Integrating Symbolic Techniques in ATPG-Based Sequential Logic Optimization

Enrique San Millán, Luis Entrena, José A. Espejo, *Silvia Chiusano, *Fulvio Corno
Universidad Carlos III de Madrid. Dpto. de Ingenería Eléctrica, Electrónica y Automática
{entrena, quique, ppespejo}@ing.uc3m.es
*Politecnico di Torino. Dipartimento di Automatica e Informatica
*{chiusano, corno}@polito.it

Abstract[#]

This paper presents a new integrated approach to logic optimization for sequential circuits. The approach is based on the Redundancy Addition and Removal algorithm, which is based on Automatic Test Pattern Generation (ATPG) techniques, and improves it using Symbolic Techniques based on BDDs. The advantage of the integrated approach lies in the ability of Symbolic Techniques to provide exact and extensive information about the sequential behavior of the portion of the circuit that is of interest to the logic optimization algorithm. Experimental results are provided that show the superiority of the approach to the original ATPG-based optimization approach.

1. Introduction

Logic optimization for synchronous sequential circuits is still an open and challenging problem. Given an initial circuit description at the gate level, sequential logic optimization aims at computing an equivalent circuit with a smaller area occupation, usually estimated through the gate, connection or literal counts. Approaches to sequential optimization have been proposed mainly at the FSM level [1]. Some approaches have also been proposed at the logic level using combinational techniques [2].

One very promising research area aims at exploiting the progresses made in the area of Automatic Test Pattern Generation (ATPG). Under some fairly general assumptions, namely for combinational circuits and for synchronous sequential circuits with a reset state, there is a correspondence between untestable faults and redundant gates and connections in the circuit. Some redundancy removal tools based on ATPG have been developed [3],[4]. These techniques have also been extended to Redundancy Addition and Removal [5]. With this approach, ATPG techniques are first used to identify a redundant connection/gate to be added to the circuit. This step temporarily increases the area of the circuit, but it is immediately followed by a redundancy identification step, again performed with ATPG techniques, to find formerly irredundant gates or connections that now became redundant, and can be removed. The goal of the algorithm is to add the redundancy in such a way to remove more hardware (gates or connections) than it adds.

The critical part of the Redundancy Addition and Removal technique is clearly the redundancy identification step based on ATPG techniques. In [5], such a step was performed through the propagation of *mandatory assignments*, with the help of unique sensitization techniques and *Recursive Learning* [6]. The exploitation of mandatory assignments allows efficient identification of inconsistencies and therefore circuit redundancies.

The goal of this paper is to enhance the degree of optimization that can be reached by proposing an integration of symbolic techniques based on BDDs [7] in the algorithm presented in [5]. Symbolic techniques are very powerful in finding exact inconsistencies and implications in both combinational and sequential circuits, but they are limited to a moderate size and a number of state elements not exceeding some tens. The approach proposed here therefore first identifies a smaller portion of the circuit (hereinafter called "macro") on which the redundancy identification procedure needs accurate results, and then delivers the required information via symbolic computations. The information that symbolic computations can extract from the macro include combinational implications between distant gates, information about the reachable states from the reset one, and justification and propagation of values through the macro.

With this integrated approach we are able to identify more mandatory assignments, and therefore to improve the circuit optimization results on sequential circuits. Some experimental results were gathered to prove the effectiveness of our solution.

The rest of this paper is organized as follows. Section 2 reviews the redundancy addition and removal techniques and their application to sequential logic optimization. Section 3 on the other hand, analyzes which kinds of

[#] This work has been supported in part by the Autonomous Community of Madrid under Project #06T/024/96

symbolic manipulations are useful for being integrated. The integrated approach is detailed in Section 4. Experimental results are finally reported in Section 5, while Section 6 presents the conclusions and points to future developments of this work.

2. Logic Optimization by Redundancy Addition and Removal

Redundancy Addition and Removal has been shown to be a powerful logic optimization method by several authors [5], [8], [9], [10], [11], [12]. With this method, a logic network is optimized by iteratively adding and removing redundancies that are identified using ATPG techniques. If the addition of *k* redundant wires/gates creates more than *k* redundant wires/gates elsewhere in the network, the removal of the created redundancies will result in a smaller area.

The basic Redundancy Addition and Removal approach can be summarized as follows. A wire is selected and tested for stuck-at fault. If no test is possible, then the wire is redundant and can be removed. Otherwise, we try to add a wire or a gate to the circuit in order to make the fault redundant. Candidates for addition are also tested for stuck-at fault in order to verify that the addition preserves the functionality of the circuit.

