Incremental Re-encoding for Symbolic Traversal of Product Machines

Stefano Quer † *
Luciano Lavagno #

Gianpiero Cabodi ‡
Ellen Sentovich $

Paolo Camurati %
R. K. Brayton †

† UCB
Dept. of EECS
Berkeley, CA
(USA)

‡ Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

% Università di Udine
Dip. di Matematica e Informatica
Udine, ITALY

Politecnico di Torino
Dip. di Elettronica
Turin, ITALY

$ Ecole des Mines de Paris
Centre de Mathématiques Appliquées
Sophia–Antipolis, FRANCE

Abstract

State space exploration of finite state machines is used to prove properties. The three paradigms for exploring reachable states, forward traversal, backward traversal and a combination of the two, reach their limits on large practical examples. Approximate techniques and combinational verification are far less expensive but these imply sufficient, not strictly necessary conditions. Extending the applicability of the purely combinational check can be achieved through state minimization, partitioning, and re-encoding the FSMs to factor out their differences. This paper focuses on re-encoding presenting an incremental approach to re-encoding for sequential verification. Experimental results demonstrate the effectiveness of this solution on medium–large circuits where other techniques may fail.

1 Introduction

An FSM is identified by its Boolean input and output domains, its initial state set, and its next state and output relations.

Many properties are related to the composite behavior of two FSMs. This behavior is captured by the *product machine*, which is an FSM composed of the two. The inputs are shared and a single output is created that records the differences between the output signals of the two.

A common problem is that of comparing two machines that are behaviorally equivalent but structurally different, for example when one FSM has been obtained from the other by means of sequential optimization (such as partial or total re-encoding, retiming and resynthesis, sequential redundancy removal, etc.). In this case the equivalence check is used to verify the validity of manual synthesis and optimization steps, or to find bugs in the synthesis algorithms. Equivalence checks can be performed on a local basis, by traversing the state space of the product machine and verifying the property on each state.

State–space traversal is typically performed via symbolic exploration of the *state transition graph* (STG) using BDDs to represent the next–state functions and reachable state sets.

We propose a technique to simplify equivalence checks by *re-encoding*. One machine is re-encoded to make it structurally more similar to the other, and hence exploit the underlying correspondence. This makes traversal–based verification more efficient, and in the limit permits using purely combinational verification. The techniques are applicable when the machines are very different, but the results will be best for *similar* machines with a non explicit correspondence (which is often the case in logic synthesis). Just as dynamic variable reordering allows one to change the shape of the canonical BDD while building it, dynamic re-encoding allows one to change

*Stefano Quer is also with the Politecnico di Torino, Dipartimento di Automatica ed Informatica, Turin, ITALY.

the shape of the reached state set while building it. The re-encoding function in product machine traversal plays a role analogous to the variable ordering in the BDD. With the increasing size of the machines to be verified, this degree of flexibility must be exploited. Experimental evidence of the effectiveness of these techniques on medium–large circuits is shown.

The remainder of the paper is organized as follows. We assume a basic knowledge of FSMs, FSM equivalence and graph isomorphism, BDDs, and image computation of sets of states. In Section 2, we analyze the requirements of re-encoding and its complexity. Section 3 describes the verification strategy based on re-encoding and combinational verification. We give some experimental results in Section 4 to justify the feasibility of this approach. Conclusions are given in Section 5.

2 Verification by Re-encoding

The overall algorithm for verification works as follows. We begin with two FSMs M^1 and M^2. If they are already combinationally equivalent (next state and output functions are equal), the process terminates. Otherwise a re-encoding function ρ is computed and used to transform M^1 into an equivalent FSM M^*. The process can be iterated replacing M^1 with M^*. This iteration yields a *sequence* of FSMs that hopefully converges from M^1 to M^2. The representation of the re-encoding characteristic function is directly related to the reachable state set of the product machine $M^1 \times M^2$.

Disregarding the trivial case in which the initial FSMs M^1 and M^2 are already combinationally equivalent, the two may be related in one of the following ways: 1) M^1 and M^2 are equivalent and their STGs are isomorphic, 2) M^1 and M^2 are equivalent but their STGs are not isomorphic, and 3) M^1 and M^2 are not equivalent.

