Transforming Boolean Relations by Symbolic Encoding

Gianpiero Cabodi[1] and Stefano Quer[1] and Paolo Camurati[2]

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy
[2] Dipartimento di Matematica e Informatica, Università di Udine, Udine, Italy

Abstract. Transforming Boolean relations and functions is an important horizontal technique that finds several applications in logic synthesis and formal verification. This paper develops a framework for analyzing input/output transformations of Boolean relations and functions. It also contributes efficient composition techniques based on partitioning the transformation. Experimental results on equivalence-preserving FSM state-space re-encoding demonstrate the feasibility of the approach.

1 Introduction

One major step in logic synthesis is *encoding*, i.e., finding a suitable correspondence between symbols of a finite set S and binary strings. We need strings whose length l satisfies the following inequality:

$$l \geq \lceil log_2(\text{card}(S)) \rceil \tag{1}$$

Boolean *relations* and *functions* describe correspondences between symbols and set elements or inputs and outputs. In combinational design, we encode input and output symbols and represent the function of the circuit. In sequential circuits, using the *Finite State Machine* (FSM) model, we encode its input, output, and state symbols and express the output and next-state functions in Boolean form. Just any of the possible encodings may not satisfy constraints related to *minimizing area*, *performance*, *power dissipation*, and to *verifying equivalence*, and so on. For this reason, *re-encoding*, i.e. replacing a binary string with another binary string, is a technique that finds an application in several cases [7], [8], [10].

For sake of readability, we do not distinguish between the initial *encoding* phase and any *re-encoding* steps and we use the term *encoding* for both.

Given an input/output mapping in the form of *Boolean Relation* (BR) $\mathcal{F}$ or *Boolean Function Vector* (BFV) f, we transform it, changing the old codes for its input (output) symbols in new ones. The input/output transformation itself is expressed as a relation or as a function. When the transformation is $1:1$ or $n:1$, both representations as function or as relation are possible. In the most general case of $n:m$ transformation, we express it as a BR, because this case occurs seldom and it's not worthwhile to investigate how to express it as a BFV. Initially, we assume to have completely specified transformations but we easily extend our results to incompletely specified ones.

This paper doesn't deal with *finding* the transformation: for this purpose we refer the reader to the literature [7], [8], [10]. We rather focus on *applying* the transformation, which is essentially the problem of composing functions or relations.

Apart from a tutorial aspect, the main contributions of this paper are:

- a theoretical framework for the application of new encodings
- a theorem for the transformation of a $n : 1$ BR into a BFV
- efficient composition techniques for both functions and relations, based on decomposition procedures and on the exploitation of the don't care set for incompletely specified functions.

The remainder of the paper is organized as follows: section 2 defines the notational framework and recalls some preliminary notions about FSMs, and Boolean operators. Section 3 describes how to encode Boolean strings. Section 4 analyzes encoding from the point of view of complexity, showing that decomposed multi-step encoding can be of help in some cases. Because of the importance of FSMs, encoding them is the subject of section 5. Section 6 reports experimental results.

2 Preliminaries

This section defines the notation used in this paper and recalls some basic notions about FSMs and Boolean operators.

2.1 Notational framework

Let $B = \{0, 1\}$. Let $I = B^n$ be an input set consisting of n-bit Boolean strings and let $O = B^m$ be an output set composed of m-bit Boolean strings.

A BFV f is defined as $f : I \rightarrow O, y = f(x) = (f_1(x), \ldots, f_m(x))$.

Let $\mathcal{F}(x, y)$ be a $n : 1$ BR. A compatible BFV f is defined as follows:

$$f : I \rightarrow O, y = f(x) \Leftrightarrow \mathcal{F}(x, y) \tag{2}$$

Given a set $S = B^k$, we can represent and manipulate efficiently its subsets using their Boolean *characteristic functions*. Let A be a subset of S. The *characteristic function* of A is the function $\chi_A : S \rightarrow B$ defined by:

$$\chi_A(a) = \begin{cases} 1 \textit{ if } a \in A \\ 0 \textit{ if } a \notin A \end{cases} \tag{3}$$

Among all sets that can be represented by characteristic functions, we consider Boolean Relations (BRs). A BR $\mathcal{F}(x, y)$ is a mapping between elements x and y of its domains I and O. It is thus a set of ordered tuples (x, y). With abuse of notation, we denote with the same symbol a relation and its characteristic function.