Redundancy Addition and Removal is also applicable to sequential circuits. In this case, redundancies are identified by sequential test generation methods using the well-known time frame model.

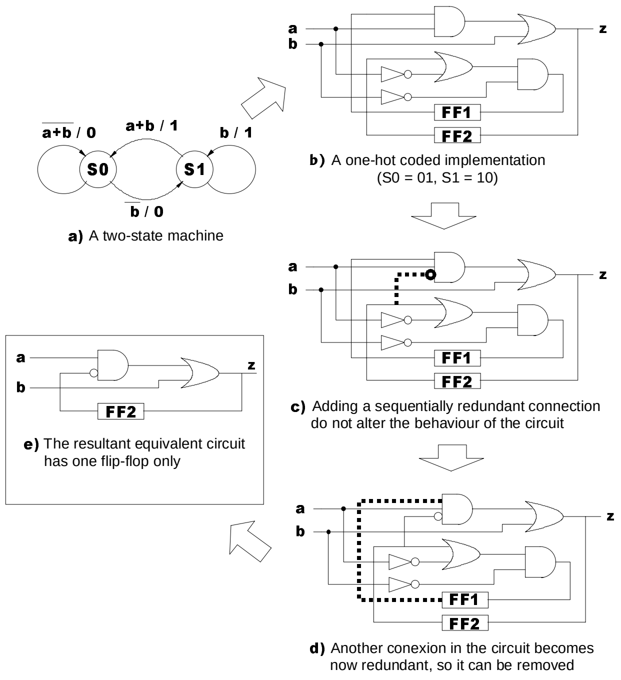
$\overline{a+b}$ / 0
a+b / 1
b / 1
S0
S1
$\overline{b}$ / 0
a) A two-state machine
a
b
z
FF1
FF2
b) A one-hot coded implementation (S0 = 01, S1 = 10)
c) Adding a sequentially redundant connection do not alter the behaviour of the circuit
d) Another conexion in the circuit becomes now redundant, so it can be removed
e) The resultant equivalent circuit has one flip-flop only

Fig. 1. Example of Sequential Redundancy Addition and Removal

Example (taken from [5]). Fig.1 shows the state graph of a two-state machine and a one-hot coded implementation of this machine. The machine has two inputs, *a* and *b*, and one output. To activate the fault *x2* stuck-at *1*, the circuit must be in state *S1 = 10*, i.e., *x1 = 1* and *x2 = 0* are mandatory assignments for this fault. Therefore, adding *x1* as an input of *g1* will block the propagation of the fault. If we add this connection, as shown in Fig. 1(c), the sequential behavior of the circuit will not change and *x2* will become sequentially redundant after this addition (Fig. 1(d)). After removing *x2* and its fanins which become floating, the circuit has only one flip-flop and is a minimal-bit encoded machine (Fig. 1(e)).

The key of ATPG-based redundancy identification is the computation of mandatory assignments. Mandatory assignments are those which are required for a test to exist and must be satisfied by any test vector. In order to identify redundancy, only the mandatory assignments need to be considered. If during the search for mandatory assignments of a given fault an inconsistency is found, then the fault is untestable.

Although it is not necessary to compute the complete set of mandatory assignments to identify redundancy, the more mandatory assignments are identified, the more redundancies may be found. Furthermore, mandatory assignments are used to identify the candidate wire/gates to be added. Unfortunately, the identification of all mandatory assignments is an NP-complete problem. Efficient techniques exist for the identification of mandatory assignments based on the concept of absolute dominators [13]. Recursive learning [6] is a recursive method that is able to identify all mandatory assignments, given enough time. By properly pruning the recursion tree, this method allows to obtain different subsets of mandatory assignments with various computational efforts.

3. Symbolic Computations for Sequential Circuits

Symbolic Techniques can represent the behavior of a synchronous sequential circuit or a portion of it through a Finite State Machine model, specified by the state transition function $y=\delta(s,x)$ (computing the next state *y* from the current state *s* and the current input *x*) and by the output function $z=\lambda(s,x)$ (computing the output *z* starting from the same information).

We adopt the standard technique based on *characteristic functions* to represent sets of inputs and of states [14]. The characteristic function χ_δ of the state transition function $y=\delta(s,x)$ is a compact representation of function δ: $\chi_\delta(s,x,y)$ takes the value 1 for all the triples (s',x',y') such that $y'=\delta(s',x')$.