In the first case there exists some $1:1$ re-encoding function that can produce combinationally equivalent FSMs (when both are restricted to the reachable state set or to an over-estimation of it). In the second one we cannot deal specifically with this problem because the final equivalence check cannot be performed combinationally when only $1:1$ re-encoding functions are considered. In the final case the re-encoding function does not exist. and to determine that the machines are not equivalent, a traversal–based verification is necessary. However in the last two cases standard traversals are performed on re-encoded and more similar machines. This traversal is possibly much simpler than with the original M^1 and M^2. The traversal–based verification may also happen when the ρ re-encoding exists but is too complex or difficult to obtain and we resort to a converging sequence of re-encodings.

Re-encoding FSMs for more efficient state–space traversal requires some support algorithms:

- *Heuristics for FSM decomposition and product machine correspondence recognition.* These are important for reducing the cost of traversal and increasing accuracy of approximate traversal.
- *Exact and/or approximate product machine traversal*, carried out as long as the representation of the reachable states remains simple. The reached states of the product machine are used as a starting point in generating the re-encoding function.
- *Extending partial re-encodings and/or reducing n:m relations to* $1:1$ *functions.* This step is the computational framework for handling incomplete or over-estimating re-encoding informations.
- *Constraining the product machine state space during traversal.* This is mandatory to reduce the complexity of approximate and exact traversals, and to reduce the complexity of the re-encoding function.

State minimization of the initial machines in some cases results in machines with isomorphic STGs and is an orthogonal technique [3] that will not be discussed here.

2.1 Transforming FSMs

An FSM M is given by $M = (I, O, S, \delta, \lambda, S_0)$, where I is the set of input variables, O the set of output variables, S the set of states, δ the set of next–state functions, λ the set of output functions, and S_0 the set of initial states. The input, output, and state variables sets are denoted x, z, and s. The reachable state set is denoted R, where $R \subseteq S$; an over-estimation of this set is denoted R^+, where $R \subseteq R^+$; the product machine of M^1 and M^2 is denoted M^p. Machine equivalence is denoted $M^1 \doteq M^2$; equivalence means identical input/output behavior from the initial state set.

The re-encoding is denoted by $\rho(s^1, s^2)$ to indicate the characteristic function of the $1:1$ mapping, and by the pair $\rho v(s^1)$ and $\rho v^{-1}(s^2)$ to indicate the Boolean functional vectors that re-express s^2 in terms of s^1 and vice-versa.

Re-encoding transforms M^1 into M^* and should preserve the input/output behavior of M^1:

$$(M^* = \rho(M^1)) \quad \Leftrightarrow \quad (M^1 \doteq M^*)$$

The goal of the transformation is to make M^* similar to M^2, where similarity is a measure of the number of equivalent states that have identical encodings. M^* is used as the new M^1 for the next re-encoding iteration. Among many equivalence–preserving re-encodings, we select ρ to be a $1:1$ mapping from S^1 to $S^* = S^2$, possibly restricted to either R^1 or R^{1+}.

Once the characteristic function of the $1:1$ mapping $\rho(s^1, s^2)$ is known two Boolean function vectors $\rho v(s^1)$ and $\rho v^{-1}(s^2)$ are generated. The next–state and output functions of M^* are computed as follows:

$$\begin{aligned}\delta^*(s^*, x) &= \rho v(\delta^1(\rho v^{-1}(s^*)), x)\\ \lambda^*(s^*, x) &= \lambda^1(\rho v^{-1}(s^*), x)\end{aligned}$$

Consequently we obtain $M^* = (I, O, S^*, \delta^*, \lambda^*, S_0^*)$. The transformation only affects the state variables, so the equivalence of M^1 and M^* is guaranteed by the existence of the inverse ρv^{-1} of ρv (since ρv is a $1:1$ function, ρv^{-1} exists).

The one step re-encoding function ρ might be too complex to represent and/or to apply. Then we try to find a sequence of simpler re-encodings.

Another possible technique to decrease the complexity of ρ, is to re-encode a subset of the state variables at a time. This strategy is particularly suited for networks of interacting FSMs or systems composed by control unit and data path, where independent encodings are applied to disjoint subsets of state variables. Let us partition the s^1 and s^2 variables into two subsets s^{1a}, s^{1b} and s^{2a}, s^{2b}. This is easily generalized to an arbitrary number of partitions. Let us express $\rho(s^1, s^2)$ as:

$$\rho(s^1, s^2) = \rho^a(s^{1a}, s^{2a}) \cdot \rho^b(s^{1b}, s^{2b})$$

If ρ^a and ρ^b are $1:1$, so is ρ.