2.2 FSMs

A *finite state machine* is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 5-tuple $M = (I, O, S, \delta, \lambda)$, where I is the input alphabet, O is the output alphabet, S is the state space, δ is the next state function ($\delta : S \times I \rightarrow S$), and λ is the output function ($\lambda : S \times I \rightarrow O$).

2.3 Boolean operators

Boolean functions are efficiently represented by Binary Decision Diagrams (BDDs) [2]. Boolean operators of great importance are ITE (if-then-else) and COMPOSE.

The if-then-else operator Given Boolean functions f, g, and h, Bryant *et al.* [1] define $\text{ITE}(f, g, h)$ as:

$$\text{ITE}(f, g, h) = f \cdot g + \overline{f} \cdot h \tag{4}$$

Function composition As this case is commonly found in our applications, we restrict investigation to a particular form of function composition where $f(x, y)$ is a Boolean function and $g(z)$ is a BFV whose set of support doesn't contain any y variables. If $y \in B^k$ and $g : B^h \rightarrow B^k$, we define function composition as:

$$f(x,\ g(z)) = \text{COMPOSE}\ (f(x,y),\ y,\ g(z)) = f(x,y)|_{y_1=g_1(z),\ y_2=g_2(z),\ldots,y_k=g_k(z)} \tag{5}$$

A standard way to perform it resorts to recursive applications of ITE:

$$\begin{aligned}\text{COMPOSE}(f(x,y),\ y,\ g(z)) = \text{ITE}(&g_1(z), \text{COMPOSE}(f_{y_1}(x, y^{(2)}),\ y^{(2)},\ g^{(2)}(z)),\\ &\text{COMPOSE}(f_{\overline{y}_1}(x, y^{(2)}),\ y^{(2)},\ g^{(2)}(z)))\end{aligned} \tag{6}$$

where superscripts, e.g. $f^{(i)}$, indicate the components with index $j \geq i$, e.g. $f^{(i)} = (f_i, f_{i+1}, \ldots, f_n)$ in the recursive formula.

Relation composition Let $\mathcal{F}(x, y)$ and $\mathcal{G}(y, z)$ be BRs. We can compose $\mathcal{F}$ and $\mathcal{G}$ by logical conjunction and existential quantification of y:

$$(\mathcal{F} \circ \mathcal{G})(x, z) = \exists y\ (\mathcal{F}(x, y)\ \cdot\ \mathcal{G}(y, z)) \tag{7}$$

3 Encoding

Let us consider a mapping between an input space I and an output space O. We express it either as a relation $\mathcal{F}(x, y)$ or as a function $y = f(x)$ (where $x \in I$ and $y \in O$). We can encode the input space, the output space, or both. Let us suppose for simplicity to encode the entire input (output) space and let the encoded input (output) space be I' (O'). The input transformation is either an encoding function e_I or an encoding relation $\mathcal{E}_I$. The same holds for the output transformation e_O or $\mathcal{E}_O$.

Relations $\mathcal{E}_I$ and $\mathcal{E}_O$ are expressed as:

$$\mathcal{E}_I : I \times I' \rightarrow B, \quad \mathcal{E}_I(x, x') \Leftrightarrow x' \text{ is a code of } x \tag{8}$$

$$\mathcal{E}_O : O \times O' \rightarrow B, \quad \mathcal{E}_O(y, y') \Leftrightarrow y' \text{ is a code of } y \tag{9}$$

Functions e_I and e_O are expressed as:

$$e_I : I \rightarrow I', \quad x' = e_I(x) \Leftrightarrow x' \text{ is the code of } x \tag{10}$$

$$e_O : O \rightarrow O', \quad y' = e_O(y) \Leftrightarrow y' \text{ is the code of } y \tag{11}$$

A graphical representation is shown in Figure 1. As the inputs to f or $\mathcal{F}$ belong to the I space, they are re-constructed from x' by applying the inverse input transformation $\mathcal{E}_I^{-1}$ or e_I^{-1}. This in turn imposes an invertibility constraint on $\mathcal{E}_I$ and e_I. In a similar way, y is transformed into y' by $\mathcal{E}_O$ and e_O. The application of the input/output encodings to f or $\mathcal{F}$ essentially requires function or relation compositions. In the compositions we identify an outer term and an inner one. $\mathcal{F}$ (f) is the outer term in input encoding and the outer one in output encoding, $\mathcal{E}_I^{-1}$ (e_I^{-1}) and $\mathcal{E}_O$ (e_O) are the corresponding inner and outer terms, respectively.

