Formally Verified Redundancy Removal

Stefan Hendricx Luc Claesen

IMEC vzw/Katholieke Universiteit Leuven
Kapeldreef 75, B-3001 Heverlee, Belgium
e-mail: Stefan.Hendricx@imec.be

Abstract

In general, logic redundancy tends to degrade design-quality by introducing additional delays in signal propagation, by increasing the gate count or simply by making the resulting hardware untestable. Since they cannot always be avoided, unwanted redundancies have to be first identified and then removed from our designs. In this paper, an alternative methodology to identify and remove redundancy is proposed, which is based on a formal, symbolic verification strategy. The formal framework underlying our approach aids in identifying redundancies and allows us to guarantee the correctness of their removal.

1. Introduction

During the electronic hardware design process, logic redundancies are sometimes deliberately introduced, in order to improve certain performance aspects — e.g. timing, stability or fault tolerance — of a given circuit. Such redundancy must however be added with care, since it does not necessarily lead to an improved design. More specifically, logic redundancies generally tend to impair a circuit's quality by introducing extra delays in signal propagation, by increasing the gate count — and thus the final silicon area — or simply by making the resulting hardware untestable.

At this point, one could simply argue not to introduce such detrimental redundancies. However, it turns out that most of the time, redundancies enter logic designs unintentionally, and therefore unnoticed. In [3], for instance, D. Bryan et al. give a clear overview of how undesired redundancies can inadvertently be created during the design process. For one, they demonstrate that structural hierarchy can cause redundancies, even though the hierarchical components themselves are proven to be irredundant.

Since they cannot always be avoided and they degrade the overall design-quality, unwanted redundancies have to be first identified and then removed. For that purpose, various approaches have been proposed and implemented [3, 7]. Common to most identification approaches, however, is that they are mainly based on deterministic test pattern generation (DTPG) algorithms — e.g. the D, the PODEM, and the state-of-the-art FAN algorithms [1].

In this paper, an alternative methodology to identify and remove redundancy is proposed, which does not make use of a DTPG-based identification strategy. Instead, our methodology is based on a formal, symbolic verification strategy. As will be demonstrated, the formal framework behind our approach aids us in identifying redundancies and allows us to formally guarantee the correctness of their removal. The key message we want to convey in this paper is that it is possible to identify logic redundancy by reasoning on a set of formal (Boolean) expressions, without the need to examine specific test patterns.

This paper is organized as follows. In the next section, some basic concepts on redundancy are defined. Section 3 describes the principles underlying our symbolic identification strategy. In section 4, we demonstrate how the identified redundancies can be removed, guaranteeing the formal correctness of the circuit. To illustrate our symbolic approach, a few practical examples are discussed in section 5. Finally, section 6 summarizes our conclusions.

2 Preliminaries

In this section, some basic concepts on logic redundancy are defined and illustrated. For reasons of simplicity, these definitions have been kept informal. They can, however, be expressed in terms of a more formal, directed-acyclic-graph (DAG) representation of combinational circuits.

Definition 1: Node N *of circuit* C *is called a* redundant node, *if the observable input-output behaviour of* C *is independent of the value of* N, *for all possible states and all possible input combinations of* C.

Based on this definition, node X in figure 1 is clearly a redundant node of the circuit, since the output is independent of the value of X.

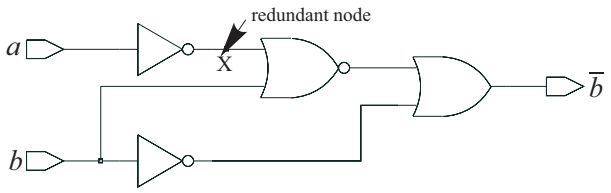


Figure 1. X is a redundant node, since the output of the circuit is independent of X's value.

Definition 2: Gate G *of logic circuit* C *is called a* prime-redundant gate, *if at least one of its inputs can be removed without affecting the observable input-output behaviour of* C, *for all possible states and all possible input combinations of* C.

Consider the example in figure 2. It is relatively easy to check that the *And*3-gate in this circuit is prime-redundant; when Boolean values *a*, *b*, *c* and *d* are applied at the inputs of the circuit, the inputs of the *And*3-gate are $a + b$, $a + \overline{c}$ and $c.d$. Clearly, the output expression $a.c.d$ only depends on the Boolean conjunction of $a + \overline{c}$ and $c.d$, whereas the third input of the *And*3-gate can be removed (together with the associated logic).