During the search for mandatory assignments a procedure based on symbolic computations is activated to detect whether the current set of assignments is consistent and possibly to find additional implications. The procedure takes all the gate values in the current time frame and tries to update them with symbolic information. The information that is used to perform these checks is both *combinational*, i.e., related to the relationships between input and output variables, and *sequential*, i.e., derived from the knowledge of the *reachable* states of the machine.

As a result, if the symbolic procedure finds inconsistencies in the assignments, the redundancy of the fault is proven, and the mandatory assignment procedure is immediately stopped. Otherwise, it may be possible to find some implication of mandatory assignments on some formerly unspecified gates: these newly found implications are inserted in the queue of mandatory assignments to be propagated.

In the implementation, the characteristic functions χ_δ and χ_λ are preliminarily extracted from the circuit netlist and the set of reachable states $\chi_R(s)$ is determined by using a classical fix point iteration [14]. A pseudo-code of the symbolic implication procedure is reported in Fig. 2. It first extracts from the circuit values the sets of allowed values at inputs and outputs. These sets, that are always cubes, are stored in the functions $\chi_x(x)$, $\chi_s(s)$, $\chi_z(z)$ and $\chi_y(y)$.

All the constraints are then intersected with the characteristic functions containing the knowledge about the circuit behavior χ_δ and χ_λ, and the characteristic function χ_{IMPLY} contains all the input/output combinations compatible with:

- the current values assumed at the gate outputs
- the function computed by the combinational network
- the set of reachable states of the machine.

This set has an arbitrary shape in the Boolean space, but we approximate it with the smallest enclosing cube, whose known variables define the *implications of mandatory assignments* .

Three different results can be returned:

- **inconsistency** (χ_{IMPLY}=0): there is no input/output combination satisfying all the constraints. The considered fault is untestable and the connection redundant.
- **no implication** ($\chi_{IMPLY}\neq 0$, but the implications of mandatory assignments coincide with the original constraints). No additional information is found by the symbolic procedure.
- **new implication** ($\chi_{IMPLY}\neq 0$, and some implication is present on at least one previously unspecified variable). A new implication has been discovered, the circuit values are updated, and the mandatory assignment procedure is restarted.

Since BDD's of reasonable size can be computed only for small and medium circuits, we apply symbolic techniques to small *macros*, only. This also allows us to neglect the variable ordering problem, since the functions have a reduced size by construction; in the experiment we use the natural ordering, i.e., the order of appearance in the netlist.

```
symbolic_imply( gate_values[] ) {
  $\chi_x$=to_bdd(PI, gate_values); $\chi_z$=to_bdd(PO, gate_values);
  $\chi_s$=to_bdd(PPI, gate_values); $\chi_y$=to_bdd(PPO, gate_values);
  compute $\chi_{IMPLY} = \chi_x \chi_s \chi_z \chi_y \cdot \chi_R \chi_\delta \chi_\lambda$ ;
  if ($\chi_{IMPLY}$ == 0) return INCONSISTENT ;
  else {
      new_gate_values[]=extract_mandatory ($\chi_{IMPLY}$ ) ;
      if (new_gate_values == gate_values)
          return NO_IMPLICATION ;
      else {
          add_implications(new_gate_values) ;
          return NEW_IMPLICATIONS ;
} } }
```

Fig. 2. Symbolic procedure

4. The Integrated Approach

The Redundancy Addition and Removal algorithm of logic optimization can be improved by enriching the mandatory assignment identification procedure with the above mentioned symbolic computations. For a fruitful integration, it is necessary to take into account the following issues:

- Symbolic procedures can deal with sequential circuits within small time and memory limits, provided that the number of flip-flops does not exceed 10-20. This means that the symbolic information must work only on a subset of the circuit, that will be called *macro*.
- The macro being considered must overlap significantly with the portion of the circuit on which there are already some mandatory assignments. Otherwise, too few input constraints arise, and no new implication can be expected to arise.
- During logic optimization the network is expected to change as a result of addition and removal of redundancies. Even if the overall circuit functionality is guaranteed not to change, the Boolean function computed by the macro may change, thus requiring re-computing all the BDD's. Therefore, the cost of the preliminary steps is relevant, and must be kept as low as possible.

The algorithm of [5] selects a *target node* in each iteration and checks for possible circuit optimizations resulting from adding and removing redundancies in the input cone of this node. The symbolic procedure builds a macro able to deliver implications in the portion of the circuit that is more likely to be explored for a given target node. Whenever the target node changes, or the circuit is changed, a new macro is identified and the BDD's are accordingly recomputed.