We must thus distinguish many cases, depending on the use of relations vs. functions for the inverse of the input transformation, for the output transformation, and for the original representation of the circuit function. Let's analyze explicitly the two limit cases:

case 1: if the three are expressed as BFVs, we use function composition:

$$y' = f'(x') = e_O(f(e_I^{-1}(x'))) \tag{12}$$

case 2: if the three are expressed as BRs, we use conjunction and existential quantification:

$$\mathcal{F}'(x', y') = (\mathcal{E}_I^{-1} \circ \mathcal{F} \circ \mathcal{E}_O)(x', y') = \exists x, y(\mathcal{E}_I(x, x') \cdot \mathcal{F}(x, y) \cdot \mathcal{E}_O(y, y')) \tag{13}$$

All other cases can be handled according to the following observations. The result of composing an outer BFV f with an inner BR $\mathcal{G}$ is a relation. BFV f is first transformed into a relation, relational composition is then performed. Composing an outer BR $\mathcal{F}$ with an inner BFV g is a plain functional composition $\mathcal{F}(x,\ g(z))$, producing a relation. A particular but important sub-case is composing a BFV with a $n : 1$ BR, i.e. a function expressed as a relation. Regardless of the mutual position (inner/outer) the BR may be converted to a compatible BFV, reducing this case to functional composition.

Let us analyze how to transform BFVs in BRs and vice-versa.

Transforming a BFV in a BR Let f be a BFV $f : B^n \rightarrow B^m, y = f(x)$. The standard way to compute the corresponding BR $\mathcal{F}(x, y)$ is to perform a conjunction:

$$\mathcal{F}(x, y) = \prod_{i=1}^{m} (y_i \equiv f_i(x)) \tag{14}$$

Transforming a BR in a BFV This case occurs in many applications, e.g., in the computation of the FSM equivalent states ($n : 1$ or when finding $1 : 1$ encodings.

In the process of converting an $n : m$ BR to a BFV, some information may be lost and the result is a BFV compatible with the original BR. We focus on $n : 1$ BRs, for which there is no information loss in the conversion process.

A possible way to compute a compatible f starting from a $\mathcal{F}$ is to individually compute each f_i function as the existential quantification of the output variables on each positive cofactor according to y_i of the BR. There is also a dual form, based on the universal quantifier. The following theorem guarantees that the result is one of the compatible BFVs of the BR:

Theorem 1. *Hp:*

- $\mathcal{F} : I \times O \rightarrow B$ *represents a* $n : 1$ *BR, i.e.,* $\mathcal{F}(x, y^*) \cdot \mathcal{F}(x, y^{**}) \Leftrightarrow y^* = y^{**}$
- $y_i = f_i(x) = \exists y \, \mathcal{F}_{y_i}(x, y), \; i = 1 \ldots n$

Th:
$\mathcal{F}(x, y) \Leftrightarrow (y = f(x)) \cdot (\exists y \, \mathcal{F}(x, y))$, *i.e., the BFV* f *is equivalent to BR* $\mathcal{F}$ *for all minterms* x *belonging to the relation.*

Proof. Informally, we proceed as follows. Given a minterm $x^* \in I$, the corresponding minterm $y^* \in O$ is unique (if it exists). Its component bits y_i, $i = 1 \ldots m$ can be computed as $y_i^* = \exists y \, \mathcal{F}(x^*, y)$.