The complexity is further reduced by observing that a partitioned re-encoding can be applied in two steps

$$\begin{aligned}\rho_1(s^1, s^2) &= \rho^a(s^{1a}, s^{2a}) \cdot \textstyle\prod_{i=1}^{|s^b|}(s_i^{1b} = s_i^{2b})\\ \rho_2(s^1, s^2) &= \rho^b(s^{1b}, s^{2b}) \cdot \textstyle\prod_{i=1}^{|s^a|}(s_i^{1a} = s_i^{2a})\end{aligned}$$

which ensure a smaller size of the intermediate BDDs, because part of the re-encoding is simply the identity function.

3 The Verification Procedure

Verification is done by using partial state–space exploration to iteratively compute a sequence of re-encoding functions. The goal is to use low–cost traversals and yet obtain enough information to compute a useful re-encoding function.

A simplified version of the code is shown in Fig. 1.

Two traversal methods are used:

- i steps of exact forward traversal of M^p, with i much less than the index of the fixed–point. This computation returns R_i^p.
- an approximate forward traversal of M^p, possibly stopping before a fixed–point is reached. This computation returns an overestimation of R^p, R^{p+}.

It is also possible to compute a mix of exact but partial information (R_i^p) and complete but approximate information (R^{p+}).

A new re-encoded machine is generated in each iteration of the outer loop. The loop terminates when equivalence is determined (a complete re-encoding function was found) or the number of allowed iterations is exceeded. In the latter case, the final re-encoded machine is used in the subsequent product machine traversal for determining equivalence.

Traversing a product machine's state space is more costly than for a single FSM. In principle the state space increases linearly in the product of the cardinalities of the state spaces. Fortunately, this is not the case in practice since the component machines are strongly coupled by primary input sharing, and they evolve in lockstep during traversal. Furthermore, if the component machines are "similar", they have similar reached state sets which implies a smaller reached state set in the product machine. One of the main issues is to try and keep corresponding next–state functions and variables interleaved.

To identify permutations of state variable names, we follow a heuristic approach inspired by Chen *et al.* [1] that computes signatures associated to input variables. We modify the method applying it to state variables. The algorithm is implemented in the PM_HEURISTIC function.

This corresponds to an initial decomposition and re-ordering of M^p. The re-encoding is generated and updated in the inner loop. At each iteration, a forward traversal of M^p is performed using either exact or approximate techniques. This generates the set reached (equal to R_i^p or R_i^{p+}). If the state graphs of the two FSMs are isomorphic, R_i^p represents a $1:1$ correspondence. In the general case both R_i^p and R_i^{p+} are an $m:n$ state correspondence.

Let ρ^- be a $1:1$ under-estimation of ρ (initially, $\rho^- = \emptyset$), and let ρ^+ be a $m:n$ over-estimation of ρ ($\rho^+ =$ reached). Information is successively added to ρ^- from ρ^+, while keeping the result $1:1$ (function EXTEND_RE-ENCODING in Fig. 1). This is done at the fixed point or periodically during traversal (when $++$level MOD step $= 0$ is true in the pseudo-code), allowing the inner steps to have more information in computing the function. ρ^- is then extended to encompass all state pairs (s^1, s^2) of R^p (function COMPLETE_RE-ENCODING in Fig. 1). This last procedure, for the sake of simplicity, extends the encoding to the whole state space, but in practice this is limited to an over-estimation of the reachable states. At the end of each cycle, the function RE-ENCODE computes the re-encoded machine M^* using the previous version and the ρ mapping.

3.1 Extending a partial $1:1$ re-encoding

Suppose we are given $A^1 \subseteq S^1$ and $A^2 \subseteq S^2$ of equal cardinality, and a $1:1$ re-encoding ρ^- that re-encodes a subset of A^1 in a subset of A^2. If a complete re-encoding is required for A^1, ρ^- must be extended to the whole A^1 set.

If A^1 and A^2 have equal cardinalities, we first extract the subsets of A^1 and A^2 included in the ρ^- function, $A^1_{\rho^-}$ and $A^2_{\rho^-}$. Then we consider $\rho^+ - (A^1_{\rho^-} \times A^2) - (A^1 \times A^2_{\rho^-})$ that indicates all possible re-encodings for states, in S^1 and S^2 that have been reached during traversal and which do not appear in the re-encoding function ρ^-. We apply the $1:1$ reduction procedure presented in 3.2 to that relation. The result is OR-ed with ρ^- to produce a $1:1$ function from A^1 to A^2. (This procedure is guaranteed to find a complete encoding given a complete graph.)

