact Forward Traversal

```
BT(δ^p, λ^p, S_0^p); {
    set_of_states Back_R, Curr, New, Prev;
    Back_R = Curr = New = λ^p;
    while (New ≠ ∅) {
        Prev = PreImg(δ^p, Curr);
        New = Prev · ¬Back_R;
        Back_R = Back_R + New;
        if ((New · S_0^p) ≠ ∅) return(Not_Equiv);
        Curr = New;
    }
    return(Equiv);
}
```

Figure 2: Exact Backward Traversal

3 Symbolic Traversal Algorithms

Let us set as a goal proving the equivalence of 2 FSMs, i.e., showing that λ^p is identically zero on all the reachable states of their product machine. The standard algorithms known from the literature are forward and backward symbolic traversals.

3.1 The standard algorithms

Exact forward traversal is a breadth-first search that, starting from the initial state set S_0^p of M^p, computes all its reachable states. At each iteration, we calculate the set of next states (Next) reached from the current one (Curr). This is accomplished by means of a symbolic image evaluation. Reached states are accumulated in R. There are two termination conditions: if there is at least an input combination for which a difference appears on the outputs on the set of newly reached states ($\lambda^p \cdot$ New $\neq \emptyset$) or if a least fixed point is found. The latter condition is tested by computing the newly reached state set (New) at each step, terminating as soon as it becomes empty (New $\neq \emptyset$). Note that the whole reachable state space needs to be explored only for equivalent machines. The pseudo-code is shown in Fig. 1.

The advantage of forward traversal is that exploration starts from a known initial state set and thus focusses only on reachable states. The limit is that it doesn't take the property under scrutiny into account and proceeds blindly so that the reachable state space might be so large that it can't be represented by a BDD.

In the exact backward algorithm, we compute the set of states where at least an input combination makes the outputs differ and we use it as a starting point for traversal. At each step a pre-image of δ^p is evaluated on the set of newly reached states (New), yielding the ones from which they can be reached in one step (Prev). Back_R accumulates discovered states. The greatest fixed point computation terminates as soon as an initial state is encountered (New $\cdot S_0^p \neq \emptyset$) or no newly reached states are found (New $\neq \emptyset$). The pseudo-code is shown in Fig. 2.

The advantage of backward traversal is that search focusses only on those states that might be relevant

```
FBT_R+(δ^p, λ^p, S_0^p); {
    set_of_states R+, Curr, New, Next;
    R+[0] = Curr = New = S_0^p;
    i = 0;
    do {
        if (λ^p · New ≠ ∅)
            if (BT_R+(δ^p, λ^p, R+, i) = Not_Equiv)
                return(Not_Equiv);
        Next = ImgApprox(δ^p, Curr);
        i = i + 1;
        New = Next · ¬R+[i-1];
        R+[i] = R+[i-1] + New;
        Curr = between(New, R+[i]);
    }
    while (New ≠ ∅);
    return(Equiv);
}
```

Figure 3: Combined approximate forward and exact backward traversals

for the proof, although many of them can't be reached from the initial ones. The limit is that it often considers too many unreachable states.

An approximate forward traversal algorithm is derived from Fig. 1 by changing the exact image computation procedure (**Img**) with an approximate one (**ImgApprox**). In this case, an over-estimation of Next results from each call to **ImgApprox**. Its limit is that it can be used to prove equivalence, but if it fails, nothing can be said for sure. Section 4 summarizes our approach to approximate image computation and compares it with the one of [4]. More details can be found in [2].

3.2 Combining traversals

Approximate forward traversals are not very expensive because approximate image computation is cheap. Their limit is that they compute at each step an over-estimation R^+ that contains in general many unreachable states. Although the outputs might differ for some input combination, this could happen on unreachable states. To rule out this case, we perform an exact backward traversal. If it terminates because it encounters S_0^p, some states on which the outputs differ are reachable and equivalence is disproven, otherwise the procedure resumes with one more approximate forward traversal step and proceeds as above. Fig. 3 shows the pseudo-code.