More formally, as

$$f_i(x) = \exists y \, \mathcal{F}_{y_i}(x, y) = \exists y \, (y_i \cdot \mathcal{F}(x, y)) \tag{15}$$

given a minterm in $y^* \in O$ and its i-th bit y_i^*

$$y_i^* = f_i(x) \Leftrightarrow y_i^* = \exists y \, (y_i \cdot \mathcal{F}(x, y)) \Leftrightarrow \exists y \, ((y_i^* \equiv y_i) \cdot \mathcal{F}(x, y)) \tag{16}$$

The second equivalence holds only if $\mathcal{F}$ is $n : 1$, which is true because of Hp. 1. Extending the above equivalence to the whole y^*

$$y^* = f(x) \Leftrightarrow \prod_{i=1}^{n} (y_i^* \equiv f_i(x)) \Leftrightarrow \prod_{i=1}^{n} (\exists y \, ((y_i^* \equiv y_i) \cdot \mathcal{F}(x, y))) \tag{17}$$

as $\exists y \, (a \cdot b) \Rightarrow \exists y \, a \cdot \exists y \, b$ we permute the conjunction and the existential quantifier

$$y^* = f(x) \Leftarrow \exists y \, (\prod_{i=1}^{n} (y_i^* \equiv y_i) \cdot \mathcal{F}(x, y)) \tag{18}$$

restricting the analysis to minterms x in $\exists y \, \mathcal{F}(x, y)$, the right expression yields $\mathcal{F}(x, y^*)$

$$(y^* = f(x)) \cdot \exists y \, \mathcal{F}(x, y) \Leftarrow \mathcal{F}(x, y^*) \tag{19}$$

The implication becomes a co-implication, proving the thesis, if we consider that no x^* can yield $(y^* = f(x^*)) \cdot \exists y \, \mathcal{F}(x^*, y)$ and not $\mathcal{F}(x^*, y^*)$.

4 On the Complexity of Encoding

A complexity analysis of encoding deals with three aspects: the complexity of computing, representing, and applying the encoding. We refer the reader to [7], [8], and [10] for a discussion on the first two points. In this paper, we focus our attention on the third one.

The application of an encoding requires essentially composing functions or relations. We assume the results published by R. Bryant in [2] on the worst-case complexity of operations on BDDs.

Let $\mathcal{F}$ and $\mathcal{G}$ be the BRs to be composed and let $|\mathcal{F}|$ and $|\mathcal{G}|$ be their size in BDD nodes. The complexity of COMPOSE is $O(|\mathcal{F}|^2 \cdot |\mathcal{G}|^2)$. Average complexity is, according to our experience, below this upper bound.

As far as functions are concerned, Bryant analyzes only the case of a BFV f composed with a function g. Let $|f|$ and $|g|$ be their size in BDD nodes. The complexity of COMPOSE is $O(|f|^2 \cdot |g|)$.

There is no analysis for the composition of two BFVs, but according to Coudert *et al.* [5], in the worst-case, this is an NP-hard problem.

In the average case, good implementations allow to cope with quite big problems [3], [9]. Variable ordering, function simplification, and efficient caching play a key role. Complexity is not only related to the final result, but also to the intermediate ones, because they require considerable space and CPU time.

We found no formal analysis nor any conjecture in literature about the cost of the intermediate steps in composition. According to our experience, we conclude that the complexity of the result of a composition depends on the complexity of the operands, namely the function/relation to be encoded and on the complexity of the encoding. The cost of computing a composition depends on the algorithm adopted, and it depends on the size of the operands, too. A good conjecture for feasible compositions is a polynomial function of the size of the operands and of the result.

Note that, if we compose BRs, the application of COMPOSE is straightforward, asymptotic complexity is better, but sometimes the BDDs for the BRs are so large that we can't even compute them. With BFVs, the individual BDDs are simpler, although their composition is harder.

Given a fixed variable ordering, Coudert *et al.* [5] show that the size of the result can be exponential. We limit our scope to problems where the result of composition can be expressed. In the following paragraphs we present techniques that simplify computation, based on the exploitation of the don't care set and on the application of encoding as a sequence of simpler functions.

4.1 Exploiting the don't care set

We do not go into the details of incompletely specified encodings. Their importance lies in the fact that properly exploiting don't-care sets may result in minimized and/or optimized BRs or BFVs.

For incompletely specified functions, the don't care set gives us a degree of freedom that we exploit to reduce the complexity of an encoding. The input don't care set $d_I(x)$ is given by the specifications. The output don't care set $d_O(x)$ is given by all those outputs that can't occur, i.e., $d_O(x) = \overline{\text{range}(f(x))}, d_O(x) = \overline{\text{range}(\mathcal{F}(x,y))}$. We leave the input (output) encoding unspecified on $d_I(x)$ ($d_O(x)$).