If A^1 and A^2 have different cardinalities, we look for a $1:1$ mapping of the smaller set onto a subset of the larger one and then extend it according to the above procedure.

The pseudo-code is shown in Fig. 2. EXTEND_RE-ENCODING extends ρ^- using ρ^+. It restricts ρ^+ to states not in ρ^-, uses the reduction procedure of Fig. 3 to reduce ρ^+ to a $1:1$ function, and adds ρ^+ to ρ^-. ρ^+ contains, in general, an over-estimation of the state correspondence which loses accuracy as the traversal proceeds; we heuristically avoid re-considering states that have already been matched. COMPLETE_RE-ENCODING uses EXTEND_RE-ENCODING to extend ρ^- to the whole space of M^p.

3.2 Reducing a re-encoding from $m:n$ to $1:1$

The last algorithm to be described is that of reducing an $m:n$ re-encoding $\rho^+ \subseteq A^1 \times A^2$ to $1:1$ $\rho^+_{1:1}$.

The relation is given in implicit form and can be represented by a bipartite graph. Then the problem is equivalent to solving a perfect bipartite matching problem.

If a perfect matching exists, it is also a maximum one, and well–known algorithms for maximum matching based on the min–cut max–flow theorem (or their symbolic versions [2]) could be used. Unfortunately the matching problem is in the inner loop of our algorithm, so exact algorithms, would still require an excessive amount time.

We propose a quick heuristic algorithm, based on Lin's *et al. compatible projection* operator without guarantee of maximality.

The pseudo-code is shown in Fig. 3. Let σ be a set of literals in the product machine state space (initially the special minterm for minimum distance selection).

After testing terminal recursion cases the procedure SELECT_LITERAL selects a splitting literal (v) from σ and uses it for recursive traversal of (in theory, all) paths from the root to a 1 leaf. Each path represents an encoding for s^1 and a corresponding state s^2. Recursion proceeds on the two cofactors of ρ^+ with respect to the splitting literal, producing the then (T) and else (E) components of the result. The reduction is guaranteed to be $1:1$ by an existential quantification on $\rho^+_{\bar{v}}$, before computing the E component that prevents the re-use of mapped states. If $v \in A^1$, existentially quantifying all s^1 results in P, i.e., all the states in A^2 in the subspace that have already been considered. The state in P should therefore not be further considered. Random choices are made in selecting a $1:1$ function contained in ρ^+.

4 Experimental Results

The verification by re-encoding technique has been implemented in a program prototype written in C. We report the results of the experiments performed on a *DEC ALPHA 7000* with *256MByte* of memory. The circuits used are taken from the well known ISCAS'89 (with the '93 addendum) and MCNC set.

We do not include data about small FSMs, because exact traversal techniques are sufficient in these cases.

In all the following tables # FF indicates the number of state variables, # P indicates the number of partitions used in approximate traversal (# P = 1 indicates exact traversal). Level is the number of traversal steps (image computations), R shows the number reachable states, and # Cycle is the number of re-encoding steps performed until equivalence is proved. Time is the CPU time in seconds.

In Table 1 we list four experiments in which we perform self-verification after manually shuffling product machine variable ordering (to emulate the case of unknown correspondence), the goal of re-encoding being to find the exact variable correspondence. $|\rho|$ represents the size of the re-encoding function in terms of BDD nodes. With the addition of the pruning constraint ($\lambda^1 = \lambda^2$) the R^{p+} computed is $1:1$ and contains the exact re-encoding. In the Time column CPU times in brackets refer to the cost of applying ρ to transform the initial machine in the re-encoded one: this appears to be a relevant phase in the overall process, except for circuit s713, in which the coupling heuristic was enough for combinational equivalence, and our re-encoding based verification was not necessary. With circuit s1423, the initial manual variable shuffling was *limited* (not far from ideal interleaving), because due to the size of the circuit we were not able to deal with a more general variable ordering. Lines s1423$_a$ and s1423$_b$ (concerning the same starting problem) differ in the accuracy of approximate traversal, and demonstrate that a looser traversal may sometimes provide an exact re-encoding in less time.