Starting from the initial state set S_0^p, the **ImgApprox** procedure computes the approximate state set (Next) reachable in one step from the current one (Curr). States that have not yet been visited (New) are identified and $\mathsf{R}^+[i]$ stores an over-estimation of the reached state space for that step. If there are new states where a difference appears on the outputs, **BT_R+** (Fig. 4) is called starting from this state set. This procedure differs from the one of Fig. 2 because the vector R^+ of the over-estimations is used to restrict the search space. If the procedure verifies that the outputs differ on reachable states, we conclude that the 2 FSMs are not equivalent and a test pattern can be returned. Otherwise, nothing can be said yet and procedure **FBT_R+** iterates. The starting set Curr is selected choosing a suitable BDD that represents all newly reached states and possibly some of the already visited ones. The least fixed point iteration terminates as soon new states are found in the approximate forward traversal step.

The function **Increase_Exactness** of [9] is based on an approximate pre-image computation, whereas the final image computation is exact. Our pre-image is exact and our image computation is approximate. We believe that this can explain our better performance, because the exact backward traversal exploits the local over-estimation R^+ to restrict the backward search space.

3.3 Enhanced exact backward traversal

The key operation in the pseudo-code of Fig. 2 is pre-image computation. Although conceptually simple, the resulting BDD might contain many unreachable states. Some intuitive observations allow us to efficiently prune the search space, once the over-estimations R^+ are available:

1. the pre-image of δ^p on Curr is of interest only inside the over-estimation $\mathsf{R}^+[i\text{-}1]$ at the previous step, i.e., it can be cofactored with $\mathsf{R}^+[i\text{-}1]$. It is easy to show that this means to cofactor the δ^p functions with $\mathsf{R}^+[i\text{-}1]$

2. there can be no states s outside the over-estimation $\mathsf{R}^+[i]$ at step i whose pre-image has a non-empty intersection with the over-estimation $\mathsf{R}^+[i\text{-}1]$ at the previous step. As a consequence, the current state set Curr whose pre-image is computed can be simplified by cofactoring it with $\mathsf{R}^+[i]$.

Note that the above remarks allow us to deal only with cofactored functions, which are in general much simpler than the original ones, with intuitive benefits on BDD size and CPU time.

Fig. 4 shows a variant to the standard backward algorithm of Fig. 2 that includes the above remarks.

```
BT_R+(δ^p, λ^p, R+, i); {
    set_of_states Curr, Prev;
    Curr = λ^p ↓ R+[i];
    while ((i > 0) and (Curr ≠ ∅)) {
        Prev =PreImg(δ^p ↓ R+[i-1], Curr);
        Curr = Prev;
        i = i-1;
    }
    if ((i = 0) and (Curr ≠ ∅))
        return(Not_Equiv)
    else
        return(Equiv);
}
```

Figure 4: Exact backward traversal with enhanced search state pruning thanks to R^+ and cofactoring

Procedure **BT_R+** is called at the i-th step of the approximate forward traversal and has access to all the over-estimations R^+ computed from step 1 to i-1. Note that there is no more need of passing S_0^p as a parameter, because it is included in $\mathsf{R}^+[0]$.

4 Computation of R^+

Algorithms for computing R^+ must satisfy several constraints:

- *speed*: computing the over-estimation must be significantly faster than performing an exact image computation
- *accuracy*, i.e., how close R and R^+ are
- *applicability*: techniques that work well on single FSMs may fail on product machines, as mutual constraints are overlooked [4].

Image computation requires the knowledge of the transition relation: hence, approximating T_M entails an approximate image computation. The approximation stems from overlooking some mutual constraints. The basic idea is to partition the n f_i functions in k groups, to build for each of them its transition relation T_{M_j} and to compute T_M as their product: $T_M = \prod_{j=1}^{k} T_{M_j}$. Functions are grouped without taking topology into account, unlike [4], rather only their order. Each machine (T_{M_j}) is assumed to be independent of the others.

It is easy to take more constraints into account, returning more accurate over-estimations, with a simple variation to the previous strategy. For example we can sort the f_i functions according to two different orderings and store the result in two arrays. We partition each array, starting from a different offset, in k groups. For each group we compute the exact transition relation. T_M is their product. We refer the reader to [2] for experimental results.

To make the two strategies efficient for product machine traversal, as already noted in [4], we keep corresponding functions in the same group.

Building transition relations for the partitions according to the definition is often impossible, as the f_is may have relatively small BDDs, but their product may grow too large. Early existential quantification could reduce the size, but it doesn't distribute with logical and. A result of [1] shows that they do distribute, provided the functions are simplified by the "*exist*" cofactor. This allows us to build transition relations for much larger circuits.