4.2 Decomposed encoding

Let us consider the general case of a BR $\mathcal{E}$ that denotes either the input or the output encoding. An equivalent result can be found by applying a sequence of simpler encodings, provided such a decomposition of $\mathcal{E}$ is available. For sake of simplicity, let us restrict investigation to just two steps. This is easily generalized to an arbitrary number of steps. In the general case, the original transformation is expressed as a composition:

$$\mathcal{E} = \mathcal{E}_a \circ \mathcal{E}_b \tag{20}$$

Applying in sequence $\mathcal{E}_a$ and $\mathcal{E}_b$ results in a relevant gain when their sizes are much lower than $|\mathcal{E}|$.

As a particular case, let us consider independent encodings applied to disjoint subsets of variables. Let v represent generically either x or y variables. Let v_a and v_b be partitions of the variables and let v'_a and v'_b be the corresponding encoded variables. The encoding $\mathcal{E}$ is the Cartesian product of $\mathcal{E}_a$ and $\mathcal{E}_b$:

$$\mathcal{E}(v_a, v_b, v'_a, v'_b) = \mathcal{E}_a(v_a, v'_a) \times \mathcal{E}_b(v_b, v'_b) \tag{21}$$

The gain can be relevant, because the size of $\mathcal{E}_a$ and $\mathcal{E}_b$ is smaller than the size of $\mathcal{E}$, especially when variables v_a (v'_a), v_b (v'_b) are interleaved in the ordering.

A simple case of such a decomposed encoding is related to don't care set exploitation. Suppose that the incompletely specified encoding $\mathcal{E}$ is a subset of the identity relation $\mathcal{I}_a$ on a subset (v_a, v'_a) of the variables, i.e., re-encoding is applied to just a subset of the code bits. Setting the values of the don't cares, we obtain the encoding $\mathcal{E}^*$, easily expressed as a Cartesian product:

$$\mathcal{E}^*(v_a, v_b, v'_a, v'_b) = \mathcal{I}_a(v_a, v'_a) \times \exists v_a, v_{a'}(\mathcal{E}_b(v_b, v'_b)) \tag{22}$$

Decomposed encodings are easily found for partitioned circuits. Partitioning according to topology relies on designer knowledge or heuristic functions, Quer *et al.* present state variable partitioning for verification in [10].

5 Encoding FSMs

FSMs play an important role in automated synthesis, formal verification, and testing. Transformations can be of great help in speeding up for example the symbolic traversal of the product machine of two FSMs, because similar state encodings make BDDs simpler [10]. They can also contribute to the exact or approximate state minimization of a machine, mapping all states belonging to the same equivalence class onto one code [4]. We therefore examine this particular case of encoding and present experimental data in the next section.

Encoding the states of FSM M may be advantageous in many applications, eg., symbolic traversals. Because of the feedback loop of Fig. 2, the transformation must be equivalence-preserving, i.e., the resulting M' must have the same input/output behavior of M. In order not to be bothered by the existence of equivalent states, we consider a particluar case of equivalence-preserving transformations, i.e., $1:1$ functions. A $1:1$ function $e(s,s')$, which is certainly invertible, serves as e_O and its inverse as $e_I = e^{-1}$ [10]. The transformation only affects the state elements on the feedback loop, so the equivalence of M and M' is guaranteed by the existence of the inverse e^{-1} of e.

Consequently, we obtain $M' = (I, O, S', \delta', \lambda')$, where the next-state and output functions are computed as follows:

$$\begin{aligned} \delta'(s',x) &= e(\delta(e^{-1}(s'),x)) \\ \lambda'(s',x) &= \lambda'(e^{-1}(s'),x) \end{aligned} \tag{23}$$

6 Experimental Results

A common verification problem is that of comparing two machines that are behaviorally equivalent but structurally different. This is the case when one FSM has been obtained from the other by means of sequential optimization (such as partial or total encoding, retiming and resynthesis, sequential redundancy removal, etc. [6]). Symbolic state space traversal is the state-of-the-art technique. Its efficiency can be increased when the FSMs have the same or similar state encodings. Transforming a state encoding to make it more similar to another one is a typical application of Boolean transformations.