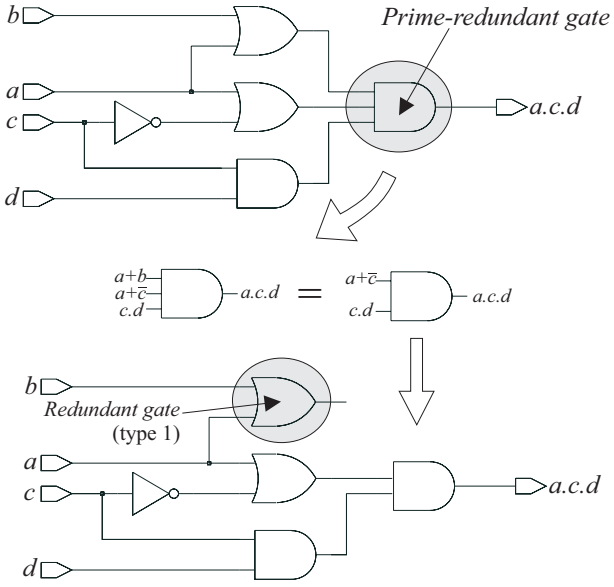


Figure 2. Example of a prime-redundant gate.

Definition 3: Gate G *of logic circuit* C *is called a* redundant gate *(type 1), if* G *can be removed without affecting the observable input-output behaviour of* C, *for all possible states and all possible input combinations of* C.

As an example, consider again the circuit in figure 2. Once the redundant input of the prime-redundant gate has been removed, the gate driving that node becomes a type-1 redundant gate. Removing that gate does not affect the working of the circuit.

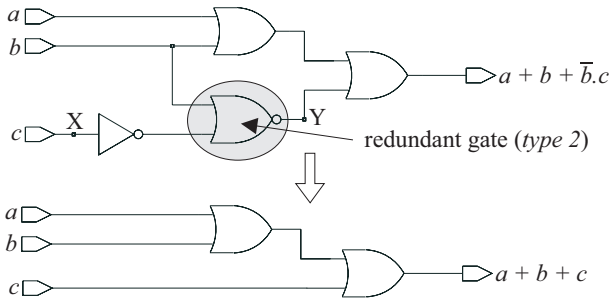


Figure 3. Example of a type-2 redundancy.

Definition 4: Gate G *of logic circuit* C *is a* redundant gate *(type 2), if the set of nodes* $\{N_i\}$ *driven by* G *can be replaced by a set of 'earlier'* [1] *nodes* $\{M_j\}$ *without affecting the observable input-output behaviour of* C, *for all possible states and all possible input combinations of* C.

To illustrate the notion of type-2 redundancy, we can take a look at the circuit in figure 3. Here, it is possible to replace node *Y* of the *Nor*-gate by the 'earlier' node *X*, without changing the output function of the circuit itself. Therefore, the *Nor*-gate can be labelled a type-2 redundant gate.

Our definition of type-2 redundancy closely resembles the general type of redundancy defined by Hayes [6]. Hayes indicated that circuits exhibit redundancy when it is possible to cut a set of *m* wires and to connect a subset of those cut wires to other wires in the circuit without changing its observable input-output behaviour. Definition 4 merely restricts Hayes' general definition to individual gates.

3 Redundancy Identification

3.1 DTPG-based Identification

The basic principle underlying DTPG-based redundancy identification is to identify redundant nodes by searching for undetectable stuck-at-faults. More precisely, DTPG-based identification compares the original circuit with the same circuit containing an assumed stuck-at-0 or stuck-at-1 fault. When no input combination (or test pattern) can be generated to distinguish the faulty circuit from the correct circuit, then — according to definition 1 — the corresponding stuck-at node is a redundant node. Once redundant nodes have been identified, simplification rules will be applied to remove the associated redundant logic [1].

[1] A set $\{M_i\}$ is called a set of 'earlier' nodes with respect to a set $\{N_j\}$, if each node M_i is (logically) independent of all the nodes in $\{N_j\}$

Curiously enough, DTPG-based approaches are only concerned with identifying redundant nodes. Gate redundancy is only dealt with indirectly through the simplification rules. For instance, if an undetectable stuck-at-1 fault is identified on an input node of an *And*-gate, that input can be removed from the gate without changing the input-output behaviour of the circuit. In our terminology, such an *And*-gate is a *prime-redundant* gate.