Table 2 presents the verification of medium–large machines composed of smaller FSMs. We used equivalent but not equal circuits (like s344 and s349) or circuits with different state assignments obtained using SIS [4]. In the latter case we start from the STG of the machine in *kiss* format, and we synthesize it with different state assignment strategies (i.e. different options for the *state_assign* command of *Nova* [5] are used to obtain equivalent machines). Using SIS implies dealing with small circuits: then we adopt *serial* (+) and *parallel* (||) composition to build larger FSMs. Column $|\rho|_{avg}$ reports the average size of the applied re-encoding functions.

In Table 3 we list some examples of independent encodings for sub-blocks of a large circuit. Given a FSM, we randomly re-encode subsets of the state variables, and check the new circuit against the original one. In the verification phase, we exploit the advantage of iterative re-encoding (see section 2.1) by restricting some re-encoding steps to subsets of the state variables. Column Re-encoding list the sizes of the re-encoded sets of state variables: e.g. $7, 7$ with 28 flip-flops means that two sets of 7 state variables each are re-encoded, while 14 state variables keep their original coding. In the case of sbc, we compare the single step re-encoding (sbc_a) with a three step one (sbc_b), and the advantage of the latter is clear in term of both size of re-encoding and CPU time. Circuits s1269 and s1512 are selected among those that we cannot traverse exactly: the ratio of re-encoded state variables is low, but the experiments prove the ability of dealing with large circuits, provided that a proper decomposition can be found for re-encoding. At present we have no exact or heuristic method for re-encoding oriented circuit decomposition, thus we used for the above experiments manual decomposition, exploiting knowledge of the partitioning used to randomly encode circuits.

Possible tunings for all these experiments involve varying the accuracy of the approximate traversal and the number of times the procedure EXTEND_RE-ENCODING is used during traversal. The experimental data result from a trade-off between accuracy and cost. The level of accuracy in the approximate forward traversal was biased to guarantee satisfactory guesses for re-encoding, while keeping the traversal time low.

5 Conclusions

We presented an incremental approach to re-encoding one FSM composing a product machine. This technique is based on an approximate forward traversal of the product machine to obtain information for re-encoding. The re-encoding functions allow us to extend the applicability of combinational checks and, in case they fail, to make traversals more efficient.

References

[1] D.I. Cheng, M. Marek–Sadowska, "Verifying Equivalence of Functions with Unknown Input Correspondence," in *Proc. EDAC'93*, February 1993, pp. 81–85

[2] G. Hachtel, F. Somenzi, "A Symbolic Algorithm for Maximum Flow in 0–1 Networks," in *Proc. IEEE ICCAD'93*, November 1993, pp. 403–406

[3] B. Lin, A. Richard Newton, "Implicit Manipulation of Equivalence Classes Using Binary Decision Diagrams," in *Proc. IEEE ICCD'91*, October 1991, pp. 81–85

[4] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, A. Sangiovanni–Vincentelli, "SIS: A system for sequential circuit synthesis," Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992

[5] T. Villa, "NOVA. State assignment of finite state machines for optimal two–level logic implementations," in *Proc. ACM/IEEE DAC'89*, June 1989, pp. 327–332

```
VERIFY_BY_RE-ENCODING (M^1, M^2, step, max_iterations)
{
M* = M^1;
n_iterations = 0;
while (1)
    {
    if (M^2 ≐ M*)
        return (∅, equivalent);
    M^p = BUILD_PM (M^2, M*);
    if (++n_iterations > max_iterations)
        /* M^p must be verified by traversal */
        return (M^p, aborted);
    /* initializations */
    level = 0;
    ρ^- = 0;
    from = reached = new = S_0^p;
    /* M^p traversal */
    M^p = PM_HEURISTIC (M^p);
    while (new != ∅)
        {
        to = IMAGE (M^p, from);
        new = to · ¬reached;
        reached = reached + new;
        if (++level MOD step == 0)
            ρ^- = EXTEND_RE-ENCODING (M^p, ρ^-, reached);
        }
    ρ^- = EXTEND_RE-ENCODING (M^p, ρ^-, reached);
    ρ = COMPLETE_RE-ENCODING (M^p, ρ^-);
    M* = RE-ENCODE (M*, ρ);
    }
}
```