We present data on the transformation of the next-state and output functions of FSMs for significant ISCAS'89 (with the '93 addendum) and MCNC benchmark circuits. We experimented on a 30 MIPS *DEC VAX 7000* with *128 MByte* of memory.

In Table 1 #P indicates the number of partitions used to encode the circuit. When #P> 1, encoding is decomposed, otherwise it is monolithic. #FF indicates the maximum number of state variables found in any partition. Column avg$|e|$ shows the size in BDD nodes of the encoding functions. The CPU time in seconds is shown in the next columns: avg_time$_{in}$ is the average time required for the input encoding of next-state and output functions, avg_time$_{out}$ is the average time for the output encoding of next-state functions, and tot_time is the total amount of time required to encode the circuit.

Decomposed encoding is always superior to monolithic encoding, except in the case of s400. Moreover, it handles cases on which the monolithic encoding approach fails because of node explosion.

7 Conclusions and Future work

Transforming Boolean functions and relations has several applications in the fields of automated synthesis and formal verification. In this paper we developed a theoretical framework about the application of encodings, contributing

Table 1. Experimental results. - means unknown, i.e. overflow in BDD nodes (with 10^6 nodes and garbage collection active).

Circuit	#P	$\#FF_{max}$	avg|e|	avg_time_{in}	avg_time_{out}	tot_time
s400	1	21	436	0.6	6.0	7.2
	3	7	161	1.0	2.0	9.8
s713	1	19	389	1.9	92.1	97.8
	2	10	195	4.5	14.1	40.5
	3	7	138	3.3	9.2	38.9
s1238	1	18	392	1.4	10.5	13.4
	3	6	129	0.9	0.4	4.3
s1423	1	74	2000	35.0	-	-
	11	7	430	20.5	75.8	1078.4
	20	4	384	15.0	29.1	904.5
s1269	1	37	808	15.7	-	-
	4	10	298	8.3	-	-
	10	4	199	7.7	97.43	1598.7
s1512	1	57	1421	10.3	-	-
	10	6	329	4.2	16.4	273.5
sbc	1	28	708	3.7	213.6	219.9
	7	4	152	0.4	1.5	17.1

in particular a theorem to transform Boolean $n : 1$ relations in BFVs. We also described efficient composition techniques for both functions and relations, based on decomposition procedures and on the exploitation of the don't care set for incompletely specified functions.

Future work will consist in a more detailed analysis of the complexity of composition, especially for BFVs, in carrying on the investigation on the exploitation of the don't care set for incompletely specified functions, and in a complete set of experimental results.

References

1. K.S. Brace, R.L. Rudell, R. Bryant: Efficient Implementation of a BDD Package. *Proc. IEEE/ACM DAC'90*, June 1990, pp. 40–45
2. R.E. Bryant: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 667–691
3. G. Cabodi, P. Camurati, S. Quer: Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversals. *Proc. IEEE EURO-DAC'94*, Grenoble (France), September 1994, pp. 22–27 *best paper award*
4. G. Cabodi, P. Camurati, S. Quer: Computing subsets of equivalence classes for large FSMs. *Proc. IEEE EURO-DAC'95*, September 1995

5. O. Coudert, J.C. Madre, C. Berthet: Verifying temporal properties of sequential machines without building their state diagrams. *AMS/DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3, 1991, pp. 75–84
6. G. De Micheli: Synthesis and optimization of digital circuits. McGraw-Hill, 1994
7. G.D. Hachtel, M. Hermida, A. Pardo, M. Poncino, F. Somenzi: Re-Encoding Sequential Circuits to Reduce Power Dissipation. *Proc. IEEE ICCAD'94*, November 1994, pp. 70–73
8. B. Lin, H.J. Touati, A. Richard Newton: Don't Care Minimization of Multi-Level Sequential Logic Networks. *Proc. IEEE ICCAD'90*, November 1990, pp. 414–417
9. C. Pixley: A computational theory and implementation of sequential hardware equivalence. *AMS/DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3, 1991, pp. 293–320
10. S. Quer, G. Cabodi, P. Camurati, L. Lavagno, E.M. Sentovich, R.K. Brayton: Incremental FSM Re-encoding for Simplifying Verification by Symbolic Traversal. *IEEE International Workshop on Logic Synthesis*, May 1995