The narrow focus on node redundancy could be considered a shortcoming of DTPG-based identification. Closer examination reveals that node redundancy is merely a 'symptom' of gate redundancy. More specifically, each redundant node indicates the existence of a redundant logic gate. The reverse, however, is not necessarily true. A logic circuit may contain redundant gates without exhibiting redundant nodes. The most obvious example of this phenomenon is depicted in figure 4. This very simple circuit contains type-2 gate redundancy, yet no redundant nodes will be identified by standard DTPG-based techniques (see note on type-2 redundancy).

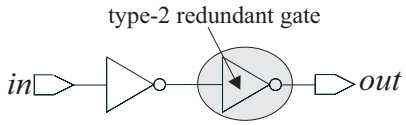


Figure 4. A logic circuit containing (type-2) gate redundancy, but no redundant nodes

The reader will agree that, instead of looking for the symptoms, it could be more efficient to immediately track down the actual causes — i.e. gate redundancies — themselves. And that is exactly the objective of the methodology presented in the remaining sections.

Note on type-2 redundancy

It is important to note that for identification techniques based on DTPG-algorithms, most of the concepts in section 2 are defined in terms of undetectable stuck-at-faults. Type-2 redundancy, however, deals with a much more general concept of redundancy than the concepts associated with the existence of undetectable stuck-at-faults. Consequently, standard DTPG-algorithms cannot directly deal with type-2 redundancy.

Indirectly, type-2 redundancy can be identified by modifying the initial circuit – e.g. by inserting well-chosen XOR-gates, as is described in [2] – and to examine the testability of these modifications. But modifying the circuit under investigation may still be seen as a disadvantage. In the following sections, we will point out that our approach is more successful in dealing directly with this more general notion of redundancy.

3.2 Symbolic Redundancy Identification

As mentioned before, our redundancy identification is based on a formal verification strategy. In this section, the concepts underlying that strategy are discussed.

3.2.1 Formal Verification Strategy

In the past, various strategies have been devised for the formal verification of electronic designs — e.g. model checking, equivalence checking and theorem proving. The work presented in this paper is based on an equivalence checking strategy.

Equivalence checking aims to prove that the observable input-output behaviours of two circuits are logically equivalent. For instance, equivalency can be proven by exhaustively comparing the truth-tables of both circuits. In a formal approach, however, the behaviour of each circuit is first formally analyzed, after which the equivalency of the derived formal expressions is examined.

As part of an experimental verification framework, we developed a symbolic analyzer for logic circuits. Symbolic analysis automatically constructs Boolean expressions for the functional behaviour of a circuit (see e.g. [5]). For that purpose, symbolic values (or functions) are applied at the inputs of the circuit and are 'propagated' through the logic network. Finally, the outputs of the circuit will be expressed in terms of the symbolic values applied at the inputs.

Formally proving the equivalence of the expressions produced by the symbolic analyzer is relatively straightforward. Internally, the symbolic values (and functions) are constructed using Binary Decision Diagrams (BDDs) — an efficient technique to represent Boolean functions [4]. To prove that two symbolic expressions are logically equivalent, we need only to check that they refer to the same BDD.

3.2.2 Formal Redundancy Identification

Using our symbolic analyzer, we can also extract a set of Boolean expressions describing the full behaviour of a circuit — i.e. expressions for *all* the nodes. Because of the memory overhead involved, such an approach is usually undesirable. For circuits that suffer from an exponential explosion in the size of their BDD-representations, as is the case for multipliers, such a description of the full behaviour may even be impossible [4]. Nevertheless, when possible, a complete set of expressions does have its own advantages. For one, it allows us to formally reason about redundancy.

Interested in the possibility of a formal approach to identify logic redundancies starting from a set of Boolean expressions, we finally established four generic identification (and removal) rules (see section 4). As the examples in section 5 will demonstrate, these simple rules allow us to identify and to remove redundancy in logic circuits. Moreover,

these rules formally guarantee the correctness of the redundancy removal.

3.2.3 Formal Correctness of Removal

A major advantage of the identification rules in section 4 is that they guarantee the functional correctness of the redundancy removal. More precisely, during the identification process, it can be guaranteed that our rules will preserve the input-output behaviour of the circuit under investigation.