5 Experimental Results

We implemented the algorithms in a fully homemade package amounting to about 15,000 lines of C-code, called FBT. BDD nodes are limited to 1,500,000.

In this section we present the results of FBT for the circuits described in Tab. 1. They come from the ISCAS'89 and MCNC suites. The columns give the name, the number of primary inputs, primary outputs, flip-flops, and gates for each circuit.

Until now, symbolic techniques have dealt with the smaller ones. We present experimental evidence on all the smaller circuits, but our main novelty is to present data on some of the larger ones, namely s1423, s5378, s13207, s15850, that, to the best of our knowledge, have never been handled symbolically in an exact way. We also present for the first time results on minmax_9, and mul_{32}, as far as symbolic ATPG is concerned.

Tab. 2 collects the ATPG statistics. We ran the experiments on a 70 MIPS VAX-Alpha. For each benchmark circuit, the table shows the total number of faults (F), the number of detectable (D), undetectable (U), and aborted (A) faults and the total CPU time (T) (in seconds, unless otherwise stated). We work on non-scannable synchronous sequential circuits composed of combinational logic and flip-flops, all controlled by the same clock. We assume the existence of an all-zero reset state and that all fault-free or faulty storage devices can be put in that state. Our model is the single stuck-at fault. We disregard faults affecting the clock or the reset lines or occurring inside the flip-flops.

FBT is exact and fully symbolic, i.e., no preliminary random pattern was fault-simulated for pruning

Circuit	PI	PO	FF	Gates
s208	11	2	8	96
s298	3	6	14	119
s344	9	11	15	160
s349	9	11	15	161
s382	3	6	21	158
s386	7	7	6	159
s400	3	6	21	162
s420	19	2	16	196
s444	3	6	21	181
s510	19	7	6	211
s526	3	6	21	193
s641	35	24	19	379
s713	35	23	19	393
s820	18	19	5	289
s382	18	19	5	287
s838	35	2	32	390
s953	16	23	29	418
s1196	14	14	18	529
s1238	14	14	18	510
s1423	17	5	74	657
s1488	8	19	6	653
s1494	8	19	6	647
s5378	35	49	179	2779
s13207	31	121	669	8652
s15850	14	87	597	10384
sbc	40	56	28	1011
minmax_9	14	9	27	863
mul_{16}	18	1	16	245
dsip_sim	229	198	224	4109
mul_{32}	34	2	32	470
big_key	229	198	224	5643

Table 1: Example statistics

the fault list. As soon as the symbolic ATPG procedure computes a new test pattern, we use simulation to drop faults. The software is still a prototype and we put no effort on optimizing its time performance, as we focussed on showing the applicability of our approach to larger circuits. We manually select the degree of approximation for R^+. It is important to note that there is a trade-off between exactness of the over-estimation and ATPG efficiency, as already experienced in [2]. It is often better to use a simple but very approximate R^+ than a complicated, but more precise one. Circuits s13207 and s15850 are sequentially very deep and this makes even approximate traversal difficult on the product machine, as already noted in [4]. We believe that most aborted faults are undetectable, but we still lack adequate approximate techniques for traversing sequentially deep circuits.

Circuit	F	D	U	A	T
s208	215	150	65	0	7
s298	308	273	35	0	5
s344	342	337	5	0	4
s349	332	325	7	0	4
s382	399	379	20	0	376
s386	384	314	70	0	4
s400	424	397	27	0	405
s420	430	204	226	0	77
s444	474	439	35	0	309
s510	564	564	0	0	14
s526	555	466	89	0	515
s641	467	408	59	0	11
s713	585	484	101	0	12
s820	850	815	35	0	40
s832	870	819	51	0	41
s838	857	303	554	0	626
s953	1079	1069	10	0	35
s1196	1242	1199	3	0	33
s1238	1355	1283	72	0	35
s1423	1515	1465	37	13	$5h$
s1488	1486	1446	40	0	73
s1494	1506	1455	51	0	90
s5378	4603	3635	778	190	$7h$
s13207$^{(1)}$	9815	1956	4860	2999	$23h$
s15850$^{(2)}$	11719	573	6804	4342	$40h$
sbc	1495	1381	34	0	180
minmax_9	1594	1247	347	0	240
mul_{16}	561	512	49	0	$8m$
dsip_sim	7233	7230	3	0	$22m$
mul_{32}	1091	1012	79	0	$30m$
big_key	15583	15123	460	0	$94m$

Table 2: ATPG results. $^{(1)}$ means that 952 faults are undetectable with sequences whose length is ≤ 100. $^{(2)}$ means that 1439 faults are undetectable with sequences whose length is ≤ 100.