Figure 1: Re-encoding–based verification.

```
EXTEND_RE-ENCODING (M^p, ρ^-, ρ^+)
{
assert (is_11 (ρ^-));
s^1 = first_machine_state_vars (M^p);
s^2 = second_machine_state_vars (M^p);
/* restrict ρ^+ to states not in ρ^- */
ρ^+ = ρ^+ · ¬∃_{s^1}(ρ^-);
ρ^+ = ρ^+ · ¬∃_{s^2}(ρ^-);
/* 1 to 1 filter */
ρ^+_{1:1} = REDUCE (M^p, ρ^+, s^1 + s^2);
/* extend ρ^- */
ρ^- = ρ^- + ρ^+_{1:1};
return ρ^-;
}

COMPLETE_RE-ENCODING (M^p, ρ^-)
{
S^p = machine_state_space (M^p);
/* complete by extension */
return EXTEND_RE-ENCODING (M^p, ρ^-, S^p);
}
```

Figure 2: Pseudo-code to extend the re-encoding function.

```
REDUCE (M^p, ρ^+, σ)
{
/* testing terminal cases */
if (ρ^+ == 0)
    return 0;
if (empty (σ))
    {
    assert (ρ^+ == 1);
    return 1;
    }
s^1 = first_machine_state_vars (M^p);
s^2 = second_machine_state_vars (M^p);
/* recursion */
v = SELECT_LITERAL (σ);
T = REDUCE (M^p, ρ^+_v, σ_v);
if (v ∈ s^1)
    P = ∃_{s^1} T;
else
    P = ∃_{s^2} T;
E = REDUCE (M^p, ρ^+_{¬v} · ¬P, σ_v);
return (BUILD_BDD (v, T, E));
}
```

Figure 3: Pseudo-code to reduce the re-encoding function.

Circuit	# FF	Verify by Exact Forward Traversal			Verify by Re-encoding			
		# Level	R	Time	# P	$\|\rho\|$	# Cycle	Time
s400	21	151	8865	13	3	689	1	7.3 (3.7)
s713	19	7	1544	2	-	0	0	0.5 (0)
s1423$_a$	74	-	-	-	16	723	1	347 (113)
s1423$_b$	74	-	-	-	8	1014	1	520 (187)

Table 1: Experimental results on large FSMs with manual re-encoding. - means unknown, i.e. overflow in BDD nodes.

Circuit$_1$/Circuit$_2$	# FF	Verify by Exact Forward Traversal			Verify by Re-encoding			
		# Level	R	Time	# P	$\|\rho\|_{avg}$	# Cycle	Time
(s344+s510$_{ie}$) (s349+s510$_{ig}$)	21	20	17249	19.8	3	244	1	3.7 (1.0)
s510$_{ioh}$+(s641$\|\|$s820$_{ia}$) s510$_{iov}$+(s713$\|\|$s832$_{ie}$)	30	47	$1.814 \cdot 10^6$	145	3	987	1	85 (57)
(s344+s510$_{ie}$)$\|\|$s1488$_{ia}$ (s349+s510$_{ig}$)$\|\|$s1494$_{ih}$	27	25	$7.77 \cdot 10^5$	423	3	734	3	39 (27)
s298$_{ig}$$\|\|$s820$_{ia}$$||$(s344+s420$_{ioh}$) s298$_{ih}$$\|\|$s832$_{ie}$$||$(s349+s420$_{iov}$)	50	-	-	-	5	1232	3	234 (115)

Table 2: Experimental results on large composed FSMs. - means unknown, i.e. overflow in BDD nodes. $_{ig}$, $_{ih}$, $_{ioh}$, etc. indicate different options for the *state_assign* command of SIS, using Nova.

Circuit	# FF	Re-encoding	Verify by Exact Forward Traversal			Verify by Re-encoding			
			# Level	R	Time	# P	$\|\rho\|_{avg}$	# Cycle	Time
s400	21	11, 10	151	8865	7.3	3	316	3	32.2 (21.2)
s713	19	10, 9	7	1544	5.1	4	139	3	23.2 (16.5)
sbc$_a$	28	7, 7	10	154592	600	2	698	1	302 (197)
sbc$_b$	28	7, 7	10	154592	600	2	187	3	101 (61)
s1269	37	5	-	-	-	13	231	2	72 (43)
s1512	57	5	-	-	-	12	257	4	231 (123)

Table 3: Experimental results on large FSMs with random encoding. - means unknown, i.e. overflow in BDD nodes.