Basically, three types of rules can be distinguished. In the first place, we have rules that explicitly remove logic gates from a circuit. Such rules preserve the correctness, because they only remove gates that have absolutely no effect on the functionality of the circuit.

The second type of rules locally transforms prime-redundant gates into functionally equivalent units by disconnecting one of the inputs. Again, the correctness of such a transformation can be guaranteed, because the symbolic expressions for the output nodes of such a gate are never changed by the rule. Therefore, such a local transformation will have no effect on the overall circuit behaviour.

The third type of rules selectively replaces circuit nodes by 'earlier' nodes. For instance, when two nodes are characterized by equivalent symbolic expressions, it is possible to replace one node by the 'earlier' node, without changing the overall functionality, thus guaranteeing the correctness of the resulting circuit. In a similar fashion, when one can replace the input node of a logic gate by an 'earlier' node, such that the output expression of that gate remains the same, the resulting circuit will be equivalent to the initial circuit.

4 Rule-based Identification and Removal System

To examine the feasibility of the proposed methodology, our symbolic circuit analyzer was extended with a rudimentary rule-based identification system. In this section, the generic set of identification (and removal) rules that have been considered so far, are presented. The next section will illustrate the application of these rules at the hand of some examples.

The first rule is responsible for explicitly removing logic gates whose outputs have been disconnected from the circuit. Such gates are typical representatives of type-1 redundant gates.

- *Rule* 1: **If** a logic gate *G* drives no output nodes of a circuit *C* and *G* drives no other logic gates in *C*, **then** remove *G* from *C*.

Clearly, when a logic gate doesn't drive other logic gates *and* doesn't drive the output nodes of a circuit, it can simply be removed without affecting the functional behaviour of that circuit. By itself, such a rule will only partially cover type-1 gate redundancy. Together with the following rules, however, rule 1 proves to be sufficient to remove redundant gates in general.

- *Rule* 2: **If** the outputs of a logic gate *G* with N inputs depend only on $N-1$ of those inputs, **then** disconnect the N^{th} input node from *G* and replace *G* by an equivalent gate with only $N-1$ inputs.

Rule 2 deals with prime-redundancy. For instance, when one of the inputs of an *And3*-gate is characterized by a constant symbolic value *1* (for all possible input combinations), rule 2 states that the redundant input node can be disconnected from the *And3*-gate *and* that the *And3*-gate itself can be replaced by an *And2*-gate of the remaining two inputs.

- *Rule* 3: **If** the symbolic value on an output node N_o of a logic gate *G* is equivalent to the symbolic value on an 'earlier' node N_e, **then** replace N_o by N_e.

Clearly, rule 3 deals with type-2 gate redundancy. For instance, when one of the two inputs of an *And2*-gate is characterized by a constant symbolic value *1* (for all possible input combinations), rule 3 states that the output node of the *And2*-gate can be replaced by the second input of that gate, without changing the behaviour of the circuit.

- *Rule* 4: **If** an input node N_i of a logic gate *G* can be replaced by an 'earlier' node N_e and still preserve the symbolic value on the outputs of *G*, **then** replace N_i by N_e.

Rule 4 deals with a much more complicated form of type-2 gate redundancy. A typical application of this rule has already been illustrated in figure 3. Node *Y* can be replaced by node *X*, without affecting the symbolic value on the output of the *Or*-gate. After disconnecting node *Y* — and successively applying rule 1 — the redundant inverter and *Nor*-gate will be removed as well.

We have implemented the generic identification rules for each logic gate supported by our verification tool. More precisely, each rule is directly integrated into the object-oriented implementation of the symbolic analyzer itself. This allows us to automatically proceed to the identification of redundancy, once the symbolic analysis has determined a set of symbolic (Boolean) expressions denoting the functional behaviour of a given circuit.

Note on search strategy

It is important to emphasize that the above rule-based system is not intended for optimization purposes. Rule-based systems for logic optimization are far more complex, since they require elaborate search strategies, backtracking

facilities, efficient management of intermediate results, precise definitions of cost and performance, etc. Our objective is simply to make a circuit irredundant, not to obtain the most 'optimal' solution.

In this sense, our rule-based system is very simple. Basically, there is no real search strategy. When a rule *can* be applied, it *is* applied, 'transforming' the circuit currently in our memory. If more than one rule applies, the rule that will be executed is determined according to some predefined priority (or simply selected randomly). In other words, we do not need to keep track of all the intermediate results and we do not need to implement backtracking, allowing a straightforward implementation of the identification system.