Our data fully agree with those coming from other symbolic approaches, like VERITAS [5]. There are discrepancies between symbolic approaches and ATPGs like STEED [6]. For example, the percentage of provably undetectable faults according to STEED for s5378 is higher than ours.

Tab. 3 compares FBT, VERITAS, and STEED in terms of test generation efficiency $\mathsf{tge} = \frac{\mathsf{D}+\mathsf{U}}{\mathsf{F}} \cdot 100$. The superiority of symbolic techniques is evident, as well as the fact that FBT handles larger circuits than VERITAS. Most other work focussed on test generation for circuits without a reset state [3], [8], [7]. As already

noted in [5], this is a different problem and, though test generation may be harder, tagging faults as undetectable may be easier. A fair comparison is therefore difficult.

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. We propose a combination of *approximate forward* and *backward traversal* that is exact and efficient, because of its enhanced search space pruning. Experimental results show that it is possible to explore and to generate the test patterns for some of the larger ISCAS'89 and MCNC circuits, that have been until now outside the scope of exact symbolic techniques.

Future developments will consist in reducing the number of aborted faults, by developing notions of fault equivalence classes and better techniques for product machine traversal of large and sequentially deep circuits.

References

[1] G. Cabodi, P. Camurati: "Exploiting cofactoring for efficient FSM symbolic traversal based on the Transition Relation," ICCD'93, Cambridge, MA (USA), October 1993, pp. 299–303

[2] G. Cabodi, P. Camurati, S. Quer: "Efficient State Space Pruning in Symbolic Backward Traversal," ICCD'94, Cambridge, MA (USA), October 1994

[3] W-T. Cheng, T. J. Chakraborty: "Gentest: an automatic test-generation system for sequential circuits," IEEE Computer, Vol. 22, n. 4, April 1989, pp. 43–48

[4] H.Cho, G.D. Hachtel, E. Macii, B. Plessier, F. Somenzi: "Algorithms for approximate FSM traversals," DAC-30, Dallas, TX (USA), June 1993, pp. 25–30

[5] H. Cho, G.D. Hachtel, F. Somenzi: "Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration," IEEE Transactions on CAD, Vol. 12, No. 7, July 1993, pp. 935–945

[6] A. Ghosh, S. Devadas, A. Richard Newton: "Test generation and verification for highly sequential circuits," IEEE Transactions on CAD, Vol. 10, No. 5, May 1991, pp. 652–667

[7] K. Hatayama, K. Hikone, M. Ikeda, T. Hayashi: "Sequential test generation based on real-valued logic simulation," ITC'92, September 1992, pp. 41–48

[8] T. Niermann, J.H. Patel: "HITEC: a test generation package for sequential circuits," EDAC'91, February 1991, pp. 214–218

[9] F. Somenzi, Private communication

Circuit	tge		
	FBT	VERITAS	STEED
s208	100	100	97.02
s298	100	100	99.02
s344	100	100	100
s349	100	100	100
s382	100	100	95.23
s386	100	100	100
s400	100	100	95.75
s420	100	100	91.16
s444	100	100	95.56
s510	100	100	99.82
s526	100	100	90.99
s641	100	100	93.08
s713	100	100	93.11
s820	100	100	100
s832	100	100	99.65
s838	100	100	80.50
s953	100	100	100
s1196	100	100	98.71
s1238	100	100	98.96
s1423	99.14	-	-
s1488	100	100	100
s1494	100	100	100
s5378	95.87	-	99.25
$s13207^{(1)}$	69.45	-	-
$s15850^{(2)}$	62.95	-	-
sbc	100	100	98.64
$minmax_9$	100	-	-
mul_{16}	100	-	-
dsip_sim	100	100	100
mul_{32}	100	-	-
big_key	100	-	-

Table 3: ATPG result comparison. - means data not available. $^{(1)}$ means that 952 faults are undetectable with sequences whose length is $\leq$ 100. $^{(2)}$ means that 1439 faults are undetectable with sequences whose length is $\leq$ 100.