The construction of macros is performed as follows:

- First, some flip-flops are selected according to some heuristic. The heuristics used for this task will be described later in this section.

- Then the logic gates surrounding these flip-flops, i.e., the gates that are both in the input cone of at least one selected flip-flop and in the output cone of some other one, are included in the macro.
- Macro inputs and outputs are identified by looking at which combinational gates are at a boundary between included gates and excluded ones.

In summary, the macro is built around a set of selected flip-flops. Therefore, the key of this approach is to find good heuristics to efficiently select which flip-flops must be included in a macro. Two heuristics have been determined:

Heuristic 1

Starting from the location of the target node, first the flip-flops that lie in the input cone of the target node are identified. If the number of such flip-flops is within the threshold to be handled by BDDs, they are all selected for the next step. If there are too many, the farthest ones to the target node are discarded; if they are too few, the input cones of these flip-flops are also explored.

Heuristic 2

According to the previous considerations, the flip-flops for the macro must overlap significantly with the portion of the circuit on which mandatory assignments already set some values. It can be shown that there is a common subset of mandatory assignments for the set of faults that are tried at one iteration. We will illustrate this idea with the following example.

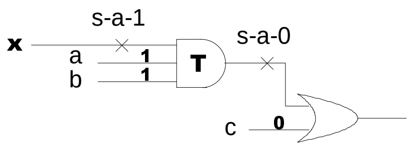

Fig. 3. Example for heuristic 2

Example. Consider the fault *T* stuck-at *0* in the circuit shown in Fig. 3. The mandatory assignments for this fault are *a=1*, *b=1* and *c=0*. Now consider the addition of a new input *x* to the target node *T* (shown as a dashed line). To check whether this added input is redundant, we test fault *x* stuck-at *1* and obtain the mandatory assignments *x=0*, *a=1*, *b=1* and *c=0*. Note that this is a superset of the mandatory assignments for the fault *T* stuck-at *0*.

In general, the faults related to the addition of wires/gates to the target node, and also the faults related to the removal of wires/gates in the input cone of the target node, have many mandatory assignments in common with the ones generated by the faults *T* stuck-at *0* and *T* stuck-at *1*, being T the target node. This observation leads to the following heuristic:

- First, try the faults *T* stuck-at *0* and *T* stuck-at *1* at the output of the target node.
- After implying, choose those flip-flops that hold a mandatory assignment for any of these stuck-at faults.
- If there are too many, we choose the closest to the target node; and if there are too few, we complete according to heuristic 1.

5. Experimental Results

In this section we present the experimental results obtained for ISCAS89 and MCNC benchmark circuits. Starting from the source code of the tool implementing the algorithm of [5], we implemented the symbolic procedures and integrated them in the optimization algorithm. The symbolic module has been implemented by using the BDD package developed by the authors. Experimental results have been run on a Sun Sparc Station Ultra-1 with 64 MB RAM, with a limit of one million of BDD nodes set.

The results obtained by running the algorithm with heuristic 2 are summarized in Table 1. This table is divided in three main column groups corresponding to the initial benchmarks, the results after running sequential optimization by Redundancy Addition and Removal only (RAR), and the results with Redundancy Addition and Removal and symbolic techniques (RAR+BDDs), respectively. For each case, the number of flip-flops (FF), gates (G) and connections (C) are shown. The last column reports the percent improvement of the number of connections.

The experimental results demonstrate that the use of symbolic techniques produces significant improvement in most examples within moderate CPU time increase. The maximum improvement is obtained for *s298*, which has 25% less gates and 21% less connections when using symbolic techniques. Heuristic 1 gives generally worse results, being better only for two examples, namely *s499* (7.67%) and *s444* (7.83%).

6. Conclusions and Future Work

Efficient logic optimization for sequential circuits is still an open issue. This paper proposed the integration of a logic optimization algorithm, based on redundancy addition and removal through the adoption of ATPG-based techniques, with symbolic computations performed via BDDs. While the overall algorithm still works at the structural level, the portions of the circuit of higher interest to the ATPG core are also analyzed through symbolic techniques, that deliver very quickly exact information about their sequential behavior.

Experimental results show that, in most cases, the circuit optimized with the proposed integrated approach are smaller than the ones obtained with the original algorithm.