5 Examples

5.1 Example 1

The logic circuit depicted in figure 5 was first examined by Bryan et al. in [3], where they showed that the given circuit contains several logic redundancies. In this section, we illustrate the identification and removal of those redundancies, using the rules discussed in the previous section.

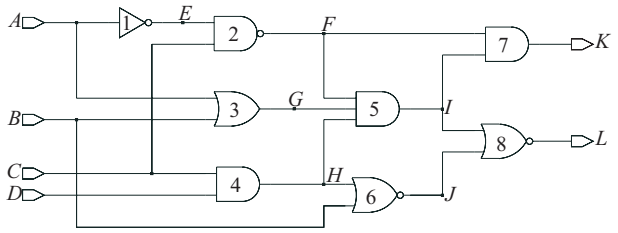
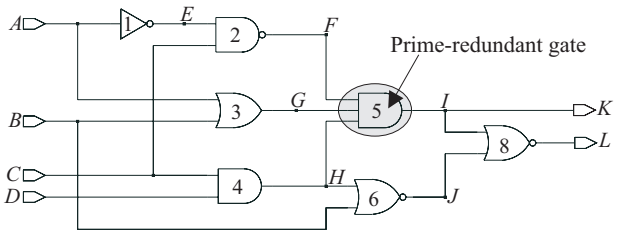


Figure 5. A circuit containing redundancies

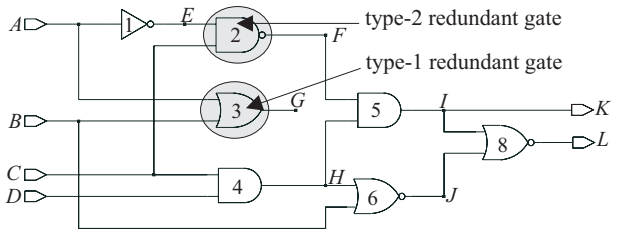
A	:	a	G	:	$a+b$
B	:	b	H	:	$c.d$
C	:	c	I	:	$a.c.d$
D	:	d	J	:	$\overline{b}.(\overline{c}+\overline{d})$
E	:	$\overline{a}$	K	:	$a.c.d$
F	:	$a+\overline{c}$	L	:	$b.(\overline{c}+\overline{d}+\overline{a})+(\overline{a}.c.d)$

Table 1. Boolean expressions obtained by symbolic analysis of the circuit in figure 5

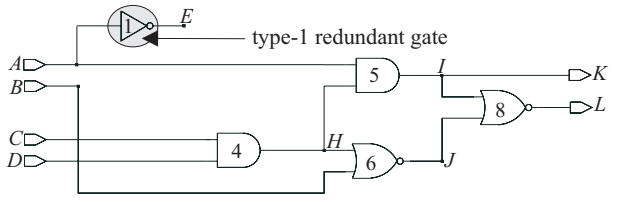
Symbolic analysis of this circuit provides us with the set of Boolean expressions depicted in table 1. Independent of the inputs, the values on nodes *K* and *I* will always be the same. This means that rule 3 can be applied to gate 7 — i.e. gate 7 is a type-2 redundant gate. Applying rule 3 interconnects nodes *K* and *I*, and disconnects the output of gate 7 from the circuit. Subsequently, rule 1 can be applied to remove gate 7, resulting in the following (equivalent) circuit.



We can also apply the second rule to gate 5. Gate 5 is a prime-redundant gate, because input *G* can be removed without changing the value on node *I*. Following rule 2, gate 5 can be replaced by an *And2*-gate and node *G* can be disconnected from the rest of the circuit. Once node *G* has been disconnected, gate 3 becomes a type-1 redundant gate, and will be removed from the circuit by applying rule 1.



The next logic redundancy to be identified is less obvious. As it turns out, rule 4 can be applied to gate 5 — node *F* can be replaced by the 'earlier' node *A* without changing the value on node *I*. Once *F* has been disconnected from gate 5, rule 1 applies to gate 2. Removal of gate 2 results in the following circuit:



Evidently, the removal of gate 2 makes gate 1 a type-1 redundant gate. After removing the redundant inverter, no more rules apply to the resulting circuit (see figure 6) — i.e. the circuit is irredundant.

5.2 Example 2

The circuit depicted in figure 7 is another example containing logic redundancy. Close examination of the circuit

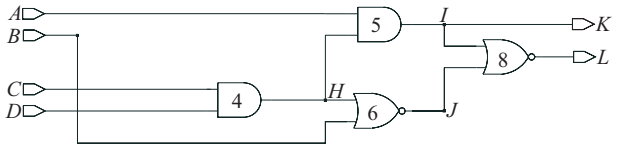
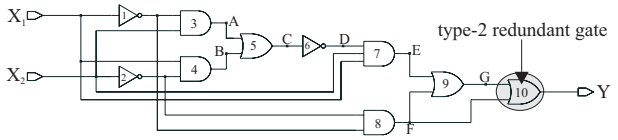


Figure 6. Irredundant version of example 1

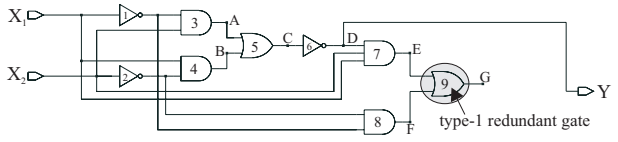
reveals that gate 10 is both *prime-redundant* and *type-2 redundant*. First, gate 10 is type-2 redundant since its output node *Y* can be replaced by the 'earlier' node *D*. In addition, gate 10 is prime-redundant since the value of its output depends only on the value of its input *G*.



X_1	:	a	D	:	$\overline{a}.\overline{b} + a.b$
X_2	:	b	E	:	$a.b$
A	:	$\overline{a}.b$	F	:	$\overline{a}.\overline{b}$
B	:	$a.\overline{b}$	G	:	$\overline{a}.\overline{b} + a.b$
C	:	$\overline{a}.b + a.\overline{b}$	Y	:	$\overline{a}.\overline{b} + a.b$

Figure 7. A logic circuit containing redundancy

Because of its dual nature, more than one rule applies to gate 10 — e.g. rule 2, rule 3 and rule 4. Here, we assume that rule 3 has the highest priority and is executed first. In that case, nodes *Y* and *D* are interconnected and gate 10 is removed, resulting in the following circuit.



Once gate 10 has been removed, gate 9 automatically becomes type-1 redundant. Rule 1 can then be applied to remove that gate. Removing gate 9, in turn, causes gates 7 and 8 to become type-1 redundant. In other words, rule 1 can again be applied (twice) to remove both gates 7 and 8. At that point, no more rules apply and the resulting circuit (see figure 8) is irredundant.

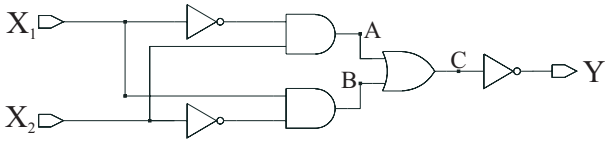


Figure 8. Irredundant version of example 2

6 Conclusion

The key message conveyed in this paper is that it is possible to identify (and to remove) logic redundancy by reasoning on a set of formal (Boolean) expressions. To demonstrate this key idea, an identification methodology based on a formal verification strategy has been proposed. To examine the feasibility of this methodology, a rule-based identification system was implemented and successfully applied to a number of examples. It appears that a limited set of identification rules proves to be sufficient to remove the redundancies in logic circuits. In addition, the formal framework underlying our approach allows us to guarantee the correctness of the redundancy removal.

Acknowledgments

The research presented in this paper was supported by a scholarship from the Flemish Institute for the promotion of Scientific-Technological Research in Industry (IWT).

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design (revised printing)*. IEEE Press, 1990.

[2] D. Brand. Verification of Large Synthesized Designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 534–537, Santa Clara (CA), Nov. 1993. IEEE.

[3] D. Bryan, F. Brglez, and R. Lisanke. Redundancy Identification and Removal. In *Proc. Intl. Workshop on Logic Synthesis – IWLS'89*, Research Triangle Park, N.C., May 23–26 1989.

[4] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[5] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[6] J. P. Hayes. On the Properties of Irredundant Logic Networks. *IEEE Transactions on Computers*, C-25(9):884–892, Sept. 1976.

[7] M. H. Schulz and E. Auth. Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques. In *International Symposium on Fault-Tolerant Computing*, June 1988.