We are still improving the tool, by aiming at broadening its applicability to larger circuits. To do this, two extensions are currently under development, namely the support of several macros inside the circuit instead of just one, and the implementation of a second symbolic implication procedure working over multiple time frames and taking also account of the values of combinational gates inside the macro, not just at its boundaries.

7. References

[1] P. Ashar, S. Devadas, A. R. Newton. Sequential Logic Synthesis. Kluwer Academic Publishers, 1992

[2] S. Malik, E. M. Sentovich, R. Brayton, A. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimizing Sequential Circuits Using Combinational Techniques. IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 10, p. 74-84. January 1991

[3] K.T. Cheng, On removing redundancy in sequential circuits, in Proc. 28th Design Automation Conf., June 1991, p. 164-169

[4] H. Cho, G.D. Hachtel, F. Somenzi. Redundancy identification and removal based on implicit state enumeration. Proc. Intl. Conf. in Computer Design (ICCD-91), October 1991, p. 77-80

[5] L. Entrena, K.-T. Cheng. Combinational and sequential logic optimization by redundancy addition and removal. IEEE Trans. on CAD, Vol. 14, No. 7, July 1995, p. 909-916

[6] W. Kunz, D.K. Pradhan. Recursive learning: an attractive alternative to the decision tree test generation in digital circuits. Proc. Intl. Test Conf., October 1992, p.816-825

[7] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys, Vol. 24, Nr. 3, 1992, p. 293-318

[8] C. Chang, K.-T. Cheng, N.-S. Woo, M. Marek-Sadowska. Layout Driven Logic Synthesis for FPGAs. Proc. DAC-94, p. 308-313. June, 1994

[9] S. C. Chang, M. Marek-Sadowska. Perturb and Simplify: Multi-level Boolean Network Optimizer. Proc. ICCAD-94, p. 2-5. November, 1994

[10] W. Kunz, P. R. Menon. Multi-level Logic Optimization by Implication Analysis. Proc. ICCAD-94, p. 6-13. November, 1994

[11] U. Gläser, K.-T. Cheng. Logic Optimization by an Improved Sequential Redundancy Addition and Removal Technique. Proc. ASP-DAC. September, 1995

[12] L. Entrena, J. A. Espejo, E. Olías, J. Uceda. Timing Optimization by an Improved Redundancy Addition and Removal Technique. Proc. EURODAC'96, p. 342-347. September 1996

[13] T. Kirkland, M. R. Mercer. A Topological Search Algorithm for ATPG. Proc. ACM/IEEE 24th Design Automation Conf., p. 502-508. June, 1987

[14] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. Symbolic Model Checking: 1020 States and Beyond. LICS'90: 5th Annual IEEE Symposium on Logic in Computer Science, p. 428–439, June 1990

	Initial			RAR				RAR + BDDs				
	FF	G	C	FF	G	C	CPU [s.]	FF	G	C	CPU [s.]	Imp. [%]
s298	14	101	255	14	72	183	5	14	54	144	23	21.31
s386	6	104	305	6	64	179	7	6	54	150	19	16.20
s444	21	125	319	21	84	230	7	21	81	213	34	7.39
s526n	21	167	449	21	104	268	13	21	83	221	71	17.54
s420	16	125	310	16	97	256	12	16	97	256	15	0.00
s499	22	138	391	22	116	339	26	22	112	330	54	2.65
s382	21	118	306	21	89	240	8	21	81	215	34	10.42
s400	21	123	317	21	88	236	8	21	81	214	40	9.32
s832	5	243	726	5	127	400	65	5	122	373	102	6.75
s967	29	306	256	29	260	679	157	29	253	664	278	2.21
bbara	4	41	147	4	41	134	1	4	41	129	4	3.73
bbsse	4	49	227	4	49	188	3	4	49	185	6	1.60
dk16	5	92	520	5	92	464	17	5	91	447	31	3.66
dk27	3	18	49	3	18	45	0	3	16	42	0	6.67
dk512	4	33	113	4	32	101	1	4	30	92	2	8.91
ex1	5	96	531	5	94	434	14	5	95	422	33	2.76
ex4	4	41	159	4	41	138	1	4	41	135	4	2.17
ex7	4	32	127	4	31	114	1	4	28	92	2	19.30
keyb	5	72	620	5	72	592	15	5	72	585	35	1.18
planet	6	146	1137	6	146	1007	52	6	144	978	167	2.88
sse	4	49	227	4	49	188	3	4	49	185	6	1.60
styr	5	134	1045	5	134	905	47	5	134	879	98	2.87

Table 1. Experimental results